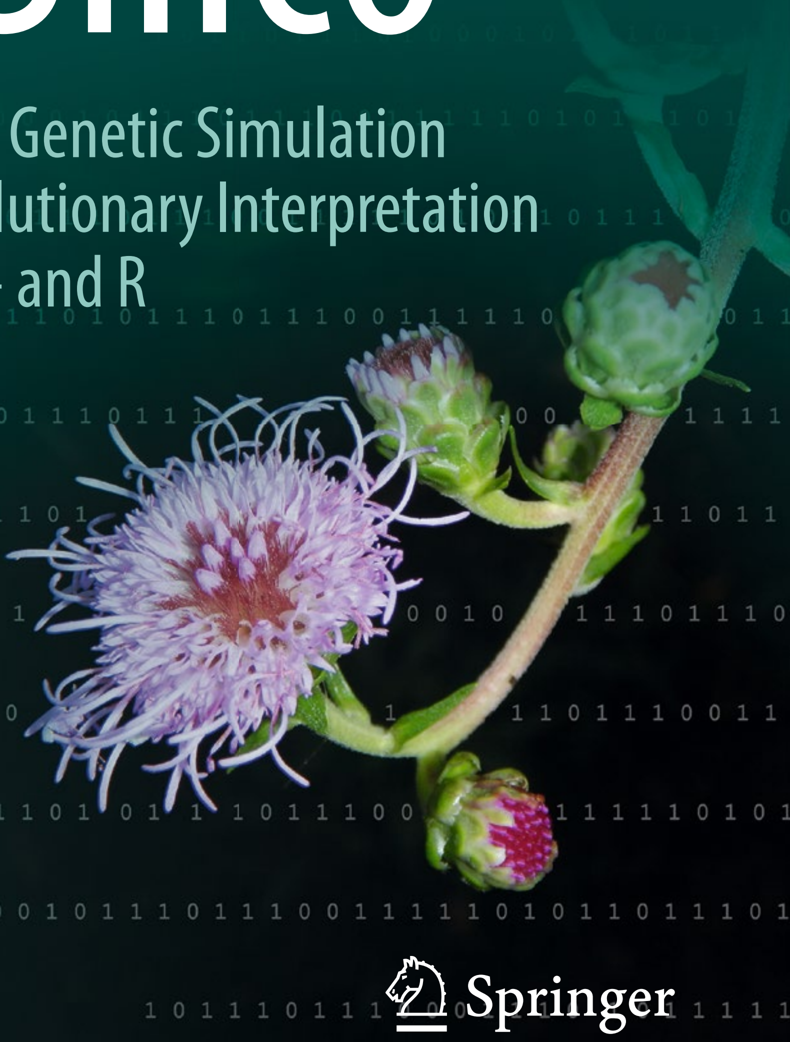
Ryan J. Haasl

Nature in Silico

Population Genetic Simulation and its Evolutionary Interpretation Using C++ and R



Springer

Nature in Silico

Ryan J. Haasl

Nature in Silico

Population Genetic Simulation
and its Evolutionary Interpretation
Using C++ and R

Springer

Ryan J. Haasl
Department of Biology
University of Wisconsin-Platteville
Platteville, WI, USA

ISBN 978-3-030-97380-3 ISBN 978-3-030-97381-0 (eBook)
https://doi.org/10.1007/978-3-030-97381-0

For my bright stars, Shannon and Charlotte

Preface

Oxen that rattle the yoke or halt in
the shade, what is that
you express in your eyes?

Leaves of Grass [*Song of Myself*]
Whitman, 1855

The ancient Greeks called the
world *κοσμος*, beauty.

Nature
Emerson, 1836

So much of what I read brings to mind evolutionary biology. Whitman's oxen *express* the ancientness of their ancestry, their genealogy, through their eyes. Their muscular strain against the yoke—the culmination of millions of years of knitting muscles now used to ease the labor of humanity. Their respites in the shade of a tree, acknowledgments of physiological signals honed in their ancestors, and their ancestors. As you see, any word of nature can send me spinning down semi-poetic thoughts of the history of life on Earth.

The material Universe is a marvel. Our local world presents glorious, ungroomed beauty (*κοσμος*), animate flesh assembled as transient bodies, the history of evolution written in their mortal (but heritable) genomes, morphology, physiology, and behavior (however broadly you might like to define that last term). As biologists, we appreciate the wildly complex interplay of abiotic and biotic factors that shape the diversity of life observed by each of us each day on Earth. Therefore, it may be difficult for purists to accept simulation of this marvel as a legitimate form of exploration in the biological sciences. However, the use of in silico in the title of this book should not detract from the often childlike wonder the organic world engenders, the existential wonder that gently directed so many of us to careers in science.

But recall that the world we find ourselves within is the current though ever-fleeting endpoint of innumerable stochastic processes that have played out since the origin of the Universe, constrained only by physical laws and what has already unfolded in the history of life. Doesn't that sound amenable to a bit of algorithmic imitation? By aping these processes in computer code, we can (with error) forecast the future direction of biological entities and (with error) infer past evolutionary processes that led to the measured characters of a contemporary genetic sample.

Some have posited a "digital physics," which (put crudely) amounts to the idea that evolution of the Universe is the output of a computer program (Zuse 1970). Imaginative explorations of this very idea are also found in the realm of science fiction, demonstrated perhaps most famously by the film *The Matrix*, which invokes gnostic thought in its conception of (1) an over-world and (2) a perceived world. Simulating perceived worlds of any complexity would require computing power well beyond our species' current capability. However, fictional representations of *reality as simulation* are instructive. They remind us that no matter the true, basal reality, we are all in some, perhaps uncomfortable, reductionist sense just data. This is good! It should mean we are capable of using computer simulation to capture something of the complicated network of interactions and quasi-rules of biology comprising "the force that through the green fuse drives the flower " (Thomas 1934).

Simulating the natural world necessarily requires us to (1) simplify, to ignore the factors of lesser effect size, and (2) question the epistemological utility of computer simulation in science (Chap. 1). Evolutionary biology is a science of history. Thus, it is natural to follow a retrospective approach in which we query the past for signal of the evolutionary factors that have produced the genetic variation we sample today in fields, ponds, forests, oceans, and the riparian zones beside rushing rivers. Retrospective (coalescent) simulation does and should play a large role in the types of intellectual investigations we focus on here (Chap. 2).

However, the majority of this volume details prospective, forward-in-time simulations, as these allow us to simulate more complex scenarios that inch us a bit closer to "reality." Chaps. 3–8 cover the causes of molecular evolution in an order of introduction that seems natural to me: mutation and population size (Chap. 3), demographic change (Chap. 4), meiotic recombination (Chap. 5), population structure and migration (Chap. 6), natural selection (Chap. 7), and the effects of natural selection on linked variation (Chap. 8). This order is not agreed upon, as a casual perusal of population genetic/genomic textbooks will reveal. My apologies to those whose preferred order is violated by my choices herein. The final chapter of this volume—Chap. 9—delves into simulating phenotypes, specifically quantitative (complex) trait values, including their evolution by artificial and natural selection.

A major goal of this volume is to walk you through the construction of a reasonably multipurpose, prospective (forward-in-time) simulation program. The program we add to as we progress through Chaps. 3–9 is named

FORward Time simUlatioN Application, or FORTUNA (see Sect. 1.4.2). An appendix provides a glossary of parameters introduced throughout. It is certainly possible for the reader to read the logic behind the coding without the need to fully digest digressions into the specific meaning of lines of code and the use of FORTUNA to investigate their research goals.

Finally, I need to mention that I am a self-taught programmer and am sure to have broken more arcane rules regarding the proper syntax and use of language idioms in R and C++. However, my philosophy has always been that if I can code something that compiles successively and reliably does what I expect it to do in a reasonable amount of compute time—*and* I can explain it to someone else—well, then we are golden.

Zuse K (1970) *Calculating Space*. New York: MIT Technical Translation AZT-70-164-GEMIT:1–98.

Platteville, WI, USA Ryan J. Haasl
January 2022

Acknowledgments

... it seemed to me suddenly that my humble life and the realms of the truth were not as widely separated as I had thought...[1]

– Marcel Proust, *Swann's Way*

As always, I must begin by thanking my wife Shannon and daughter Charlotte for putting up with me sitting in front of a terminal for days and nights on end as I fret about the scaling of curves and the demands of data structures. My mother, Mary, and maternal grandparents, Dawne and Leonard, are also largely responsible for my maturation as a scholar. Growing up in small-town Central Wisconsin, they reliably fostered my intellect despite limited means. I am also deeply indebted to my PhD advisor Bret Payseur, who encouraged my innate wonder of the natural world and ultimately helped me become a real scientist. Many thanks to the thousand and more students I have taught in genetics as well as systematics and evolutionary analysis at the University of Wisconsin-Platteville. Collectively, they have helped me become a better teacher of admittedly difficult concepts. I also appreciate the home I have found in the Biology Department at UW-Platteville, peopled with down-to-earth colleagues who strive each day to support each other and the students they serve. Finally, sincere thanks to those who have aided in the publication of this volume at Springer Nature, including Shina Harshavardhan, Rivka Kantor, and Kenneth Teng. In particular, I want to thank Janet Slobodien who was gracious enough to consider and, ultimately, accept a book proposal from a junior faculty member at a small university.

[1] Quoted with permission from the translation by Lydia Davis, 2004, Penguin Classics.

Contents

Acronyms and Symbols

AFS	Allele frequency spectrum
bp	Base pair
LD	Linkage disequilibrium
MRCA	Most recent common ancestor
NFD	Negative frequency-dependent
QTL	Quantitative trait locus
SFS	Site frequency spectrum
SNP	Single nucleotide polymorphism
SNV	Single nucleotide variant
d	Number of demes
D	A measure of linkage disequilibrium
D'	Normalized measure of linkage disequilibrium
D_t	Tajima's D
f_A	Frequency of allele A
$f_{A/a}$	Frequency of genotype A/a
F	Observed heterozygosity
F_{ST}	Fixation index, a measure of population structure
h	Dominance coefficient
h^2	Narrow-sense heritability
h_i	Frequency of haplotype i
H_{exp}	Expected heterozygosity
H_{obs}	Observed heterozygosity
H^2	Broad-sense heritability
K	Carrying capacity or number of unique haplotypes
μ	Mutation rate (per-site/per-generation)
μ_{site}	
μ_{locus}	Mutation rate when the distinction is needed between per-single-site and per-locus
m	Generic migration rate symbol
$m_{i,j}$	Fraction of individuals in deme j that are immigrants from deme i
M	Population migration rate ($4N_em$ in diploid)

N	Census population size or generic symbol for population size
N_e	Effective population size
N_t	Population size, effective or not, at time t
π	Nucleotide diversity
p_i	Quantitative trait value of individual i
$P(e)$	Probability of event e
ρ	Recombination rate parameter
r	Exponential growth rate or recombination frequency
s	Selection coefficient
S	Number of segregating (polymorphic) sites
t	Generic variable for time
θ	Population mutation rate ($4N_e\mu$ in diploid)
θ_W	Watterson's θ
t_k	Time to next coalescence among k lineages
V_P	Phenotypic variance (of a quantitative trait)
V_E	Environmental variance (of a quantitative trait)
V_G	Total genetic variation (of a quantitative trait)
V_A, V_D, V_I	Additive, dominance, and epistatic genetic variance, respectively (of a quantitative trait)
$\bar{w}$	Mean population fitness
w_A	Marginal fitness of allele A
$w_{A/A}$	Relative fitness of genotype A/A
$w_{AB/ab}$	Relative fitness of two linked genes A and B (a dihybrid in cis arrangement in this example)

1 Simulation as a Form of Scientific Investigation

But that there were natural causes to all these things I am willing to concede, for the resources of nature are infinite apparently.
– Samuel Beckett, *Molloy*

1.1 Simulations as Enlivened Models

In the first episode of the *Discovery Channel*'s 2011 series *Planet Dinosaur*, the Cretaceous dinosaur *Spinosaurus sp.* is brought to life on the screen. The presented animation required a model of the dinosaur's anatomy based on fossil discoveries in Egypt and Morocco from the early twentieth century as well as more complete, more recent finds. Models are commonly imperfect representations of an entity or system that is either too complex for us to represent with exactitude or will always be impossible for us to experiment upon because the dynamics of the system evolve over expansive time periods or are not directly accessible. In evolutionary biology, models are used for all of these reasons. For example, current anatomical models of the full body of *Spinosaurus* necessarily involve uncertainty as only some of its various bony parts have been uncovered. In addition, its status as an extinct species precludes us from directly observing its true biomechanics, behavior, ecology, or fleshed-out appearance. Thus, paleontologists debate whether *Spinosaurus* was both an aquatic and terrestrial predator or one or the other and argue whether its enormous neural spines supported a fin or hump-like structure.

R. J. Haasl, *Nature in Silico*, https://doi.org/10.1007/978-3-030-97381-0_1

As another example, although the rapid evolution of phenotype occurs in response to artificial and natural selection, there are many objects of evolutionary interest that occur over time periods too lengthy to observe and monitor directly. Indeed, when we consider the radiation of a clade, the time period in question commonly spans millions of years. Moreover, the radiation of a clade from a single, ancestral species implies a host of details that the imperfect model can probably safely ignore: daily temperatures, most intra- and interspecific interactions, earthquakes, floods, wind speeds, etc.

For the purposes of this volume—and largely ignoring the extensive literature on scientific modeling (with no intended disrespect to those who think deeply on the subject)—I will treat a simulation as an *enlivened* model. Returning to the example of *Spinosaurus*, paleontologists have used the more-or-less complete fragments of individual *Spinosaurus* fossils to create models of the overall body of *Spinosaurus*. With the aid of anatomists and biophysicists, the capacity of this dinosaur to run or swim or bite was also modelled. Even at this stage, however, the model of *Spinosaurus* is a static thing, though it speaks to its potential energy to run and bite and swim. Only upon the application of computer-generated imagery (CGI) to the model do we transition to a simulation of the living *Spinosaurus*. In other words, CGI *Spinosaurus* is an enlivened simulation of the model.

Our models will be less fearsome. We will, for example, model mutation and genetic drift, the effects of demographic change on genetic diversity, and the evolution of a complex trait over time. However, the same analogy holds. A model of mutation will specify parameters such as the length of the sequence monitored, the point mutation rate, and the size of the population. This parameterized model is static, but it contains potential energy, if you will. Once instantiated in code that is run on a computer, the model comes alive and outputs evolutionary data that track the frequencies of derived alleles in the simulated sequence from generation to generation.

1.2 Borges: How Detailed Should a Model Be?

In his landmark (though regarded by many academics as specious) treatise *Simulacra and Simulation*, Jean Baudrillard (1994) begins with the example of Jorge Luis Borges' one-paragraph fiction, "On Exactitude in Science."[1] The beauty and relevance of this text make it worth quoting in its entirety:

[1] Quoted with permission from *Collected Fictions*, 1999, translator Andrew Hurley, Penguin Classics.

> ... In that Empire, the Art of Cartography attained such Perfection that the map of a single Province occupied the entirety of a City, and the map of the Empire, the entirety of a Province. In time, those Unconscionable Maps no longer satisfied, and the Cartographers Guilds struck a Map of the Empire whose size was that of the Empire, and which coincided point for point with it. The following Generations, who were not so fond of the Study of Cartography as their Forebears had been, saw that that vast Map was Useless, and not without some Pitilessness was it, that they delivered it up to the Inclemencies of Sun and Winters. In the Deserts of the West, still today, there are Tattered Ruins of that Map, inhabited by Animals and Beggars; in all the Land there is no other Relic of the Disciplines of Geography.

The text is attributed to a fictional Suárez Miranda; in other words, the story itself simulates the model of academic writing. More to the point, however, the story addresses the map-territory relation. A map is a representation/model of the territory, and as Alfred Korzybski famously remarked, “The map is not the territory.” Model and reality are joined by a sign of approximation rather than identity.

Yet taking a geographical map as a pedestrian example, models may offer significant utility. Despite their stripped-down detail, we use maps to successfully navigate the real world. A paper or electronic road map allows us to find the location of a social gathering even when the house of our host is not explicitly included. More radically, the road map certainly does not include the blue jay splashing in the bird bath when we arrive or the letters and packages in our host’s mailbox. These minute, real-time details are simply not necessary for us to find our destination using the map as a guide.

When the Cartographer’s Guild of the story goes big and constructs a point-for-point map of the Empire at the scale of 1:1, people of the Empire find the map “Useless.” How could a map as big as the world it represents serve as a practical navigational aid or provide a digestible synopsis of the Empire’s reach? The title Borges ascribes to the story is telling. As scientists who model the natural world, the essential tension of the model lies between too much detail and not enough. That is, what level of “exactitude” is required for usefulness?

As the example of the road map shows us, a model can be immensely useful despite far-less-than-perfect precision. Furthermore, Borges’ fiction warns that construction of a perfect model entails considerable labor not well spent. Baudrillard (1994) writes,“To simulate is to feign to have what one doesn't have. [...] But it is more complicated than that because simulating is not pretending." Models and enlivened simulations may only ape nature, given that each ephemeral bubble of the stream and each genetic locus are not represented. However, when we engage in modeling and simulating the natural world, we are not simply pretending—i.e., *playing* at world building. Successful simulation requires us to identify a model that includes the essential details of our natural subject and put them in motion; the output of the simulation thus references nature.

1.3 A Short, Selective History of Computer Simulation

1.3.1 Origination of the Monte Carlo Method

Georges-Louis Leclerc, Comte de Buffon was a French polymath who is often referred to in histories of evolutionary thought as one of the first humans to publicly contemplate questions that concern evolutionary biologists (Mayr 1981). In addition, the **Monte Carlo method** may be indirectly traced to his so-called "needle experiment," first proposed in 1733 (Buffon 1733) and published with an analytical solution in 1777. The *short-needle* variant of this experiment estimates the probability that a needle of length l when thrown randomly upon a wooden floor composed of an array of boards of equal width w (with $l < w$) will lay across the intersection between two floor boards (Fig. 1.1a). It can be shown that this probability $P = \frac{2l}{w\pi}$. Although Buffon did not propose the needle experiment for this purpose, rearrangement of this equation

$$\pi = \frac{2l}{wP} \tag{1.1}$$

suggests we can perform the needle experiment with known values of l and w to obtain an empirical estimate of P and therefore π. To be explicit, if we know l and w, unknowns π and P remain. However, we can obtain an approximation of P by actually throwing needles of length l at an array of floorboards whose widths are w. Plugging our estimate $\hat{P}$ into Eq. 1.1 then provides us with the estimate $\hat{\pi}$.

We next implement the needle experiment as an R function as an example of the Monte Carlo method and experimentation. We pass a single argument (n) to the function, which is the number of "needles to throw."

```
buffon <- function(n)
{
  r <- 0.25
  hits <- 0
  x0 = runif(n, 0, 20)
```

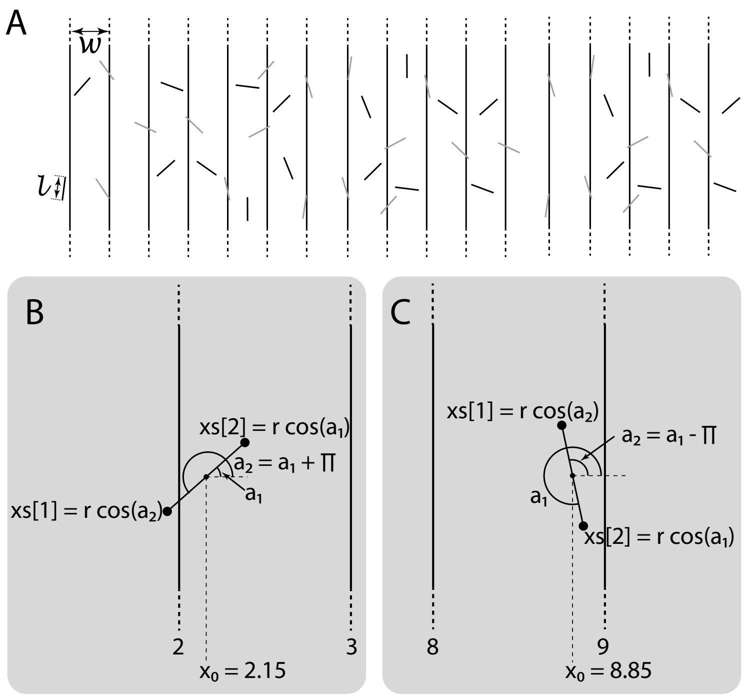


Fig. 1.1 Buffon's needle experiment. (**a**) The setup in which needles of length l are thrown on a floor whose boards are of width ω. Gray needles straddle a floorboard edge, while black needles land wholly on one floorboard. (**b**) An example of finding the endpoints of a thrown needle based on the midpoint of the needle along the x-axis (x_0) and angles a_1 and a_2; in this case, the needle counts as a "hit" because its lower endpoint overlaps the floorboard intersection at $x = 2$. (**c**) Another example, where the needle does not overlap either of its flanking floorboard intersections

```
  a1 = runif(n, 0, 2*pi)
  results <- list(vector(), vector())
  for (i in 1:n) {
    a2 <- 0
    if (a1[i] > pi) {
      a2 <- a1[i] - pi
    } else {
       a2 <- a1[i] + pi
    }
    x1 <- r*cos(a1[i]) + x0[i]
    x2 <- r*cos(a2) + x0[i]
    xs <- sort(c(x1, x2))
    if ( xs[1] <= floor(x0[i]) | xs[2] >= ceiling(x0[i]) ) {
      hits <- hits + 1
    }
    if (i %% 20 == 0) {
      results[[1]] <- c(results[[1]], i)
```

```
23        results[[2]] <- c(results[[2]], 1/(hits/i))
24      }
25    }
26    return(results)
27  }
```

In this function, we assume $w = 1$ and $l = 0.5$, which reduces Eq. 1.1 to $\pi = 1/P$.

Monte Carlo experiments use random sampling to solve a numerical problem; in many cases, the problem may be intractable. Famously, in 1954, Fermi, Pasta, Ulam, and Tsingou (Fermi et al. 1955) used random sampling to study a nonlinear problem in physics that (like many others) was not mathematically tractable:

> We decided to try a selection of problems for heuristic work where in absence of closed analytic solutions experimental work on a computing machine would perhaps contribute to the understanding of properties of solutions.

In the case of the needle experiment, where the random sample is obtained by either physically throwing the needles or simulating their throws as we do, the goal is to estimate P, which is then used to estimate π.

To simulate the landed position of a thrown needle, we need to generate two random numbers: (1) the point along the x-axis where the midpoint (`x0`) of the needle falls on a floor with 20 floorboard intersections (line 5) and (2) the angle in radians at which the axis of the needle lands relative to the x-axis (`a1`; line 6)—i.e., an angle of zero means the needle is parallel with the x-axis and has zero slope—from which the second angle (`a2`) is calculated (lines 9–14). Figure 1.1b–c provides two examples that illustrate the subsequent calculation of the two endpoints of a thrown needle and the determination of whether or not the needle straddles a floorboard intersection (lines 15–20). If the needle does cross a floorboard intersection, it is added to the tally of `hits` or successes. Every 20 needles, the updated estimate of $\pi = 1/P =$ `1/(hits/i)` is recorded (lines 21–24), and the results are ultimately returned by the function after all needles are thrown (line 26).

Figure 1.2 shows the results of three independent simulations in which 1,000,000 needles are thrown. The average of all three estimates upon simulating the throws of 1,000,000 needles is very near the true value of π. Note that we use the known value of π to find the endpoints of thrown needles (lines 10–14). This seems rather circular, but we are simply using our modern knowledge of trigonometry to accurately model where real needles would land. The random (Monte Carlo) determination of `x0` and `a1` is really what drives our estimate of π in the simulation.

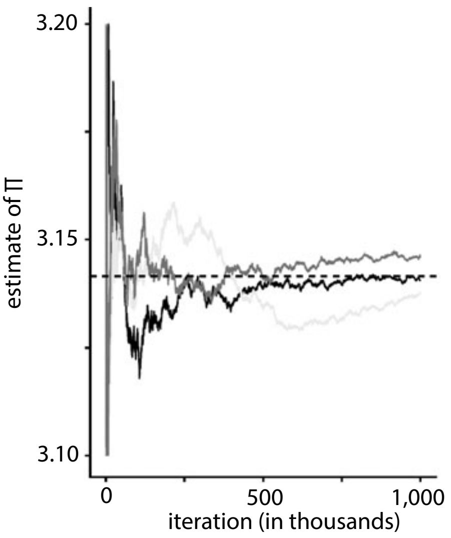


Fig. 1.2 Three independent simulations of Buffon's needle experiment. In each simulation, one million needles were thrown. The traces show the estimate of π versus the number of iterations (i.e., needles thrown). The mean estimate of π across the three replicates is very close to its true value (dashed line)

1.3.2 Early Computer-Based Simulation

Coincident with the culmination of World War II, the first generation of general-purpose electronic computers was developed. Notable among these computers was the Electronic Numerical Integrator and Computer (ENIAC), one of the first general-purpose electronic computers. The programmable capacity of these computers provided a means for scientists to work on problems that were mathematically intractable; the era of simulation-based inference and electronic number crunching was born. It is an unfortunate skeleton in the closet of scientific computing that the very first use of the ENIAC was to test the practicability of a thermonuclear weapon. In late 1945, Edward Teller, Nicholas Metropolis, and Stanislaw Ulam simulated the analytically intractable dynamics of a thermonuclear reaction.

John von Neumann and Ulam understood that machines like the ENIAC could run Monte Carlo experiments with much greater efficiency than the contemporary practice of legions of women—the original bearers of the title computer—performing massive quantities of hand calculations. In other words, it made sense to replace physical throwers of needles with electronic ones. Although early, computer-based Monte Carlo experimentation continued to be used in weapons research, the computational methods developed in this era paved the way for their later, more benign application. For example, by the late 1940s, Ulam developed the method of Markov chain Monte Carlo, which modern biologists use in a wide variety of statistical applications including Bayesian inference of phylogenetic trees (Huelsenbeck and Ronquist 2001) and the estimation of population structure from empirical genetic data (Pritchard et al. 2001).

In 1958, Keith Tocher developed the first general-purpose simulator named, rather obviously, the General Simulation Program (GSP), which was used for the specific task of simulating the workings of an industrial plant. Tocher later penned the first book on computer simulation, *The Art of Simulation* (1963). Shortly thereafter, Geoffrey Gordon developed a computing language named the General Purpose Simulation System (GPSS). It too was initially developed with business efficiency in mind and was often used for so-called waiting line problems—i.e., What are the dynamics of queues when there are more customers than service providers? During the same period, two important papers examined the problems associated with digital simulation (Conway 1963; Conway et al. 1959). Although the authors focused on *systems simulation*—as a modern example, consider the video game *SimCity*, where the output of a large number of interacting variables is simulated—they were motivated by a desire to describe *general* problems facing the practitioner of a digital computer simulation. Some of these problems still require careful consideration today, including management of memory and dealing with error associated with discretization of continuous quantities such as time.

1.3.3 *Early Simulations of Biological Evolution*

Next, I focus on two groups of biological researchers who pioneered the application of computer simulation to questions of evolutionary biology. I first discuss work by population geneticists in the 1960s, who were motivated by a desire to check the validity of their mathematical models using Monte Carlo experimentation. Second, we examine the early work of a group of rebellious paleontologists (and one ecologist) who simulated phylogeny as a stochastic process and helped drive the growth of a more quantitative paleontology with a decidedly evolutionary bent: paleobiology.

1.3.3.1 The Molecular Population Geneticists

The mid-to-late 1960s were banner years for the empirical and theoretical study of genetic variation in populations. Harris (1966), Hubby and Lewontin (1966), and Lewontin and Hubby (1966) produced empirical data documenting the existence of considerable variation in human and fruit fly (*Drosophila pseudoobscura*). In a real sense, these seminal papers provided theoretical population geneticists with a justification for their existence. Although phenotypic variation in nature was always evident, theory required underlying genetic variation for there to be anything to talk about. In the words of Hubby and Lewontin (1966):

> ... even without mathematics it is clear that genetic change caused by natural selection presupposes genetic differences already existing ...

In other words, the mathematics of population genetics only reached its full potential once empiricists began to characterize the grist (abundant genetic variation) for the mill (populations subject to evolutionary factors). Documented differences in electrophoretic mobility of an enzyme revealed by Lewontin and Hubby were distinct, co-dominant *phenotypes* themselves. Importantly, it was also true that these electrophoretic alleles were likely to be neutral and have no effect on conspicuous structural or physiological phenotypes.

Two years earlier, the infinite alleles model (IAM; Kimura and Crow 1964) had posited that (1) the "wild-type allele" is often a set of distinct DNA sequences that all map to the wild-type phenotype and (2) for the purposes of modeling genetic variation, it is quite safe to assume that each new mutation produces a previously nonexistent allele at the genetic level, without reference to the phenotype associated with each allele. In conjunction with steadily accumulating evidence of abundant genetic variation in natural populations, the idea that a protein-coding gene could be mutated in a number of ways with next-to-no effect on phenotype led Motoo Kimura to formulate his neutral theory of molecular evolution. The first expression of neutral theory (Kimura 1968) emphasized the role of genetic drift:

> Finally, if my chief conclusion is correct, and if the neutral or nearly neutral mutation is being produced in each generation at a much higher rate than has been considered before, then we must recognize the great importance of random genetic drift due to finite population number in forming the genetic structure of biological populations.

This focus on the abundance of neutral variants, whose frequencies in a population are determined by genetic drift and not natural selection (referred to as non-Darwinian evolution by King and Jukes (1969)), led Kimura to revisit Kimura and Crow (1964). He did so with a twist by modeling dynamics of the IAM using diffusion (Kolmogorov differential) equations and assuming only mutation and genetic drift as the stochastic drivers of allele frequencies and the number of alleles in a population (Kimura 1968).

Unlike the standard Wright-Fisher model, diffusion models treat allele frequency and time as continuous variables. Given an initial allele frequency, diffusion models provide a *probability distribution* of allele frequency at any given time. Thus, output of diffusion models explicitly acknowledges the probabilistic, random nature of microevolution under the IAM: given the starting frequency of an allele $x_{t=0}$ and effective population size N_e, we can quantify the probability, for example, that $0 \geq x_{t>0} < 0.01$.

The probabilistic results of the analytically derived diffusion model detailed in Kimura (1968) are attractive because they allow us to easily quantify the probability of allele frequency at some future time $t > 0$. However, tractability of diffusion models relies on some approximation—namely, replacing an explicit accounting of allele frequency (e.g., $1/2N$ in a diploid model, where N is population size) with a continuous variable. Thus, Ewens and Ewens (1966) and Kimura (1968) used Monte Carlo simulations of the IAM to assess potential loss of accuracy associated with the diffusion model approach.

Kimura (1968) used two programs written in Fortran and run with punch-card input on an IBM 7090 to implement his Monte Carlo experiments. Before moving on to the simulations, first, consider how the fully transistorized computer used by Kimura was marketed by IBM at a cost of nearly three million dollars (equivalent to 25 million USD today; IBM Data Processing Division 1960):

> Although the IBM 7090 is a general purpose data processing system, it is designed with special attention to the needs of engineers and scientists, who find computation demands increasing rapidly. As a scientific computing system, the 7090 will greatly speed the design of missiles, jet engines, nuclear reactors and supersonic aircraft.
>
> Four IBM 7090 systems are incorporated in the Air Force's Ballistic Missile Warning System, the 3000-mile radar system in the far north designed to detect missiles fired at southern Canada or the United States from across the polar region.
>
> Two IBM 7090 systems are being used by Dr. Wernher Von Braun's development group at the George C. Marshall Space Flight Center of the National Aeronautics and Space Administration, in Huntsville, Alabama.

The IBM 7090 and the follow-up 7094 were used to control the Mercury and Gemini space flights. Although renting the use of a 7090 cost hundreds of thousands of dollars per month (in today's dollars), it was as an old but reliable beast by the time Kimura used it in 1968; the first IBM 7090 was installed in 1959. However, access to such a machine was a privilege, particularly if you were a population geneticist and not working for NASA or a defense contractor.

In Kimura's 1968 simulations, each allele was accounted for each generation. A pseudo-random number generator was used to implement (1) mutation and (2) the random sampling of alleles from the previous generation. The diploid version of Kimura's program began with as many alleles as there were gene copies—i.e., $2N$ alleles. After a sufficient number of discrete generations, the factors of sampling error (drift) and mutation reached an equilibrium of sorts. As Kimura notes regarding a simulation where $N = N_e = 100$ (N_e is *effective* population size):

Starting with 200 alleles, the balance between mutation and random extinction of alleles had been reached well before generation 100.

For each of the nine runs of the diploid model, Kimura calculated the average value of the number of effective alleles (n_e) and the number of alleles (n_a) across all simulated generations following achievement of mutation-drift balance. n_e is the number of *equally frequent* alleles required to achieve the same level of heterozygosity as observed, while n_a is the actual number of distinct alleles in the simulated population. Unless the simulated alleles are of equal frequency, $n_a > n_e$. Although Kimura's conclusion is subjective and not subject to a goodness-of-fit test, Kimura was pleased with his comparison of observed values of n_e and n_a in the simulations to the expected values under the diffusion model:

Despite the smallness of population number assumed in these experiments, agreement between observed and expected is fairly good, except that the diffusion approximation tends to underestimate n_a.

As one example of his results, when he simulated $N = N_e = 100$ and $2N\mu = 1$, where μ is mutation rate, the run produced average $n_a = 9.68$ and $n_e = 3.13$; expected values were $n_a = 8.61$ and $n_e = 3.0$.

Ewens and Ewens (1966) only ran one Monte Carlo simulation, while Kimura (1968) ran two simulations of his haploid program and nine simulations of his diploid program. Considering that we could perform one of these Monte Carlo experiments in well less than a second today using a standard desktop computer, it is tempting to view these numerical experiments as laughable. However, these were pioneering experiments. Rather than adopting a cynical attitude toward their sluggishness, we should consider how any one of us can harness the immense computing power available to us today to best advance the science of population genetics. In my case, I know I will never possess the genius of a Kimura. So I view today's computing power as something of an equalizer; we can ask computers to simulate scenarios of a complexity that reaches well beyond what pure mathematical analysis would ever allow.

1.3.3.2 The Paleobiologists

Eldredge and Gould (1972) countered the assumption of phyletic gradualism (PG) inherent to the Modern Synthesis and Darwin's own thinking with an alternative hypothesis they called punctuated equilibria (PE; Eldredge and Gould 1972). The opposing concepts of PG and PE explain the origin of morphological divergence among species by either emphasizing the role of anagenesis (PG) or cladogenesis (PE). PG hypothesizes that the majority of divergence is due to anagenesis—changes within species lineages over long stretches of time—while PE suggests most divergence is due to a burst of morphological change at and shortly after (on a geologic scale) the cladogenetic event of speciation. Eldredge and Gould were particularly interested in the disparity between the phylogenetic expectations under PG and PE because numerous empirical case studies from the paleontological record supported morphological stasis within species over millions of years—i.e., anagenesis had a relatively small influence. Moreover, the early paleobiologists were keen to argue for macroevolutionary mechanisms that were decoupled from microevolution. The prevailing opinion, assumed at least since the Modern Synthesis, was that macroevolutionary patterns such as the presence of speciose, "bushy" clades versus small, species-poor clades should be viewed as the accumulated effect of microevolution.

Enter David Raup, a pioneer of a more-quantitative paleontology, the use of computer simulation in evolutionary biology, and the new discipline of paleobiology. In conjunction with like-minded paleontologists and ecologist Daniel Simberloff, Raup participated in the so-called MBL group, named after the site of their early meetings at the Marine Biological Laboratory in Massachusetts. The MBL group devised a simple simulation of something akin to a stochastic birth-death process, the results of which were visualized as phylogenetic trees and clade diversity graphs. For each discrete and unitless step/iteration of the simulation, one of the following happened to an extant lineage: (1) continued as its own lineage, (2) split into two species/lineages, or (3) went extinct.

Each simulation began with a single species and, initially,

$$P(speciation) > P(extinction), \tag{1.2}$$

which allowed for the tree to grow. In addition to the parameters for speciation probability and extinction probability, Raup et al. (1973) specified an equilibrium diversity parameter. Upon achieving an equilibrium, the relative probabilities of speciation and extinction were set to

$$P(speciation) = P(extinction). \tag{1.3}$$

Finally, two other parameters were required: (1) a damping factor, which controlled the magnitude of variation around the equilibrium diversity level, and (2) a clade size parameter, which specified the size of a monophyletic grouping (as the sum of the lineages' ages) required to simulate what amounts to an automated taxonomist declaring this or that monophyletic group as a clade.

The motivation for this project was to "test for correspondence between real data and the randomly generated results" (Raup et al. 1973); it should not escape our notice that this comparison of empirical and simulation-generated data is the beating heart of the modern suite of inferential methods called approximate Bayesian computation (ABC; Beaumont 2010). Equal extinction and speciation probabilities, following the attainment of equilibrium diversity level, were meant to model an evolutionary world in which microevolution, particularly natural selection, played no role. If simulation of the macroevolutionary processes of cladogenesis and extinction alone could produce cladograms similar to those estimated from nature, it might suggest there are unknown macroevolutionary (above-species-level) processes at work in producing macroevolutionary patterns visualized in cladograms.

The lack of any *directly* simulated microevolutionary factors in the MBL simulation implied an agnosticism to the causes of extinction and speciation. In the words of the authors:

> ... we suggest that an evolutionary event may depend upon the joint occurrence of many underlying causes, each achieving a specific probability of occurrence at a given time, so that the event itself can be predicted only in a statistical sense – even though it does, in fact, have a conventional cause.

So the authors treat cause of speciation as an integration of multiple and difficult-to-realistically-simulate factors. Again, avoiding direct simulation of microevolution suggests that any correspondence between simulated and empirical cladograms supports the existence of unknown macroevolutionary (above-the-species-level) processes at work in producing macroevolutionary patterns visualized in cladograms.

A stronger case for the lack of importance of natural selection to macroevolutionary pattern would be made by a simulation that explicitly includes natural selection and other microevolutionary factors (Turner 2011) whose results indicate that no matter how we vary the magnitude of these factors, we still obtain the same macroevolutionary results. This would provide strong support for a true decoupling of microevolution and macroevolution and perhaps speak to the relative importance of PG and PE models of macroevolutionary change. To the latter point, these first results reported by the MBL group are generated by a simulation that explicitly disallows speciation by anagenesis, which the authors refer to as "pseudo-extinction" of the ancestral species due to considerable within-lineage change. Still, the authors emphasize their belief in the relevance of their results to the natural world based on a comparison of simulated phylogenies and those of an empirical diversity record of reptiles. Particularly interesting is the observation that in one representative simulation, three distantly related taxa go extinct at the same time step. In practice, this simultaneous extinction of multiple, unrelated clades would likely be interpreted as evidence of a common cause of extinction. Given the independence of each lineage in the simulation, however:

> ... the stochastic approach invites the alternate hypothesis that the three extinctions had totally independent causes and that their coincidence in time was due to chance – well within the expectations of the appropriate stochastic model.

Due to limitations in computing power, only 24 independent runs of the program were performed. As with the population genetic simulations, references to the practical aspects of simulation in Raup et al. (1973) may strike us as remarkably outdated.

> Because of the limitations of computer storage, the number of lineages produced by any run was restricted to 500.

> Lineages are shown as a series of vertical dashes, one for each unit of time; the ancestry of lineages formed by branching is shown by horizontal series of commas. This form of output can be produced rapidly on a line printer.

However, this publication along with slightly earlier studies in adjacent disciplines (e.g., geology; Harbaugh and Bonham-Carter 1970) laid the foundation on which modern simulations of biologically relevant stochastic processes are built. Moreover, it is not a bad practice when building simulation programs to consider a world in which we have greater restrictions on computer memory and efficiency than we actually do. If taken seriously, this mental exercise can spur us to create more efficient code. Of course, deadlines exist and we may not have time to craft perfectly elegant code; sometimes, it is necessary and acceptable to write programs of brute force that take advantage of the vastly increased computing power currently available to us.

1.4 Philosophy and Simulation

1.4.1 Plato and Representational Fiction

One section of Plato's *Republic*, whatever its limitations as gauged by professional philosophers, may be, is worth starting with to engage our minds in the task of assessing the epistemological implications of computer simulation. In essence, can—and if so how—computer simulations help us learn about the natural world?

In Book III of *The Republic* Plato—in the voice of Socrates and the brothers Glaucon and Adeimantus—dialogue ensues upon the educational rules for the children to become Guardians (philosopher kings/rulers) in a utopian city free of the disparate hazards presented by democracy and oligarchy. These Guardians would rule not for personal gain but because their aptitudes and dispositions made it their clear destiny.

Socrates begins by examining the content of the literature (poetry and drama) appropriate to training budding Guardians. He distinguishes between *representational* and *narrative* fictions, and to make the point clear, Socrates recalls the very opening of *The Iliad* in which the priest Chryses implores Agamemnon to retrieve his daughter. Up to a point in this epic poem, Chryses is viewed in the third person as a priest who does this and that. But then, Homer switches to a perspective in which it is as if Chryses is voicing the story rather than Homer. The former is narrative fiction and the latter is representational fiction, in which we as the reader are expected to inhabit the mind of the character. Plato believed representational fiction should be banned from the curriculum of the learning Guardians. They should only inhabit thoughts of justice and excellence. Stunningly, the very act of reading or, as an actor, inhabiting and playing the part of a less-than-excellent character can corrupt:

> They should neither do a mean action, nor be clever at acting a mean or otherwise disgraceful part on the stage for fear of catching the infection in real life (Plato 395c).

The education of the Guardians relates to scientific simulation in an important way. When we code an evolutionary scenario in silico, we work from models or, more specifically, deeply thought-out *representations* of the components of our model. Although my authorship of this volume clearly signals the value I place in the efficacy of scientific simulation, we must not allow ourselves to fall beholden to the clean, hermetic atmosphere of digitally represented worlds. It is important that the representational fictions we instantiate in the computer are treated as another tool in our arsenal that allows scientists to attack the truth along lines tangential to it.

1.4.2 Fortuna and Chance Ontology

> *The return of Fortuna corresponded to the world feeling of chance ontology . . .*[2]
> –Peter Sloterdijk, *In the World Interior of Capital*

The Roman goddess of chance, Fortuna, spun her *Rota Fortunae*, or Wheel of Fortune, to randomly determine the fates of man and woman (Fig. 1.3). Her presence among the Roman pantheon acknowledged the recognizably stochastic nature of human life. Today, you and I also recognize the large role chance plays in our everyday lives. In addition, we understand that probabilistic outcomes are implicated in the fates of *all* lifeforms as well as the physical structure of Earth and the abiotic elements of its overlain, ever-fluid, and skin-like biosphere.

To borrow Sloterdijk's term – though in a rather different context – *chance ontology* is an honest accounting of *being* that acknowledges the considerable influence of chance and risk taking. Evolutionary genetic theory embraces and codifies the deeply probabilistic engine of life. Mutation, meiotic recombination, migration, natural selection, and genetic drift all incorporate elements of chance. Deterministic models certainly have their role in evolutionary biology. For example, it is helpful to know that an adaptive allele is *expected* to take considerably more time to move from a frequency of 0.99 to fixation when heterozygotes have the same fitness as homozygotes for the

[2] Quoted with permission from the translation by Wieland Hoban, 2013, Polity Books, Cambridge.

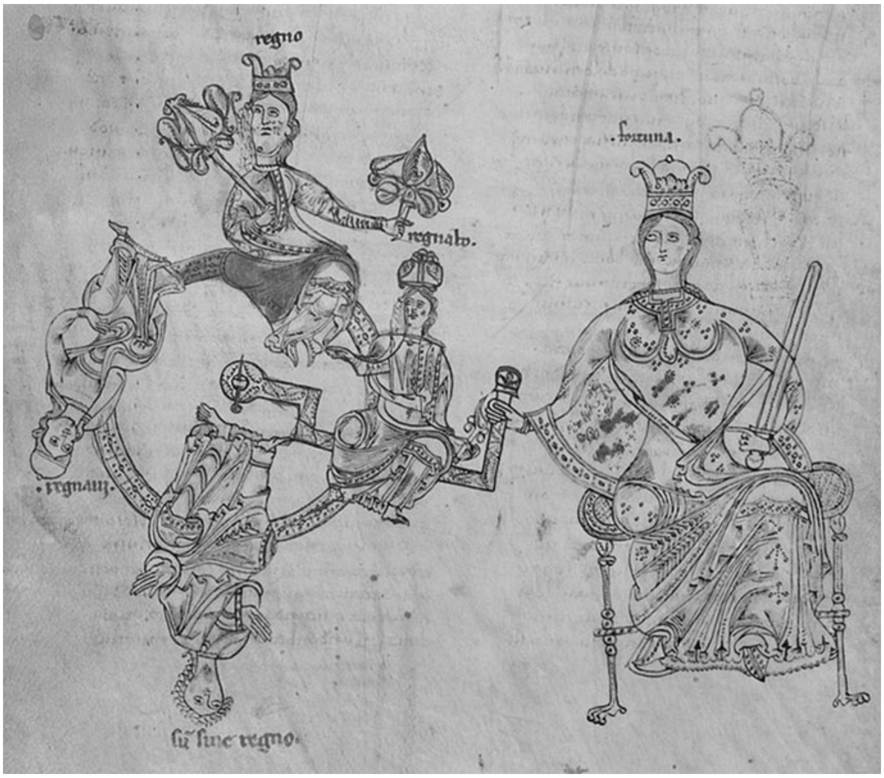


Fig. 1.3 A Medieval illustration of Fortuna added in the twelfth century to a tenth-century copy of *Moralia in Job* by St. Gregory the Great (540–604AD). Fortuna turns the Wheel of Fortune between different fates, which, clockwise from the top, are ruling, about to rule, without a kingdom, and having ruled. The two upside-down figures with less *fortunate* fates look to Fortuna, as if asking for her intervention (Image in the public domain)

adaptive allele than when they do not. Yet including chance in our enlivenment of models by simulation is one detail we should include whenever possible. In a world of chance ontology, expectations are guide posts but hardly certainties; stochastic simulations offer us a means to explore the ways in which expectations are circumnavigated. The ubiquitous presence of pseudorandom number generators in the code detailed herein suggests they are the modern acolytes of Fortuna.

1.4.3 *Epistemological Concerns*

Epistemology is the study of knowledge. How do we distinguish fact from opinion? How is knowledge generated? The act of simulation and simulation output require us to think about what simulated data represent. How do simulated data differ from empirical data? Should a computer simulation be counted as an experiment?

Although we will not explore the epistemology of simulation in detail, it is worth taking a moment to contemplate the seeming weirdness of simulation experiments in population genetics. Often, coalescent simulations (Chap. 2) are run thousands or millions of times in the service of inference methods such as approximate Bayesian computation. Are these repetitions and the data they generate in some way analogous to Mendel's repetitions of genetic crosses and the resultant phenotypic data?

Perhaps I'm asking too many questions, but these are real concerns regarding simulation. We want our work to be meaningful and applicable to our academic concerns. So we had best be confident that our methodology (simulation) provides us with knowledge. I would argue that simulations are real experiments that provide us with data that can be used both for inference and for building intuition concerning natural processes. The results of in vitro experiments are commonly accepted with the caveat that things might turn out different in a living organism. Similarly, we should use the results of in silico experiments to our advantage, but keep in mind that things might be different in the natural world.

For discussion of serious philosophical thought on the topic of the epistemological nature of simulation, I would recommend (Winsberg 2010) as an excellent starting point. Winsberg (2010) makes a convincing case that population genetic simulation is both a legitimate form of scientific experimentation and that simulated data are a source of furthering our understanding of natural processes:

> ... simulation is a process of knowledge creation, and one in which epistemological issues loom large ... simulation is in fact a deeply creative source of scientific knowledge ...

1.5 Whom This Book Will Benefit

The decreasing costs of computing power and genetic data acquisition over the last two decades present biologists with exciting opportunities to draw nuanced inferences from population genetic/genomic data (Hudson 2002). Depending on the research question addressed, one of two distinct approaches to population genetic simulation may be appropriate (Chap. 2). The most efficient of these two approaches by far is retrospective, coalescent simulation. This approach is suitable in a wide variety of contexts and should be chosen whenever appropriate. However, because the computing power of even a single, modern desktop computer is quite remarkable, it is now possible to conduct prospective, forward-in-time simulations that model individuals as separate objects with complex life histories and genetic processes.

This volume is therefore geared toward advanced undergraduate students, graduate students, and professional academics who need a gentle push toward incorporating population genetic simulation into their research. Although I spend time covering retrospective coalescent simulation in the next chapter, *my hope is that this volume will help researchers realize the power of prospective simulation*. I find prospective simulation to be empowering because it allows me to explore evolutionary theory in a more transparent way than retrospective simulation; every detail of the simulated evolutionary process must be spelled out in the code. Therefore, when prospective simulation points to an unexpected result, it is easier to validate that the surprise result is not an artifact of the simulation process itself. In addition, prospective simulation allows us to simulate more complex evolutionary scenarios only limited by our informed imagination and the extra compute time this added complexity entails.

1.6 Required Background Knowledge and Online Resources

To avoid making the volume overly long, I assume a *basic* working knowledge of the data structures and syntax of C++ and R. However, we will not find it necessary to venture into the arcane aspects of these two languages in this volume. The reader with general knowledge of control structures in computer programming should be able to learn by example through study of the numerous instances of example code found throughout. I devote considerable verbiage to elaborating details of the code I introduce throughout the text. The reader interested solely in the use of simulation code found in the book and a tour of some of the scenarios that can be simulated—their results interpreted—can safely skip these technical accounts of code logic and implementation. However, I hope a minority of readers might find these prosaic interruptions useful.

Because I wrote this volume with an eye toward readers well versed in the concepts of evolutionary biology and genetics, I also assume the reader is familiar with terms such as allele, polymorphism, and fitness. Nevertheless, each of the subsequent chapters includes exposition on basic theoretical knowledge that will aid the reader's understanding of the simulation details. As a practical aid, all of the code listings presented here can be found at github.com/deltafortuna organized by chapter. Updates to the code, errata, and future expansions of FORTUNA will also be found here. The Appendix provides a comprehensive list of the parameters used to run distinct simulation scenarios.

References

Baudrillard J (1994) Simulacra and Simulation. The University of Michigan Press, New York

Beaumont MA (2010) Approximate Bayesian computation in evolution and ecology. Annu Rev Ecol Evol Syst 41:379–406

Buffon G (1733) Editor's note concerning a lecture given 1733 by mr. le clerc de buffon to the royal academy of sciences in paris. Histoire de l'Académie Royale des Sciences pp 43–45

Conway R (1963) Some tactical problems in digital simulation. Manag Sci 10:47–61

Conway R, Johnson B, Maxwell W (1959) Some problems of digital systems simulation. Manag Sci 6:92–110

Eldredge N, Gould SJ (1972) Punctuated equilibria: an alternative to phyletic gradualism. In: Schopf TJM (ed) Models in Paleobiology. Freeman, Cooper and Co., San Francisco, pp 82–115

Ewens WJ, Ewens PM (1966) The maintenance of alleles by mutation—monte carlo results for normal and self-sterility populations. Heredity 21:371–378

Fermi E, Pasta J, Ulam S (1955) Studies of non linear problems. Los Alamos National Laboratory Document LA-1940:977–989

Harbaugh JW, Bonham-Carter G (1970) Computer simulation in geology. John Wiley, New York

Harris H (1966) Enzyme polymorphisms in man. In: Proceedings of the Royal Society B Biological Sciences, vol 164, pp 298–310

Hubby JL, Lewontin RC (1966) A molecular approach to the study of genic heterozygosity in natural populations i. The number of alleles at different loci in *Drosophila pseudoobscura*. Genetics 54:577–594

Hudson RR (2002) Generating samples under a Wright-Fisher neutral model of genetic variation. Bioinformatics 18:337–338

Huelsenbeck JP, Ronquist F (2001) Mrbayes:bayesian inference of phylogenetic trees. Bioinformatics 17:754–755

IBM Data Processing Division (1960) IBM 7090 Processing system technical fact sheet. IBM, New York

Kimura M (1968) Evolutionary rate at the molecular level. Nature 217:624–626

Kimura M, Crow JF (1964) The number of alleles that can be maintained in a finite population. Genetics 49:725–738

King JL, Jukes TH (1969) Non-darwinian evolution. Science 164:788–797

Lewontin RC, Hubby JL (1966) A molecular approach to the study of genic heterozygosity in natural populations. ii. amount of variation and degree of heterozygosity in *Drosophila pseudoobscura*. Genetics 54:595–609

Mayr E (1981) The growth of biological thought. Harvard University Press, Harvard

Pritchard J, Stephens M, Donnelly P (2001) Inference of population structure using multilocus genotype data. Genetics 155:945–959

Raup DM, Gould SJ, Schopf TJM, Simberloff DS (1973) Stochastic models of phylogeny and the evolution of diversity. J. Geol. 81:525–542
Turner D (2011) Paleontology: a philosophical introduction. Cambridge University, Cambridge
Winsberg E (2010) Science in the age of computer simulation. The University of Chicago Press, Chicago

2 Retrospective and Prospective Simulation

Δt approaching zero, eternally approaching, the slices of time growing thinner and thinner, a succession of rooms each with walls more silver, transparent, as the pure light of the zero comes nearer.[1]

– Thomas Pynchon, *Gravity's Rainbow*

With notable exceptions, including experimental evolution, much of the research effort in evolutionary biology focuses on the past. Which gene or genetic variant was targeted by natural selection in the past? What is the demographic history of a population? When did the MRCA of a clade live on Earth?

The fossil record provides us with direct, if imperfect, insights into the past morphology of extinct species. Genomic data of extant organisms provide us with a less-direct record of the past, and the record is messy. Although modern genomes have been shaped by historical events and are, in that sense, historical records, new mutations overwrite old ones, chromosomal mutations such as inversions and translocations change the positions of genes, etc. Therefore, genomic data offer a "document" for inferring the past, but the constant erosion of the "text" in this document makes our task of inferring the past difficult.

For a moment, consider that you come across a grand array of dominoes fallen upon each other on a sidewalk. What can you infer about the past from the observed trackways of fallen dominoes? Was there a point in the past when some individual provided the kinetic energy to set the falling cascade in motion? Or was it the wind? Regardless, *when* did the dominoes fall? As an outsider hypothesis, is it possible that some strange individual assembled the dominoes in their observed fallen state? The problem here is that current patterns might be explained by a multitude of historical causes.

The use of modern genetic data to infer information about the past is therefore beset by at least two broad problems:

[1] Quoted with permission. 1973. Penguin Classics.

1. Genetic patterns indicative of past events, such as the signatures of a selective sweep due to positive natural selection (see Chap. 8) decay due to new mutation and recombination.
2. Modern patterns of genetic variation are often equally well explained by distinct evolutionary causes.

However, simulation experiments (Chap. 1) provide a means for addressing these problems. For example, decreased frequencies of derived alleles in a localized region of the genome might be explained by positive or purifying natural selection. Simulating genetic data under both scenarios and comparing the simulated data to the one empirical data set has the potential to help us identify subtle differences that support purifying rather than positive natural selection or vice versa.

In this chapter, we mainly focus on a retrospective (backward-in-time) approach to evolutionary genetic simulation. We also compare retrospective simulation to the focus of the rest of this volume—prospective, or forward-in-time simulation. Because evolutionary biology is a largely historical science, it is perhaps more natural for us to think retrospectively. One key point to keep in mind, however, is that prospective simulations can take a past timepoint—t_{past}—as the starting generation; we can then run the prospective simulation for $t_{now} - t_{past}$ generations to produce a simulated data set corresponding to the current time. In other words, both retrospective and prospective simulations have the ability to generate simulated genetic data sets that we can compare to an empirical data set sampled from an extant population.

2.1 Background: Retrospective Versus Prospective Simulation

Retrospective simulations, based on coalescent theory (Wakeley 2008), provide us with simulated sequence samples that, in the context of inference, are compared to present-day empirical samples. By varying parameter values controlling evolutionary factors such as mutation, demographic change, population structure, recombination, and more, comparisons between simulated and empirical sequence data sets empower the evolutionary geneticist to estimate probability densities on parameters of interest. Coalescent simulations are retrospective in the sense that the first step is to simulate the *past* genealogy leading to a modern-day sample. The resulting shape of the genealogy may be influenced by (1) recombination, (2) demographic change during specified periods of time, (3) migration among semi-isolated populations, and (4) simple forms of natural selection. The simulated genealogy serves as the “skeleton” upon which mutations (following some model) are placed. Derived variants then “flow” to all sequences that are descendants

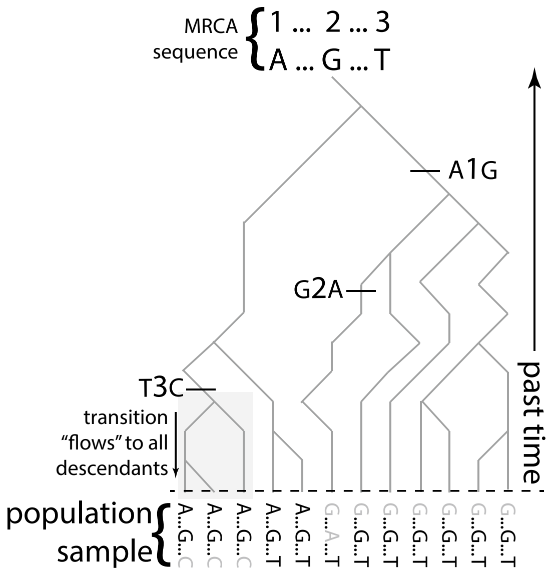


Fig. 2.1 In retrospective, coalescent simulation, a genealogy is first simulated; this tree connects all sequences in the current generation to their most recent common ancestor (MRCA). Once a genealogy is produced, a mutational model is used to add mutations to random positions on the branches of the genealogy. In the illustrated tree, we focus on three sites within the sequence of the MRCA: A, G, and T and sites 1–3; the ellipses indicate intervening, unspecified sequence. Three point mutations that occur are randomly determined. As illustrated by the T3C mutation, the derived cytosine residue is inherited by all descendants of the lineage on which the mutation occurred. In this case, the result of the three point mutations is a set of three single nucleotide polymorphisms: A/G, position 1; G/A, position 2; T/C, position 3

of the branches where mutations occurred (Fig. 2.1). Importantly, the simulation of only those lineages leading to the modern-day sample (i.e., the genealogy) obviates the need to account for those lineages that went extinct in the past and are therefore not ancestral to the modern-day sample (Fig. 2.2a–b).

Prospective simulations begin with an initial *population* of individuals whose patterns of genetic variation are determined by the researcher. This category of simulation is often individual based, meaning that each generation we account for all individuals of the population, each of which acts as its own agent: migrating, incurring mutation, mating, etc. The primary advantage to prospective simulation is that the complexity of the simulated evolutionary model is only limited by the imagination and coding ability of the researcher. As one example, we can code complex forms of natural selection such as overdominant or frequency-dependent selection that are difficult or impossible to implement in retrospective simulations. In addition, because prospective simulations begin with the initial population, move for-

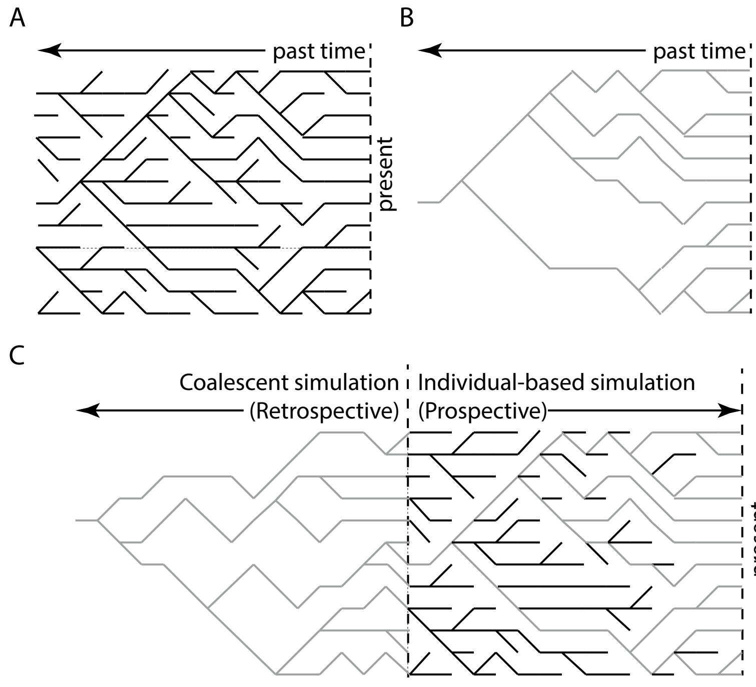


Fig. 2.2 (**a**) Transmission of gene copies toward the present in a population of constant size $N = 10$. (**b**) The underlying genealogy/coalescent of the current population in (**a**), which makes it evident that all ten individuals in the present population are descendants of a single, most recent common ancestor 13 generations in the past. (**c**) A simulation tactic used frequently in this volume. The starting genetic variants for prospective simulation will often be drawn from a coalescent simulation (left half of panel)

ward in time generation by generation, and account for all individuals, these simulations are often easier to conceptualize. Retrospective simulations are more opaque in nature, meaning there is a greater chance that the researcher is not simulating exactly what he or she thinks. The primary disadvantage to prospective simulations is that they require considerably greater compute time. This disadvantage stems from the need to keep track of *every* individual *every* generation, as opposed to retrospective simulations that only require us to consider the lineages leading to the modern-day sample.

The central tension at the heart of our choice between retrospective and prospective simulation is therefore computational efficiency versus model complexity. The choice is simple if the model of evolution is relatively simple. For example, if one wants to investigate how changes to the point mutation rate affect the accumulation of genetic differences between two populations experiencing limited gene flow, this is easily accomplished using a retrospective approach. In this case, the computational speed of coalescent simulation makes it the clear choice. On the other hand, if one wishes to simulate a metapopulation with hundreds of source and sink sites, prospective simulation may be the only way to accomplish this goal.

In Chaps. 3–9, we will systematically construct a more-and-more complex prospective simulation program. However, we will often use retrospective simulation to derive an initial population for prospective simulation (Fig. 2.2c). Because so much of this volume is devoted to prospective simulation, in this chapter we focus on coalescent theory and the mechanics of performing retrospective simulations in R. The intent is that this chapter will serve as a standalone primer on retrospective simulation as well as a reference point for later chapters in which retrospective simulation is used.

2.2 Background: Coalescent Theory

Coalescent simulation is considerably more efficient because we begin with a sample of k genes and ask, How are these genes related to one another? To answer this question, we simulate a genealogy connecting the n genes to a common ancestor based on a few simple probabilistic expectations. The efficiency of this process derives from our ability to ignore any past lineages of the population/species that fail to transmit genetic information to the present sample time. In other words, extant genetic variation derives only from mutations to the lineages that are part of the genealogy. The reader interested in detailed treatments of coalescent theory has several volumes from which to choose (Hahn 2018; Hein et al. 2005; Wakeley 2008) in addition to the primary literature.

Consider a sample of $k = 2$ genes from a diploid population of $2N_e$ gene copies. If N_e is large, technically infinite, the probability that the two genes find their common ancestor (i.e., coalesce) in the preceding generation is $\frac{1}{2N_e}$. This is an intuitive probability; there are $2N_e$ gene copies in the preceding generation, and given that one of the two genes had to come from one of these potential parents, the chance that the other gene copy in the sample is derived from the same parent is simply one out of the $2N_e$ possible parents. Clearly, the probability that coalescence does not occur in the previous generation is $1 - \frac{1}{2N_e}$. The time in generations to coalescence of the two gene copies is geometrically distributed as

$$P_{T_2}(t_2) = \frac{1}{2N_e}\left(1 - \frac{1}{2N_e}\right)^{t-1}, \tag{2.1}$$

where T_2 is a random variable for the time to coalescence of two randomly sampled sequences. In other words, looking backward, the path to the first "success" (coalescence) requires $t - 1$ generations in which coalescence does *not occur* followed by the generation in which coalescence finally occurs.

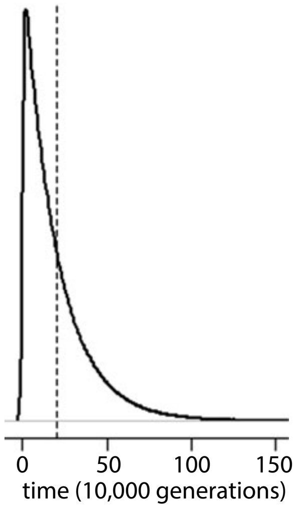


Fig. 2.3 Exponential approximation of coalescent times for two genes. $N_e = 10{,}000$ and mean coalescent time is $2N_e = 20{,}000$ generations (dotted line)

The continuous approximation is an exponentially distributed time to coalescence:

$$P_{T_2}(t_2) = \frac{1}{2N_e}e^{-\frac{t-1}{2N_e}} \tag{2.2}$$

The mean and standard deviation of an exponential distribution are both equal to the inverse of the exponential parameter λ, which equals $\frac{1}{2N_e}$ in this case. Thus, the expected time to coalescence of two genes is simply $2N_e$. However, this expectation is not particularly predictive given that the standard deviation on this random variable is also $2N_e$ (Fig. 2.3).

Again looking to the past, the next coalescence among any pair of the k genes is exponentially distributed with parameter $\lambda = \binom{k}{2}$ as

$$P_{T_k}(t_k) = \binom{k}{2}e^{-\binom{k}{2}t_k}, \tag{2.3}$$

where time is scaled in units of $2N_e$ *generations for a diploid population*. Thus, when $N_e = 10{,}000$, the expected time to coalescence of $k = 2$ genes is $1/\binom{2}{2} = 1$, or $2N_e \times 1 = 20{,}000$ generations. If $k = 10$ sampled genes are present in the genealogy, the expected time to the first coalescence between any two lineages is $1/\binom{10}{2} = 0.022$, or $2N_e \times 0.022 = 444$ generations. These two cases demonstrate the intuitive result that expected coalescent times decrease as the number of lineages left to coalesce increases; simply put, there are more potential pairs of genes that may share a common ancestor in the previous generation, each with a probability of $1/2N_e$.

Two important metrics of the genealogy are (1) the *tree height* or time to the most recent common ancestor (T_{MRCA}) and (2) the *total time* in the tree, which is the sum of all branch lengths (T_t). The expected values, again scaled in units of $2N_e$ generations, are

$$E[T_{MRCA}] = 2\left(1 - \frac{1}{k}\right) \tag{2.4}$$

$$E[T_t] = 2\sum_{1}^{k-1} \frac{1}{i}. \tag{2.5}$$

It is worth noting that coalescent simulation in R, whether we use the most popular coalescent simulator MS (Hudson 2002) or SCRM in R, will require us to scale time in units of $\theta = 4N_e$ generations for diploid populations (see Sect. 2.3). This is important as the theoretical results just discussed are all scaled in units of $2N_e$ generations.

To simulate a genealogy for sample size k, we need only draw coalescent times from Eq. 2.3 and choose a random pair of genes/lineages to coalesce at each coalescent time. The realized total branch length of the tree is then

$$T = \sum_{i=2}^{k} it_i \tag{2.6}$$

To assign actual simulated sequences to the sample of k genes, we add random mutation events to the branches. As intuition suggests, longer branches will have greater probability of bearing mutation(s). Similarly, genealogies with larger values of T will bear more total mutations and, therefore, segregating sites among the sampled genes.

Assuming the sampled sequences evolve neutrally, the expected number of differences between two randomly drawn sequences (where the expected time to coalescence is $2N_e$ generations) is $\theta = 4N_e\mu$. Because mutation in either lineage leads to a difference between the two sampled sequences, on average, a total of $\theta/2$ mutations occur along each lineage (Fig. 2.4). In general, the number of mutations, m, that occurs along a branch is conditional on branch length t and Poisson distributed with parameter $\frac{\theta t}{2}$:

$$P_{M|t}(m|t) = \frac{\left(\frac{\theta t}{2}\right)^m}{m!} e^{-\frac{\theta t}{2}} \tag{2.7}$$

Thus, after simulating a genealogy with explicit branch lengths, we can use Eq. 2.7 to draw random numbers of mutations per branch and determine the sampled sequences by allowing these mutations to flow down the genealogy to the sampled sequences as illustrated in Fig. 2.1.

We can also use coalescent simulation to produce sequence samples under simple non-neutral and non-equilibrium conditions. For example, coalescent

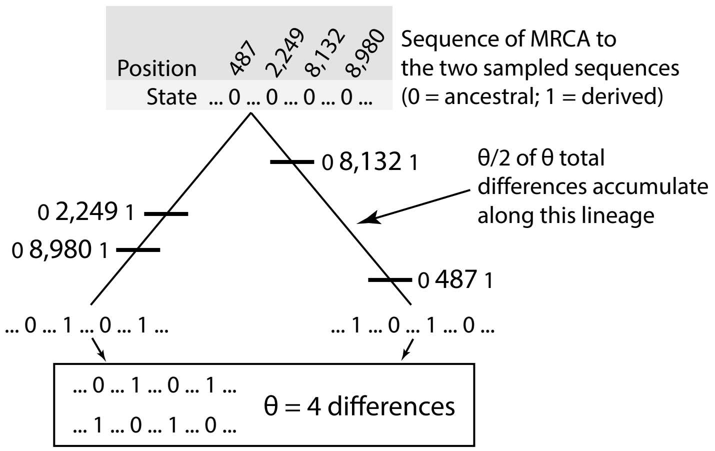

Position 487 2,249 8,132 8,980
State ... 0 ... 0 ... 0 ... 0 ...
Sequence of MRCA to the two sampled sequences (0 = ancestral; 1 = derived)
0 8,132 1
θ/2 of θ total differences accumulate along this lineage
0 2,249 1
0 8,980 1
0 487 1
... 0 ... 1 ... 0 ... 1 ...
... 1 ... 0 ... 1 ... 0 ...
... 0 ... 1 ... 0 ... 1 ...
... 1 ... 0 ... 1 ... 0 ...
θ = 4 differences

Fig. 2.4 Consider a sequence 10,000 bp long with a point mutation rate of 10^{-8}. Then, the sequence-wide mutation rate is $\mu_{sequence} = 10^4 \times 10^{-8} = 10^{-4}$ and in a population where N_e = 10,000, $\theta = 4N_e\mu = 4 \times 10^4 \times 10^{-4} = 4$. Under the neutral coalescent, the expected number of differences between two randomly chosen sequences with an expected coalescent time of $2N_e$ generations is equal to θ. Mutation in either of the lineages leads to a difference between the two sequences; thus, on average, $\theta/2$ mutations will occur on each lineage to yield the expected number of differences, θ. Here, the ancestral sequence is represented by four zeros at the four sites that will be mutated in one of the lineages. The time and identity of each mutation from the ancestral 0 allele to the derived 1 allele are indicated

simulators allow us to model changes in population size over time, population structure with or without gene flow, and simple selective regimes. To demonstrate, consider a population that undergoes an exponential increase in population size from 100 to 10,000 individuals beginning 100 generations in the past. Because there are so many more individuals in the current population than in the past, this means coalescent events are less likely to happen in more recent generations. Rapid population increase changes the topology of the genealogy dramatically, tending to produce so-called star genealogies in which coalescent events cluster further back in time when the population was small. In other words, the terminal branches of the genealogy are much longer than under the equilibrium coalescent. This leads to an abundance of singleton variants because most mutation events occur along these long, terminal branches and therefore only affect the sequence of one of the sampled genes.

Modeling complex forms of selection, including multilocus and frequency-dependent selection, under the coalescent framework is difficult or impossible and one motivation for performing prospective simulations. Recent coalescent simulators, such as `discoal` (Kern and Schrider 2016), are quite useful in that they allow us to rapidly simulate gene sequences overlapping or linked to the target of a hard or soft selective sweep. However, this program only simulates selection for a single adaptive allele. Similarly, Berg and Coop (2015) approximate a soft selective sweep using coalescent theory. Again, this is useful when one is interested in simulating this important scenario. However, more complex selective dynamics require us to code our own prospective simulations.

2.3 Coalescent Simulations in MS and R

In this section, we use the R packages `scrm` (Staab et al. 2015) and `coala` (Staab and Metzler 2016) to simulate several evolutionary scenarios of differing demographic history and population structure (Fig. 2.5). We will also use R to analyze the output data.

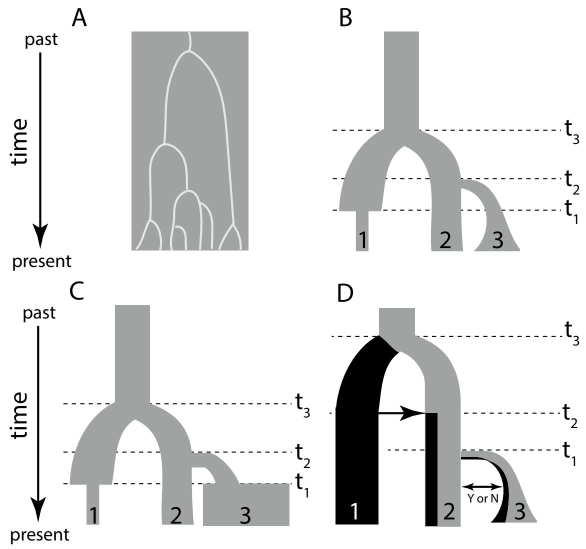


Fig. 2.5 The four scenarios modeled in this section using coalescent simulation in R. Time flows forward from top to bottom. (**a**) A single population of constant size; the gray box represents the population and the white lines the genealogy leading to the sample of eight genes. In all other panels, genealogies are not illustrated; the widths of populations on the horizontal axis represent relative size of the population. (**b**) Population structure without gene flow, including an instantaneous population bottleneck in population 1 at time t_1 and exponential population growth in population 3. The two population splits occur at times t_2 and t_3. (**c**) The same as scenario B, with the difference that population 3 undergoes an instantaneous population expansion at time t_1. (**d**) Population structure with admixture and gene flow. The ancestral population splits into two wholly isolated populations 1 and 2 at t_3. At time t_2, a fraction of individuals in population 2 are drawn from population 1 individuals (admixture), and population 2 splits into populations 2 and 3 at time t_3. Population 3 expands exponentially after its origin, and gene flow may occur between populations 2 and 3. We also incorporate recombination in simulations of this scenario

2.3.1 Package SCRM

The coalescent simulator SCRM (Staab et al. 2015) is an alternative to the commonly used coalescent simulator MS (Hudson 2002) and is tailored to provide rapid (though approximate) coalescent simulation of genome-size data sets. SCRM is easily implemented in R using the `scrm` package and can be used without approximation to simulate shorter sequences as well. Most of the arguments used in SCRM are the same as those used in MS, with some useful additions, including admixture with a population that is simulated "on the side."

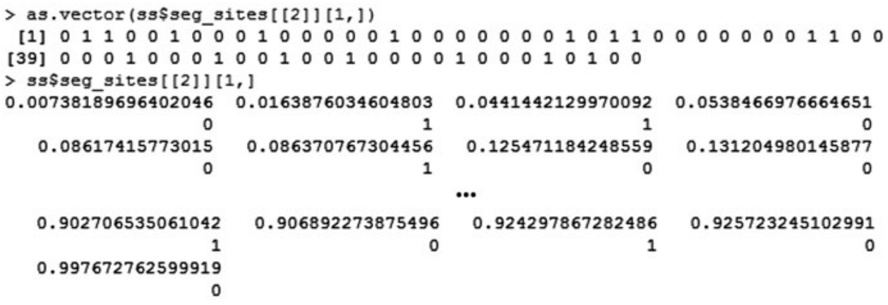
2.3.1.1 Scenario A

We begin with the simplest form of coalescent simulation: a single population of constant size (Fig. 2.5a). MS, SCRM, and COALA (see Sect. 2.3.2) scale all events—coalescence, mutation, recombination, migration, and population splits—in terms of the population mutation parameter, $\theta = 4N_e\mu$ (assuming a diploid population). Therefore, the first step in coalescent simulation is to choose a focal population and calculate its value of θ. We assume that mutations are point mutations; the output will therefore be a sequence of segregating sites for each gene sampled, expressed as a vector of zeros and ones, where ones represent derived alleles. Because sequences are being returned, we must calculate the mutation rate μ for the sequence as a whole, which means we must explicitly decide upon the length of the sequence to be simulated. Let us assume a 10,000-bp sequence with a point mutation rate of 1×10^{-8}. Then $\mu_{sequence} = 10^4 \times 10^{-8} = 10^{-4}$. We will further assume that effective population size $N_e = 50{,}000$. With these parameters in hand, $\theta_{seq} = 4 \times N_e \times \mu_{sequence} = 4 \times (5 \times 10^4) \times 10^{-4} = 20$.

The following R listing generates two data sets of 20 sequences sampled from a population in which $\theta = 20$, stored in object `ss`, and provides some basic investigation of the simulated data:

```
1  library(scrm) # load package
2  ss <- scrm('20 2 -t 20 -T')
3  names(ss)
4  as.vector(ss$seg_sites[[2]][1,])
5  ss$seg_sites[[2]][1,]
6  round(as.numeric(colnames(ss$seg_sites[[2]])) * 10000)
7  dim(ss$seg_sites[[1]])
8  dim(ss$seg_sites[[2]])
```

In line 2, the leading "`20 2`" specifies simulation of two data sets of 20 sequences, the value of θ is specified following the "`-t`" flag, and the "`-T`" flag indicates we want to store the genealogy in addition to the sequences. The object `ss` now contains two lists, which are revealed by the `names` function in



```
> as.vector(ss$seg_sites[[2]][1,])
 [1] 0 1 1 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 1 1 0 0
[39] 0 0 0 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1 0 0 0 1 0 1 0 0
> ss$seg_sites[[2]][1,]
0.00738189696402046 0.0163876034604803 0.0441442129970092 0.0538466976664651
                  0                  1                  1                  0
   0.08617415773015  0.086370767304456  0.125471184248559  0.131204980145877
                  0                  1                  0                  0
                                     ...
   0.902706535061042  0.906892273875496  0.924297867282486  0.925723245102991
                   1                  0                  1                  0
   0.997672762599919
                   0
```

Fig. 2.6 Output from lines 4 and 5 of the previous listing. Note that many positions have been omitted from the positional data for the sake of space

line 3: `ss<currencydollar>trees` and `ss<currencydollar>seg_sites`. Line 4 prints the sequence of the first of 20 sequences from data set 2, while line 5 prints the same sequence with positions of the segregating variants indicated (Fig. 2.6). Note that the positions are given as real numbers between 0 and 1. To convert this to an actual position in the simulated sequence, we multiply the column names (positions converted to numeric) by the sequence length of 10,000 (line 6). The dimensions of the data sets (lines 7–8) indicate 20 rows (for the 20 sequences) and 91 and 65 columns, the number of segregating sites, for data sets 1 and 2, respectively.

Next, we use the R package `ape` (Paradis and Schliep 2019) to visualize the genealogies of the two simulated data sets. The objects in the list `ss<currencydollar>trees` are Newick formatted trees, which represent the relationships between the sequences as well as the branch lengths. For example, within the Newick formatted tree, the code `(13:0.006,11:0.006):0.068` indicates that (1) sequences 13 and 11 are sister sequences, (2) the branches leading from their common ancestor are 0.006 long, and (3) the branch leading from the coalescent event that links these sequences to another sequence or sequences is 0.068 long. The following listing provides R code for plotting the two genealogies (Fig. 2.7).

```r
library(ape)
len = length(ss$trees)
trees <- list(length = len)
for (i in 1:len) {
  trees[[i]] <- read.tree(text = paste0(ss$trees[[i]]))
}
par(mfcol=c(1,2)) # create plot device with two panels
plot(trees[[1]])
plot(trees[[2]])
```

Next, we check that the expected values of T_{MRCA} and T_t are realized in a sample of 10,000 independently simulated genealogies. Assuming a sample size of $n = 20$, the expected values of these two quantities are

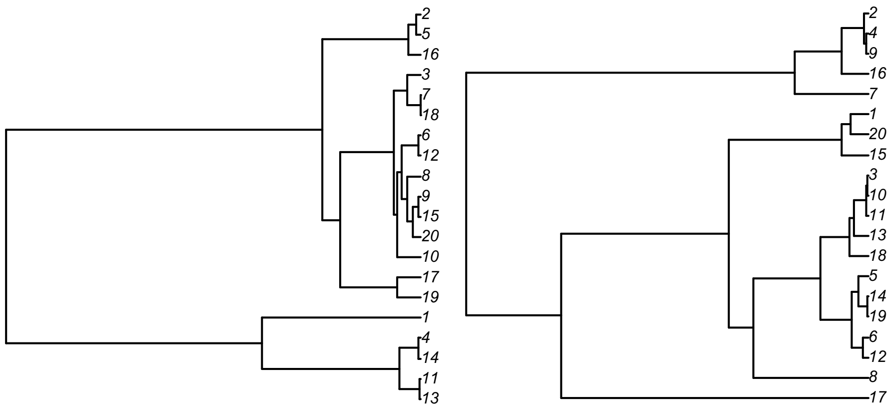


Fig. 2.7 The two simulated genealogies visualized using APE. Of course, genealogies you simulate will have different topologies and tip labels. Note that most coalescent events are clustered near the present

$$E[T_{MRCA}] = 2\left(1 - \frac{1}{20}\right) = 1.90 \tag{2.8}$$

$$E[T_t] = 2\sum_{1}^{20-1} \frac{1}{i} \sim 7.10. \tag{2.9}$$

However, these expectations are expressed in units of $2N_e$ generations, and the branch lengths reported by SCRM are reported in units of $4N_e$ generations. *We must therefore rescale the expected values to* $E[T_{MRCA}] = 0.95$ *and* $E[T_t] = 3.55$ *by dividing by 2.*

Once a genealogy is loaded as a tree object in APE, we can calculate a multitude of metrics of interest to us. Currently, our goal is to calculate the T_{MRCA} and T_t for each tree. Assume you have a tree object in APE named `tr`. The function `coalescent.intervals()` returns several values, one of which is `total.depth`—i.e., T_{MRCA}. Thus, `coalescent.intervals(tr)` `<currencydollar>total.depth` returns the T_{MRCA} of `tr`. The tree object itself has a value `<currencydollar>edge.length`, which is a vector of all branch (edge) lengths. Thus, `sum(tr<currencydollar>edge.length)` returns T_t.

We are now ready to run a proving experiment that calculates the *average* values of T_{MRCA} and T_t across 10,000 independently simulated genealogies to see how well they compare to the previously calculated expectations.

```r
ss <- scrm('20 10000 -t 20 -T')
Tt <- 0
Tmrca <- 0
for (i in 1:10000) {
   tr <- read.tree(text = paste0(ss$trees[[i]]))
   Tt <- Tt + sum(tr$edge.length)
   Tmrca <- Tmrca + coalescent.intervals(tr)$total.depth
}
Tt / 10000 # returns the average value of T_t
Tmrca / 10000 # returns the average value of T_MRCA
```

In my case, the average values of T_{MRCA} and T_t were 0.953 and 3.553, respectively, which are clearly good approximations to the expected values of 0.95 and 3.55.

2.3.1.2 Scenario B

We now turn our attention to the scenario shown in Fig. 2.5b. Table 2.1 lists the parameter values used for each of the three populations in this scenario. We again assume a point mutation rate of $\mu = 1 \times 10^{-8}$ and a sequence 10,000 bp long; therefore, $\mu_{sequence} = 10^4 \times 110^{-8} = 10^{-4}$. Because it does not undergo demographic change, we will use population 2, with $N_e = 10{,}000$, as our focal population to calculate $\theta = 4 \times 10^4 \times 10^{-4} = 4$. The time points listed in Table 2.1 are expressed in units of absolute generations; we need to convert these to units of $4N_e = 40{,}000$ generations to use the SCRM simulator. 10,000, 20,000, and 40,000 generations are therefore expressed as 0.25, 0.5, and 1.0 $4N_e$ generations, respectively. Finally, we need to calculate the exponential rate that causes the increase of population 3 from 100 individuals at t_2 to 100,000 individuals at t_0. Exponential growth follows the equation $N_t = N_0 e^{rt}$. In the current case, $N_0 = 100$ at the start of the expansion, $N_t = 100{,}000$ as the current population size, and $t = t_2 = 0.5$. We solve for the exponential rate using a rearrangement of the previous equation:

$$r = ln\left(\frac{N_t}{N_0}\right)t^{-1} = ln\left(\frac{100{,}000}{100}\right)\frac{1}{0.5} = 13.8 \tag{2.10}$$

Now, we are ready to run the simulation using SCRM.

Table 2.1 Parameters used to simulate scenario B shown in Fig. 2.5b. Dashes indicate that a given time point is not relevant to the population. *instant* = instantaneous bottleneck; *exp* = exponential growth

Parameter	Population 1	Population 2	Population 3
$\mu_{\mathbf{sequence}}$	1×10^{-4}	1×10^{-4}	1×10^{-4}
$\mathbf{N_e}$	10,000 → 1000 (*instant*)	**10,000 (*constant*)**	100 → 100,000 (*exp*)
sample size	10 sequences	10 sequences	10 sequences
$\mathbf{t_1}$	10,000 generations	–	–
$\mathbf{t_2}$	–	20,000 generations	20,000 generations
$\mathbf{t_3}$	40,000 generations	40,000 generations	–

```
ss <- scrm('30 2 -t 4 -I 3 10 10 10 -eg 0 3 13.8 -en 0 1 0.1 -en 0.25 1 1
    ↪ -ej 0.5 3 2 -ej 1.0 1 2 -T')
trees <- list(length=2)
for (i in 1:len) {
  trees[[i]] <- read.tree(text = paste0(ss$trees[[i]]))
}
plot(trees[[1]])
```

The command passed to `scrm()` in line 1 of the previous R listing introduces a number of additional flags that allow us to implement the demography shown in Fig. 2.5b.

- `-I`: Used to simulate population structure. The first number following is the number of populations, which is then followed by the number of genes sampled from each of these populations.
- `-eg`: Used to simulate exponential population growth. The first number following indicates the time *before which* exponential growth applies; the second number indicates the population to which growth applies, and the third number is the growth rate as calculated in Eq. 2.10.
- `-en`: Used to simulate instantaneous changes in population size. The first number indicates the time *before which* the population size (relative to N_e) specified by the third number applies. The second number indicates the population to which this change applies.
- `-ej`: Used to simulate the splitting of an ancestral population (looking *forward* in time) or, thinking retrospectively, the joining of two populations looking *backward* in time. The first number is the time when the split/join occurs, the second number indicates the population number that is eliminated by the join (looking retrospectively), and the third number is the population that subsumes the lost population.

Now we can deconstruct the command passed to `scrm()` in line 1:

- `30 2 -t 4`: Specifies simulation of two data sets with $k = 30$, $\theta = 4$.
- `-I 3 10 10 10`: Specifies simulation of three modern populations, from each of which ten sequences are sampled. Note that the total number of sequences drawn from all populations must equal the total sample size (30 in this case).
- `-eg 0 3 13.8` specifies that from the present (time 0) *past-ward*, an exponential growth rate r of 13.8 applies to population 3.
- `-en 0 1 0.1 -en 0.25 1 1`: The first `-en` sets the size of population 1 to $0.1 \times N_e$ starting a time=0 and moving backward. However, the second `-en` sets the size of population 1 to $1 \times N_e$ specified at time $t_1 = 0.25 \times 4N_e$ generations ago and before. Looking *forward* in time, these two `-eN` flags should be interpreted as an instantaneous population bottleneck that occurs at t_1 and reduces the population to one-tenth of its previous size.
- `-ej 0.5 3 2 -ej 1.0 1 2`: The first -ej specifies that, looking backward in time, population 3 becomes part of population 2 at time $t_2 = 0.5 \times 4N_e$

generations ago. Subsequently, the second -ej command indicates that the remaining two lineages (1 and 2) join to form the ancestral lineage at time $t_1 = 1.0 \times 4N_e$ generations ago. Looking *forward* in time, these two `-ej` flags should be interpreted as a population split at t_1 into lineages 1 and 2, followed by a split of lineage 2 into populations 2 and 3 at t_2.

As mentioned previously, this MS syntax can become rather opaque. Particularly given the scaling of time parameters, it is very easy to simulate something other than what you intended.

Figure 2.8 shows the resulting genealogy from one run of scenario B (Fig. 2.5b). All three populations are resolved as monophyletic groups, although repeated simulation of this population model will frequently yield population 2 and/or 3 as (a) paraphyletic group(s). On the other hand, population 1 is always resolved as a monophyletic group, likely due to two factors: (1) it is isolated from the other two populations for 40,000 generations and (2) it undergoes a rather severe bottleneck at $t_1 = 10{,}000$ generations ago. This latter point also explains the extremely short branch lengths for population 1 seen in Fig. 2.8. We expect population bottlenecks to decrease the expected time for coalescent events relative to a population of constant size. Intuitively, the reason for shorter branch lengths is that the probability of coalescence, $1/2N_e$, becomes larger as N_e becomes smaller. As we have seen already, the opposite is expected during population expansions, which cause coalescent times and branch lengths to increase. It may therefore seem surprising that the average coalescence time among population 3 sequences is clearly shorter than among population 2 sequences. However, we need to remember that population 3 initially undergoes a severe bottleneck from 10,000 to 100 at time t_2 before the population expands inexorably (and exponen-

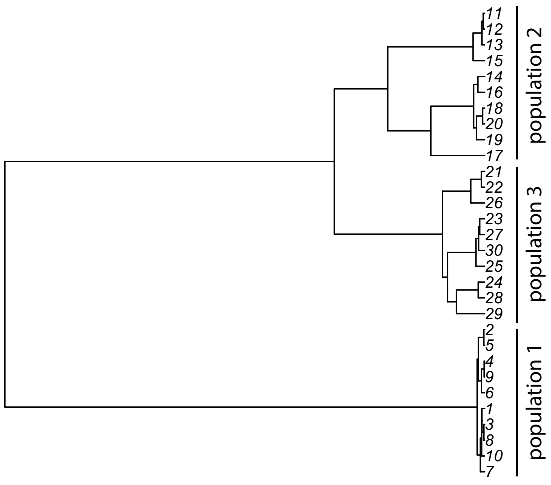


Fig. 2.8 A representative genealogy resulting from simulation of scenario B

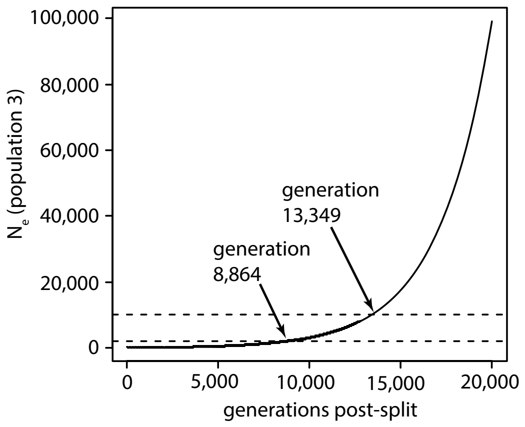


Fig. 2.9 Population size of the exponentially increasing population 3 in scenario B. Starting population size is $N_e = 100$. The two horizontal dashed lines correspond to $N_e = 2000$ and $N_e = 10{,}000$

tially!) toward its current size of 100,000. To see why this matters, consider the population size trajectory of population 3 in scenario B (Fig. 2.9). It takes 8864 generations for the population to attain 2000 individuals, still a fifth of the pre-bottleneck population size, and 13,349 generations to regain the pre-bottleneck population size of $N_e = 10{,}000$. While it is true that the population then expands from $N_e = 10{,}000$ to $N_e = 100{,}000$ in the remaining 6650 generations, a majority of its time is spent at a size less than $N_e = 10{,}000$. Indeed, we can use the harmonic mean of all N_e to calculate effective population size over the 20,000-generation period:

$$\frac{1}{N_e} = \frac{1}{t}\sum_{i=1}^{t}\frac{1}{N_i}, \tag{2.11}$$

where N_i is population size at time $t = i$. This results in an effective population size of just 691 individuals, substantially less than the constant effective size of population 2, which is 10,000.

2.3.1.3 Scenario C

Scenario C differs from scenario B in two ways: (1) at time t_2, population 3 begins at the same size as population 2 ($N_e = 10{,}000$), and (2) the population expansion at time t_1 is instantaneous, increasing to $N_e = 100{,}000$. We again assume that population 2 has a constant population size of $N_e = 10{,}000$ and use this to scale time and rates. Thus, $t_3 = 1$, $t_2 = 0.5$, and $t_1 = 0.25$ in units of $4N_e$ generations. The code for implementing 100 replicates of this scenario and extracting and visualizing trees is as follows:

```r
ss <- scrm('30 100 -t 4 -I 3 10 10 10 -en 0 3 10 -en 0.25 3 1 -en 0 1 0.1
    ↪ -en 0.25 1 1 -ej 0.5 3 2 -ej 1.0 1 2 -T')
trees <- list(length=100)
for (i in 1:len) {
  trees[[i]] <- read.tree(text = paste0(ss$trees[[i]]))
}
plot(trees[[1]]) # to plot other trees replace 1 with an integer between
    ↪ 2 and 100
```

Line 1 is dissected as follows:

- `30 100 -t 4 -I 3 10 10 10`: Specifies 100 data sets of size 30, $\theta = 4$, and three populations from each of which ten sequences are drawn.
- `-en 0 3 10`: Specifies that at time 0 and moving backward in time, population 3 is set to ten times the size of the focal $N_e = 10{,}000$, or 100,000.
- `-en 0.25 3 1`: Specifies that at time $t_1 = 0.25$, population 3 is instantaneously set to the same size as that of the focal population (i.e., 10,000 sequences).
- `-en 0 1 0.1 -en 0.25 1 1`: Specifies that at time zero, looking backward, the size of population 1 is set to 1/10th that of the focal population size (i.e., 1000) and at time $t_1 = 0.25$, its size increases (again, looking backward in time) to the same size as that of the focal population (i.e., 10,000).
- `-ej 0.5 3 2 -ej 1.0 1 2`: Specifies that population 3 merges with population 2 (looking backward in time) at time $t_2 = 0.5$ and population 1 merges with population 2 at time $t_3 = 1.0$.

Figure 2.10 shows a representative genealogy resulting from coalescent simulation of scenario C. Note that sequences drawn from the bottlenecked

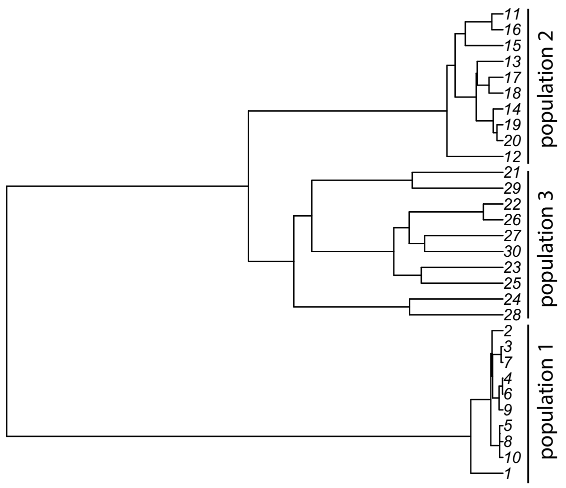


Fig. 2.10 A representative genealogy resulting from simulation of scenario C

population again coalesce very rapidly. Relative to scenario B, however, it is now clear that the population expansion encountered by population 3 has led to elongated branch lengths. The full effect of a population expansion—to generate long branches and lots of single nucleotide variants at low frequencies—is realized. This is the result of the different ways in which we modeled a population expansion in scenarios B and C. In scenario C, population 3 instantaneously increases to a size of 100,000 sequences and maintains this size for 10,000 generations. This is quite different from the slow but inexorable growth of population 3 in scenario B, which, as we have seen, leads to an effective population size considerably less than 10,000 despite the fact that population 3 has attained a size of 100,000 sequences by current time in both scenarios.

2.3.1.4 Scenario D

When running coalescent simulations for scenario D (Fig. 2.5d), we introduce three additional details: a one-time admixture event, recombination, and continuous gene flow. Once again, we assume that the effective population size of population 2 is $N_e = 10{,}000$ and use this to scale all parameter values. We also use deeper time points, setting $t_3 = 80{,}000$ generations ($2 \times 4N_e$ generations), $t_2 = 40{,}000$ generations ($1 \times 4N_e$ generations), and $t_1 = 25{,}000$ generations ($0.625 \times 4N_e$ generations). The relevant events, moving forward in time, are then as follows:

- At t_3, the common ancestral population splits in two, moving forward in time.
- At t_2, one-half of the sequences in population 2 are drawn from population 1, representing a one-time admixture event.
- At t_1, population 3 originates as a population with size $N_e = 100$, after which it undergoes exponential growth to a population size of $N_e = 100{,}000$ by t_0 (present day).

Exponential Growth of Population 3

Because the time allotted for exponential growth from 100 to 100,000 sequences is longer (25,000 generations) relative to that simulated in scenario B (where it was 20,000 generations), we need to calculate the exponential growth rate anew:

$$r = ln\left(\frac{100{,}000}{100}\right)\frac{1}{0.625} = 11.05 \tag{2.12}$$

Continuous Migration

We will simulate a very high symmetric migration rate between populations 2 and 3 of $m = 0.01$, which indicates that every generation, 1% of the sequences in populations 2 and 3 consist of migrants from the opposite

population. The population-scaled migration rate $M = 4N_e m$. Recall that the N_e in question is that of the constant-sized population 2—i.e., 10,000. Thus, gene flow from population 2 to 3 and vice versa is $M = 4 \times 10^4 \times 10^{-2} = 400$.
Recombination

Let $\rho = 4N_e r$ represent the population-scaled recombination rate, where r is the probability that a cross-over occurs at a specific site in the simulated sequence (note that this is a distinct r from the exponential growth rate calculated immediately above). A reasonable *per-site* value for r is 10^{-8}; therefore, $\rho_{site} = 4 \times 10^4 \times 10^{-8} = 4 \times 10^{-4}$. This will need to be multiplied by the sequence length simulated to obtain what is essentially the probability that a cross-over occurs somewhere along the length of a simulated sequence. We will simulate sequence lengths of 10,000 bp and 250,000 bp, which will therefore have values of $\rho_{sequence} = \rho_{site} \times 10^4 = 4$ and $\rho_{sequence} = \rho_{site} \times 2.5 \times 10^5 = 100$, respectively.

We simulate scenario D, specifying 25 data sets of 30 10,000 bp sequences each, ten sets from each population:

```
dat <- scrm('30 25 -t 4 -r 4 10000 -I 3 10 10 10 -m 2 3 400 -m 3 2 400
  ↪ -eg 0 3 11.05 -ej 0.625 3 2 -eps 1.0 2 1 0.5 -ej 2.0 1 2')
```

Because the 25 data sets of 30 sequences are simulated independently of one another, we can treat them as 25 *unlinked* 10,000 bp sequences. The command is dissected as follows:

- `30 25 -t 4`: Because we aim to simulate 25 unlinked sequences of 10,000 bp ($\theta = 4$ for each of the 25 sequences), we require 25 independent coalescent simulations with a sample size of 30 in each case. As shown below, we then combine the 25 independent (i.e., unlinked) samples together. We will, for example, concatenate the first sample from each of the 25 data sets for a *total* data set that represents the 25 unlinked sequences sampled from a single individual. Because each of the 25 simulations simulates sequence evolution at unlinked, independent loci, it is safe for us to randomly combine sequences. Essentially, we are performing post hoc independent assortment. One thing we must not do, however, is concatenate sequence data from separate populations.
- `-r 4 10000`: This flag indicates that meiotic recombination in the form of crossing over should be simulated during the generation of each of the 25 data sets. The parameter values are the values of ρ (calculated above) and the length of the sequence (in base pairs).
- `-I 3 10 10 10`: As before, this flag indicates that there are three populations at current time, from each of which a sample of ten haplotypes is drawn. It is important to understand that the first ten rows of the resulting table of segregating sites are drawn from population 1, the next ten from population 2, and the final ten from population 3.

- `-m 2 3 400 -m 3 2 400`: The lowercase `m` flag is used to indicate migration rate between a pair of specific populations in a single direction. In each case, the first two parameters are the population number from which migrants emigrate and the population number to which they immigrate, respectively. The third parameter is the population-scaled migration rate (M) calculated above. We need both flags to simulate *symmetric* rather than one-way migration between the two populations.
- `-eg 0 3 11.05 -ej 0 0.625 3 2 -eps 1.0 2 1 0.5 -ej 2.0 1 2`: The growth and join flags are used similarly to their use in scenarios B and C. However, the `-eps` flag is new. It states that at $1 \times 4N_e$ generations ago, one-half (fourth parameter) of the sequences in population 2 were derived from population 1 (second and third parameters, respectively). In other words, this is an example of one-way admixture.

Next, we combine the segregating sites from all 25 loci into one data frame and provide column/marker names:

```r
ss <- data.frame(dat$seg_sites[[1]])
marker_pre = rep("M1", dim(dat$seg_sites[[1]])[2])
for (i in 2:length(dat$seg_sites)){
  marker_pre = c(marker_pre, rep(paste("M", i, ".", sep = ""),
      ↪ dim(dat$seg_sites[[i]])[2]))
  ss <- cbind(ss, dat$seg_sites[[i]])
}
marker_suf <- seq(1:dim(ss)[2])
names(ss) <- paste(marker_pre, marker_suf, sep="")
names(ss) <- marker_labs
```

I then wrote the data frame ss to a file, which I input to the popular program STRUCTURE (Pritchard et al. 2001) using the admixture model, 240,000 MCMC iterations (40,000 burn-in), ploidy of one, three as the assumed number of populations, and all other parameter values set to default.

In essence, we are treating the simulated data as an empirical data set and using the program STRUCTURE to assess whether the given evolutionary history produces evidence for three separate populations or not. Figure 2.11a shows the results of the STRUCTURE analysis. Note that because each row of the input file was analyzed as data from a haploid individual, each vertical bar in the figure shows the admixture proportions of a "haplotype" of 25 unlinked sequences and *not* a diploid individual. If the simulated data were indeed empirical data, we see there is no indication of independent evolution of populations 2 and 3 as all 20 samples from these populations show ~ 100% ancestry to a single population (represented by dark gray). Population 1 is clearly distinct from both of these populations.

This result is not surprising, given that the level of gene flow between the two populations is very high ($M = 400$). A well-known—perhaps in itself surprising—rule of thumb is that when $M >> 1$ (which often represents a very small absolute number of migrants, Nm) between two defined groups of individuals of the same species, these groups behave as a single, panmictic population.

Remember, however, that the data set analyzed was a concatenation of 25 unlinked 10,000-bp loci. When single 10,000-bp loci from the same data set are analyzed in STRUCTURE, the results are highly variable (Fig. 2.11b$_{1-3}$). Sometimes, we obtain nearly the same result as when the total data set is used (Fig. 2.11b$_1$). However, in many cases, we obtain results that would seem to split the sampled sequences from populations 2 and 3 into groups that do not follow their a priori defined populations (Fig. 2.11b$_1$ and b$_2$). Comparison of the STRUCTURE results in Fig. 2.11a and b nicely demonstrates that simulation results can be used to perform a power analysis that provides us with valuable insight into the type and quantity of data we should gather from natural populations of interest. The lesson here would clearly be that we need to gather sequence data from multiple unlinked positions in the genome to make trustworthy inferences regarding the population structure of the focal species.

Finally, I took this idea one step further and ran another coalescent simulation of a single 250,000-bp sequence using all the same demographic and time parameters as before. Importantly, this produces a simulated data set of exactly the same number of assayed nucleotides as a data set of 25 10,000-bp sequences. STRUCTURE analysis produced results highly divergent from those of the previous analyses in that all three populations were characterized as being highly isolated from one another (Fig. 2.11c). While it is likely that many simulations of a single 250,000-bp sequence would produce STRUCTURE results analogous to those shown in Fig. 2.11a, the result shown in Fig. 2.11c speaks to the need to sample multiple, unlinked loci, which provide independent assessments of evolutionary history and prevent one anomalous locus from leading us to draw incorrect conclusions.

2.3.1.5 Interpreting Genealogical Results When Recombination Is Simulated

Although we simulated recombination in our treatment of scenario D, we simply used the sequence data and did not address the underlying genealogical information. When recombination is included in a coalescent simulation, looking backward in time, there are two competing processes that result in the genealogy: merger of two sequences (coalescence) and splitting of a single sequence into two (recombination). The complex tree documenting the history of recombination and coalescence that links the sampled sequences to a common ancestral sequence is referred to as an **ancestral recombination graph** (Hudson 1990). In practice, the complex history of coalescence and recombination means that a sequence of a given length can be broken into subsequences whose borders are defined by recombination breakpoints—sites where crossing over took place. Each of these subsequences has its

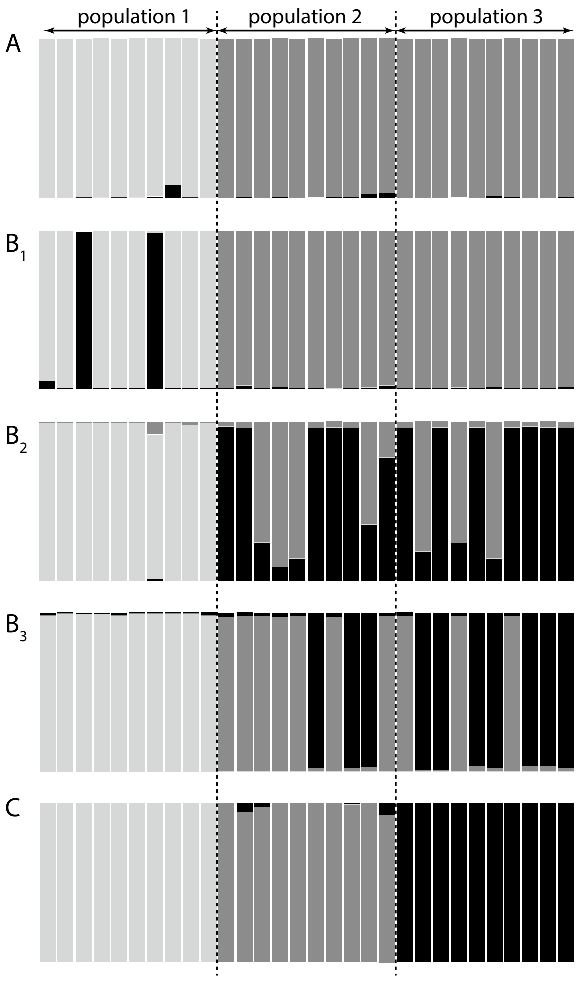


Fig. 2.11 STRUCTURE results for differently sized data sets generated according to evolutionary scenario D. (**a**) Based on a data set of 25 unlinked loci of 10,000 bp each. (**b**) Three separate results based on *one of the* 10,000-bp loci from (**a**). (**c**) Based on a data set of one 250,000-bp data set

own genealogy. As a result, there is an MRCA for each subsequence of the sampled sequence and a grand most recent common ancestor, or GMRCA, which is the common ancestral sequence to all subsequences of the sampled sequence (Fig. 2.12a). The ancestral recombination graph can be separated into individual genealogies for each subsequence (Fig. 2.12b), and the sampled sequences incorporate mutations on each of these trees (Fig. 2.12b–c).

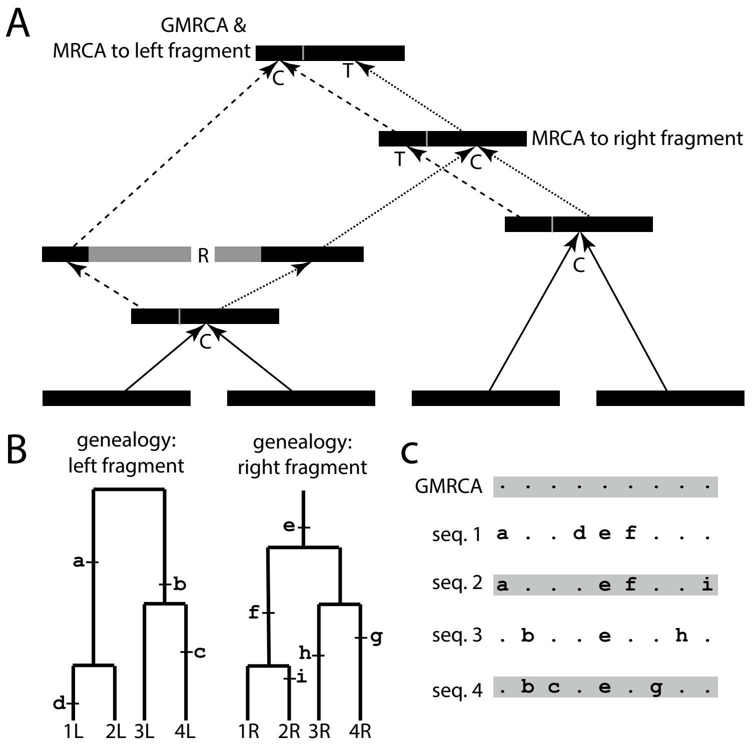


Fig. 2.12 The ancestral recombination graph shows the history of coalescent and recombination events leading from a grand most recent common ancestor (GMRCA) to a sample of sequences. (**a**) The ancestral recombination graph. A single recombination event is shown with the breakpoint (vertical gray lines) splitting the overall sequence sampled into left and right fragments. Solid lines show paths common to the genealogies of both fragments, dashed lines show paths specific to the genealogy of the left fragment, and dotted lines show paths specific to the genealogy of the right fragment. C stands for a coalescent event, R for a recombination event, and T for transmission of a sequence. Gray sequence is sequence not represented in the sample. (**b**) The reconstructed genealogies of the left and right fragments; note that the MRCA to the right fragment occurs in the more recent past than that of the left fragment. Mutations denoted by small arabic letters. (**c**) The sequences of the sample based on mutations shown in (**b**). Dots indicate the alleles of the GMRCA, while letters indicate derived mutations

2.3.2 *Package COALA*

The R package COALA allows the user to simulate sample sequences using SCRM or other simulators including MS. The chief advantage to using COALA is that the user specifies the requisite parameter values and the command line fed to MS or SCRM is generated automatically. This is particularly advantageous in cases where the simulated model is complex, which makes it easy to incorrectly specify the order and/or details of the command line.

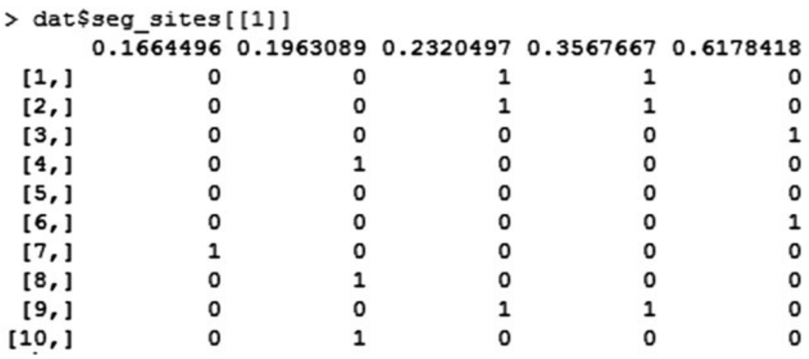
Let us begin with a simple model to familiarize ourselves with the basic syntax used in COALA functions and simulation. Assuming you have installed the `scrm` and `coala` packages in R, the following code stores the results of coalescent simulation in the object `dat`:

```
1  library(coala) # loads SCRM as well
2  mod <- coal_model(10,1) + feat_mutation(1) + sumstat_seg_sites()
3  dat <- simulate(mod)
```

Model `mod` is specified in line 2 using the function `coal_model()`, one of many *feature* functions (`feat_mutation()`), and one of many *summary statistic* functions (`sumstat_seg_sites()`). Note that feature and summary statistic functions are strung together using a "+". The arguments to `coal_model()` specify a *single* sample size of ten haploid sequences. The value of the population mutation parameter θ is the argument to `feat_mutation()`, and `sumstat_seg_sites()` indicates we want to collect the state of each segregating site in all of the sampled sequences. In line 3, we then pass the specified model to the function `simulate()`, and all output is stored in the object `dat`. By default, the SCRM simulator is used, but you can gain access to the simulator MS, for example, by installing the package PHYCLUST and adding the `activate_ms()` function with a suitable priority in order to use MS instead (see COALA package documentation for full details).

To view the SCRM command line associated with the model, we type `dat<currencydollar>cmds`, which prints the `list` of commands used for each separate simulation. In our case, we ran only one simulation, and the value of `dat<currencydollar>cmds[[1]]` is `"scrm 10 1 -t 1"`. In order to view the polymorphism data, we simply enter `dat<currencydollar> seg_sites[[1]]`, which provides the data shown in Fig. 2.13. Columns are segregating sites, and rows are individual sequences. The column headings specify, as a real number, the position of each segregating site. To convert these to nucleotide positions (i.e., integers), we need to consider the length of the simulated sequence. One way to satisfy our simulated parameter value $\theta = 1$ is to imagine a population of 2500 individuals, a point mutation rate of 10^{-8}, and a sequence that is 10,000 bp long: $\theta = 4N_e\mu = 4 \times 2.5 \times 10^3 \times 10^{-8} \times 10^4 = 10^0 = 1$. Then the first segregating site is at nucleotide 1664 or 1665 depending on how you decide to round numbers.

Now let us simulate a substantially more complex scenario with two populations, one of which experiences a recent exponential population expansion (population 1) and the other which experiences a recent population bottleneck (population 2; Fig. 2.14a). We assume that population 1 consists of $N_e = 10{,}000$ individuals at time 0 (which is used to scale all times), a sequence 10,000 bp in length, and a point mutation rate of 10^{-8}. Furthermore, we assume that 100 (or 0.0025 $4N_e$) generations ago, population 1 only held 100 individuals while population 2 consisted of 20,000 individuals. At this time point, population 1 began exponential population growth at a rate of $r = 1842.1$, and population 2 instantaneously declined to a size 0.01 of



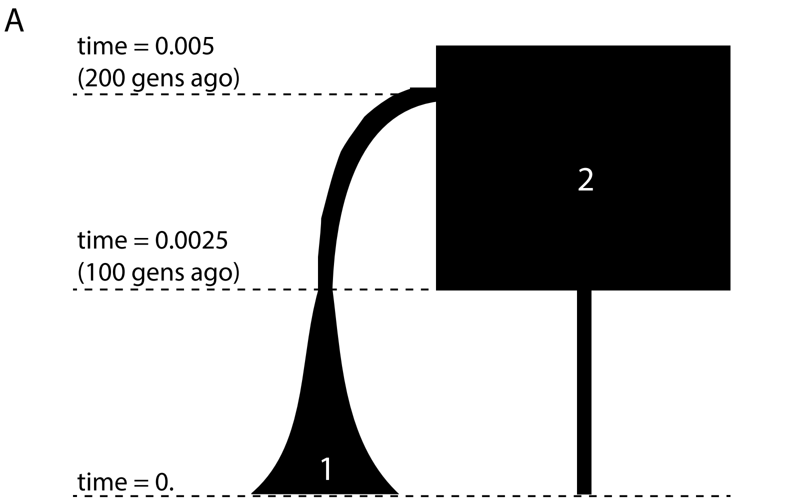
```
> dat$seg_sites[[1]]
       0.1664496 0.1963089 0.2320497 0.3567667 0.6178418
 [1,]          0         0         1         1         0
 [2,]          0         0         1         1         0
 [3,]          0         0         0         0         1
 [4,]          0         1         0         0         0
 [5,]          0         0         0         0         0
 [6,]          0         0         0         0         1
 [7,]          1         0         0         0         0
 [8,]          0         1         0         0         0
 [9,]          0         0         1         1         0
[10,]          0         1         0         0         0
```

Fig. 2.13 Polymorphism data generated using COALA

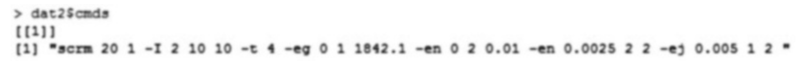
N_e = 10,000, or 100 individuals. Finally, 200 (or 0.005 $4N_e$) generations ago, the ancestral population split into the extant populations. The following code specifies this model and simulates according to the model:

```
mod2 <- coal_model(c(10,10),1) + feat_mutation(4) + feat_growth(1842.1,
  ↪ population=1, time="0.") + feat_size_change(0.01, population=2,
  ↪ time="0.") + feat_size_change(2, population=2, time="0.0025") +
  ↪ feat_pop_merge(0.005,1,2) + sumstat_seg_sites() +
  ↪ sumstat_nucleotide_div(population=1) + sumstat_sfs(population=2)
dat2 <- simulate(mod2)
```

Figure 2.14b–c shows the SCRM command associated with the model and the simulated data from the first ten of many segregating sites. The final two commands in line 1 of the previous listing introduce two additional summary statistics that can be calculated on the simulated data. First, `sumstat_nucleotide_div(population=1)` specifies calculation of nucleotide diversity on the data from population 1 (the first ten sequences). See Sect. 3.4.6.1 for a discussion of what nucleotide diversity represents. For now, suffice it to say that nucleotide diversity is the mean number of pairwise differences between sampled sequences. In our case, nucleotide diversity (`dat2<currencydollar>pi`) equals zero because there is no variation; all sequences in population 1 are identical (the first ten rows/haplotypes listed in Fig. 2.14c). Unfortunately, COALA does not provide a means to output nucleotide diversity for more than one population; we can calculate nucleotide diversity across sequences from all populations using the argument `population="all"`, but this would be quite inappropriate in this case as the two populations have experienced dramatically different histories during the last 100 generations. Second, `sumstat_sfs(population2)` calculates the site frequency spectrum *for population 2* (the second ten rows/haplotypes in Fig. 2.14c). `dat2<currencydollar>sfs` holds the vector `6 2 8 0 3 0 2 0 0`; this indicates there are six segregating sites in which only one sequence bears the derived allele, two segregating sites in which two sequences (of 10)

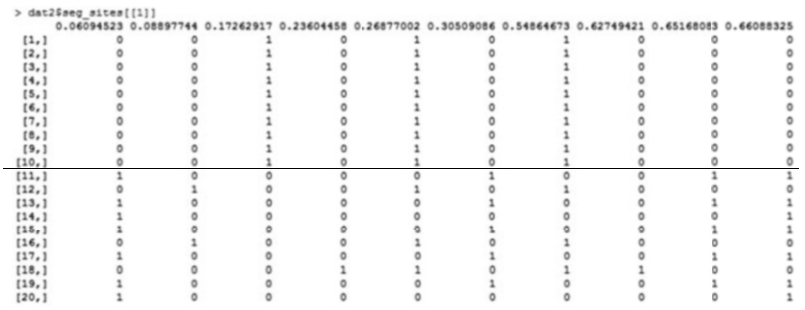


B



```
> dat2$cmds
[[1]]
[1] "scrm 20 1 -I 2 10 10 -t 4 -eg 0 1 1842.1 -en 0 2 0.01 -en 0.0025 2 2 -ej 0.005 1 2 "
```

C



```
> dat2$seg_sites[[1]]
      0.06094523 0.08897744 0.17262917 0.23604458 0.26877002 0.30509086 0.54864673 0.62749421 0.65168083 0.66088325
 [1,]          0          0          1          0          1          0          1          0          0          0
 [2,]          0          0          1          0          1          0          1          0          0          0
 [3,]          0          0          1          0          1          0          1          0          0          0
 [4,]          0          0          1          0          1          0          1          0          0          0
 [5,]          0          0          1          0          1          0          1          0          0          0
 [6,]          0          0          1          0          1          0          1          0          0          0
 [7,]          0          0          1          0          1          0          1          0          0          0
 [8,]          0          0          1          0          1          0          1          0          0          0
 [9,]          0          0          1          0          1          0          1          0          0          0
[10,]          0          0          1          0          1          0          1          0          0          0
[11,]          1          0          0          0          0          1          0          0          1          1
[12,]          0          1          0          0          1          0          1          0          0          0
[13,]          1          0          0          0          0          1          0          0          1          1
[14,]          1          0          0          0          0          0          0          0          0          1
[15,]          1          0          0          0          0          1          0          0          1          1
[16,]          0          1          0          0          1          0          1          0          0          0
[17,]          1          0          0          0          0          1          0          0          1          1
[18,]          0          0          0          1          1          0          1          1          0          0
[19,]          1          0          0          0          0          1          0          0          1          1
[20,]          1          0          0          0          0          0          0          0          0          1
```

Fig. 2.14 A model simulated in COALA. (**a**) Summary of the demographic changes in the model. Two hundred generations ago, population 1 split from the ancestral population (looking forward in time). One hundred generations ago, population 1 began growing exponentially, and population 2 experienced a severe and instantaneous population bottleneck. (**b**) The SCRM command line assembled by COALA. (**c**) Polymorphism data for all 20 sequences at the first ten of 21 total segregating sites in the sample. The horizontal line separates sequences sampled from populations 1 and 2. Note the absence of variation among sequences drawn from population 1 (sequences 1–10)

bear the derived allele, eight segregating sites in which three sequences (of 10) bear the derived allele, etc. Again, calculation of this summary statistic in COALA suffers from the same limitation; it can either be calculated across all populations or in just one of the populations. Nevertheless, this is a very handy package for coalescent simulation. It is possible to simulate selection at a single locus (`feat_selection()`), calculate a wide variety of summary statistics, and more.

As a test of your population genetic intuition, posit an explanation for why the sampled sequences from population 1 are devoid of variation while considerable variation is found among the sampled sequences from population 2.

2.4 The Utility of Retrospective, Coalescent Simulation

It bears mentioning again that the majority of this volume focuses on forward simulation. This is not because coalescent simulation is obsolete or inapplicable to evolutionary inference. Coalescent simulation is perfectly appropriate for a variety of inferential tasks. In fact, whenever possible, the use of coalescent simulation is preferred due to its rapidity.

As we move into Chap. 3, however, our focus shifts to and stays with forward-in-time simulation. One reason for this focus is that, to my knowledge, there are few texts that provide a primer on forward-in-time simulation. My hope is that as you, the reader, make your way through the subsequent chapters, you will feel empowered to use forward simulation to tackle problems that cannot be met by coalescent simulation's limitations.

References

Berg J, Coop G (2015) A coalescent model for a sweep of a unique standing variant. Genetics 201:707–725

Hahn MW (2018) Molecular Population Genetics. Sinauer Associates, Oxford, UK

Hein J, Schierup MH, Wiuf C (2005) Gene Genalogies, variation and evolution. Oxford University, Oxford, UK

Hudson RR (1990) Gene genealogies and the coalescent process. In: Futuyma D, Antonovics J (eds) Oxford surveys in evolutionary biology, vol 7. Oxford University, Oxford, pp 1–44

Hudson RR (2002) Generating samples under a Wright-Fisher neutral model of genetic variation. Bioinformatics 18:337–338

Kern A, Schrider D (2016) Discoal: flexible coalescent simulations with selection. Bioinformatics 32:3839–3841

Paradis E, Schliep K (2019) ape 5.0: an environment for modern phylogenetics and evolutionary analyses in R. Bioinformatics 35:526–528

Pritchard J, Stephens M, Donnelly P (2001) Inference of population structure using multilocus genotype data. Genetics 155:945–959

Staab P, Zhu S, Metzler D, Lunter G (2015) SCRM: efficiently simulating long sequences using the approximated coalescent with recombination. Bioinformatics 31:1680–1682

Staab PR, Metzler D (2016) Coala: an R framework for coalescent simulation. Bioinformatics 32(12):1903–1904

Wakeley J (2008) Coalescent theory: an introduction. Roberts and Company Publishers, Greenwood Village

3 Mutation and Genetic Drift

And who's that, through the crack in the orange shade, breathing carefully? Watching? And where, keepers of maps, specialists at surveillance, would you say the next one will fall?[1]

– Thomas Pynchon, *Gravity's Rainbow*

3.1 Background

The vast majority of new variants generated by mutation are destined for loss in the near future. The reason is intuitive. In a diploid population of effective size N_e, a point mutation yields a new autosomal variant that begins as a single copy among $2N_e$ total copies of the nucleotide, or at a frequency of $1/2N_e$. Given its initial rarity, the new allele can easily be lost by chance in one or a few generations. There is no guarantee the allele will be sampled and transmitted to the next generation. As a concrete example, if the individual carrying the mutant allele does not mate, the mutant allele is lost after only one generation of existence. Even if the carrier of the mutant variant does mate, by Mendel's principle of equal segregation, the mutant allele will be transmitted with uncertainty (probability = 0.5).

[1] Quoted with permission. 1973, Penguin Classics.

Qualitatively, then, allele frequencies may evolve each generation solely due to random sampling of the finite number of alleles in the previous generation. Are there quantitative expectations related to this phenomenon? Yes. Perhaps the most important *simple* result of theoretical population genetics is that the fixation probability of a *neutral* allele (i.e., the probability it attains a frequency of 1.0) equals its current frequency (Kimura 1962). The fixation probability of a new neutral allele is therefore $1/2N_e$ in a diploid population; note this implies that loss of a new variant becomes more likely as population size increases. Even when a new genetic variant confers a considerable selective advantage, its survivorship is still highly improbable when at very low frequency in the population.

The precarious existence of new variants, coupled with their steady generation by mutation, leads to the characteristic shape of the *neutral* allele frequency spectrum (AFS). The AFS is the distribution of allele counts in a population sample, documenting the number of derived variants found as n copies in the population. The neutral AFS is geometrically distributed; the most common class of variants consists of those that exist as only one copy in the sample, which are called *singletons*. Although new alleles tend to be lost rapidly, mutation repeatedly counteracts these losses by introducing new variants.

The primary consequence of genetic drift is loss of genetic diversity due to the loss of alleles and, therefore, polymorphism. One way to measure this loss is by examining the decay of heterozygosity over time. In the absence of mutation, genetic drift leads to a predictable decay in heterozygosity at a polymorphic locus. In a diploid population, heterozygosity at a polymorphic locus declines by $H/2N_e$ each generation, where H is the current value of heterozygosity (Wright 1931). A deterministic recursion equation specifies what H will be after t generations of genetic drift:

$$H_t = H_0(1 - 1/2N_e)^t,$$

where H_0 is the starting value of heterozygosity.

In this chapter, excepting Sect. 3.2, we model both mutation and genetic drift. Given sufficient time, the population will reach *mutation-drift equilibrium*, at which point the generation of new variants by mutation is balanced by the loss of genetic variants due to genetic drift. For a given mutation rate and effective population size, we can predict the equilibrium level of genetic diversity, measured in a diploid population as the parameter $\theta = 4N_e\mu$. Although individual new alleles are more likely to be lost in a large population than in a small population, a greater total number of new alleles enter the large population each generation. Thus, the expected level of genetic diversity in a population at mutation-drift equilibrium increases with the size of the population. For example, given the same point mutation rate for a 10,000-bp locus of $\mu = 10{,}000 \times 10^{-8} = 10^{-4}$, genetic diversity $\theta = 4 \times 10{,}000 \times 10^{-4} = 4$ for a diploid population of size $N_e = 10{,}000$ and $\theta = 4 \times 100{,}000 \times 10^{-4} = 40$

for a diploid population of size $N_e = 100{,}000$. Importantly, as detailed in Sect. 3.6, the theoretical expectation of genetic diversity (θ) at a neutral locus at mutation-drift equilibrium provides us with an important metric for validating the simulation program FORTUNA, which we begin constructing in this chapter.

3.2 A Textbook Simulation

Introductory biology textbooks often contain a figure that shows results from simulation of genetic drift on an existing genetic variant for different population sizes. This figure provides the reader of the textbook with an intuitive understanding of the relationship between population size and the magnitude of genetic drift.

A primary goal of this volume is to empower the reader to use simulation in order to build intuition regarding processes of much greater complexity. Therefore, it is worth taking a few moments to code this simple simulation to reinforce the value of simulation in the science of evolutionary genetics and prepare for more challenging coding tasks. In the following program, we ignore mutation and assume that a standing variant begins at a frequency of 0.5. Our expectation is that intergenerational change in allele frequency will be dramatic in small populations and muted in large populations.

Ignoring the idiosyncratic details of different life histories and, to put it coldly, reproduction is a biological means for sampling the previous generation's alleles to produce the next generation's set of alleles. The central limit theorem dictates that the magnitude of generation-to-generation flux in allele frequency due to finite sampling (and not selection, migration, etc.) is negatively correlated with population size. Our simple program of genetic drift in a population of *haploid* organisms models reproduction as sampling with replacement. The count of the focal allele in the next generation is generated by drawing a random binomial variable with parameters N, haploid population size, and p, the frequency of the focal allele in the previous generation. Because population size does not change, N remains constant, but p potentially changes every generation. Each haploid individual of the next generation results from a random Bernoulli trial in which the chance of "success" (focal allele *is* sampled) equals p. Unlike the prospective simulation program we construct in this volume, which explicitly simulates each mating, we draw a single binomially distributed random variable that represents the count of focal alleles in the population in the next generation. A single source file is sufficient for the program.

drift.cc

```
 1  \\ includes
 2  #include <random>
 3  #include <fstream>
 4  #include <string>
 5  #include <map>
 6  #include <vector>
 7
 8  int main(int argc, char *argv[]) {
 9
10    // command line arguments
11    double N = atof(argv[1]);
12    std::string suffix = argv[1];
13    int simnum = atoi(argv[2]);
14    int gen = atoi(argv[3]);
15
16    std::mt19937 engine(std::time(0)); //initialize the random engine
17    std::map<int, std::vector<double>> data;
18
19    for (int i = 0; i < simnum; ++i) { // once per simulation replicate
20      double freq = 0.5; //start all sims with frequency = 0.5
21      std::binomial_distribution<> b(N, freq); //binomial sampler
22      for (int j = 0; j < gen; ++j) { // once per generation
23        data[i].push_back(freq);
24        freq = b(engine) / N;
25        b.param(std::binomial_distribution<>::param_type(N, freq));
26      }
27    }
28
29    // print data held in the map, data, to output file
30    std::string fname = "drift_N" + suffix;
31    std::ofstream output;
32    output.open(fname.c_str());
33      //print header line
34    output << "gen ";
35    for (int set = 0; set < simnum-1; ++set)
36      output << set << " ";
37    output << simnum << std::endl; // avoid extra space after last entry
38      //print data for each sim and each generation
39    for (int j = 0; j < gen; ++j) {
40      output << j << " "; // print generation number
41      for (int i = 0; i < simnum-1; ++i)
42        output << data[i][j] << " " ; // print focal allele frequency
43      output << data[simnum-1][j] << std::endl;
44    }
45    output.close();
46
47    return 0;
48  }
```

drift.cc takes three command-line arguments in the order (1) haploid population size, (2) the number of replicate simulations to run, and (3) the number of generations for which each replicate should run. Lines 11–14 use these inputs to define the variables `N`, `simnum`, and `gen`, respectively. Note that `N` is declared as a `double`; this is because `N` will be used as the denominator to calculate the frequency (a real number) of the focal allele each generation.

The C++11 standard introduced random number generators for several useful probability distributions. To draw random variables from any of these distributions, it is necessary to initialize a random number engine (line 16), which is passed as an argument whenever a random variable is generated (e.g., line 24). Here and throughout the volume, we deploy the reliable and widely used Mersenne Twister 19937 (`std::mt19937`) pseudorandom number generator.

The program populates a `map` named `data`, whose key is the replicate number and whose value is a `vector` of allele frequencies for the simulation in question. Each generation (lines 22–26), the current frequency is added to `data` (line 23), and a binomial sample of the focal allele is drawn, from which the new frequency (variable `freq`) is calculated (line 24). The new frequency is then used to reparameterize the binomial distribution `b` (line 25). Finally, the contents of `data` are printed to a file whose name is *drift_N** (lines 29–45), where * is the sample size specified in the command-line argument.

Next, we compile the program and run five replicates each for $N = 50$ and $N = 50{,}000$ producing the output data files *drift_N50* and *drift_N50000*, respectively.

compile and run `drift.cc`

```
g++ -std=c++11 drift.cc -o drift // -std flag allows use of <random>
./drift 50 5 100 // small number of generations needed for focal allele
    ↪ to fix or be lost
./drift 50000 5 50000
```

Now, we plot the simulation results in R.

`plot_drift.r`

```
library(reshape2) // allows us to "melt" results from all five simulations
library(ggplot2)
library(cowplot)
d50 = read.table(file = "drift_N50", header = T)
d50000 = read.table(file = "drift_N50000", header = T)
melted50 = melt(d50, id = "gen")
melted50000 = melt(d50000, id = "gen")
ggplot(metled50, aes(gen, value, factor=variable)) + geom_line() //
    ↪ Figure 4.1A
ggplot(melted50000, aes(gen, value, factor=variable)) + geom_line() +
    ↪ xlim(0,100) // Figure 4.1B
ggplot(melted50000, aes(gen, value, factor=variable)) + geom_line() //
    ↪ Figure 4.1C
```

The resulting plots are shown in Fig. 3.1. Comparison of Fig. 3.1a and b shows the erratic swings in allele frequency associated with small population size versus the muted changes to allele frequency from generation to generation when population size is large. Although the focal allele is fixed or lost in all five simulations in less than 60 generations when $N = 50$, polymorphism is maintained for 50,000 generations in three of five simulation replicates when $N = 50,000$ (Fig. 3.1c).

We next consider the relationship between population size and the mean and variance of p across multiple simulations. Although the plots in Fig. 3.1 show that each simulation run results in deviation from the initial frequency of the focal allele ($p = 0.5$), the expected value of p is 0.5. Yet this expectation is realized in a somewhat peculiar manner. Given sufficient time and no countervailing factors such as natural selection, the focal allele is either lost or fixed. Because the probability of fixation of an allele is equal to its initial frequency, we expect that half of all simulations will result in loss of the allele while the other half will result in fixation of the allele. Therefore, over multiple simulations, the average frequency of a focal allele with $p_{initial} = 0.5$ is expected to be 0.5. If we run 1000 simulations, $E[p] = 0.5$ is attained rapidly when population size is small and more slowly when population size is large (Fig. 3.2a). Following the same logic, if half of all simulations result in $p = 1$ and the other half in $p = 0$, then $E[\sigma_p^2] = 0.25$. Figure 3.2b shows that this expected variance is met in short order when $N = 50$ and has still to be attained after 100,000 generations when $N = 50,000$.

We now check that simulated data using our binomial-based process model of genetic drift match the theoretical decay of heterozygosity due to drift. Once again using `drift.cc`, I simulated 100 replicates of haploid $N = 50,000$ populations for 50,000 generations and plotted the expected heterozygosity versus time (see Sect. 3.1) and compared this to the observed mean value of heterozygosity across all 100 simulations (Fig. 3.3). The match is quite good. However, it is important to note that few *individual* simulations track expected heterozygosity very well. For example, many simulated populations lost one allele or the other, after which point heterozygosity equals zero. This is another example where a statistical expectation is met only when we average multiple replicates. Admittedly, this is the definition of statistical expectation, but we should remind ourselves that an expected value might actually be unexpected in a single population.

Although forward simulations are much slower than coalescent simulations, forward simulations provide two major advantages that even the simple `drift.cc` makes clear. First, a forward model of evolution is easier to conceptualize and therefore code for many people. In nature, genetic drift has a forward-in-time effect on genetic variation after all; as mentioned above, offspring are produced via the sampling "method" of reproduction. It is simple to encode this insight in a simulation program as random sampling of a statistical distribution. As we move on to individual-based, forward simulations, the value of defining models that agree with our perception of the flow

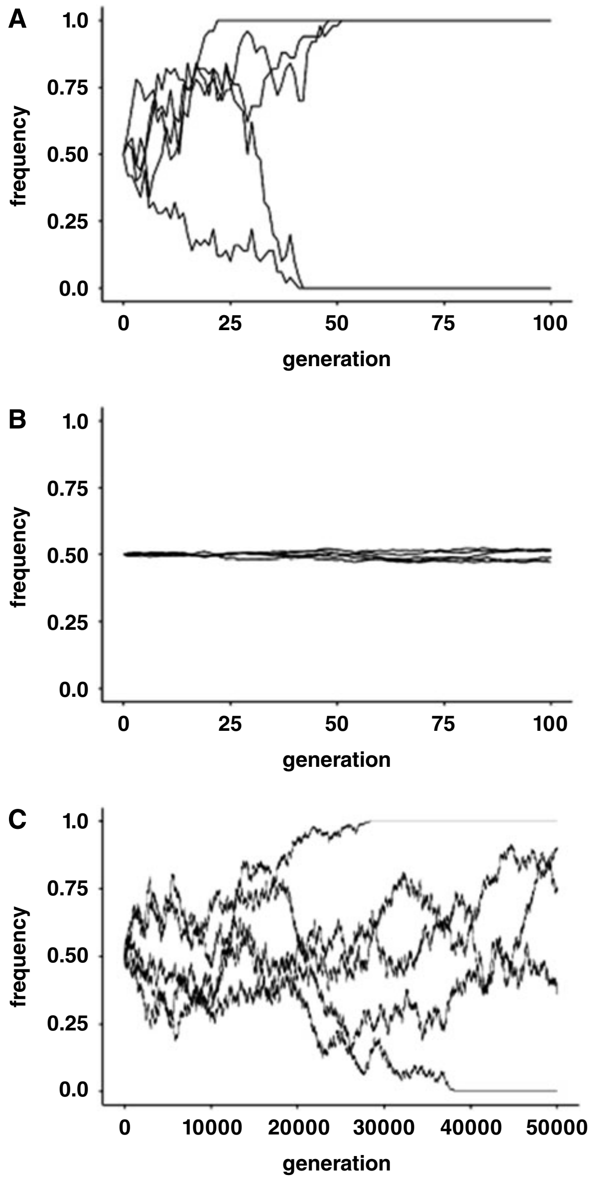


Fig. 3.1 Simulations of genetic drift on a focal variant beginning at a frequency of 0.5. (**a**) $N = 50$. (**b** and **c**) $N = 50{,}000$. Note the change in the scale of the time axis between (**b**) and (**c**)

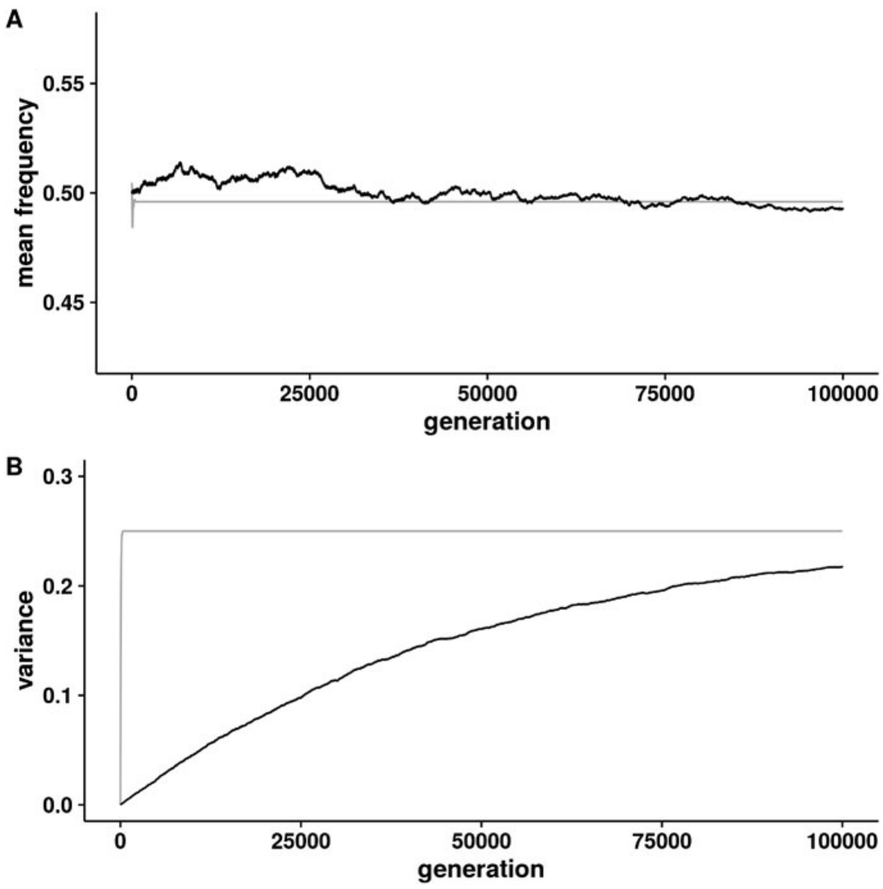


Fig. 3.2 Mean (**a**) and variance (**b**) of the frequency of a focal allele beginning at a frequency of 0.5 across 1000 simulations for $N = 50$ (gray lines) and $N = 50{,}000$ (black lines)

of time will become even more apparent. Second, forward simulation allows us to document complete historical information regarding genetic variation. Coalescent simulation only considers the genealogy leading to the alleles of the sample (see Chap. 2); extinct alleles are ignored. While this simplification is profound and central to the efficiency of coalescent simulation, historical information is often of interest. Coalescent simulation cannot provide the data shown in Figs. 3.1 and 3.2 because it only provides us with information regarding a terminal generation. On the contrary, we can track the frequency of an allele every generation using forward simulation (Fig. 3.4). Furthermore, we can store the fate of each and every allele that emerges over the course of a simulation. Comprehensive historical data such as this may be quite helpful depending on the goals of the researcher.

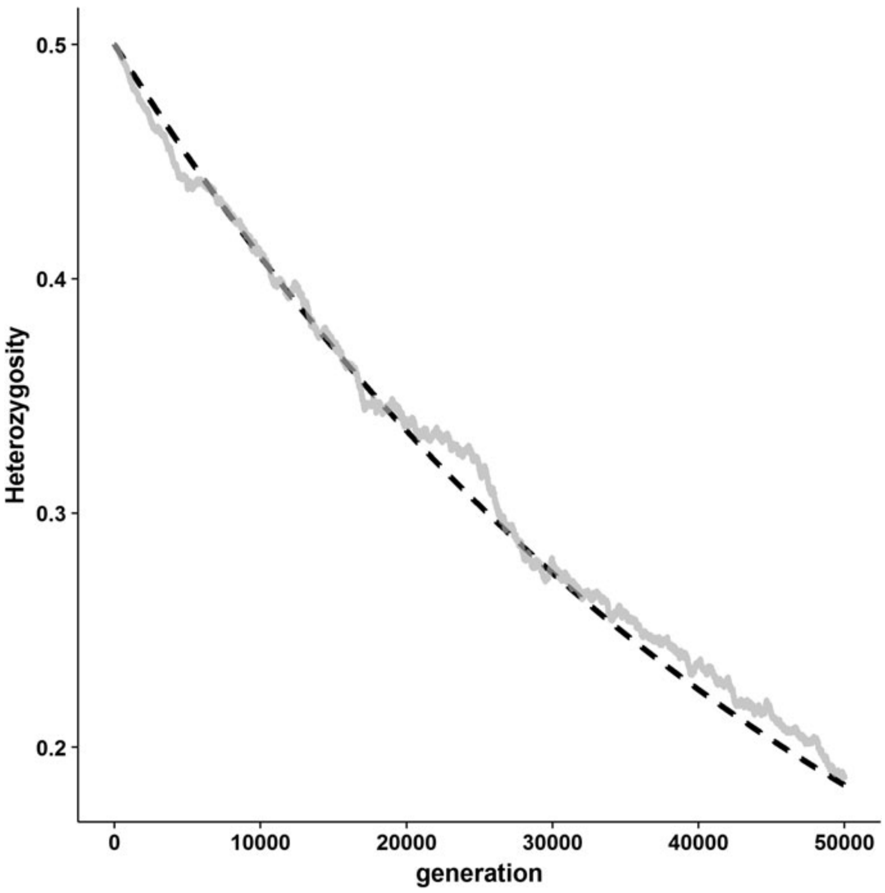


Fig. 3.3 The decay of heterozygosity at a diallelic locus in the absence of mutation. The gray line plots mean heterozygosity across 100 independent simulations; each simulated locus began with two alleles at a frequency of 0.5 and, therefore, starting heterozygosity of $2 \times 0.5 \times 0.5 = 0.5$. The dashed line plots the expected decline in heterozygosity for a haploid population where $N_e = 50{,}0000$ and $H_0 = 0.5$

3.3 Some Practicalities

3.3.1 Efficient Representation of a Genetic Sequence

In most cases, the direct product of our simulations will be genetic sequences. We should therefore consider the best means of representing genetic sequences in the programs we code. Perhaps the most straightforward approach would be to store each sequence as a `string`, array of `char`, or a `bitset` in which each site in the sequence is tracked. This would be an inefficient sequence representation for the simple reason that most positions

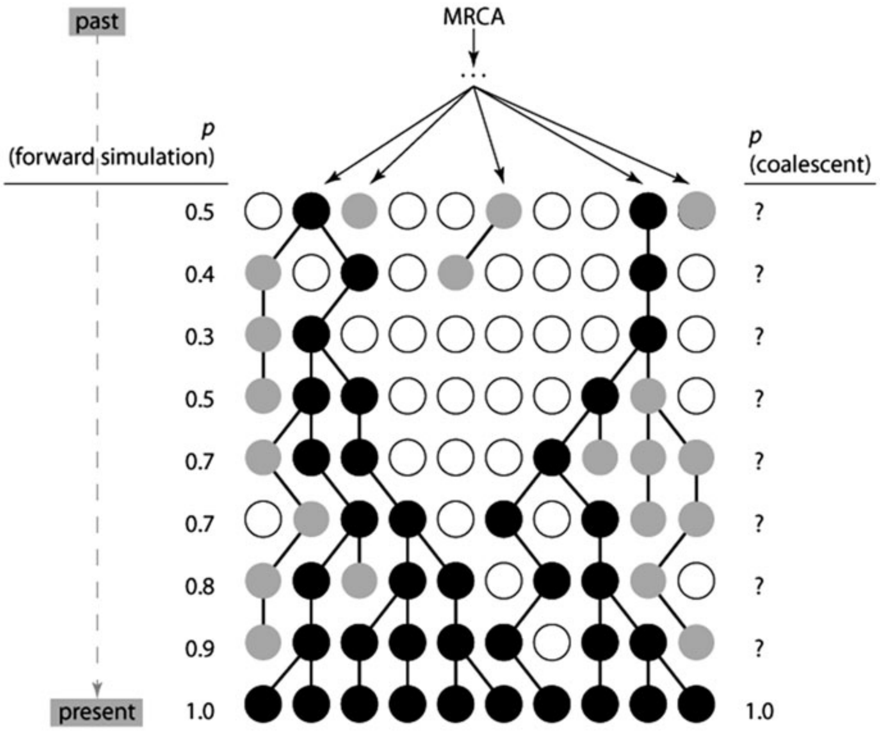


Fig. 3.4 The descent of a derived allele. Simulation only provides us with the genealogy connecting black circles. Forward simulation, on the other hand, allows us to record all genetic details of past generations. As a simple example demonstrating the value of a comprehensive history, forward simulation allows us to calculate the frequency of the derived allele (p) each generation, while coalescent simulation only provides us with p in the final, sampled generation. Black circles represent alleles that are directly ancestral to the sampled alleles. The alleles represented by gray circles are identical by descent to those alleles ancestral to the sample, but they are members of an extinct genetic lineage. Open circles represent alleles that are not identical by descent to the derived allele

in the sampled sequence will be invariant even when $\theta = 4N_e$ is realistically large. It is substantially more efficient to track only those sites within the sequence that are polymorphic in the simulated population.

Throughout the volume, therefore, we represent each *derived* allele as an object in its own right. The `Allele` object stores `private` variables such as its count in the population, generation of origin, and position in the sequence (Fig. 3.5, left column). Each sequence in the population is represented as a `vector` of the derived allele positions present in the sequence and diploid individuals as a collection of two of these sequences (Fig. 3.5, middle column). When calculating summary statistics on a sample of sequences and/or

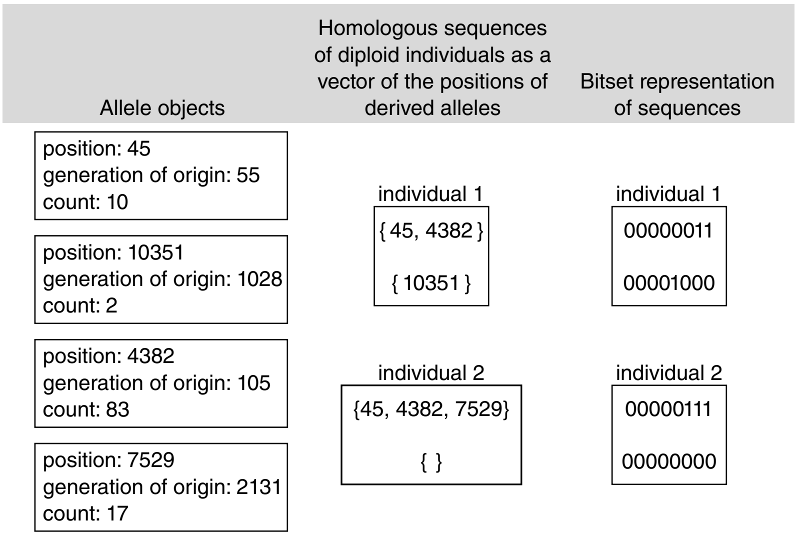


Fig. 3.5 An `Allele` object stores information regarding each derived allele (left column). We represent gene sequences as a `vector` of derived allele positions present in the sequence and individual, diploid organisms as a set of two sequences (middle column). For the purpose of summary statistic calculation in particular, we also use `bitsets` to represent sequences; the state at each segregating site is specified as a `0` (ancestral allele) or `1` (derived allele). The segregating sites documented in the `bitset` are ordered such that the state of the lowest segregating position (45 in this example) will be coded in the `bitset` as the rightmost (index `[0]`) bit; the four leftmost 0s of the `bitset` in this example provide room for the coding of segregating sites that emerge during the simulation due to mutation

printing sequences to output files, we will represent each sequence as an ordered `bitset` of 0s and 1s—ancestral and derived alleles, respectively—for each polymorphic site in the population (Fig. 3.5, right column). This representation is particularly useful for rapid calculation of quantities such as the number of pairwise differences between two sequences using `bitset` operators.

In addition to how we represent sequences, we need to consider additional output details. Example considerations include the following: Do we want the program to record the history of all derived alleles? Do we want to store samples (or the entire population) at specific time points? Do we want the program to calculate summary statistics at specific time points? It is important to consider these desired outputs because they will impact the structure and efficiency of the program we code. In the following sections, I demonstrate how each of these outputs may be accommodated.

3.3.2 *Simulating Point Mutation on a Sequence*

The probability that a point mutation changes the state of a specific nucleotide within a specific sequence is generally very low. For example, recent estimates of this probability for human autosomes, which is the *per-nucleotide mutation rate* ($\mu_{\texttt{nt}}$), are 10^{-8}, or one in 100 million (Lynch 2010; Roach et al 2010). Intuitively, however, the probability that a point mutation occurs *somewhere* along a sequence (*per-locus mutation rate*, $\mu_{\texttt{locus}}$) will be larger. For example, on average, one in 100 copies of a homologous, 1-Mbp sequence in a population will incur mutation somewhere along their length every generation. This is because the per-locus mutation rate equals sequence length times per-nucleotide mutation rate. Assuming $\mu_{\texttt{nt}} = 10^{-8}$, then $\mu_{\texttt{locus}} = 10^6 \times 10^{-8} = 0.01$.

Complicating matters somewhat, we must allow for the smaller probability that more than one mutation occurs in a given sequence. Technically, the number of point mutations incurred by a specific sequence is binomially distributed with the number of "trials," n, equal to the number of nucleotides in the sequence and the probability of "success," p, equal to $\mu_{\texttt{nt}}$. In our simulations of sequence variation, n is generally large and p is very small. Thus, it is safe to use the *Poisson approximation* to the binomial distribution, setting the Poisson parameter to $\lambda = np$. Random draws from a Poisson distribution so parameterized can then be used to simulate the number of mutations incurred by a given sequence. Unless the simulated sequence is very long, the result will usually be zero, and the transmitted sequence will be identical to that of the parent. When the number of mutations is greater than 0, we determine the random position(s) where the mutation(s) occurred and whether the current state of the nucleotide is ancestral or derived; see the discussion of the member function `mutate()` below for details.

3.4 Forward Simulation of Mutation and Genetic Drift

In this section, we begin our construction of the forward-in-time simulator FORTUNA. We define critical C++ classes that will grow in size throughout the book and produce a program that simulates a diploid population represented by sequences of a specified length. The program will sample sequences from the population periodically; this sample is then used to calculate summary statistics of population-level genetic variation. I will also introduce functionality to record the history of each variant/allele, including the generation of origin and the generation of either extinction or fixation. This functionality requires considerably greater computing time and need only be included in a working program if this information is necessary for

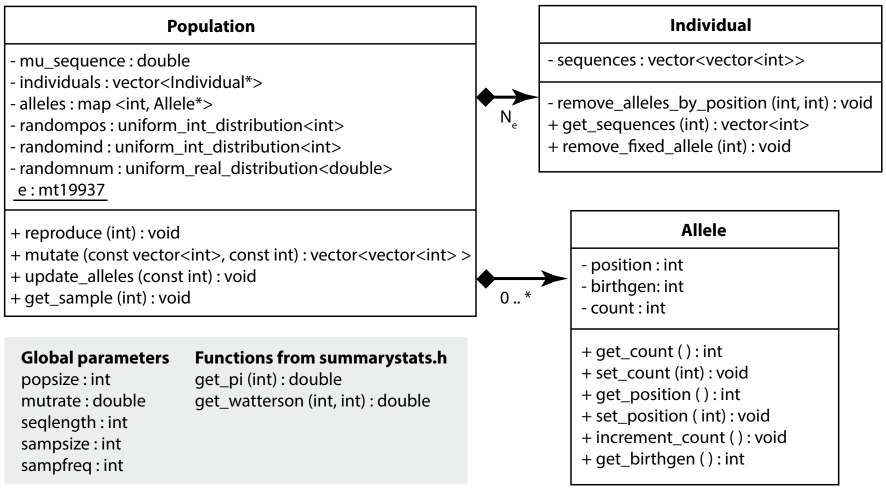

Population
- mu_sequence : double
- individuals : vector<Individual*>
- alleles : map <int, Allele*>
- randompos : uniform_int_distribution<int>
- randomind : uniform_int_distribution<int>
- randomnum : uniform_real_distribution<double>
e : mt19937
+ reproduce (int) : void
+ mutate (const vector<int>, const int) : vector<vector<int> >
+ update_alleles (const int) : void
+ get_sample (int) : void
Individual
- sequences : vector<vector<int>>
- remove_alleles_by_position (int, int) : void
+ get_sequences (int) : vector<int>
+ remove_fixed_allele (int) : void
N_e
Allele
- position : int
- birthgen: int
- count : int
+ get_count ( ) : int
+ set_count (int) : void
+ get_position ( ) : int
+ set_position (int) : void
+ increment_count ( ) : void
+ get_birthgen ( ) : int
0 .. *
Global parameters
popsize : int
mutrate : double
seqlength : int
sampsize : int
sampfreq : int
Functions from summarystats.h
get_pi (int) : double
get_watterson (int, int) : double

Fig. 3.6 Markup of the three classes critical to program FORTUNA

the research question at hand. Because the basic simulator in this chapter accounts for relatively few evolutionary factors, the parameters specified are few. The most important of these are mutation rate, effective population size (N_e), and sequence length.

The markup shown in Fig. 3.6 outlines the three classes introduced in this chapter: `Population`, `Individual`, and `Allele`. The program begins by reading in the parameter values specified in the `parameters` file. These parameters are declared in `params.h`, and values from the `parameters` file are read and initialized by `params.cc`. The main source file `fortuna_ch3.cc` instantiates an object of class `Population`, which includes a vector of pointers to N_e diploid individuals of class `Individual` and a map of extant alleles, where the key is the position of the variant in the sequence and the value is a pointer to an allele of class `Allele`. Each individual possesses two homologous copies of the sequence, each represented as a `vector` of derived allelic positions. In addition to its position, each allele includes a variable for the generation it arose—the variable `birthgen`—and its current count in the population, variable `count`, although `birthgen` is only recorded in the case where a full accounting of alleles is required. All sequences are initially devoid of genetic variation, but once-a-generation calls to the method `Population::reproduce()`—which draws random mating pairs as well as the chromosome transmitted by each parent and checks for new mutation—gradually drive the population toward mutation-drift equilibrium. Figure 3.7 provides a summary of program execution.

```
INITIALIZE global parameter values
OPEN output files
INSTANTIATE starting population of size $N_e$
FOR gen generations
     FOR $N_e$
          reproduce(): create Individual objects of the next generation by sampling
                       current generation
               mutate(): create new Allele objects and change transmitted sequences
               IF sampling generation
                    update_alleles(): update allele counts; delete monomorphic sites
                    get_sample(sampsize)
                         get_pi(): calculate and record nucleotide diversity
                         get_watterson(): calculate and record Watterson's $\theta$
CLOSE output files
```

Fig. 3.7 Pseudocode of program execution

3.4.1 Parameters

To use simulation as a tool for evolutionary inference, our simulation program must allow for easy manipulation of the parameter values to be simulated. In this section, we introduce the file `parameters`, which is read by FORTUNA at the beginning of program execution. After compiling FORTUNA, `parameters` can be edited and saved to run the program with different parameter values. In the context of inference, which requires us to run simulations using parameter values drawn from a prior distribution, the wrapping inference program can automate the relevant editing of `parameters`. At this stage, a small number of parameters are necessary:

parameters

```
popsize 1000
mutrate 1e-08
seqlength 2.5e05
sampsize 100
sampfreq 100
```

where `popsize` is diploid N_e, `mutrate` is the per-nucleotide mutation rate, `seqlength` is the length of the sequence to be simulated in base pairs, `sampsize` is sample size (<= `popsize`), and `sampfreq` is the frequency (in generations) at which samples are taken from the simulated population. The number of generations to be simulated is implemented as a command-line argument. Note that (1) you can change parameter values in the `parameters` file without the need to recompile the program and (2) scientific notation is written using e-notation—e.g., 2.5e05 represents 2.5×10^5.

Our next step is to write two files. The header file `params.h` declares program variables that will hold parameter values, and `params.cc` populates these variables with the values specified in `parameters`.

params.h

```
#ifndef PARAMS_H
#define PARAMS_H

// must be constant at compile time
extern const int bitlength = 1000;

// the following are defined by values in the parameters file
extern int popsize;
extern double mutrate;
extern int seqlength;
extern int sampsize;
extern int sampfreq;

#endif
```

The use of the keyword `extern` ensures that these variables are accessible by any file that includes this header file. Because we will use a `bitset` to represent samples taken during the simulation, and because the length of a `bitset` must be constant at compile time, we directly specify this length (`bitlength`) in the header file. Values larger than 1000 are required for larger values of N_e and `seqlength`. Until you get a sense for how many segregating sites are generated by your simulation, you should be liberal with this parameter value to avoid segmentation faults. If the variable `bitlength` is changed, *recompilation is required*; all other parameter values can be changed in the `parameters` file without the need for recompilation.

params.cc

```
#include "params.h" // access to declarations of global parameter values

map<string, string> read_parameters_file(const string &parameters_fn)
{
  //map<int, map<string, string> > params_by_block;
  map<string,string> params;
  ifstream paramfile(parameters_fn.c_str());
  string line;
  while(getline(paramfile, line)) {
    istringstream iss(line.c_str());
    string key, nextone, value;
    iss >> key;
    while (iss >> nextone)
      value += nextone + " ";
    params[key] = value;
  }
  return params;
}

map<string, string> parameters = read_parameters_file("parameters");

// variable names
int popsize, sampsize, seqlength, sampfreq;
```

```
24  double mutrate;
25
26  int process_parameters() {
27       popsize = atoi(parameters["popsize"].c_str());
28       mutrate = atof(parameters["mutrate"].c_str());
29      sampsize = atoi(parameters["sampsize"].c_str());
30      seqlength = atof(parameters["seqlength"].c_str());
31      sampfreq = atoi(parameters["sampfreq"].c_str());
32      return 1;
33  }
34
35  int good_parameters = process_parameters();
```

Having declared program variables in `params.h`, the next step is to populate them with the values listed in the `parameters` file, which is accomplished when `params.cc` is run (through an `#include` statement) early in the execution of the program. First, function `read_parameters_file()` is declared and defined (lines 3–18). The `map` returned by this function stores the parameter name as the *key* and the remaining line specifying the value(s) of the parameter as the *value*. The function is run on line 20, storing the returned `map` in a variable named `parameters`. After declaring the names of the variables (lines 23–24), function `process_parameters()` converts values from type `string` to the desired type (lines 26–35).

3.4.2 `main()` Function

The `main()` function specified in `fortuna_ch3.cc` has several important roles, including the following:

- Allows access to required standard library elements and other files of the program using `#include` statements
- Reads in the number of generations to be simulated from the command line
- Initializes a `Population` object
- Evolves the population for the specified number of generations by calling the `reproduce` function of the `Population` class

`fortuna_ch3.cc`

```
#include <algorithm>
#include <string>
#include <iterator>
#include <bitset>
#include <random>
#include <cmath>
#include <iostream>
#include <fstream>
```

```
 9  #include <vector>
10  #include <map>
11
12  using namespace std;
13
14  #include "params.h"
15  #include "params.cc" // inclusion causes parameter values to be read
16  #include "allele.h"
17  #include "individual.h"
18  #include "population.h"
19
20  int main(int argc, char *argv[]) {
21
22    int gens = atoi(argv[1]);
23    mt19937 engine(time(0)); //initialize the random engine
24    Population::e = engine;
25    Population pop;
26
27    // simulate for gens generations
28    for (int i =0; i < gens; i++)
29      pop.reproduce(i);
30    pop.close_output_files();
31
32    return 0;
33  }
34
35  // static variables for population class
36  mt19937 Population::e;
```

The number of generations to be simulated is read as a command-line argument on line 22. Subsequently, the `mt19937` random number engine, declared as a `static` variable for use in class `Population` on line 36, is initialized on lines 23–24. Next, an object of class `Population` named `pop` is instantiated (line 25), and the `Population` member function `reproduce()` is called `gens` times to evolve the population (lines 28–29). Finally, output files that record summary statistics and other data are closed (line 30).

3.4.3 Class Population

Class `Population` is central to the execution of FORTUNA. An instantiated `Population` object is the basic unit of evolution in simulations. It includes containers that store pointers to important sets of objects: diploid `Individuals` of the population as well as `Alleles` currently segregating in the population. To begin with, let us look at the basic `private` and `static` variables in the class as well as the `Population` constructor.

3.4.3.1 Basic Class Structure: Constructor and Private Variables

population.h

```cpp
#ifndef POPULATION_H
#define POPULATION_H

#include "params.h" // provides access to global parameter values (extern)
#include "summarystats.h" // provides access to summary statistic
    ↪ calculations/functions

class Population {

private:
double mu_sequence;
vector<Individual*> individuals;
map<int, Allele*> alleles; /// int key is position of the allele
uniform_int_distribution<int> randompos;
uniform_int_distribution<int> randomind;
uniform_real_distribution<double> randomnum;
poisson_distribution<int> randommut;
ofstream pi_file;
ofstream watterson_file;
ofstream allele_file;

public:
Population () {
  // initialize random number distributions
  mu_sequence = seqlength * mutrate;
  randompos.param(uniform_int_distribution<int>::param_type(1,seqlength));
  randomind.param(uniform_int_distribution<int>::param_type(0,popsize-1));
  randomnum.param(uniform_real_distribution<double>::param_type(0.,1.));
   randommut.param(poisson_distribution<int>::param_type(mu_sequence));

  individuals.reserve(popsize*2);
  for (int i=0; i<popsize; ++i) {
    vector<int> s1; vector<int> s2;
    vector<vector<int>> ses{s1,s2};
    individuals.push_back( new Individual(ses) );
  }

  string fname = "nucleotide_diversity";
  pi_file.open(fname.c_str());
  pi_file << "gen summary.stat" << endl;

  fname = "watterson_estimator";
  watterson_file.open(fname.c_str());
  watterson_file << "gen summary.stat" << endl;

  fname = "allele_info";
  allele_file.open(fname.c_str());
  allele_file << "position birthgen lifespan extinct.fixed" << endl;
}
49 static mt19937 e;
50 };
51
52 #endif
```

The class constructor takes no arguments and begins by calculating the per-sequence mutation rate `mu_sequence` and initializing four random number generators (lines 24–28):

- `randompos`: Used to draw random sites within the simulated sequence
- `randomind`: Used to draw random individuals in the population
- `randomnum`: Used to draw a real number on the interval [0,1]
- `randommut`: Used to draw the number of mutations incurred by a transmitted sequence

Next, the vector `individuals` is populated with N_e (`popsize`) objects of class `Individual` (lines 31–35). `individuals` (line 11) is of type `vector<Individual*>`—i.e., a `vector` of pointers to objects of class `Individual`. At the start of the simulation, the population is devoid of genetic variation; thus, the variable `alleles` (line 12) is empty, and the two sequences of each individual consist of empty `vectors` (lines 32–33). Line 34 instantiates each `Individual` object as well as the pointer in `individuals` using the keyword `new`. As we will see, new mutation will gradually increase genetic variation in the population, and eventually mutation-drift equilibrium will be attained. Later in this chapter, we will use sequence output from coalescent simulation to initialize the `Population`, which will eliminate the need for waiting to achieve equilibrium. Finally, in lines 37–47, the constructor defines and opens three files—*nucleotide_diversity*, *watterson_estimator*, and *allele_file*—that will store the summary statistics π and θ_{wat} calculated every `sampfreq` generations as specified in the `parameters` file and the details of new alleles arising by mutation. One note: this latter file should be commented out if the detailed *history* of polymorphism is not required, as it adds appreciable compute time.

3.4.3.2 Member Functions

The `public` member function `reproduce()` advances the simulation one generation at a time. `reproduce()` itself calls the `private` member functions `mutate()`, `update_alleles()`, and `get_sample()`, which we consider in turn below. As the name implies, `reproduce()` is responsible for producing the next generation of individuals. Producing each member of the next generation requires several steps:

- Choose two parents randomly from among `individuals`.
- For each parent, choose one of their two (implicitly autosomal) sequences for transmission.
- Assess transmitted sequences for mutation.

As we add complexity (and realism) to FORTUNA in succeeding chapters, additional steps will be needed. For now, however, these steps will suffice to simulate mutation and genetic drift. Note below—in the discussion of function `mutate()`—that we do not mutate a parental sequence until *after* each parental chromosome has been randomly selected for transmission. This choice is rooted in biological reality. Consider two siblings that, with probability 0.5, inherit separate *copies* of the same chromosome from one of their parents. Because the copies of this chromosome are products of independent meioses, any errors in DNA replication will be independent in each case, leading to gametes that may bear distinct mutations to the parental template. The same ordering of events will be necessary when we consider meiotic recombination in Chap. 5—namely, the products of recombination are derived *after* the parent and its chromosomes are selected randomly. If we did not do this, our simulation model would imply that all gametes of an individual result from a single meiosis.

population.h: function `reproduce()`

```
 1  public:
 2  void reproduce(int gen) {
 3
 4     for (int i=0; i< popsize; ++i) {
 5        vector<int> parents;
 6        parents.push_back(randomind(e));
 7        parents.push_back(randomind(e));
 8
 9        // create descendant of individuals parents[0] and parents[1]
10        individuals.push_back( new Individual(individuals[parents[0]],
              ↪ individuals[parents[1]], mutate(parents, gen)) );
11     }
12
13     // delete dynamically allocated individuasl of the last generation
14     for (auto iter = individuals.begin(); iter != individuals.end() -
            ↪ popsize; ++iter)
15        delete *iter;
16     // remove orphaned pointers from individuals
17     individuals.erase(individuals.begin(), individuals.end()-popsize);
18
19     // update allele counts on sample generations
20     if (gen % sampfreq == 0 )
21        update_alleles(gen);
22
23     if ( gen != 0 && gen % sampfreq == 0 ) {
24        random_shuffle(individuals.begin(), individuals.end() ) ;
25        get_sample(gen);
26     }
27  }
```

For each `individual` of the next generation, lines 4–11 of `reproduce()` carry out the random choice of parents using the keyword `new` to instantiate an `Individual` object. However, an alternative, intra-simulation constructor is used to instantiate the `new` individuals (lines 39–51 of `individual.h`; Sect. 4.4.4), which requires us to pass three arguments: the pointers to parents 1 and 2 as well as a call to `mutate()`, which will determine the positions of any mutations incurred by the transmitted sequences from parents 1 and 2 (line 10). New individuals of the next generation are added to the already-existing `vector individuals`.

Following instantiation of all of the next generation's individuals, the vector `individuals` holds all individuals of this *and* the next generation. Thus, individuals of the previous generation are deleted, freeing the memory they used (lines 14–15), and individuals of the previous generation are erased from `individuals`, returning the size of the `vector` to population size N_e (line 17). *If* the next generation is a sampling generation (i.e., `samplefreq` is a factor of `gen`), the function `update_alleles()` is called (lines 20–21). In addition, if it is not the first generation, a sample of the population's genetic variation will be recorded (lines 23–26) in terms of the summary statistics calculated by the function `get_sample()` detailed in Sect. 3.4.6. Note that before `get_sample()` is called, the `individuals` vector is randomly shuffled (line 24) so that the first `sampsize` individuals in the vector represent a random sample of individuals.

`population.h`: function `mutate()`

```
1  private:
2  vector<vector<int> > mutate(const vector<int> &parents, const int &gen) {
3     vector<vector<int> > mutation_results;
4  
5     // determine which, if any, positions are mutated
6     vector<int> mutnum{randommut(e)};
7     mutnum.push_back(randommut(e));
8  
9     mutation_results.push_back({mutnum[0]});
10    mutation_results.push_back({mutnum[1]});
11 
12    // determine which of the two homologs is transmitted by each parent
13    mutation_results.push_back({(randomnum(e)<0.5) ? 0 : 1});
14    mutation_results.push_back({(randomnum(e)<0.5) ? 0 : 1});
15 
16    // resolve any mutation(s) that did occur
17    for (int i=0; i<2; ++i) {
18       for (int j = 0; j < mutnum[i]; ++j) { // loop not entered if no
             ↪ mutation (i.e., mutnum[i] == 0)
19          int position = randompos(e);
20          if (alleles.find(position) == alleles.end()) { // new mutation to
                ↪ a derived allele in the population
21             alleles.insert({position, new Allele(position, gen)});
22             mutation_results[i].push_back(position);
23          } else { // mutation present in POPULATION; determine if derived
                ↪ allele found in the considered sequence
24             vector<int> seq =
                   ↪ (*(individuals[parents[i]])).get_sequence(mutation_
25                    results[i+2][0])
26          vector<int>::iterator p = find(seq.begin(), seq.end(),
               ↪ position);
27          if (p != seq.end()) // back mutation
28            mutation_results[i].push_back(position * -1); // negative
                  ↪ position signals removal of allele by back mutation
29          else
30            mutation_results[i].push_back(position);
31        }
32      }
33    }
34    return mutation_results;
35  }
```

As mentioned, the function `reproduce()` calls the function `mutate()` as an argument to the intra-simulation constructor of class `Individual`. The results of `mutate()` are evaluated and ultimately passed to this constructor. The function `mutate()` itself requires (1) a vector of two `int`s, which are the positions of the parent individuals in the `individuals` vector, and (2) the current generation. The variable `mutation_results` (line 3), a `vector` of four `vector<int>`s, will hold the results of mutation. The first two `vectors` record details of mutation(s) to the sequences transmitted by `parents[0]` and `parents[1]`, respectively. In each case, the first number of the vector is the number of mutations incurred by the transmitted sequence as determined by draws from the Poisson random number generator `mutnum` (lines 6–10). Lines 13–14 determine which one of each parent's two sequences is transmitted; the results of this determination are the sole entries in the last two `vectors` of `mutation_results`. The nested `for` loops on lines 16–32 are used to resolve several important aspects of any mutations incurred. First, the position of each new mutation within the sequence is determined on line 19, and the standard library algorithm `find()` is used to determine whether or not a derived allele already exists at that position as an `Allele` object (line 20). If not, a new `Allele` object is created (line 21), and its position is pushed to the `[0]` or `[1]` position of `mutation_results` (line 22). Otherwise, lines 23–29 determine whether the transmitted sequence has an ancestral or derived allele at that position. If the allele is *ancestral*, the position is appended to the `[0]` or `[1]` position of `mutation_results` (lines 28–29); if the allele is *derived*, back mutation is indicated by adding the *negative* position to the back of the `[0]` or `[1]` position of `mutation_results` (lines 26–27). Recall that the results of `mutate()` are passed to the `Individual` intra-simulation constructor. Specifically, the four-`vector` `mutation_results` returned by function `mutate()` will be interpreted during instantiation of the new `Individual` object (lines 39–51 in `individual.h`; Sect. 4.4.4).

population.h: function update_alleles()

```
private:
void update_alleles(const int &gen) {
  // reset all counts to zero
  for (auto iter = alleles.begin(); iter != alleles.end(); ++iter)
    (*(iter->second)).set_count(0);
  map<int, int> new_allele_counts;
  for (auto iter = individuals.begin(); iter != individuals.end();
      ↪ ++iter) {
    for (int i=0; i<2; ++i) {
      const vector<int> &a = (**iter).get_seq(i);
      for (auto iter2 = a.begin(); iter2 != a.end(); ++iter2) {
        ++new_allele_counts[*iter2];
        (*alleles[*iter2]).increment_count(); // NOTE: alleles[*iter]
            ↪ returns a reference (mapped_type) to the pointer to the
            ↪ Allele object at position *iter2. The leading *
            ↪ dereferences the pointer, granting access to the data.
      }
    }
  }

  for (auto iter3 = new_allele_counts.begin(); iter3 !=
      ↪ new_allele_counts.end(); ++iter3)
    (*alleles[iter3->first]).set_count(iter3->second);

  // identify lost and fixed alleles and print to allele history file
  vector<int> to_remove;
  for (auto iter = alleles.begin(); iter != alleles.end(); ++iter) {
    int current_count = (*(iter->second)).get_count();
    if (current_count == 0) { // allele LOST from population
      to_remove.push_back(iter->first); // first is position
      int birthgen = (*(iter->second)).get_birthgen();
      allele_file << iter->first << " " << birthgen << " " << gen -
          ↪ birthgen << " 0" << endl;
    }
    if (current_count == popsize*2) { // derived allele FIXED in
        ↪ population
      to_remove.push_back(iter->first);
      int birthgen = (*(iter->second)).get_birthgen();
      for (auto iter2 = individuals.begin(); iter2 !=
          ↪ individuals.end(); ++iter2)
        (**iter2).remove_fixed_allele(iter->first); // removed fixed
            ↪ allele's position from all individuals' sequences
            ↪ (currently stored in nextgen)
      allele_file << iter->first << " " << birthgen << " " << gen -
          ↪ birthgen << " 1" << endl;
    }
  }

  // free memory associated w/ lost/fixed alleles and remove entry from
      ↪ alleles container
  for (auto iter = to_remove.begin(); iter != to_remove.end(); ++iter) {
    delete alleles[*iter]; // free memory from Allele object itself
    alleles.erase(*iter); // erase alleles map entry corresponding to
        ↪ the deleted allele
  }
}
```

The function update_alleles() performs the important tasks of

- Calculating the current counts of all derived alleles (lines 3–16)
- Identifying allele objects no longer segregating in the population, either because they were lost from the population or reached fixation in the population (lines 21–36)
- Deleting these objects as well as erasing pointers to these objects in the vector alleles (lines 39–42)

This function is computationally expensive. Therefore, we only invoke the function (via reproduce()) in generations we intend to harvest a sample of genetic variation from the simulation. However, if you wish to collect a comprehensive list of derived alleles that emerge during the simulation, you must set samplefreq equal to 1 so that the lifespan of alleles is properly calculated. If this is not necessary for your purposes, however, you should set samplefreq to something substantially greater than 1 and ignore the output recorded in the allele_info file.

population.h: functions get_sample() and close_output_files()

```
private:
void get_sample(int gen) {
  vector<bitset<bitlength>> sample;
  map<int, int> allele_counts; // note that a map is always sorted by keys
  int count = 0;
  for (auto iter = individuals.begin(); iter !=
      ↪ individuals.begin()+sampsize; ++iter) { // determines which
      ↪ alleles are present in sample
    vector<int> haplotype = (**iter).get_sequence(0);
    for (auto iter2 = haplotype.begin(); iter2 != haplotype.end();
        ↪ ++iter2)
      ++allele_counts[*iter2];
  }
  for (auto iter = individuals.begin(); iter !=
      ↪ individuals.begin()+sampsize; ++iter) { // creates haplotypes
      ↪ for each sequence in the sample and populates bitset
    vector<int> haplotype = (**iter).get_sequence(0);
    sort(haplotype.begin(), haplotype.end());
    string hap;
    for (auto iter = allele_counts.begin(); iter != allele_counts.end();
        ↪ ++iter)
      if ( binary_search (haplotype.begin(), haplotype.end(),
          ↪ iter->first))
        hap += "1";
      else
        hap += "0";
    sample.push_back(bitset<bitlength> (hap));
  }
  int S = allele_counts.size();
   pi_file << gen << " " << get_pi(sample) << endl;
   watterson_file << gen << " " << get_watterson(sample, S) << endl;
}
```

```
26
27 public:
28 void close_output_files () {
29   watterson_file.close();
30   pi_file.close();
31   allele_file.close();
32 }
```

Each generation a sample is required, the function `get_sample()` is called from within `reproduce()` to calculate summary statistics of interest on the random sample of `individuals` using the functions provided in `summarystats.h` (Sect. 3.4.6). In this chapter, we limit ourselves to calculating two estimators of the population mutation rate $\theta = 4N_e\mu$: nucleotide diversity (π) and Watterson's estimator of θ, θ_W. The `for` loop listed in lines 6–10 stores the sequences of the `sampsize` individuals in a `vector` named `haplotype`. Recall that before calling `get_sample()`, the `individuals` vector was shuffled so that we can simply use the first `sampsize` individuals in the vector as a random sample of individuals. Further, recall that sequences are stored as vectors of `int`s, where each integer is the position index of a nucleotide where the `individuals`'s state is derived. Lines 8–9 populate the `map` `allele_counts` with counts of alleles found in the sample; thus, `allele_counts` ends up holding the index of any derived allele found at least once in the sample and excludes any alleles that are segregating in the *population* but not found in the *sample* of sequences. Lines 11–21 reconstruct each sequence of the sample as a `bitset` of 0s and 1s and add this bitset to the vector of bitsets, `sample` (declared on line 3). Finally, the number of segregating sites, S, is calculated, and calls to `get_pi()` and `get_watterson()` calculate and print π and θ_W to separate files (lines 21–24). See Sect. 3.4.6 for details of how these summary statistics are calculated.

The function `close_output_files()` is called by `main()` following completion of the simulation. This function simply closes the output streams that record summary statistics and allele information (lines 29–31).

3.4.4 Class Individual

Objects of class `Individual` are primarily defined by the two copies of the simulated homologous sequence they store in the variable `sequences`. In addition, the class provides the `public` functions `get_sequences()` (line 17) and `remove_fixed_alleles()` (lines 10–14). The former (line 17) returns the specified sequence (indexed by 0 or 1), while the latter removes fixed, derived alleles from both sequences of the individual—which is called from the `update_alleles()` function of class `Population` and removes the position number of a recently fixed allele from the sequences if present in the individual.

Two constructors are listed. The "generation 0" constructor (lines 27–29) is used at the beginning of the simulation to produce an individual with *empty* sequences. The "intra-simulation" constructor (lines 32–43) takes two `Individual` objects as well as the results of the `mutate()` function as arguments and initializes the offspring `Individuals` using this information. Note that within this constructor, back mutations (from derived to ancestral allele; identified by a negative position number) are dealt with by calling the `private` function `remove_allele_by_position()` (lines 10–14), which removes the position of the derived allele from the appropriate sequence.

individual.h

```
 1  #ifndef INDIVIDUAL_H
 2  #define INDIVIDUAL_H
 3
 4  #include "allele.h"
 5
 6  class Individual {
 7
 8  private:
 9     vector<vector<int>> sequences;
10     void remove_allele_by_position (int seqnum, int position) {
11        auto pos = find(sequences[seqnum].begin(), sequences[seqnum].end(),
              ↪ position);
12        if (pos != sequences[seqnum].end()) // ensures position in vector
13           sequences[seqnum].erase(pos);
14     }
15
16  public:
17     inline vector<int> get_sequence(int whichseq) { return
            ↪ sequences[whichseq]; }
18     void remove_fixed_allele(int to_remove) {
19        for (int i = 0; i<2; ++i) {
20           vector<int>::iterator p = find(sequences[i].begin(),
                  ↪ sequences[i].end(), to_remove);
21           if (p != sequences[i].end()) // i.e., element not found
22                 sequences[i].erase(p);
23         }
24     }
25
26     // generation 0 constructor
27     Individual (vector<vector<int>> seqs): sequences(seqs) {
28           ;
29     }
30
31     // intra-simulation constructor
32     Individual (Individual *p1, Individual *p2, vector<vector<int> >
            ↪ mutation_results) {
33        sequences.push_back((*p1).get_sequence(mutation_results[2][0]));
34        sequences.push_back((*p2).get_sequence(mutation_results[3][0]));
35        for (int i=0; i<2; ++i) {
36           for (int j=1; j<mutation_results[i].size(); ++j) {
37              if (mutation_results[i][j] > 0)
```

```
          sequences[i].push_back(mutation_results[i][j]);
        else
          remove_allele_by_position(i, -1 * mutation_results[i][j]);
      }
    }
  }
};

#endif
```

3.4.5 *Class Allele*

Objects of class `Allele` store information on derived alleles resulting from mutation. **I am using *allele* in the sense of a single nucleotide variant, not the more extensive sense of a unique gene *sequence*.** In most cases, we are only interested in the `position` and current `count` of a given derived allele. However, the generation in which a derived allele is first created is also recorded in the `private` variable `birthgen`. This information can be used for more detailed analysis of mutational output and will be used to validate the correctness of the mutation process in Section 3.6.

allele.h

```
#ifndef ALLELE_H
#define ALLELE_H

class Allele {

private:
  int position;
  int birthgen;
  int count;

public:
  inline int get_count() { return count; }
  inline void set_count(int ccount) { count = ccount; }
  inline int get_position() { return position; }
  inline void set_position(int pposition) { position = pposition; cout <<
      ↪ pposition << ": " << position << endl; }
  inline void increment_count() { ++count; }
  inline int get_birthgen() { return birthgen; }

  // constructor
  Allele (int pos, int gen): position(pos), birthgen(gen) {
    count = 0;
  }

};

#endif
```

3.4.6 Summarystats Header File

The file summarystats.h provides functions for calculation of various summary statistics. For now, we limit ourselves to the calculation of nucleotide diversity (π; not the mathematical constant) and Watterson's θ (θ_W). In subsequent chapters, we will add additional functions to calculate a diversity of summary statistics. The functions in this file are made available to programs through the #include summarystats.h statement in population.h; the functions in summarystats.h are invoked by the get_sample() function of Population objects.

summarystats.h

```
#ifndef SUMMARYSTATS_H
#define SUMMARYSTATS_H

#include "params.h"

// declarations
double get_pi (vector<bitset<bitlength>> &sample);
double get_watterson (vector<bitset<bitlength>> &sample, int S);

// definitions
double get_pi (vector<bitset<bitlength>> &sample) { // pass by reference
   double sumdiffs = 0.;
   double numcomp = 0.;

   #pragma omp parallel for collapse(2)
   for (int i = 0; i< sample.size() - 1; ++i) {
      for (int j = i+1; j<sample.size(); ++j) {
         sumdiffs += (sample[i] ^ sample[j]).count();
         numcomp+=1.;
      }
   }
   return (sumdiffs/numcomp/seqlength);
}

double get_watterson (vector<bitset<bitlength>> &sample, int S) { // pass
    ↪ by reference
   double denominator = 0.;
   for (double i=1.; i<sampsize; ++i)
      denominator += 1./i;
   return (S/denominator/seqlength);
}

#endif
```

3.4.6.1 Nucleotide Diversity, π

In words, π is the average number of differences between a pair of sequences in the sample. The number of unique sequence pairs in a sample is $\binom{n}{2}$, where n is the number of sequences in the sample—i.e., the value of parameter `samplesize`. Nucleotide diversity is calculated as

$$\pi = \binom{n}{2}^{-1} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij}, \tag{3.1}$$

where d_{ij} is the number of differences between sequences i and j. For example, if sequence i represented as a `bitset` is `0101001` and sequence j is `1101101`, then $d_{ij} = 2$. The number of unique pairwise comparisons is large for realistic sample sizes—e.g., 4950 for $n = 100$ and 19,900 for $n = 200$. One way to increase the efficiency of these comparisons is to take advantage of the multicore architecture characteristic of modern processors. In the function `get_pi()` (lines 11–23), the `pragma` on line 15 signals a nested `for` loop that assigns subsets of the total number of comparisons to parallel calculations on separate available cores. Finally, note that the return value of `get_pi()` is divided by `seqlength` (line 22); in other words, the *per-site* rather than the *sequence-wide* value of π is reported.

3.4.6.2 Watterson's θ, θ_W

At mutation-drift equilibrium, both π and θ_W serve as estimators of the population mutation parameter $\theta = 4N_e\mu$. However, the estimator θ_W is based on a very different summary of the genetic variation inherent to the sample of sequences: the number of segregating (variable) sites in the alignment of sampled sequences, S. Importantly, even if only one of the n sequences shows a different nucleotide than the other $n-1$ sequences at a site (what is commonly referred to as a *singleton*), the site counts toward the total number of segregating sites. S is therefore a rather coarse summary of genetic variation as both a singleton and a site where the frequency of the derived allele is 0.5 add one to the sum S. Furthermore, θ_W is not simply equal to S because sample size influences S. Clearly, larger sample sizes provide greater opportunity to discover segregating sites for which the minor allele frequency is low. The definition of θ_W is therefore normalized by sample size:

$$\theta_W = \frac{S}{a_n}, \tag{3.2}$$

where $a_n = \sum_{i=1}^{n-1} \frac{1}{i}$, which is calculated in the function `get_watterson()` (lines 25–30) and stored in the variable `denominator` (line 26). Note that the return value of `get_watterson()` is also divided by `seqlength` (line 29); the function thus returns the *per-site* value of θ_W.

3.4.6.3 The Inferential Merit of π and θ_W

Although we expect $\pi = \theta_W$ in an equilibrium population, perturbations to equilibrium such as demographic change or various forms of natural selection affect the two summary statistics differently. Consider a scenario in which a large population is drastically reduced in size due to environmental catastrophe. This population bottleneck will reduce the quantity of genetic variation in the population, thereby lowering both π and θ_W. If the remnant population then expands over time, π and θ_W will increase at *different* rates due to the distinct ways in which these statistics are calculated. Specifically, θ_W will increase more rapidly than π because there will be an enrichment of rare alleles resulting from new mutation. While rare alleles stand an appreciable chance of being captured by a sample, each increasing S by one, these rare alleles will impact π—the *average* number of differences between sampled sequences—very little. Comparisons of π and θ_W can therefore be used to test the null hypothesis of equilibrium ($\pi = \theta_W$) and potentially allow us to infer the cause of non-equilibrium when the null hypothesis is rejected. For example, a large value of $\theta_W - \pi$ may be interpreted as evidence of recent population expansion. As discussed in subsequent chapters, somewhat more sophisticated test statistics such as Tajima's D (Tajima 1989) are used to facilitate inferences such as this.

3.5 Aspects of Genetics, Population Biology, and Environment Not Yet Modeled

It is important to take a moment and consider biological realities that our current model (and simulation program) of the evolution of population genetic variation does not include. At the genetic level, we do not include meiotic recombination. The process and consequences of recombination will be introduced in Chap. 5. In addition, the only form of mutation considered is point mutation; although point mutations remain the focus throughout this volume, my intent is to add additional code at `driftlessevolution.com` that will allow simulation of other categories of mutation (e.g., microsatellites and indels).

Note that the simple version of FORTUNA detailed in this chapter assumes *neutral* evolution. In other words, the genetic variants introduced by point mutation neither increase nor decrease the fitness of individuals who carry them. We wait until Chaps. 7–8 to consider natural selection and its effects on linked sequence variation.

At the level of the population, there are several potential complexities not addressed by the current model. First, all matings produce a single offspring. Second, generations are nonoverlapping; all individuals are assumed to reproduce at the same time. Third, we assume a single panmictic population and ignore the sex of potential parents. Although the current model precludes selfing, all potential mating pairs between separate individuals are equally likely. This also means that population structure is not modelled. To do so, we must create separate objects of class `Population` and specify the rate of migration between these populations (see Chap. 6 for details). Fourth, we do not consider nonrandom mating in any of its forms, such as inbreeding, positive assortative mating, or negative assortative mating. Fifth, population size is constant; bottlenecks, expansions, and founder events are not considered. All of these additional complexities and more will be considered in subsequent chapters.

Thus, we begin simply and add additional complexities incrementally. I again emphasize that what we sacrifice in terms of computational efficiency when choosing a forward-in-time simulation is balanced by our ability to simulate the complexities mentioned in a relatively straightforward and intuitive manner. If it were possible to incorporate all of these complexities in a retrospective, coalescent simulation, there would be no upside to forward simulation.

3.6 Validation: Comparing Simulation Output with Theoretical Expectations

If a simulation program compiles, runs without crashing, and generates output, this does not ensure that the simulation has simulated what we think it did. Whenever possible, it is therefore important to compare simulated data to theoretical expectations. In the current case—simulation of a Wright-Fisher population with constant population size, no recombination, all neutral variants, and random mating—there are a number of comparisons we can make to reassure ourselves that the simulation program is working properly. First, both π and θ_w are expected to equal $\theta = 4N_e\mu$. Second, the probability that a new variant ultimately fixes in the population equals its initial frequency of $1/2N_e$. Third, for those alleles that do fix, mean time to fixation is expected to be $4N_e$. All three of these expectations are calculable because our simulation program requires us to provide explicit values of N_e and μ.

We will use R to compare simulation results to theoretical expectations. First, I read the two data files that contain the running calculations of π and θ_w each `samplefreq` generations. We then plot π and θ_w versus time and compare these values to the theoretical expectation (Fig. 3.8).

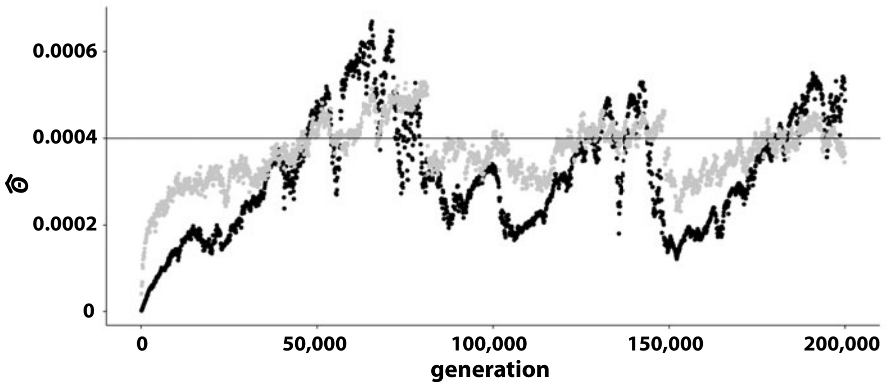


Fig. 3.8 Values of nucleotide diversity (black) and Watterson's θ (gray) sampled every 100 generations (n = 400) for 200,000 generations. Note that because we begin with no genetic variation in the simulated 250,000-bp sequence, time is required for mutation to add variation. For the parameter values simulated (`popsize`=10,000, `mutrate`=10e-8), the expected value of $\theta = 4 \times N_e \times \mu = 4 \times 10{,}000 \times 10^{-8} = 4 \times 10^{-4}$, which is denoted by a solid horizontal line

```
> pi = read.table(file = "nucleotide_diversity", header = T)
> wat = read.table(file = "watterson_estimator", header = T)
> ggplot(NULL, aes(gen, summary.stat)) + geom_point(data=pi) +
    ↪ geom_point(data=wat, colour="gray") + geom_hline(yintercept=2e-06)
```

Regarding Fig. 3.8, there are two points worth noting here. First, π and θ_w track each other as we would expect with roughly equal mean values, but π demonstrates noticeably greater variance and occasional, dramatic swings in value. A good example is found at roughly 135,000 generations, where π decreases sharply and θ_W remains roughly static around its expected value. The different behaviors of these two summary statistics is central to the statistic Tajima's D, which we will revisit when discussing demographic change and the inference of natural selection in subsequent chapters. Even in the neutral case, however, it is clear these two summary statistics respond at different rates to changes in levels of genetic variation over time. Second, the expected value of θ for this simulation is 4×10^{-4}, as denoted by the horizontal bar. Because we start the simulation with a population devoid of variation, it takes some time for the population to reach this mutation-drift equilibrium value of θ. It therefore makes sense to discard the results from early generations of the simulation before we check whether the data meet theoretical expectations. Note, however, that even after the population has obtained sufficient genetic variation ($\approx$ 50,000 generations), values of these two θ estimators still oscillate about the expected value of 0.0004. In other words, the equilibrium value is just that; we expect the *mean* value of θ to be 4×10^{-4} over long periods of time.

I next ran a 200,000-generation simulation in which per-nucleotide mutation rate was 10^{-9}, population size was 500, and simulated sequence length was 5 Mbp. Expected $\theta = 4 \times 500 \times 10^{-9} = 2 \times 10^{-6}$. In R, I then loaded objects `pi` and `wat` as before and ran the following code.

```
1  > pi.trunc = pi[pi$gen>50000,] // burn-in of 50,000 generations
2  > wat.trunc = wat[wat$gen>50000,]
3  > mean(pi.trunc$summary.stat) // find mean value of pi over the remaining
       ↪ 150,000 generations
4  [1] 1.99867e-06
5  > mean(wat.trunc$summary.stat)
6  [1] 2.000395e-06
```

Lines 1–2 enforce a burn-in period during which time the population is acquiring genetic variation and not yet at equilibrium. As the results in lines 4 and 6 show, both estimators of θ conform nicely to the theoretical expectation of $\theta = 4N_e\mu = 2 \times 10^{-6}$.

The `update_alleles()` function reports the lifespan of each derived allele—i.e., the difference between `birthgen` and the generation in which the derived allele fixes or is lost, which quantifies the number of generations for which the derived allele was segregating in the population. In addition to the position and lifespan of a derived allele, the file `allele_info` records a column named `extinct.fixed` that holds a zero if the derived allele was lost or a one if the derived allele fixed. As mentioned previously, proper calculation of the lifespan of a derived allele requires that `update_alleles()` be run every generation, which happens if the parameter `samplefreq` is set to 1. This adds substantially to the execution time of the program. However, it is valuable for certain purposes, including validation of the underlying code as we now see. My hope is that the results shown below will provide you with sufficient confidence that the program is operating correctly, obviating the need for you to run a simulation of unnecessarily great execution time.

Let us look at an example. For the sake of continuity, I only use alleles "birthed" in the last 150,000 generations of the same simulation analyzed in the previous listing of R code.

```
1  > a = read.table(file = "allele_history", header = T)
2  > a.trunc = a[a$birthgen>50000,] // burn-in of 50,000 generations
3  > a.trunc.fixed = a.trunc[a.trunc$extinct.fixed == 1,] // select alleles
       ↪ that fixed
4  > nrow(a.trunc.fixed) / nrow(a.trunc) // nrow gives number of alleles in
       ↪ each data.frame
5  [1] 0.0009833301 // proportion of new alleles that fixed
6  > mean(a.trunc.fixed$lifespan) //
7  [1] 1955.18 // mean time to fixation
```

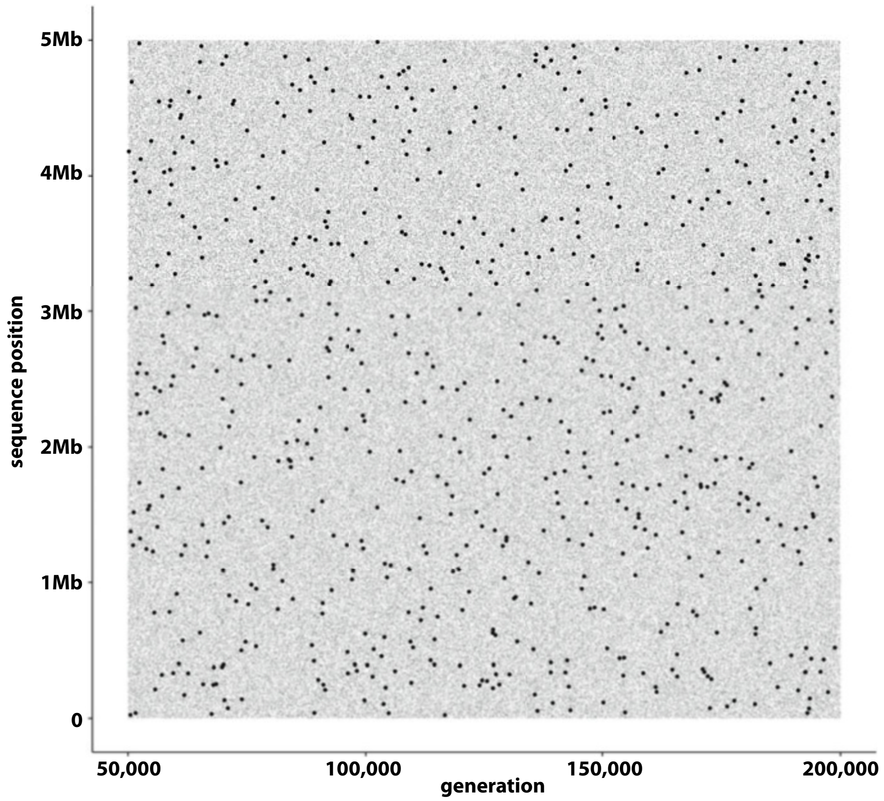


Fig. 3.9 Position of each new mutation versus generation. Nearly 750,000 new mutations were generated within the 150,000 generations shown (gray and black dots). Although the vast majority of these new mutations were rapidly lost from the population (the gray dots that are so frequent they appear as a background square of gray), a small number of new mutations became fixed (black dots). By sight, the position of new mutations and those destined for fixation appears evenly distributed

These results are in keeping with theoretical expectations. Namely, the expected fixation probability of a new allele equals $1/2N_e = 1/(2\times500) = 0.001 \approx 0.00098$ and the expected mean time to fixation equals $4N_e = 4\times500 = 2000 \approx 1955$.

Finally, it is worth checking that random number generation is behaving in a suitably random manner. One check is to plot sequence position of mutations versus time. If random number generation is working properly, we expect a random cloud of points. In other words, we do not expect mutational hot or cold spots. In our example, this expectation is thankfully met, and it is clear that alleles destined for fixation are also evenly distributed by sequence position (Fig. 3.9).

3.7 Avoiding the Burn: Coalescent Simulation Followed by Forward Simulation

As seen in Fig. 3.8, a pure forward simulation requires beginning with no variation in the population and thousands of generations of simulation before mutation-drift equilibrium is achieved. In consequence, we waste computation time and, for most analyses of the simulated data, must specify a burn-in time that avoids calculation on pre-equilibrium generations. The time required to achieve mutation-drift equilibrium increases dramatically with increases in population size and/or mutation rate. Thankfully, we can usually avoid these two related problems by first running a rapid coalescent simulation and then using the output of the coalescent simulation to establish the first generation in our forward simulation. This allows us to begin our forward simulation with a population that is already at mutation-drift equilibrium. A standard program for performing coalescent simulations is MS (Hudson 2002). In the following discussion, I assume the compiled MS executable is located in the same directory as the compiled FORTUNA program.

To facilitate this new functionality, two additional parameters are added to the `parameters` file: `useMS`, which specifies whether the program should begin by calling MS, and `mscommand`, which specifies the command to be run in MS. We will assume $N_e = 50{,}000$ diploids, $\mu_{per-site} = 5 \times 10^{-9}$, and a 100,000-bp sequence. Our goal is to simulate 100,000 sequences (the total number of sequences in the population) at mutation-drift equilibrium. The only parameter for the MS command that needs to be calculated is per-locus θ, which is $4 \times N_e \times \mu_{per-site} \times$ `seqlen`:

$$\theta = 4 \times 50{,}000 \times 5 \times 10^{-9} \times 1 \times 100{,}000 = 100 \tag{3.3}$$

Given our need for 100,000 simulated sequences and $\theta = 100$, the mscommand is therefore `./ms 100000 1 -t 100 >ms_output`. Note that output from MS is sent to a file named `ms_output`, which will be parsed for its sequences that are in turn input for an expanded `Population` class constructor (detailed below).

We next take a look at the minor additions to the files `parameters`, `params.h`, `params.cc`, and `mutdrift_forward.cc`:

preemptive coalescent simulation: minor modifications to program files

```
// additions to parameters
useMS 1
mscommand ./ms 100000 1 -t 100 >ms_output

// additions to params.h
extern bool useMS;
extern string mscommand;

```

```
// additions to params.cc
bool useMS;
string mscommand;

int process_parameters() {
    ...
    useMS = atoi(parameters["useMS"].c_str());
    mscommand = parameters["mscommand"];
    ...
}

// addition to fortuna_ch3.cc
#include <sstream> // stringstream parsing of ms_output
#include <regex> // enables parsing of ms_output
```

Setting the Boolean `useMS` to 1 impels FORTUNA to run `mscommand`, parse the `ms_output` file, and use the output genetic sequences as the starting point for forward simulation. Setting `useMS` to 0, on the other hand, means that all sequences will, as before, lack genetic variation at the beginning of the simulation.

Next, we look at a more significant change to the constructor of `population.h`, which will parse `ms_output` and use the sequence data to populate the Population class at generation 0. Again, the advantage to this is that from the very first generation of forward simulation, we have sequence data at mutation-drift equilibrium.

population.h: modified constructor

```
if (useMS) { // start population with MS generated variation
   cout << "using MS to initialize population ..." << endl;
   system(mscommand.c_str());
   ifstream ms_output("ms_output");
   string ms_line;
   regex query("positions");
   bool trigger = false;
   vector<int> allele_positions;
   while(getline(ms_output, ms_line)) {
      if (regex_search(ms_line, query)) {
         trigger = true;
         istringstream iss(ms_line);
         string s;
         iss >> s; //skip the first subpart, which is "positions:"
         while (iss >> s) { // read decimal positions,
                                 // convert to base pair position,
                  // and create new allele at that position
            int position = seqlength * atof(s.c_str());
            allele_positions.push_back(position);
            alleles.insert( { position , new Allele(position,-1) } );
         }
         continue;
      }

25      if (trigger) { // allele positions determined;
26         vector<int> s1, s2;
27         for (int i=0; i < ms_line.length(); ++i)
28            if (ms_line[i] == '1')
29               s1.push_back(allele_positions[i]);
30         getline(ms_output, ms_line);
31         for (int i=0; i < ms_line.length(); ++i)
32            if (ms_line[i] == '1')
33               s2.push_back(allele_positions[i]);
34         vector<vector<int>> ses{s1,s2};
35         individuals.push_back( new Individual(ses) );
36      }
37   }
38 } else { // constructor code from earlier listing
39      ...
40   }
41 }
```

If `useMS` is set to 1, line 1 ensures that lines 2–37 are executed rather than the constructor detailed previously (lines 38–40). Line 3 runs MS. The `while` loop (lines 9–37) first identifies the line that begins with `positions:` in the `ms_output` file and creates new alleles after converting the decimal positions to base pair positions (lines 10–23). Note that the a value of -1 is passed to the constructor of `Allele` (line 20), which indicates we do not know how many generations ago the derived allele arose because MS output does not provide this information.

Line 10 uses the <`regex`> library's functionality to search for the line that matches the string "positions." If it is found, this indicates that all subsequent lines in the `ms_output` file contain sequences (really, haplotypes). The Boolean variable `trigger` is then set to `true` (line 11) to indicate that the program has found the critical line below which the genetic data are given. Lines 25–36 of the `while` loop are executed when `trigger` is true. These lines extract the haplotype data from `ms_output` two lines at a time to create new `Individual` objects using the MS simulated haplotype data. Again, if `useMS` is set to 0, the previously detailed constructor (line 39, placeholder) will execute instead, and each individual will begin with no derived alleles.

Figure 3.10 compares π across 400,000 generations of forward simulation between the case where we begin with MS-derived data (black dots) and the case where we do not (gray dots). Clearly, coalescent simulation is helpful. If we begin with nonvariable haplotypes, $> 100,000$ generations of simulation are required to reach mutation-drift equilibrium for the parameter values simulated.

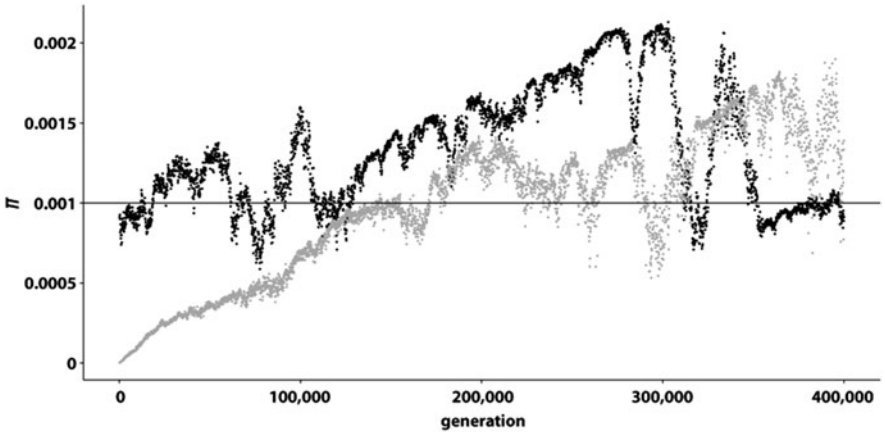


Fig. 3.10 Beginning the simulation with output from a coalescent simulation assures that the simulated population begins the forward simulation at mutation-drift equilibrium. As the figure illustrates, this approach eliminates the need for a burn-in period. The simulation beginning with output from MS (black dots) therefore begins near the expected equilibrium value of $\pi = 0.001$ (horizontal line). Indeed, the mean value of π for the first 100,000 generations equals 0.00107. The pure forward simulation (gray dots), where we begin with no genetic variation, does not reach mutation-drift equilibrium until well after 100,000 generations. Though not always possible, beginning a forward simulation from coalescent output therefore eliminates useless computation and wasted time. The parameters specified in both simulations were $N_e = 50,000$, $\mu_{per-site} = 5\times10^{-9}$, and sequence length of 100,000 bp. π was calculated every 100 generations based on a sample of 400 individuals

References

Hudson RR (2002) Generating samples under a Wright-Fisher neutral model of genetic variation. Bioinformatics 18:337–338

Kimura M (1962) On probability of fixation of mutant genes in a population. Genetics 47:713

Lynch M (2010) Rate, molecular spectrum, and consequences of human mutation. Proc Natl Acad Sci USA 107(3):961–968. https://doi.org/10.1073/pnas.0912629107

Roach JC, Glusman G, Smit AFA, Huff CD, Hubley R, Shannon PT, Rowen L, Pant KP, Goodman N, Bamshad M, Shendure J, Drmanac R, Jorde LB, Hood L, Galas DJ (2010) Analysis of genetic inheritance in a family quartet by whole-genome sequencing. Science 328(5978):636–639. https://doi.org/10.1126/science.1186802

Tajima F (1989) Statistical method for testing the neutral mutation hypothesis by dna polymorphism. Genetics 123:585–595

Wright S (1931) Evolution in mendelian populations. Genetics 16:97–159

4 Demographic Change

The workers have taken it into their heads that they, with their busy hands, are the necessary, and the rich capitalists, who do nothing, the surplus population.
– Friedrich Engels, *The Condition of the Working Class in England*

4.1 Background

In the previous chapter, we considered a population that maintains constant population size. However, we often need to simulate a population whose size is, or was, in flux. Brief consideration brings to mind many such cases. The population size of our own species has expanded dramatically, particularly over the last century; total census population of the human species grew from ~2 billion in 1927 to ~7 billion in 2011. Invasive species are primarily defined as a category of alien species by their rapid population growth following introduction to a new locality. Declines in population size are also common in the natural world. Nascent island populations—the result of immigration from mainland populations—are necessarily much smaller than the parent population. Anthropogenic changes to the environment have caused countless species throughout the world to decline in size precipitously.

Demographic change has a *genome-wide* effect on genetic variation because changes in population size change the number of individuals and therefore the number of copies of entire genomes present in a population. In other words, the number of copies of a specific locus or chromosome is not altered *to the exclusion* of most other loci/chromosomes. This contrasts sharply with natural selection, which generally only affects quantity and pattern of genetic variation in the immediate vicinity of the locus targeted by selection.

Note that my use of the term "demographic change" is limited to changes in population size. We will not explicitly model changes in birth/death rates

R. J. Haasl, *Nature in Silico*, https://doi.org/10.1007/978-3-030-97381-0_4

or the age structure of populations. In this chapter, N_t represents population size at time t, while N_{100} represents population size at generation 100. A symbological distinction is not made between census and effective population sizes, but we assume *effective* population size throughout the chapter.

4.1.1 Models of Demographic Change

Consider a population whose census size increases monotonically from 1000 individuals to 10,000 individuals over the course of 100 generations. We can imagine a number of trajectories the population might follow to bring about this increase (Fig. 4.1). The simplest are *instantaneous* population expansion in which the increase occurs in one generation and a *linear* expansion in which $(10{,}000 - 1000)/100 = 90$ individuals are added each generation. Somewhat more complicated are the widely used *exponential* and *logistic* models of population growth, which we now consider in turn.

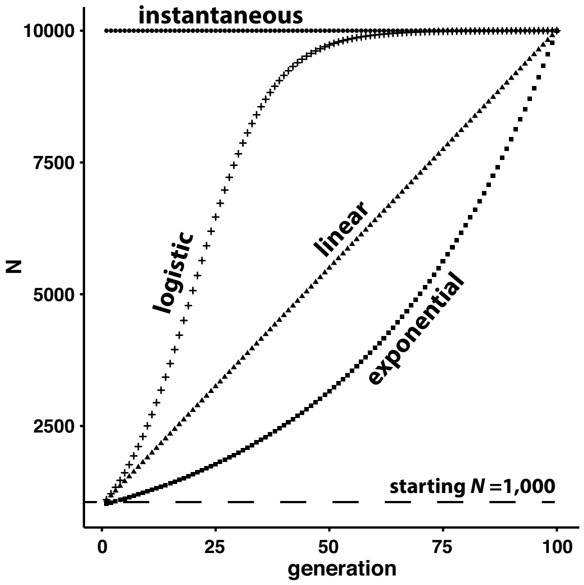


Fig. 4.1 Four models of population expansion from $N = 1000$ to $N = 10{,}000$ over the course of 100 generations. For the logistic model, the low density growth rate $r \approx 0.114$. For the exponential model, the intrinsic rate of increase $r \approx 0.023$

4.1.1.1 Exponential Population Growth or Decline

Each generation, exponential population growth increases population size by a constant fraction of the current population size. Because the base population size increases each generation, the number of individuals added each generation grows rapidly, ultimately leading to runaway population growth. Following t generations of exponential increase, beginning with a population size of N_0, population size N_t is

$$N_t = N_0 e^{rt}. \tag{4.1}$$

Given t, N_0, and N_t (i.e., parameters we are likely to specify for our simulation), we can calculate the corresponding value of r, the *intrinsic rate of increase*, using the following rearrangement of Eq. 4.1:

$$r = \ln\left(\frac{N_t}{N_0}\right)t^{-1} \tag{4.2}$$

Thus, for the example described at the beginning of this section and shown in Fig. 4.1, $r = \ln(10) \times 0.01 \approx 0.023$. Exponential population *decline* follows the same equations, but r is negative in sign. For example, the inverse scenario in which population size declines from $N_0 = 10{,}000$ to $N_{100} = 1000$ requires $r \approx -0.023$.

4.1.1.2 Logistic Population Growth

Runaway population growth is ultimately constrained by limited resources—an idea famously promulgated by Thomas Malthus. A small population may grow exponentially, but as population size approaches its *carrying capacity*, K, growth slows down. Logistic growth is therefore characterized by a sigmoidal growth curve, where growth from a small population size begins at near-exponential rates but then declines precipitously as population size approaches K. The modifying influence of K on growth rate is easily seen in the discrete logistic equation:

$$N_{t+1} = N_t + rN_t\left(1 - \frac{N_t}{K}\right). \tag{4.3}$$

When $N_t << K$, nearly rN_t individuals are added to current population size by the next generation. Under the logistic model, r is therefore sometimes named the *low density growth rate*. However, at carrying capacity ($N_t = K$), rN_t is multiplied by zero, resulting in constant population size of K. Given K and N_0, the logistic model for population size after t generations is

$$N_t = \frac{KN_0}{N_0 + (K - N_0)e^{-rt}} \tag{4.4}$$

4.1.2 *Using Coalescent Simulation to Build Intuition Regarding the Genetic Consequences of Demographic Change*

To build basic intuition regarding the effects of population increase *and* decrease on genetic variation, we use coalescent simulation implemented in the R package `coala` (Staab and Metzler 2016, see Chapter 2). To make the effects of population growth and decline as clear as possible, we will compare the genealogies and site frequency spectra of a population at mutation-drift equilibrium ($N_e = 25{,}000$ diploid individuals) to simulations of two extreme scenarios: (1) an instantaneous population expansion from $N_e = 50$ to $N_e = 25{,}000$ and (2) an instantaneous population bottleneck from $N_e = 25{,}000$ to $N_e = 50$. In both cases, the genetic sample is drawn 50 generations following the sudden change in population size. We draw samples of $n = 20$ and $n = 100$ to show the effect of demographic change on the structure of genealogy (left column panels in Fig. 4.2) and the SFS (right column panels of Fig. 4.2), respectively. A bottleneck is a biologically relevant scenario; environmental catastrophe may rapidly decimate a local population, and a small, founder population derived from a large population may in one generation establish itself elsewhere. An instantaneous population expansion of the size simulated here is less relevant from a biological/ecological standpoint. Nevertheless, its extremity is used to showcase the effects of population expansion.

In all scenarios, a point mutation rate of 1×10^{-8} and a 20,000-bp sequence were simulated. We first examine the baseline results from simulating a population at mutation-drift equilibrium. Looking backward in time, the $n = 20$ sequences of a sample *initially* coalesce with rapidity, while coalescent times (the waiting time to the merger of sequence lineages) increase as the number of lineages to coalesce decreases (Fig. 4.2a). This is characteristic of the neutral coalescent process, in which the probability of coalescence between an *unspecified* pair of lineages in the previous generation equals $\frac{n(n-1)}{4N_e}$. As n—the remaining number of genes that have not coalesced—decreases, the probability of coalescence decreases, and coalescence time increases on average.

Under the neutral coalescent, the site frequency spectrum should be distributed geometrically. However, this expectation is only met when spectra are averaged across numerous unlinked loci. Although the SFS of one simulation is not geometrically distributed, it does show hallmarks of what we expect the distribution of genetic variation to look like under neutral, equilibrium conditions (Fig. 4.2b). The most abundant types of polymorphic sites (SNPs) are those in which the frequency of derived alleles is low. In particular, the most abundant type of SNP is a *singleton* in which only one of the 100 sampled sequences shows a derived allele. The SFS for this simulation also shows a declining trend in the number of SNPs with high frequencies of the derived allele. Again, because the results are shown for a single simulation (and, therefore, a single locus), we do see some clear exceptions to this trend. For example, the numbers of SNPs with derived allele counts are remarkably common — 5 and 13, respectively (Fig. 4.2b).

Figure 4.2a and b is derived from the results of one simulation. Devise a hypothesis to explain the bimodal SFS in Fig. 4.2b given the genealogy shown in Fig. 4.2a.

Both the genealogy and SFS under instantaneous population expansion (Fig. 4.2c,d) show a clear departure from that of the equilibrium scenario. The genealogy is comb-like (Fig. 4.2c); all 20 sampled sequences evolve independently of each other until they rapidly coalesce in a compressed period of time in the recent past. The SFS is notable for (1) the small number of polymorphic sites (cf. scale of the y-axis in Fig. 4.2b and d) and (2) derived allele counts exclusively ≤ 3 out of 100 sampled sequences. We expect a drastic increase in population size to alter patterns of genetic variation, but the specific question of interest is why these particular alterations are characteristic of population expansion.

Prior to the expansion, the population of 50 diploid individuals collectively carries just 100 copies of the simulated locus. As detailed in the last chapter, small populations harbor less variation than large populations. It is possible, even likely, that the initial population was devoid of variation at the locus simulated. However, the in-one-generation increase in population size to 25,000 diploid individuals provides a large reservoir of sequences that can incur mutations. In other words, the small population with depauperate genetic variation becomes a large population with depauperate genetic variation. The simulated sequence is a (nearly) blank canvas upon which new variants may be written. However, the long terminal branches of the genealogy mean the vast majority of derived alleles found in a sample are singletons. This process of occasionally generating derived alleles within a genetically depauperate sequence yields the SFS seen in Fig. 4.2d. Polymorphic sites are rare because we have only allowed 50 generations for new mutations to arise on the "blank canvas." Moreover, those polymorphic sites that are identified in the sample show low derived allele counts because they have had little time to spread through the population.

We next consider the genesis of the distinct topology associated with a large and instantaneous population expansion: in our example, looking backward in time from the present, *no* coalescence of the $n = 20$ sequences for an extended period followed by a flurry of coalescent events in the past (Fig. 4.2c). As you might guess, this cluster of coalescent events dates back to the much smaller, pre-expansion population. The long absence of coalescence following the expansion is a direct consequence of the sudden increase in

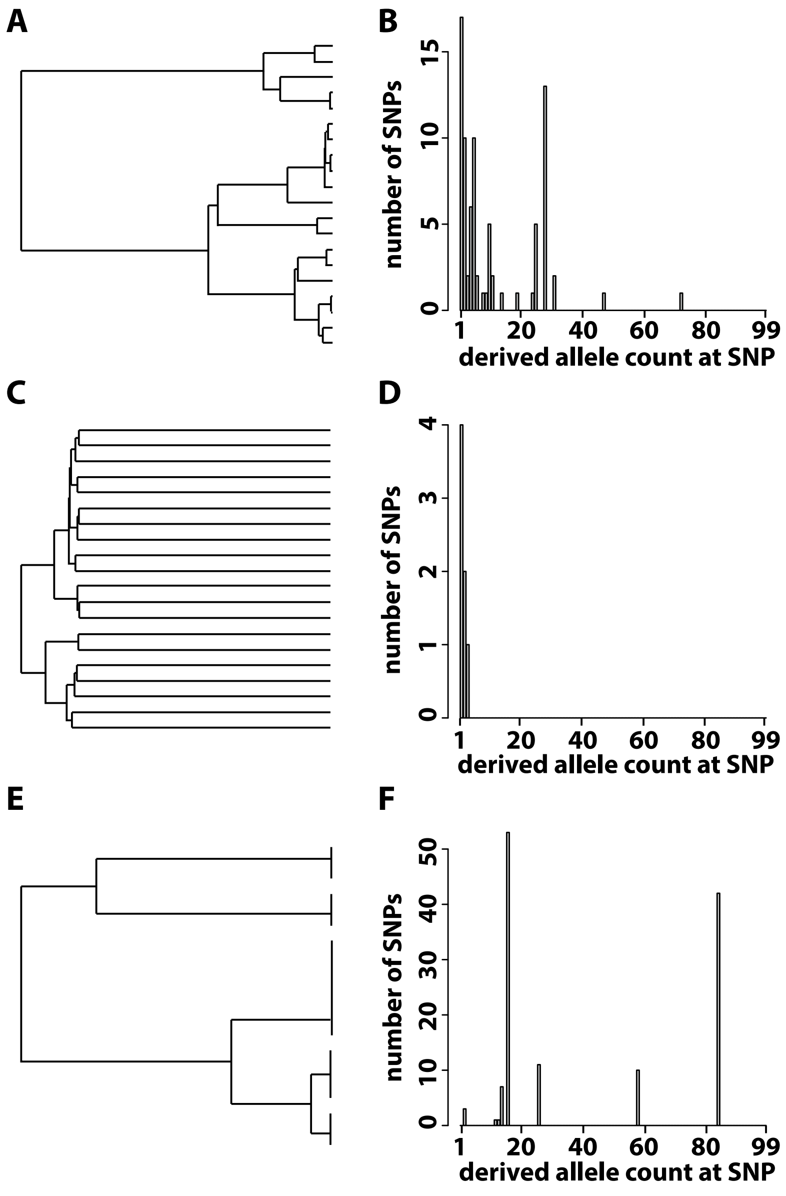


Fig. 4.2 Typical genealogies and site frequency spectra for a neutral, equilibrium model (**a**,**b**), an instantaneous population expansion (**c**,**d**), and an instantaneous population bottleneck (**e**,**f**). Although there are 20 branch tips in (**f**), coalescent events take place so rapidly moving backward in time that it appears the tips are too short to visualize. For clarity, the genealogies are based on a sample of just 20 sequences; the site frequency spectra are calculated from a larger sample of 100 sequences. Samples for the population bottleneck and expansion scenarios were drawn 50 generations following the demographic event. In the genealogies, time flows forward from left to right

the number of sequences present in the population. The probability $P(n)$ that no coalescent event takes place in the previous generation is equal to

$$P(n) = \frac{2N_e - 1}{2N_e} + \frac{2N_e - 2}{2N_e} + \ldots + \frac{2N_e - n + 1}{2N_e}, \tag{4.5}$$

where n is the number of sequences in the sample. It can be shown that $P(n) \approx 1 - \binom{n}{2}\frac{1}{2N_e}$ (Hudson 1990). In our current example, then, the probability that *none* of the lineages coalesce for the 50 generations since the expansion—with $n = 20$ and $2N_e = 50{,}000$—equals

$$P(20)^{50} \approx \left[1 - \binom{20}{2}\frac{1}{50{,}000}\right]^{50} \approx 0.827 \tag{4.6}$$

In other words, greater than 80% of the time, *none* of the $\binom{20}{2} = 190$ pairs of sequences will coalesce in the 50 generations following the expansion. Contrast this with the small, pre-expansion population of just 50 individuals ($2N_e = 100$) where the probability that one pair of sequences will coalesce with a common ancestor in the previous generation is a near certainty. Although an instantaneous, 500-fold increase in population size is an extreme example, these ideas show that a comb-like topology (perhaps, more commonly named a star topology) is characteristic of genealogies associated with population expansions.

The average effect of a population bottleneck on a genealogy is opposite to that of a population expansion (Fig. 4.2e); most coalescent events take place post-bottleneck, while the last coalescent events are often much older and traceable to the larger, *pre*-bottleneck population. The resulting SFS (Fig. 4.2f) shows isolated peaks associated with mutations occurring on the long branches that become derived alleles shared by the descendant sequences of these long branches.

Finally, we look at the summary statistics—and estimators of θ—nucleotide diversity π and Watterson's estimator θ_W for the same demographic scenarios just discussed. Our expectations are the following: (1) for the scenario of no demographic change, both estimators should zero-in on the per-locus expected value of $\theta = 20$, and (2) because both summary statistics are directly related to the site frequency spectrum and we have just seen that demographic change has profound impacts on the SFS, both summary statistics should be greatly perturbed by the strong demographic change modeled. Both of these expectations are met by this simple simulation study (Fig. 4.3). In addition, notice that under neutral, equilibrium conditions, θ_W shows superior resolution to π as an estimator of per-locus θ (Fig. 4.3a).

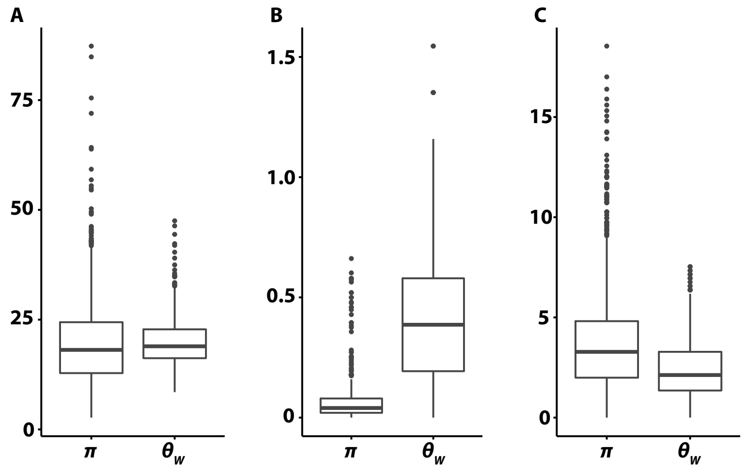


Fig. 4.3 Boxplots showing the diversity in *per-locus* values of nucleotide diversity π and θ_W across 1000 replicate simulations for each of three demographic models. Panels (**a**), (**b**), and (**c**) here correspond to the same demographic models used to produce the results shown in Fig. 4.2 panels (**a-b**), (**c-d**), and (**e-f**), respectively—namely, ((**a**)) No demographic change. ((**b**)) Instantaneous population expansion. ((**c**)) Instantaneous population bottleneck. $N_e = 25{,}000$, per-site $\mu = 1 \times 10^{-8}$, `seqlength` = 20,000 bp, and `sampsize` = 100. **Note the different y-axis scales between panels.** Samples for the population bottleneck and expansion scenarios were drawn 50 generations following the demographic event. On average, π - θ_W is zero for a neutral locus in a population of constant size, a negative value for population expansion, and a positive value for a population decline; the results shown here validate those theoretical expectations. The median estimate of per-locus θ (for *both* estimators) at a neutral locus in a population of constant size (panel A) matches the expectation of $\theta = 4N_e\mu \times seqlen == 4 \times 50{,}000 \times 10^{-8} \times 10{,}000 = 20$. Both summary statistics are therefore good estimators of θ when the atmosphere is ostensibly boring—neutral and static. **In addition, this is another example of validation in which our simulation results match theoretical expectations**

4.2 Forward Simulation of Demographic Change

We now modify the forward simulation program FORTUNA—introduced in Chap. 3—to facilitate simulation of varied demographic scenarios. Modifications also include updates to `summarystatistics.h` that enable calculation of the summary statistic Tajima's D.

4.2.1 *Requisite New Parameters*

Imagine that the modeled population is subject to a series of changes in population size. The information necessary to model each size change includes the following: (1) type, e.g., instantaneous or logistic; (2) generation numbers for the onset and conclusion of the demographic regime; (3) population size at onset of the demographic regime; and (4) any additional parameter(s) required to specify the demographic model. We first add program parameters to the `parameters` and `params.h` files.

modifications to parameters and params.h files to implement demographic change

```
 1  // additions to parameters
 2  demography 0 1 4
 3  dem_parameter 0 -9900 0.02
 4  dem_start_gen 0 1001 1501
 5  dem_end_gen 1000 1500 20000
 6  carrying_cap 0 0 10000
 7
 8  // additions to params.h
 9  extern vector<int> demography;
10  extern vector<double> dem_parameter;
11  extern vector<int> dem_start_gen;
12  extern vector<int> dem_end_gen;
13  extern vector<int> carrying_cap;
14  extern vector<int> pop_schedule;
```

Pay special attention to the parameter values listed on lines 2–6; each is followed by numbers that specify details of three sequential demographic events in the population. For example, the `demography` parameter specifies the type of demographic event and takes one of five values:

- 0 = constant population size—i.e., no change
- 1 = instantaneous change
- 2 = linear change
- 3 = exponential change
- 4 = logistic change

Thus, the entry `demography 0 1 4` indicates the population first maintains the constant size specified by parameter `popsize` (Chap. 3), experiences an instantaneous change, and then enters a phase of logistic change (growth in this case). The values listed after the other four correspond to these three demographic regimes in the same order. Look again at the parameter values in the previous listing. Take a moment to understand that numbers −9900, 1001, 1500, and 0 provide the needed details to model the second selective regime (an instantaneous change in population size).

The parameters dem_start_gen and dem_end_gen specify the starting and ending generation for a given demographic event, respectively. The value of dem_parameter provides a necessary parameter for a given demographic model:

- Absolute change in population size for instantaneous change
- Per-generation change in population size for linear change
- Rate parameter for both exponential and logistic change

Logistic change requires specification of a *second* model parameter—carrying capacity—provided to the parameter carrying_cap. Each parameter is stored in a vector, and multiple values for a given parameter should be separated by whitespace of any size.

The parameter values in the previous listing will be used in the next section. Collectively, they specify no change in population size for the first 1000 generations, followed by an instantaneous loss of 9900 individuals (out of 10,000, as specified by the parameter popsize) that lasts for 500 generations, and finally logistic growth at a rate of 0.02 with a carrying capacity of 10,000. The logistic demographic regime holds from generation 1501 until the end of the simulation at generation 20,000.

File params.h also declares a vector<int> called pop_schedule, which holds the population size at every generation (generations 0–20,000 in the current case) based on the demographic model parameters just discussed. The values of pop_schedule are calculated in params.cc using the following additions to the file:

additions to **params.cc**

```
...
vector<int> get_multi_int_param(const string &key)
{
  vector<int> vec;
  istringstream iss(parameters[key].c_str());
  string param;
  while(getline(iss, param, ' '))
    vec.push_back(atoi(param.c_str()));
  return vec;
}

vector<double> get_multi_double_param(const string &key)
{
  vector<double> vec;
  istringstream iss(parameters[key].c_str());
  string param;
  while(getline(iss, param, ' '))
    vec.push_back(atof(param.c_str()));
  return vec;
}

```

```
vector<int> create_pop_schedule()
{
   vector<int> ps;
   int i=0;
   int cursize = popsize;
   for (int step = 0; step < demography.size(); ++step) {
     for (; i<dem_start_gen[step]; ++i)
        ps.push_back(cursize);
     for (; i <= dem_end_gen[step]; ++i) {
        switch(demography[step]) {
          case 0: ps.push_back(cursize); // no size change
                 break;
          case 1: if (i == dem_start_gen[step])
                  cursize += dem_parameter[step];
                ps.push_back(cursize); // instantaneous
                break;
          case 2: cursize += dem_parameter[step];
                ps.push_back(cursize); // linear
                break;
          case 3: cursize *= exp(dem_parameter[step]); // exponential
                ps.push_back(cursize);
                break;
          case 4: cursize = (carrying_cap[step] * cursize) /
                        (cursize + (carrying_cap[step] -
                            ↪ cursize)*exp(-1*dem_parameter[step])); //
                            ↪ logistic
                ps.push_back(cursize);
        }
     }
   }
   return ps;
}

// additional variable declarations
vector<int> demography;
vector<double> dem_parameter;
vector<int> dem_start_gen;
vector<int> dem_end_gen;
vector<int> carrying_cap;

 int process_parameters() {
 ...
   demography = get_multi_int_param("demography");
   dem_parameter = get_multi_double_param("dem_parameter");
   dem_start_gen = get_multi_int_param("dem_start_gen");
   dem_end_gen = get_multi_int_param("dem_end_gen");
   carrying_cap = get_multi_int_param("carrying_cap");
...
}
...
vector<int> pop_schedule = create_pop_schedule();
```

The functions `get_multi_int_param( )` and `get_multi_double_param( )` (lines 2–20) allow read-in of multi-valued parameters and are used in the calculation of `pop_schedule` carried out by function `create_pop_schedule( )` (lines 22–51). Population size begins at the value specified by the parameter `popsize` and is stored as current population size in the variable `cursize` (line 26). The `for` loop beginning at line 27 will step through each demographic event and, for each generation of the individual demographic regime, calculate population size. Note that once the starting generation of a demographic event is reached, as determined by the control statement at line 28, the program enters the `for` loop from line 30 through line 48 until the last generation of the demographic event is reached. Each iteration of this `for` loop employs a switch statement (lines 31–47), which queries the type of demographic change (line 31), updates population size (`cursize`) accordingly, and pushes `cursize` to the population schedule (`pop_schedule`). Finally, the calculated population schedule is returned (line 50).

Small changes to `population.h` are also required. However, we will cover these changes in Sect. 4.2.3 following a brief discussion of how to calculate the summary statistic Tajima's *D*.

4.2.2 *Calculating Tajima's D*

As discussed in Chap. 3, Tajima's *D* quantifies the difference between two estimators of θ: nucleotide diversity (π) and Watterson's estimator (θ_W) (Tajima 1989). At equilibrium, we expect these estimators to provide roughly equal estimates, yielding a value of $D = 0$. Positive or negative deviations from zero are characteristic of various evolutionary events, including demographic change. Specifically, Tajima's *D* is defined as

$$D = \frac{\pi - \theta_W}{\sqrt{\mathtt{Var}(\pi - \theta_W)}} \tag{4.7}$$

Now let $a_1 = \sum_{i=1}^{n-1} \frac{1}{i}$, $a_2 = \sum_{i=1}^{n-1} \frac{1}{i^2}$, n be sample size (the number of sequences), and S be the number of segregating sites in the sample. Then, Tajima (1989) defines $\mathrm{Var}(\pi - \theta_W)$ as

$$\mathtt{Var}(\pi - \theta_W) = S\left(\frac{1}{a_1}\right)\left(\frac{n+1}{3(n-1)} - \frac{1}{a_1}\right) + S(S-1)\left(\frac{1}{a_1^2 + a_2}\right)\left(\frac{2(n^2+n+3)}{9n(n-1)} - \frac{n+2}{a_1 n} + \frac{a_2}{a_1^2}\right) \tag{4.8}$$

Keep in mind that $\theta_W = \frac{S}{a_1}$, where the denominator controls for sample size, which is necessary because large samples uncover a greater number of segregating sites than small samples. Similarly, the expected range of Tajima's *D* is expected to be greater for smaller sample sizes; the variance of

D in the denominator controls for this fact, as it is a function of n, a_1, and a_2, the latter two themselves being functions of n. The following function for calculating and returning Tajima's D is added to `summarystats.h`:

summarystats.h: function get_tajimas_d( )

```
 1  double get_tajimas_d (double pi, double watterson, int S) {
 2     double d = pi - watterson; // numerator
 3     double a1 = 0.;
 4     double a2 = 0.;
 5     for (double i=1.; i < sampsize; ++i) {
 6        a1 += 1./i;
 7        a2 += 1./(i*i);
 8     }
 9     double n = sampsize; // for easier expression
10     double var = watterson * ( (n+1) / (3*(n-1)) - 1/a1 ) +
11         ( S * (S-1) * ( 1 / (a1*a1+a2)) *
12        ( (2*(n*n + n+3))/(9*n*(n-1)) - (n+2)/(a1*n) + a2/(a1*a1) ) );
13     return(d / sqrt(var));
14  }
```

As reflected by the arguments to this function (line 1), it can only be called after π and θ_W have been calculated. Furthermore, calculation of Tajima's D requires us to use the per-locus rather than per-site estimates of θ. Therefore, the return values of the functions `get_pi( )` and `get_watterson( )` were also modified to reflect this necessary change. Specifically, `return (sumdiffs / numcomp)` rather than `return (sumdiffs / numcomp / seqlength)` returns the per-locus value of π. Similarly, the change to `return (S / denominator)` returns the *per-locus* number of segregating sites.

4.2.3 *Final Changes to Program Files*

Minor modifications to `population.h` are required to complete the implementation of additional functionality—namely, the ability to simulate demographic change and calculate Tajima's D. Because we are now calculating three summary statistics, we also make modifications to `population.h` that cause all summary statistic output to print to one file.

population.h: Chap. 4 modifications

```
// changes to private variables
ofstream sumstat_file; // all sumstats printed here

// change to function update_alleles( )
   if (current_count == pop_schedule[gen]*2) { // replaces popsize*2 in
        ↪ ch3 listing

```

```
 7 // changes to function reproduce ( )
 8   randomind.param(uniform_int_distribution<int>::param_type(0,pop_schedule
        ↪ [gen]-1));
 9    ...
10   for (int i=0; i< pop_schedule[gen==0 ? gen : gen-1]; ++i) { // replaces
        ↪ popsize in ch3 listing
11    ...
12    for (auto iter = individuals.begin(); iter != individuals.end() -
         ↪ pop_schedule[gen==0 ? gen : gen-1]; ++iter) // replaces
         ↪ popsize in ch3 listing
13    ...
14    individuals.erase(individuals.begin(), individuals.end()-
         ↪ pop_schedule[gen==0 ? gen : gen-1]); // replaces popsize in
         ↪ ch3 listing
15
16 // additions to function get_sample( )
17   double pi = get_pi(sample);
18   double watterson = get_watterson(sample, S);
19   double tajimasd = get_tajimas_d(pi, watterson, S);
20   sumstat_file << gen << " " << pi << " " << watterson << " " << tajimasd
        ↪ << endl;
21
22 // addition to function close_output_files( )
23   sumstat_file.close();
24
25 // changes to Population constructor
26 randomind.param(uniform_int_distribution<int>::param_type(0,pop_schedule[0]
     ↪ - 1)); // replaces popsize in ch3 listing
27    ...
28    for (int i=0; i<pop_schedule[0]; ++i) { // replaces popsize in ch3
         ↪ listing
29   ...
30   fname = "sumstats";
31   sumstat_file.open(fname.c_str());
32   sumstat_file << "gen pi watterson tajimasd" << endl;
```

Changes to the `population.h` file shown in lines 2, 17–20, 23, and 30–32 drive calculation of Tajima's *D* and output of all summary statistics to a single file named `sumstats`.

The remaining changes specified account for the potentially variable value of population size. In Chap. 3, where population size remained constant, we could simply use the value `popsize` every generation. Now, however, the population size at any given generation `gen` is stored in `pop_schedule[gen]`. Because population size may change each generation when modeling demographic change, the random variable `randomind` must be updated each generation so that potential indices of individuals chosen as parents fall between 0 and current population size less one. Initialization of the `randomind` is performed in the constructor (line 26) and updated each generation at the beginning of function `reproduce( )` (line 8). Variable population size also necessitates changes to the test expressions of several `for` loops (lines 10, 12, and 28) as well as an `erase( )` function (line 14). In line 10, the test expression is written as `i < pop_schedule[gen==0 ? gen : gen-1]`. We use the conditional operator for the index because `pop_schedule[-1]` is undefined; thus, for generation 0, we need to use the initial population size `pop_schedule[0]` as the test condition.

4.3 Simulating a Bottleneck Followed by Logistic Growth

Having modified the forward simulation program to allow demographic change and calculation of Tajima's D, we can now use the program to model a broadly realistic case of demographic change. Imagine a population of $N_e = 100{,}000$ diploid individuals at mutation-drift equilibrium. Catastrophic environmental change causes an instantaneous population bottleneck that reduces population size to $N_e = 100$, which lasts for 500 generations (Fig. 4.4a). After this, the environment recovers, and logistic growth ensues at a rapid growth rate of $r = 0.02$ and a carrying capacity of $N_e = 10{,}000$ (Fig. 4.4a).

The demographic parameter values required to specify the demographic model shown in Fig. 4.4a are those focused on at the beginning of Sect. 4.2.1. We sample 100 non-recombining sequences with a per-site mutation rate of $\mu = 1 \times 10^{-8}$ and a length of 250,000 bp from the simulated population every ten generations. Each simulation uses an initial coalescent simulation to generate sequences for generation 0 in the forward simulation. `mscommand` is set to `./ms 20000 1 -t 100 >ms_output` in `parameters`. Because `popsize` remains 10,000 and we are simulating a diploid locus, MS must generate 20,000 sequences. Furthermore, the value of $\theta = 100$ as the *per-locus* value of θ was obtained as follows: $\theta = 4N_e\mu \times 2.5 \times 10^5 = 4 \times 10^4 \times 2.5 \times 10^{-8} \times 2.5 \times 10^5 = 100$.

Figure 4.4b,c shows per-site estimates of θ for two independent replicates of the simulation setup detailed in the previous paragraph. Although the starting quantity of genetic variation generated by coalescent simulation differs in each case, qualitatively similar behavior is seen in both replicates. The instantaneous bottleneck immediately and drastically reduces genetic variation. Over the course of the next 18,500 generations, as population size increases rapidly to carrying capacity and holds there, genetic variation is slowly restored to expected equilibrium levels. Importantly, π recovers more rapidly than θ_W. Given that the numerator of Tajima's D is $\pi - \theta_W$, we expect this behavior to yield consistently negative values of D. This expectation is observed; roughly between generations 1500 and 5000, Tajima's D is < -2 in both simulations (Fig. 4.4d). In general, Tajima's D greater than 2 or less than -2 is considered a significant indicator of evolutionary change. Although a variety of evolutionary factors may be responsible for this deviation (see next section), we know the cause in our simulated population: rapid population

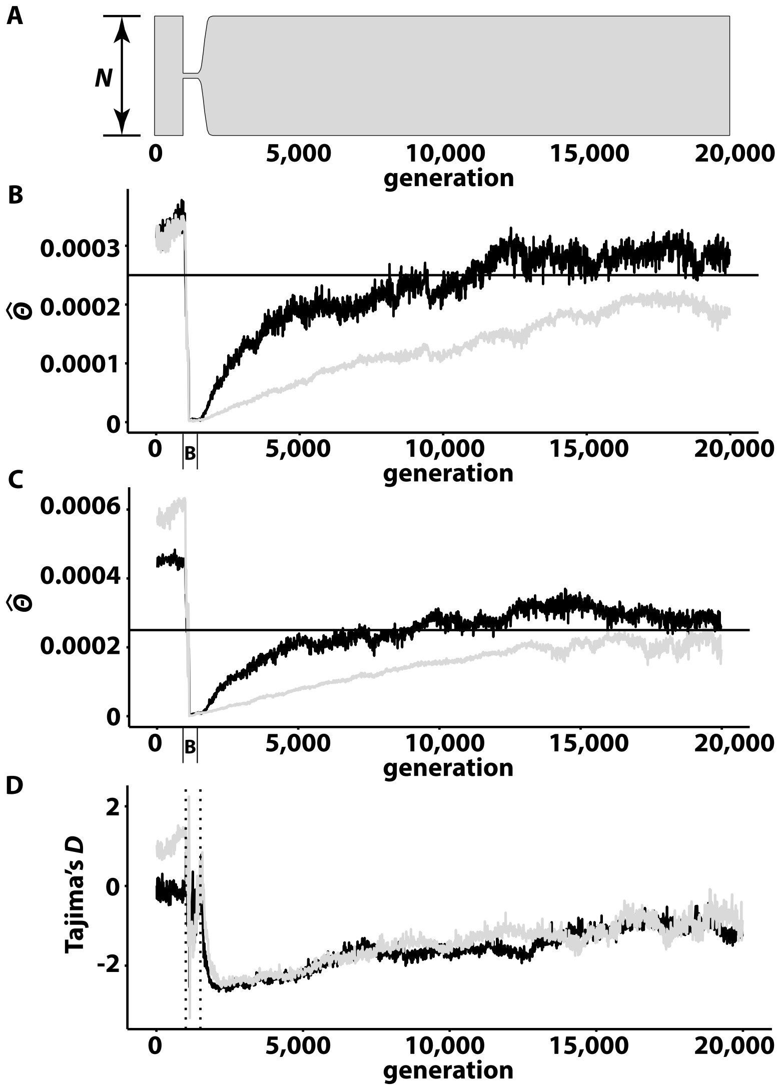


Fig. 4.4 Modeling demographic change. (**a**) The demographic model; a population of $N_e = 10,000$ diploid individuals is simulated using MS; forward simulation consists of 1000 generations at $N_e = 10,000$, followed by an instantaneous bottleneck that reduces the population to $N_e = 100$ diploid individuals for 500 generations, followed by logistic growth at a rate of $r = 0.02$ and a carrying capacity of $K = 10,000$. (**b** and **c**) Results from two independent replicates of the simulation, sampled every ten generations and summarized as two independent estimators of θ, θ_W (black lines) and π (gray lines). (D) The summary statistic Tajima's D for the simulation shown in (**b**; black line) and (**c**; gray line). The period of the 500-generation bottleneck is indicated by **B** or vertical dashed lines

expansion. In other words, Tajima's D provides a rather long-lived signal of population expansion.

Interestingly, Tajima's D does not provide us with evidence of the drastic population bottleneck that occurs at generation 1000. During the bottleneck, values of D become highly variable, but we fail to note a consistent value of $D > 2$, which is indicative of significant population decline (Fig. 4.4d). The loss of nearly all genetic variation with the instantaneous bottleneck explains this. Values of π and θ_W are both nearly zero immediately following the bottleneck, which does not provide a great enough difference between the two estimators to yield a clear signal of population decline. In the next section, we will model a gradual population decline, which does produce a clear increase of D to greater than 2. The population expansion provides a much clearer signal because the bottleneck produces a genetically depauperate population that sets the stage for a slow recovery of genetic variation as population size increases. The relative difference in recovery of π and θ_W then yields the significantly negative values of Tajima's D shown in Fig. 4.4d.

4.4 The Varying Utility of Summary Statistics for Inference

We now simulate a gradual, linear decline in population size. This situation is of practical relevance, as conservation biologists are often interested in assessing current population dynamics in order to determine if a population merits intervention. The results of this simulation will raise the issue of which summary statistics are best suited for a specific inferential task.

Consider a population of 50,000 diploid individuals at mutation-drift equilibrium. A linear decline of ten individuals per generation begins at generation 1000 until, at generation 5900, the population stabilizes at 1000 diploid individuals. We assume a per-site point mutation rate of $\mu = 1 \times 10^{-8}$ and a 50,000-bp sequence. Once again, we initialize a `Population` object using the results of coalescent simulation in MS. The `parameters` file that describes this population model (Fig. 4.5a) is

`parameters` for the model detailed in Sect. 4.4

```
popsize 50000
mutrate 1e-08
seqlength 5e04
sampsize 100
sampfreq 10
demography 0 2 0
dem_parameter 0 -10 0
dem_start_gen 0 1001 5901
dem_end_gen 1000 5900 8500
carrying_cap 0 0 0
useMS 1
mscommand ./ms 100000 1 -t 100 >ms_output
```

Figure 4.5b shows the per-site values of π (gray) and θ_W (black) sampled every ten generations. During the population decline, a *very* gradual increase in Tajima's D is observed (Fig. 4.5c). Furthermore, while θ_W begins to decline at 3000 generations, likely due to the loss of rare variants, π remains elevated until after the population decline ends at 5900 generations (Fig. 4.5b). Shortly after the decline stops at generation 5900, significant values of D are observed and maintained during the next 2000 generations (Fig. 4.5c).

At roughly generation 7000, there is a sharp drop in both θ_W and π. Generate a hypothesis that explains this seemingly anomalous "blip."

From the practical standpoint of the conservation biologist hoping to determine if a population is subject to a sustained population decline, these results are worrisome. During the nearly 4000 generations that the population is bleeding individuals, both π and Tajima's D fail to signal such. Moreover, the decline in θ_W is only visible to us because we have access data from the entire history of the population decline. θ_W at any one sample point would not signal a population decline.

On another note of caution, Fig. 4.4 makes clear that Tajima's D drops below the critical value of -2 following a drastic, though transient, population bottleneck. As will be shown in the last chapter of this volume, strong positive selection also drives Tajima's D to less than -2. As is often the case in population genetics, distinct evolutionary scenarios produce the same pattern of genetic variation. On a more hopeful note, we can sometimes find an additional layer to the genetic pattern that allows us to go further with our inference. Returning to our example, both a population bottleneck and selective sweep can produce a significantly negative value of Tajima's D. On average, the former will be observed across the whole genome, while the latter will be limited to a region proximate to the molecular target of natural selection.

Given the nosiness of our summary statistic clues, one inferential tactic is to create as many unique ways of summarizing the genetic data as possible in the hope that a legion of summary statistics will somehow collectively capture the nuance of a sequence alignment. Superficially, the recent use of convolutional neural networks (CNN) on input "images" of sequence alignments seems to obviate the need for summary statistics (Flagel et al. 2018). Of course, the CNN is learning high-dimensional summaries of the data that we would never imagine natively (i.e., as humans). So summary statistics are still in play, but they are of the multidimensional chess variety.

Nevertheless, there are summary statistics that intuitively should capture aspects of the sequence alignment not tracked by π or θ_W. For example, how many unique haplotypes are found in a sample? The answer is the summary statistic K. Each of the K haplotypes has a frequency in the sample.

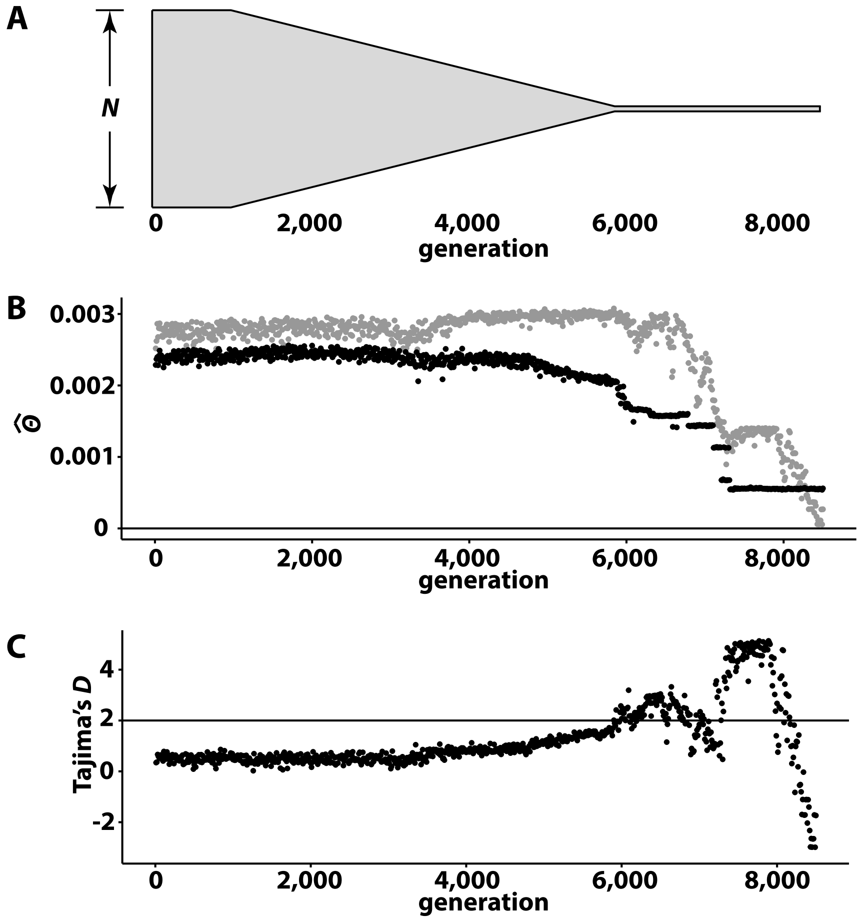


Fig. 4.5 Modeling demographic change: a gradual, linear decline over the course of 4000 generations. (**a**) The demographic model: a population of $N_e = 50,000$ diploid individuals at mutation-drift equilibrium is simulated using MS; forward simulation begins with a further 1000 generations of equilibrium followed by a linear decline lasting 4900 generations in which N_e declines by ten individuals each generation. At this point, $N_e = 1000$ diploid individuals and a further 4600 generations are simulated. Results of simulation of this model include (**b**) the per-site values of θ_W (black dots) and π (gray dots) as well as (**c**) Tajima's D

The summary statistic K is the discrete frequency distribution of each of the K haplotypes. Sometimes, variant-specific summaries such as observed and expected heterozygosity, or their difference, can prove valuable.

Successful inference often requires us to act creatively, conjuring up new summaries of the data as well as creating compound summary statistics (such as Tajima's *D*) that contrast individual summaries of the data. Imagine an evolutionary scenario and a sequence alignment sampled from a population. What other summary statistics can you come up with that might facilitate insight into the population's evolution?

References

Flagel L, Brandvain Y, Shrider D (2018) The unreasonable effectiveness of convolutional neural networks in population genetic inference. Mol Biol Evol 36:220–238

Hudson RR (1990) Gene genealogies and the coalescent process. In: Futuyma D, Antonovics J (eds) Oxford surveys in evolutionary biology, vol 7. Oxford University Press, pp 1–44

Staab PR, Metzler D (2016) Coala: an R framework for coalescent simulation. Bioinformatics

Tajima F (1989) Statistical method for testing the neutral mutation hypothesis by dna polymorphism. Genetics 123:585–595

5
Meiotic Recombination

The magician was one who knew how to enter into this system, and use it, by knowing the links of the chains of influences ... [1]
– Frances A. Yates *Giordano Bruno and the Hermetic Tradition*

5.1 Background

Thus far we have treated mutation as the only source of genetic variation. While mutation is the source of all genetic *variants*, this viewpoint limits our thinking on issues of evolutionary genetics. DNA sequences are strings of specific *combinations* of variants. It is reasonable to conclude that different combinations of variants might, for example, vary in terms of the fitness they confer to individuals. Meiotic recombination in diploid organisms is responsible for producing potentially novel combinations of genetic variants, i.e., unique multisite patterns of genetic variation referred to as *haplotypes*. The shuffling of genetic variants via recombination is evolutionarily important because specific combinations of variants may be selectively advantageous. In addition, haplotypic variation generated by recombination is often of value to inference. As discussed in the final section of the previous chapter, haplotype data are the basis for several additional summary statistics that we can add to our inferential arsenal.

5.1.1 *Crossing-Over and Independent Assortment*

Because meiotic recombination alters *combinatorial* genetic variation, discussion of recombination requires us to focus on two or more variable loci in

[1] Quoted with permission. 1991, University of Chicago Press.

a population. Depending on whether the loci in question are linked or unlinked, two very distinct forms of meiotic recombination are responsible for shuffling variants found in gametes. In the case of linked polymorphisms, crossing-over during Prophase I of meiosis is responsible for recombination. Consider two *linked* diallelic SNPs, where the specific single-nucleotide variants are G/C and A/T. Considering only these two sites, four unique combinations (i.e., haplotypes) are possible:

```
G A
C A
G T
C T
```

Let K_n stand for the number of possible, unique haplotypes in a sequence with n diallelic polymorphisms. As the number of polymorphic sites increase, the number of unique haplotypes increase exponentially: $K_n = 2^n$.

Based on the simple example just discussed, imagine crossing-over occurs between the two polymorphisms during meiosis of a cell in the doubly heterozygous individual `CT/GA`, where the forward slash separates the haplotypes of the two homologous chromosomes. Two of the four daughter cells (gametes) will harbor *recombinant gametic types*—`CA` and `GT`—while the other two daughter cells will hold *parental types*—`CT` and `GA`—named so because these are the very haplotypes contributed by the parents of the `CT/GA` individual (Fig. 5.1).

The genetic distance between two loci, measured in centiMorgans (cM), quantifies how frequently recombinant types are generated in cells of the double heterozygote. If the two loci are physically very close (i.e., tightly linked), crossing-over between the loci in question will be rare and the *recombination frequency* (r, the proportion of gametes that are recombinant) will be low. On the other hand, if the two loci are physically very far apart from one another nearly every meiosis will include a cross-over between them, each of which will generate two recombinant and two parental gametic types (Fig. 5.1, left). In this case $r = 0.5$, which is an upper limit on recombination frequency. The reason r cannot be greater than 0.5 is that each cross-over involves only one of the sister chromatids from each of the homologous chromosomes. The chromatids of each homologous chromosome not participating in the cross-over event retain their original, parental haplotype (Fig. 5.2).

Let me stress this point by relating it to how we model and simulate recombination. Recombination frequency among two linked loci measures the frequency at which crossing-over leads to *transmission* of its products (recombinant chromosomes and the gametes that carry them), *not* the frequency of chiasma formation between two loci. Again, the extreme case is most illustrative of this point. When r between two linked loci is maximal at 0.5, *every* meiosis is expected to entail crossing-over between the two loci. Recombination frequency is 0.5 rather than 1 because only half of the products of this meiosis will be recombinant. In other words, recombination frequency measures the fraction of gametes carrying recombinant sequences and *not* the frequency at which a recombination event takes place between two loci in meiocytes. This simplifies the model of recombination in the computer because we can simply inquire whether a recombinant sequence is passed on (with probability r) and eschew simulating chiasmata that ultimately will have no effect on the pattern of diversity in the simulated, focal segment of DNA (Fig. 5.2b).

If the two polymorphisms in question are unlinked, meiosis still produces recombinant gametic types with combinations of genetic variants that differ from either parent's contribution. Meiotic recombination of unlinked loci is referred to as *independent assortment*, which means that each homologous pair independently "determines" which side of the metaphase plate the maternal and paternal chromosomes position themselves. For two homologous pairs as shown in Fig. 5.1 (right), there are two distinguishable alignments of the homologous pairs. If we consider all homologous pairs of, say, human, independent assortment leads to a very large number of distinguishable assortments ($2^{c-1} = 2^{22} \approx 4.2$ million, where c is the haploid number of chromosomes in a dividing cell).

The upper bound on r is still 0.5. Consider the double heterozygote `C/G;T/A`, where the semi-colon indicates that the two polymorphisms reside on non-homologous chromosomes and the forward slash still separates the variants found on each of the two homologous pairs. Independent assortment of nonhomologous pairs leads to a recombination frequency of $r = 0.5$ because one of the two possible alignments during Meiosis I yields solely parental types, the other possible alignment yields solely recombinant types, and both alignments are equally probable (Fig. 5.1, right).

5.1.2 Linkage Disequilibrium

The shuffling action of meiotic recombination implies that it is a randomizing mechanism. If it is currently the case that a nonrandom association of alleles—e.g., `CT` and `GA` haplotypes are found much more frequently than `CA` and `GT` haplotypes—crossing-over should lead to the decay of this nonrandom association between alleles over time absent countervailing mechanisms. The nonrandom association of alleles is commonly called *linkage disequilibrium* (LD). Unfortunately, this term is somewhat of a misnomer as LD may also be observed between unlinked loci. For example, epistatic selection in which *the combinations* of an allele on chromosome 2 and an allele on chromosome 5 are selected for (or against) because this specific combination of alleles leads to greater (or lesser) fitness will enrich for a particular combination of unlinked alleles. In other words, epistatic selection can lead

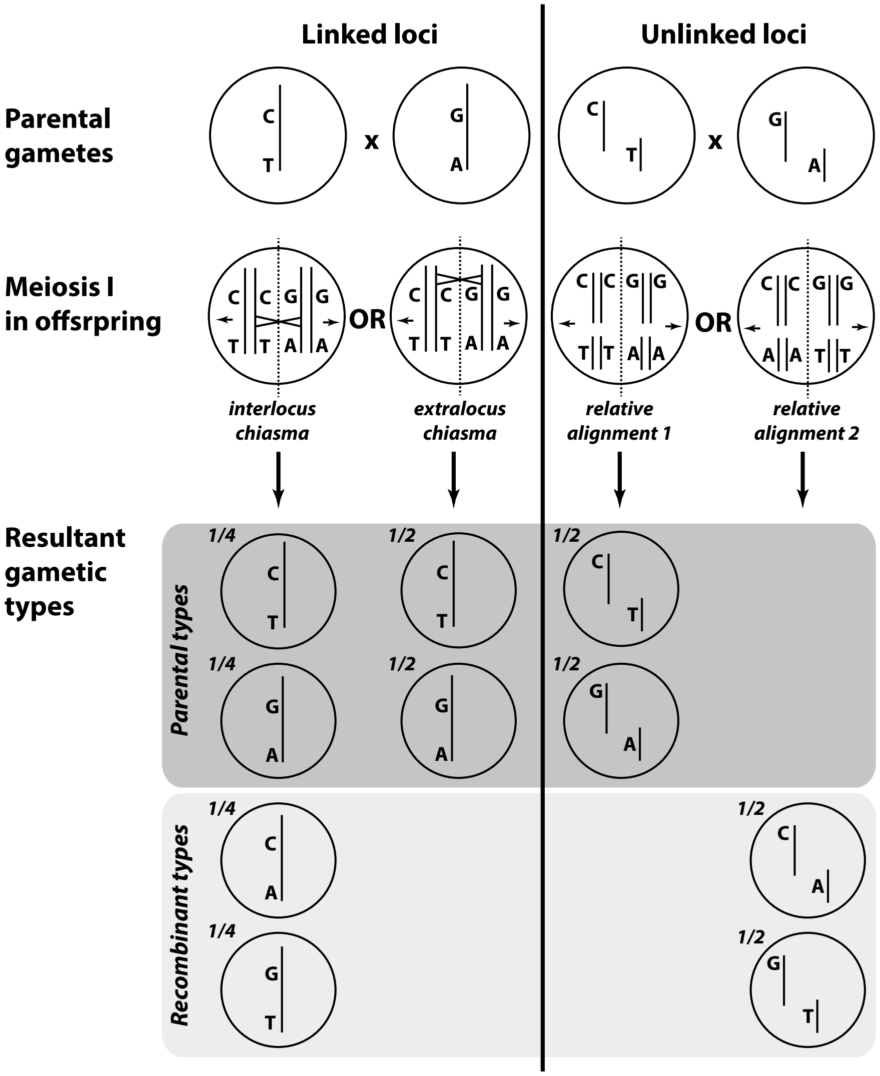


Fig. 5.1 Recombinant gametic types, which are new combinations of alleles in a gamete relative to those found in the parental gametes, are generated by crossing-over in the case of linked loci (left) and independent assortment in the case of unlinked loci (right). Regarding linked loci, recombinant types are only generated if the chiasma forms between the two loci in question (an *interlocus chiasma*); even then, only half of the resultant gametes are recombinant because two of the chromatids are uninvolved in crossing-over and retain the parental combination of alleles. Importantly, the frequency at which interlocus chiasmata form is positively correlated with physical distance between loci. Regarding unlinked loci, there are two possible relative alignments of the relevant homologous pairs. One produces 100% parental types, while the other produces 100% recombinant types. Because both alignments are equally likely, meioses produce 50% recombinant types on average (i.e., $r = 0.5$). In Meiosis I illustrations, vertical dashed line represents the cell's equator or metaphase plate

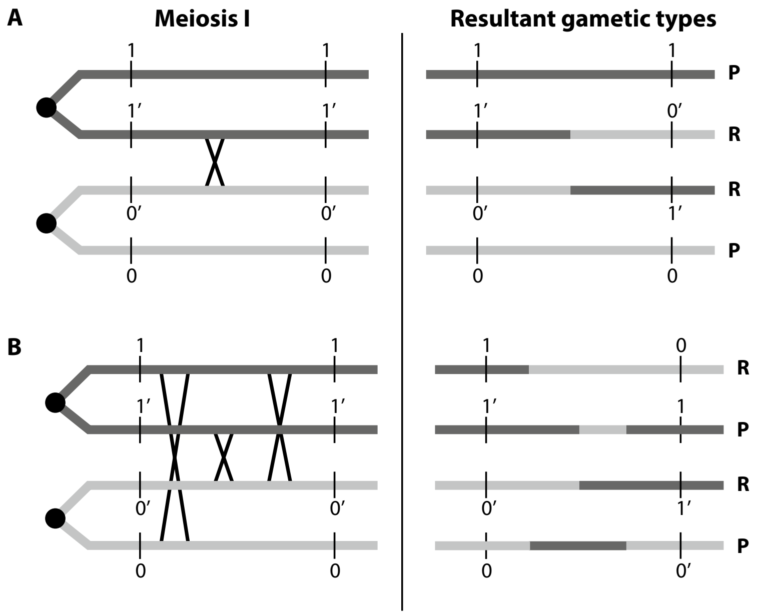


Fig. 5.2 (**a**) Even for loosely linked loci, between which a chiasma forms *every* meiosis, r cannot exceed 0.5 because a given chiasma only involves two chromatids. This result is easy to interpret; the two chromatids not involved in crossing-over remain parental types. (**b**) Perhaps more surprisingly, multiple chiasmata involving different pairs of chromatids still yield only half recombinant types *with respect to the two loci shown.* Also note that our simulation program does not need to account for the complexity shown in the right-hand panel of (**b**). We can simply use r as a probability to assess whether or not a *transmitted* sequence is recombinant or not

to a nonrandom association of unlinked alleles. As a result, LD is sometimes referred to as *gametic* disequilibrium in the literature.

The presence and magnitude of LD can be a powerful inferential tool, but several questions related to LD need to be answered: (1) How does LD arise?; (2) How do we quantify LD?; and (3) At what rate does crossing-over cause LD between linked loci to decay?

Epistatic selection is one means by which LD arises. If a specific combination of alleles at two loci is selectively favored, that combination will increase in frequency in the population producing a nonrandom association of alleles. Conversely, if a specific combination of alleles at two loci is deleterious, that combination will decrease in frequency in the population, leading to a conspicuously low frequency of the combination. Another source of LD is simple mutation. Consider a `G/A` polymorphism linked to a monomorphic `T`. Now imagine that one copy of this `T` nucleotide is mutated to `C` on a chromosome that bears a `G` at the already-polymorphic site. If the two sites are sufficiently close to one another and the new `C` allele randomly increases in frequency, `GC` haplotypes will also rise in frequency, representing a nonrandom association of alleles.

The most common means of *quantifying* LD is to calculate coefficients of linkage disequilibrium. Consider two linked loci with two alleles each: `G/A` and `T/C`. Furthermore, imagine that both polymorphic loci are at Hardy-Weinberg equilibrium and that LD is complete. In other words half of the chromosomes in a sample are of haplotype `GC` and the other half are of haplotype `AT`. The frequencies of the four possible haplotypes are then $P_{GC} = P_{AT} = 0.5$ and $P_{GT} = P_{AC} = 0$ and all allele frequencies (e.g, p_G at site 1) are 0.5. If the association among alleles at the two loci was completely *random*, then the difference $P_{GC} - p_G p_C$ is expected to equal zero; the frequency of a given haplotype should simply be equal to the product of the two component allele frequencies. The coefficients of LD are calculated in this way:

$$\begin{aligned} D_{GC} &= P_{GC} - p_G p_C = 0.25 \\ D_{AT} &= P_{AT} - p_A p_T = 0.25 \\ D_{GT} &= P_{GT} - p_G p_T = -0.25 \\ D_{AC} &= P_{AC} - p_A p_C = -0.25 \end{aligned} \tag{5.1}$$

Thus, we see that regardless of the haplotype on which we focus, the absolute value of the coefficient is 0.25. And it really is the absolute value that indicates the relative magnitude of LD. As LD decays (see below), both negative and positive coefficients approach zero, which represents complete random association among alleles, or *linkage equilibrium*.

One problem with this standard set of LD coefficients is that the maximum absolute value obtainable is dependent on the frequencies of the component alleles. When all four alleles are at equal frequencies of 0.5, $D_{\mathtt{max}} = 0.25$. In any other case, however, $D_{\mathtt{max}} < 0.25$, which you can confirm by plugging four unequal allele frequencies into the formula for $D_{\mathtt{max}}$:

$$D_{\mathtt{max}} = \mathtt{min}\{p_1 p_2, q_1 q_2\}, \tag{5.2}$$

where p_1 and p_2 are the frequencies of the two alleles at one focal locus and q_1 and q_2 are the frequencies of the two alleles at the other.

Because LD coefficients are dependent on allele frequencies, we cannot reliably compare our measure of LD between different pairs of loci. Lewontin (1964) proposed a set of normalized coefficients termed D':

$$D' = \frac{D}{D_{\mathtt{max}}} \tag{5.3}$$

Although D' is also somewhat dependent upon component allele frequencies, it is an improved measure for comparison of LD among different pairs of loci. See Hedrick (1987) for a discussion of the pitfalls related to quantifying LD as well as the benefits to using different metrics for different purposes.

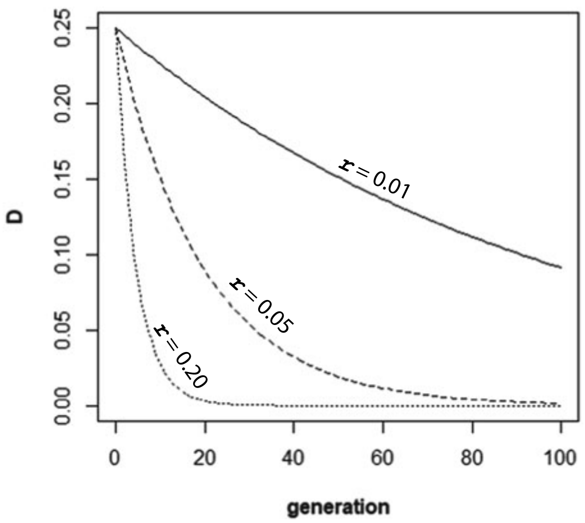


Fig. 5.3 Decay of linkage disequilibrium in 100 generations for three different values of r

Decay of LD among linked loci is a function of their genetic distance as measured by recombination rate r. Specifically, after one generation the coefficient of LD changes by a factor of $1 - r$:

$$D_{t+1} = D_t(1 - r), \tag{5.4}$$

and after t generations declines by a factor of $(1 - r)^t$:

$$D_t = D_0(1 - r)^t, \tag{5.5}$$

where D_0 is the starting value of the LD coefficient. Figure 5.3 shows the decay of LD for different values of r. The dynamics illustrated in this figure are easily explained; greater genetic distances (r) reflect more frequent crossing-over between the two loci in question, the shuffling effect of which leads to more rapid decay of LD.

5.1.3 Variation in Recombination Rate

When simulating a relatively short sequence of DNA, it is appropriate to use a single recombination rate—for example, 10^{-8} per nucleotide. However, simulation of a long DNA sequence requires us to recognize that fine-scale recombination rates vary along chromosomes. Recombination ″hotspots″ are local regions of a chromosome that show recombination rates much greater than the genomic average (Lichten and Goldman 1995).

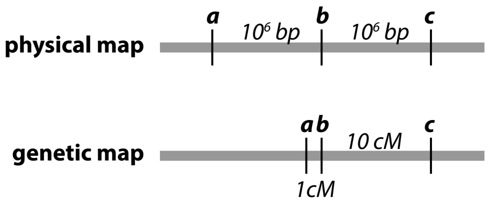


Fig. 5.4 Locus b is separated from loci a and c by the same physical distance of 1 million base pairs. For illustrative purposes, let the mean *per-nucleotide* value of r between loci a and b be 10^{-8} and that between loci b and c be 10^{-7}. Then, genetic distances between the two loci are not equal to each other. Specifically, the recombination rate between loci a and b, $r_{ab} = 10^6 \times 10^{-8} = 0.01 = 1\,\texttt{cM}$ while $r_{ab} = 10^6 \times 10^{-7} = 0.01 = 10\,\texttt{cM}$. In terms of recombination, b is ten times closer to a than c

A *physical map* of genetic loci is drawn in units of base pairs, while a *genetic map* is drawn in units of cM. Because recombination rate varies along chromosomes, distances between loci on physical and genetic maps are not proportional (Fig. 5.4). We will briefly explore the consequences (in terms of LD coefficients) of variation in recombination rate between six loci in Sect. 5.2.

In Sect. 5.3, we will add to and modify the FORTUNA program files by adding cross-over events to the simulated sequence. When we focus on an entire sequence, it is possible for us to explicitly classify a segment of the overall sequence as a recombination hotspot. In general, recombination hotspots are on the order of 1–2 kb long. So, as an example, consider a simulated 10,000 bp sequence where bases 4000–5999 represent a hotspot in which per-base recombination rate is $r_h = 10^{-7}$, while the rest of the bases in the sequence possess a recombination rate of $r_c = 10^{-8}$. Now, let r_t represents the total probability of a cross-over somewhere along the sequence: $r_t = 2000 \times r_c + 8000 \times r_c = 2.8 \times 10^{-4}$. Finally, let l be the sequence length, a be the lower bound of the hotspot, and b be the upper bound of the hotspot. Then, given a cross-over event, the cumulative probability distribution of the sequence position p where the cross-over occurs is defined as the piecewise function:

$$C(p) = \begin{cases} \frac{pr_c}{r_t} & \text{if } 0 \leq p < a \\ \frac{ar_c+(p-a)r_h}{r_t} & \text{if } a \leq p < b \\ \frac{(a+p-b)r_c+(b-a)r_h}{r_t} & \text{if } b \leq p \leq l \end{cases} \tag{5.6}$$

Figure 5.5 shows the cumulative distribution of cross-over position for the parameter values discussed above. Note that although the 2 kb hotspot only occupies 20% of the physical sequence, roughly 70% of cross-overs will map to a position within this hotspot. In Sect. 5.3, where we will extend our forward simulation to include crossing-over, we will also add functionality allowing us to model a recombination hotspot. In order to do this, when a

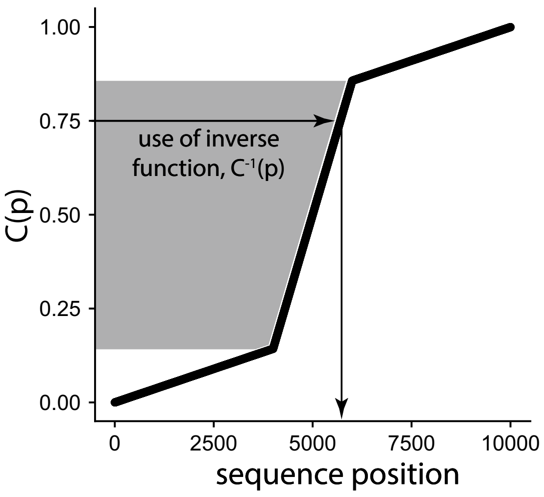


Fig. 5.5 Cumulative probability distribution $C(p)$ of cross-over position on a 10,000 bp sequence. The per-nucleotide probability of cross-over is 10^{-8} for all nucleotides outside the recombination hotspot on the range [4000,6000], where the per-nucleotide probability of cross-over is an order of magnitude higher, 10^{-7}. If we use a random number generator to draw a real number on [0,1], the arrows show how we can map this value using the inverse function of $C^{-1}(p)$ to a position on the sequence. The gray box illustrates the region along the y-axis that maps to a position in the hotspot

cross-over event occurs, we will need to map a random real number on the range [0,1] to a specific position on the sequence where cross-over occurs (Fig. 5.5). This requires the inverse cumulative probability function, which returns a sequence position for a given random number between 0 and 1 inclusive. The inverse of Eq. 5.6 is:

$$C^{-1}(p) = \begin{cases} \frac{cr_t}{r_c} & \text{if } 0 \leq c < \frac{ar_c}{r_t} \\ \frac{cr_t + a(r_h - r_c)}{r_h} & \text{if } \frac{ar_c}{r_t} \leq c < \frac{br_h - a(r_h - r_c)}{r_t} \\ \frac{cr_t - (b-a)(r_h - r_c)}{r_c} & \text{if } \frac{br_h - a(r_h - r_c)}{r_t} \leq c \leq 1 \end{cases} \tag{5.7}$$

This rather monstrous looking equation will be simple to program. The arrowed lines in Fig. 5.5 show its utility. We can simply plug in a randomly generated value of c to the appropriate term in Eq. 5.7 in order to obtain a position of cross-over.

Use of this inverse function does produce the desired results. Using the expanded version of FORTUNA outlined in Sect. 5.3, I kept track of the position of the first 5000 cross-overs sites on a 200,000 bp sequence with a recombination hotspot at bases 100,000–101,999, a base recombination rate of 10^{-8} per base and a "hot" recombination rate of 10^{-7} per base. The empirical probability density of cross-over sites shows a very large increase in cross-over occurrence within the narrow window of the hotspot (Fig. 5.6).

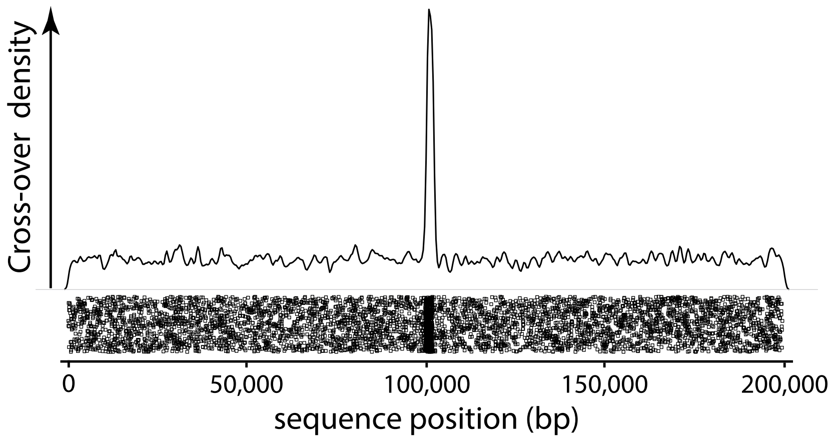


Fig. 5.6 Distribution of 5000 random cross-over positions along a 200,000 bp sequence, with a recombination hotspot from bases 100,000–101,999. The strip chart at the bottom of the figure, with noise added in the vertical dimension, shows a clear build-up of cross-overs coinciding with the hotspot. The kernel density estimate on top shows the magnitude of cross-over density increase within the hotspot

5.2 Forward Simulation of Meiotic Recombination Among Multiple Linked and Unlinked Loci

Before expanding FORTUNA code, we write a standalone program to simulate six diallelic loci in which alleles are represented by 0 and 1. Loci *a*, *b*, and *c* are linked, locus *d* is unlinked to all other loci, and loci *e* and *f* are linked (Fig. 5.7a–b). We use output from this simulation to examine the effect of population size on the decay of LD.

The simulation begins with an admixture event between two populations of the same species wholly isolated from each other for hundreds of generations. We also assume that during their isolation the two populations have fixed opposite alleles at each of the six loci. In other words, each individual of one population is homozygous for ″0″ alleles and each individual of the other population is homozygous for ″1″ alleles. At generation 0 of the simulation, a new admixed population is formed with equal numbers of individuals from each population (Fig. 5.7c). Then, at each of the studied loci the frequencies of ″0″ and ″1″ alleles are both 0.5. Initially, complete LD ($D = 0.25$) exists between each pair of loci because each individual in the generation of admixture either possesses all ″0″ or all ″1″ alleles.

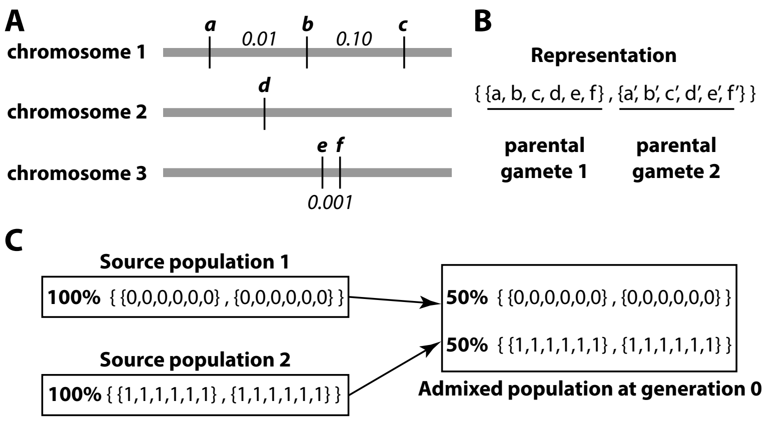


Fig. 5.7 (**a**) Hypothetical chromosomes 1–3 with the six loci modeled; recombination frequency *r* between linked loci is indicated. (**b**) Each individual is represented as a `vector` of two `vectors`, and ultimately stored in a `map` that represents the population (see text). Each of the two vectors contains the alleles transmitted by one parent. (**c**) The simulation detailed in this section begins with an admixture event that combines equal numbers of individuals from two source populations which have fixed for opposing alleles. As a result, the initial admixed population shows complete LD at all six loci

5.2.1 Code

Given the relative simplicity of this program, we will code a single source file `multilocus_ld.cc` that declares the two functions `recombine()` and `print_data()` followed by the main function:

multilocus_ld.cc: declarations and `main` function

```
#include <random>
#include <iostream>
#include <fstream>
#include <string>
#include <map>
#include <vector>
#include <algorithm>

using namespace std;

map<int, vector<vector<int> > > recombine(map<int, vector<vector<int> >
    ↪ >&, double&);
void print_data(int g, ofstream&, map<int, vector<vector<int> > >&,
    ↪ double&);

double ab_r, bc_r, ef_r;
uniform_int_distribution<int> randomind(0,9999);
uniform_int_distribution<int> random_chromatid(0,3);
uniform_real_distribution<double> randomnum(0,1);
```

```
18 mt19937 engine(time(0));
19
20 int main (int argc, char *argv[] ) {
21    int gen = atoi(argv[1]);
22    double numind = atof(argv[2]);
23    ab_r = atof(argv[3]);
24    bc_r = atof(argv[4]);
25    ef_r = atof(argv[5]);
26
27    randomind.param(uniform_int_distribution<int>::param_type(0,numind-1));
28
29    string fname("ld_coefs");
30    ofstream ofile;
31    ofile.open(fname.c_str());
32
33    ofile <<
        ↪ "gen\tDab\tDac\tDad\tDae\tDaf\tDbc\tDbd\tDbe\tDbf\tDcd\tDce\tDcf\
34            tDde\tDdf\tDef" << endl;
35
36    map<int, vector<vector<int> > > population;
37    vector<int> p1{1,1,1,1,1,1};
38    vector<int> p2{0,0,0,0,0,0};
39
40    // parent population 1
41    for (int i=0; i < numind/2; ++i) {
42       population[i].push_back(p1);
43       population[i].push_back(p1);
44    }
45    // parent population 2
46    for (int i=numind/2; i<numind; ++i) {
47       population[i].push_back(p2);
48       population[i].push_back(p2);
49    }
50
51    for (int g=0; g<gen; ++g) {
52       print_data(g, ofile, population, numind);
53       population = recombine(population, numind);
54    }
55
56    ofile.close();
57
58    return 0;
59 }
```

Lines 11–12 declare functions `recombine( )` and `print_data( )`, which are defined in the next listing. Lines 14–18 declare the variables for recombination rates between linked loci as well as three random number generators and the `mt19937` algorithm that powers them. The program takes five command line arguments (lines 20–25), which in order are: (1) number of generations to run; (2) number of diploid individuals; and (3) recombination rates between the linked paris of loci *a* and *b*, *b* and *c*, and *e* and *f*, respectively. On line 27, the `randomind` number generator is updated with the user-defined population size and lines 29–33 open the output file and print the header line. As shown in Fig. 5.7, the six alleles from each parent are stored as a vector and an individual is represented by a vector of two of these vectors (one for each parent). Finally, the population is represented as a map of individual `vector<vector<int> >`s (line 35). The population is initialized with half individuals from a parent population in which all alleles are "0" and half individuals from a parent population where all alleles are "1" (lines 36–48). The heart of the program is the `for` loop (lines 50–53) that prints output and simulates recombination followed by mating for `gen` generations.

The two functions called on lines 51–52 are now defined:

multilocus_ld.cc: `recombine` and `print_data` function definitions

```cpp
map<int, vector<vector<int> > > recombine(map<int, vector<vector<int> > > &p, double &n) {
  map<int, vector<vector<int> > > recombined;
  for (int i=0; i < n; ++i){
    vector<vector<int> > parent1 = p[randomind(engine)]; //e.g., {0,0,0,0,0,1} & {0,1,1,0,0,1}
    vector<vector<int> > parent2 = p[randomind(engine)];

    // ab crossing-over
    // first parent
    if (randomnum(engine) <= ab_r)
      swap(parent1[0][1], parent1[1][1]);
    // second parent
    if (randomnum(engine) <= ab_r)
      swap(parent2[0][1], parent2[1][1]);

    // bc crossing-over
    if (randomnum(engine) <= bc_r)
      swap(parent1[0][2], parent1[1][2]);
    if (randomnum(engine) <= bc_r)
      swap(parent2[0][2], parent2[1][2]);

    // ef crossing-over
    if (randomnum(engine) <= ef_r)
      swap(parent1[0][5], parent1[1][5]);
    if (randomnum(engine) <= ef_r)
      swap(parent2[0][5], parent2[1][5]);

    vector<int> chr1{randomnum(engine) < 0.5 ? 0 : 1, randomnum(engine) < 0.5 ? 0 : 1 };
    vector<int> chr3{randomnum(engine) < 0.5 ? 0 : 1, randomnum(engine) < 0.5 ? 0 : 1 };

    vector<int> variants1;
    vector<int> variants2;

    for (int j=0; j<3; ++j) {
      variants1.push_back(parent1[chr1[0]][j]);
      variants2.push_back(parent2[chr1[1]][j]);
    }

    variants1.push_back(parent1[randomnum(engine) < 0.5 ? 0 : 1][3]);
    variants2.push_back(parent2[randomnum(engine) < 0.5 ? 0 : 1][3]);

    for (int j=4; j<6; ++j) {
      variants1.push_back(parent1[chr3[0]][j]);
      variants2.push_back(parent2[chr3[1]][j]);
    }

    recombined[i].push_back(variants1);
    recombined[i].push_back(variants2);
  }
  return recombined;
}

void print_data(int g, ofstream &o, map<int, vector<vector<int> > > &p,
   ↪ double &n) {
  vector<vector<double> > coefs;
  vector<double> zeroes;
  zeroes.assign(6,0.);

  double nn = 2.*n; // number of alleles

  for (int i=0; i<6; ++i)
    coefs.push_back(zeroes);

  for (int i=0; i<5; ++i) {
    for (int j=i+1; j<6; ++j) {
      double freq1 = 0.;
      double freq2 = 0.;
      for (int k = 0; k<n; ++k) {
        for (int l=0; l<2; ++l) {
          freq1 += p[k][l][i];
          freq2 += p[k][l][j];
          if (p[k][l][i] ==1 && p[k][l][j] == 1)
            coefs[i][j]++;
        }
      }
      coefs[i][j] /= nn;
      coefs[i][j] -= (freq1/nn * freq2/nn);
    }
  }

  o << g << "\t";
  for (int i=0; i<5; ++i) {
    for (int j=i+1; j<6; ++j) {
      o << coefs[i][j] << "\t";
    }
  }
  o << endl;
}
```

Function `recombine()` receives a reference to `population` and population size and an empty `map` called `recombined` is declared as a holder for individuals of the next generation (lines 1–2). The `for` loop spanning lines 3–48 iterates population-size times. It begins by picking two random parents from the population (lines 4–5). Then, crossing-over among linked loci on the parental chromosomes is implemented. Let us focus on cross-over events between loci *a* and *b*. For parent 1, we ask if a random number between 0 and 1 inclusive is less than or equal to the recombination rate between *a* and *b*. If so, we conclude that cross-over has occurred and the alleles at the a and b loci will be swapped between the two homologs. To accomplish this the standard algorithm `swap( )` is used; the arguments to this function are the alleles at position 1 (the *b* locus allele) of the first parent's two chromosomes. The algorithm will, as named, swap these two alleles within their respective `vectors`. The same task is performed for parent 2 and again for linked loci *b* and *c* and *e* and *f* (lines 11–25). Lines 27–31 choose between the two parental homologs of each parent for chromosome 1 and 3 and declare two temporary `vectors` to contain the parental contributions from each parent. Lines 33–44 add the allelic contributions of each parent to these `vectors`. Finally, the `map` to be returned (`recombined`) is populated with individual *i*'s alleles. I leave it for the reader to decipher the mechanics of the function `print_data`, simply stating that the function calculates LD coefficient *D* for each unique pair of loci and prints to file.

5.2.2 *Results*

I ran `multilocus_ld` two times each for $N_e = 10{,}000$ and $N_e = 100$. For the larger population size, the decay of LD among all three pairs of linked loci closely follows the decay of LD expected according to Eq. 5.5 (Fig. 5.8, left-hand panels). When $N_e = 100$, however, much greater generation-to-generation variance of LD coefficients is observed. The results for D_{ab} (Fig. 6.7, upper right-hand panel) are illustrative. In the case of the simulation tracked by a gray line, D_{ab} declines rapidly and has dropped from 0.25 to around -0.075 by 175 generations, after which it rapidly rises to nearly 0.1 by 200 generations. The simulation tracked by the black line hews closely to the expected line, but just after 150 generations flat lines at $D_{ab} = 0$ due to the loss of an allele. The frenetic behavior and greater disparity between observation and expectation of the small population's values of *D* are the results of genetic drift. Small population size leads to large swings in allele frequencies, which are components of Eq. 5.1. In the most extreme case, one of the alleles is lost from the population. Because the program calculates all LD coefficients with reference to the 1–1 haplotype, if the "1" allele is lost at locus *a* or *b* all terms in Eq. 5.1 are zero and D_{ab} therefore remains zero.

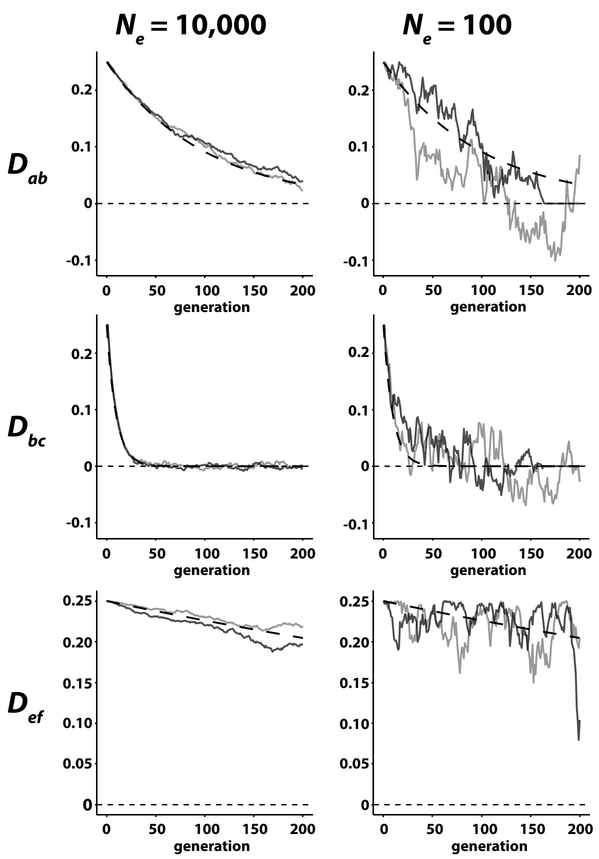


Fig. 5.8 Decay of linkage disequilibrium (LD) between three pairs of *linked* loci for effective population sizes of 10,000 and 100. In each graph, the dashed curve shows the expected decay of LD according to Eq. 5.5. In each case, results of two independent simulations are shown (gray and black solid lines) and $D = 0.25$ at generation 0. When $N_e = 10{,}000$ results conform to theoretical expectations rather well (left column); this is not true of the small $N_e = 100$ population (right column). Both D_{ab} and D_{bc} ($N_e = 100$) remain at 0 in one of the two replicates due to loss of an allele

What about the LD coefficients between *unlinked* loci? Take D_{af} as an example. Like all of the loci, D_{af} begins at 0.25, but should rapidly decline since $r = 0.5$ among unlinked loci. Table 5.1 shows the value of D_{af} for the first ten generations of evolution for the same four simulations summarized in Fig. 5.8. When $N_e = 10{,}000$, simulated decay of LD among unlinked loci follows expectations closely. On the other hand, D_{af} can oscillate somewhat dramatically when effective population size is only 100. I have bold-faced a few cells in the $N_e = 100$ simulation results to show that, even among unlinked loci, the influence of genetic drift allows appreciable values of LD to reemerge.

Table 5.1 Expected and simulated decay of LD among *unlinked* loci *a* and *f*. Bold-faced values highlight notable deviations from the long-term decline of LD due to the imperfect sampling associated with small population size

		$\mathbf{N_e = 10{,}000}$		$\mathbf{N_e = 100}$	
Generation	Expected	Sim 1	Sim 2	Sim 1	Sim 2
0	0.2500	0.2500	0.2500	0.2500	0.2500
1	0.1250	0.1280	0.1265	0.0920	0.1095
2	0.0625	0.0655	0.0640	0.0708	0.0632
3	0.0313	0.0277	0.0301	0.0292	0.0331
4	0.0156	0.0150	0.0110	-0.0145	0.0129
5	0.0078	0.0076	0.0036	0.0122	0.0123
6	0.0039	0.0048	0.0024	-0.0140	**0.0421**
7	0.0020	0.0046	-0.0025	0.0087	**0.0238**
8	0.0010	0.0017	-0.0013	**0.0250**	0.0120
9	0.0005	0.0027	0.0003	**0.0341**	**0.0290**

(1)
The output from `multilocus_ld` shows declining D for *all* pairs of loci, including those that are unlinked such as loci a and f (Fig. 5.7a). This happens despite the fact that function `recombine( )` only resolves recombination via crossing-over. Look at the definition of `recombine( )` and work out why D for unlinked loci declines each generation (as it should). In other words, how is independent assortment accomplished in this code?

(2)
Add to the code of `multilocus_ld.cc` to simulate epistatic selection in which the relative fitness of the four possible genotypes at loci a and b are $w_{0-0} = 1$, $w_{1-1} = 0.975$, $w_{0-1} = 0.95$, and $w_{1-0} = 0.95$.

5.3 Forward Simulation of Crossing-Over Along a Sequence

We now turn our attention to modification of the forward simulation program FORTUNA. Specifically, we will add the potential for crossing-over along the simulated sequence. The simulation coded in Sect. 5.2 modeled the occurrence of crossing-over in a binary manner (i.e., either crossing-over occurred between loci a and b or it did not). Because we will now work with sequences holding many polymorphic loci rather than pairs of polymorphic loci, added realism is required. Specifically, we must now consider the possibility of multiple cross-over events between any two loci that affect the haplotype of the transmitted chromosome. To do so, we will model the number of cross-over events using a Poisson distribution for the same reasons we used a Poisson-distributed random variable to model mutation:

probability of cross-over at a specific nucleotide is very small and the number of nucleotides (each a chance/trial for occurrence of cross-over) are quite large. This adjustment relative to what we did in Sect. 5.2 is important because an even number of cross-overs between two loci will not change the combination of alleles *at* these loci. Failure to account for this fact would lead to simulation output that harbors inaccurate recombinant haplotypes. In particular, alleles at distantly linked loci would switch too often.

For the reader familiar with basic experimental genetics, a review of the three-point test cross will remind her that empirical estimation of recombination rate requires correction for those small number of instances where a transmitted chromosome has incurred two cross-overs between the two outer loci. For example if three loci are found in the order *a-b-c* and one cross-over occurs between *a* and *b* during the same meiosis that a cross-over occurs between loci *b* and *c*, test cross results will provide evidence for recombination within the *a-b* and *b-c* intervals but initially fail to identify the fact that *two* cross-overs occurred between the interval *a-c*. By allowing for the possibility of multiple cross-overs and ensuring that these rare events are resolved correctly, we obtain haplotypes that are in keeping with what we would expect to find in nature.

5.3.1 *Additional Parameters Required to Model Recombination Along a Sequence*

Two boolean parameters—`useRec` and `useHotRec`—will be used to control whether the simulation incorporates recombination and a recombination hotspot, respectively. Baseline recombination rate and the recombination rate at the hotspot are stored in the parameters `recrate` and `hotrecrate`, respectively. In the case that a recombination hotspot is simulated, the starting and ending positions of the hotspot within the simulated sequence are specified by `hotrecStart` and `hotrecStop`. Beginning with this iteration of FORTUNA, we introduce code that calculates window-based statistics instead of statistics for the entire sequence. This requires three additional parameters: (1) `getWindowStats`, which instructs the program to calculate window-based statistics or not; (2) `windowSize`, which specifies the length in base pairs of each window; and (3) `windowStep`, which specifies the number of base pairs to slide the window forward. The next listing shows the updated `parameters` file as well as the values that will be used to simulate the results detailed at the end of this section. Note that we are modeling a population of constant size, which is why the `demography` parameter is set to 0.

parameters

```
popsize 10000
mutrate 1e-08
useRec 1
```

```
useHotRec 1
recrate 1e-09
hotrecrate 1e-06
hotrecStart 80000
hotrecStop 81999
seqlength 2e05
sampsize 100
sampfreq 25
demography 0
dem_parameter 0
dem_start_gen 0
dem_end_gen 100002
carrying_cap 0
useMS 1
getWindowStats 1
windowSize 10000
windowStep 5000
mscommand ./ms 20000 1 -t 80 >ms_output
```

The next two listings provide the additional code added to files `params.h` and `params.cc` to properly load values of the new parameters provided in the `parameters` file.

params.h: additions for this chapter

```
extern double recrate;
extern double hotrecrate;
extern bool useRec;
extern bool useHotRec;
extern int hotrecStart;
extern int hotrecStop;
extern bool getWindowStats;
extern int windowSize;
extern int windowStep;
```

params.cc: additions for this chapter

```
// new parameter declarations
int hotrecStart, hotrecStop, windowSize, windowStep;
double recrate, hotrecrate;
bool useRec, useHotRec, getWindowStats;

// populate parameters with values
recrate = atof(parameters["recrate"].c_str());
hotrecrate = atof(parameters["hotrecrate"].c_str());
hotrecStart = atoi(parameters["hotrecStart"].c_str());
hotrecStop = atoi(parameters["hotrecStop"].c_str());
windowSize = atoi(parameters["windowSize"].c_str());
windowStep = atoi(parameters["windowStep"].c_str());
useRec = atoi(parameters["useRec"].c_str());
useHotRec = atoi(parameters["useHotRec"].c_str());
getWindowStats = atoi(parameters["getWindowStats"].c_str());
```

5.3.2 *Modifying* `population.h`

The primary changes to `population.h` include (1) modification of the function `get_sample()` to allow calculation of window-based summary statistics; (2) a new function `recombine()` that simulates cross-over events; and (3) necessary changes to the class constructor.

In the constructor, conditional `ifs` are used to control whether or not the total recombination pressure (`r_sequence`; line 2) should be calculated. If recombination in the absence of a hotspot is modeled, `r_sequence` is simply the `recrate` times `seqlen` (line 59). On the other hand, if a hotspot is modeled, calculation of `r_sequence` requires consideration of the disparate recombination rates and the length of sequence for which each rate holds (lines 55–58). With `r_sequence` in hand, the Poisson random variable generator (`randomrec`; line 3) is parameterized in line 60. In addition, we add lines that open the output file and print the appropriate column headers depending on whether summary statistics will be calculated for windows (lines 63–68) or for the entire sequence as in earlier chapters (line 69). In the case of window-based summary statistics, the column headings consist of the generation, the specific statistic, and the mid-point of each window. For example, if a 200,000 bp sequence is simulated and windows 10,000 bp wide are used with a step size of 5000 bp, the column headings for windows will run from *w5000* for the [0,10000]bp window to *w195000* for the [190000, 200000]bp window.

Regarding summary statistic calculation, the function `get_sample()` is modified to allow for calculation of window-based statistics. Lines 7–8 use the populated `map<int, int> allele_counts`, where the key is the position and the value is the number of sequences in the sample with the derived allele at that position, to populate the `positions` vector with ordered positions of alleles. The block executed when `getWindowStats` is `true` (lines 10–19) calls the `get_window_stats()` function defined in `summarystats.h` and prints the results (returned as the `map stats`) to the summary statistics output file. Note that the key in `stats` is the name of the statistic (e.g., S) and its value is a vector of this statistic calculated for each window (see `summarystats.h` listing below for detail).

The intrasimulation constructor for class `Individual`, called within the function `reproduce()` includes a call to the new function `recombine()` as its last argument (line 49). Function `recombine()` (lines 26–46) determines the positions of break points (i.e., chiasmata) formed between the two chromatids already selected for transmission to the new individual in function `mutate()`. If `useRec` is `false`, an empty vector is returned as an argument to the Individual constructor and no recombination will take place. On the other hand, if recombination is simulated, the number of chiasmata are drawn from `randomrec` (line 29). The block in lines 31–39 is executed, which implements the bias toward break points within the hotspot via Eq. 5.7 as detailed in section 5.1.3. In the absence of a recombination hotspot, random break points along the sequence length are chosen from randompos (line 40). Finally, the break points are sorted in ascending order (line 43) and returned to the intrasimulation constructor (line 45).

population.h: additions for this chapter

```
 1  // additions to private variables
 2  double r_sequence;
 3  poisson_distribution<int> randomrec;
 4
 5  // additions, modifications to private function get_sample()
 6     vector<int> positions;
 7     for (auto iter = allele_counts.begin(); iter != allele_counts.end();
           ↪ ++iter)
 8        positions.push_back(iter->first);
 9
10     if (getWindowStats) {
11        map<string, vector<double> > stats = get_windowStats(positions,
              ↪ sample, S); // function to be added to summarystats.h
12        for (auto iter = stats.begin(); iter != stats.end(); ++iter) {
13           sumstat_file << gen << " ";
14           sumstat_file << iter->first << " ";
15           for (auto iter2 = (iter->second).begin(); iter2 !=
                 ↪ (iter->second).end(); ++iter2)
16              sumstat_file << *iter2 << " ";
17           sumstat_file << endl;
18        }
19     } else { // previous code in this block
20        double pi = get_pi(sample);
21        ...
22        sumstat_file << gen << " " << pi << " " << watterson << " " <<
              ↪ tajimasd << endl;
23     }
24
25   // new private function recombine()
26   vector<int> recombine() {
27     vector<int> breakpoints;
28     if (useRec) { /// if false, empty vector passed to Individual
           ↪ constructor
29        int chiasmata = randomrec(e);
30        for (int i=0; i<chiasmata; ++i) {
31           if (useHotRec) {
32              double c = randomnum(e);
33              if (c < hotrecStart * recrate / r_sequence )
34                 breakpoints.push_back( c * r_sequence / recrate );
35              else if (c < (hotrecStop*hotrecrate -
                    ↪ hotrecStart*(hotrecrate-recrate)) / r_sequence)
36                 breakpoints.push_back( (c*r_sequence +
                       ↪ hotrecStart*(hotrecrate - recrate)) / hotrecrate);
37              else
38                 breakpoints.push_back( (c*r_sequence -
                       ↪ (hotrecStop-hotrecStart)*(hotrecrate-recrate) ) /
                       ↪ recrate );
39           } else {
```

```
            breakpoints.push_back(randompos(e));
         }
      }
      sort(breakpoints.begin(), breakpoints.end());
   }
   return breakpoints;
}

// modification to reproduce() line
individuals.push_back(new Individual(individuals[parents[0]],
     ↪ individuals[parents[1]], mutate(parents, gen), recombine()) );

// additions, modifications to constructor
Population () {
...
   if (useRec) { // new block
      if (useHotRec) {
         int hotspot_length = hotrecStop - hotrecStart + 1;
         r_sequence = ( hotspot_length * hotrecrate ) + ((seqlength -
              ↪ hotspot_length) * recrate);
      } else
         r_sequence = seqlength * recrate;
      randomrec.param(poisson_distribution<int>::param_type(r_sequence));
   }
...
   if (getWindowStats) { // new block
      sumstat_file << "gen stat ";
      for (int w=0; w + windowSize <= seqlength; w += windowStep)
         sumstat_file << "w" << w+windowStep << " ";
      sumstat_file << endl;
   } else
      sumstat_file << "gen pi watterson tajimasd" << endl; // old line
...
}
```

5.3.3 *Modifying* `individual.h`

Resolution of the break points determined in the `Population` class function `recombine()` occurs during instantiation of the new object of class `Individual`. The modified intrasimulation constructor accepts the break points as input (next listing: line 35). If a transmitted sequence incurs a new mutation (line 38), the positions of derived alleles in the sequence are sorted in ascending order (line 40). After mutation is resolved, if one or more break points were passed to the constructor, the new combination of alleles in the transmitted sequences are determined by a call to the new function `resolve_crossover()` (lines 2–32).

The work of the rather complicated function resolve_crossover() is summarized in Fig. 5.9 for the case of three break points. Essentially, the break points divide each of the transmitted sequences into numbreaks + 1 segments. Then, every other segment (really, its derived alleles), beginning with the second segment of each of the sequences, is swapped between sequences. Figure 5.9a shows the setup, while Fig. 5.9b shows its resolution for the specific case illustrated.

At the start of the function, the number of break points is calculated as numbreaks (line 3), iterators for determination of segments are declared (line 4), a map to hold the derived allele positions of each segment, and a vector to store recombined sequences are declared (lines 5–6). Next, we loop through the two transmitted sequences in turn (lines 8–18). The iterator lower is initialized with the index of the first derived allele in a sequence (line 9), after which a loop through the break point positions (lines 10–15) identifies the index of the last derived allele position that is less than the position of the next break point (stored as iterator upper) using the standard library algorithm upper_bound() (line 11). The derived allele positions of the segment are stored in the dummy vector newvec using standard library function assign( ), lower is set to upper, and the derived alleles of the segment are added to the segments map (lines 12–14). Finally, any derived allele positions in the last segment downstream of the last break point are determined (lines 16–17).

The final loop of function resolve_crossover (lines 20–31) loops numbreaks + 1 times to build the recombined sequences. In the first pass, when the loop variable i == 0, recombined sequences are first initialized with the derived allele positions of each sequence's first segment (line 22). In each subsequent iteration, derived allele positions of the opposite sequence's segment are appended to the recombined sequences if i%2 == 0 (lines 23–27) or the original derived allele positions of the sequence are appended (lines 27–30). The return value of this function is void because sequences are private variables of class Individual and the function has modified these variables internally.

individual.h: additions and modifications

```
1  // new private function
2    void resolve_crossover (vector<int>& breaks) {
3      int numbreaks = breaks.size();
4      vector<int>::iterator lower, upper;
5      map<int, vector<int> > segments;
6      vector<int> newvec;
7
8      for (int seq = 0; seq < 2; ++seq) {
9        lower = sequences[seq].begin();
10       for (int i=0; i<numbreaks; ++i) {
11         upper = upper_bound(sequences[seq].begin(),
             ↪ sequences[seq].end(), breaks[i]);
12         newvec.assign(lower, upper);
```

```
            lower = upper;
            segments[i + seq*(numbreaks+1)] = newvec;
         }
         newvec.assign(lower, sequences[seq].end());
         segments[numbreaks + seq*(numbreaks+1)] = newvec;
      }

      for(int i=0; i<= numbreaks; ++i) {
         if (i%2 == 0) {
            if (i == 0) {sequences[0] = segments[0]; sequences[1] =
                 ↪ segments[numbreaks+1];}
            else {
               sequences[0].insert(sequences[0].end(), segments[i].begin(),
                    ↪ segments[i].end());
               sequences[1].insert(sequences[1].end(),
                    ↪ segments[i+numbreaks+1].begin(),
                    ↪ segments[i+numbreaks+1].end());
            }
         } else{
            sequences[0].insert(sequences[0].end(),
                 ↪ segments[i+numbreaks+1].begin(),
                 ↪ segments[i+numbreaks+1].end());
            sequences[1].insert(sequences[1].end(), segments[i].begin(),
                 ↪ segments[i].end());
         }
      }
   }

// modifications to intrasimulation constructor
Individual (Individual *p1, Individual *p2, vector<vector<int> >
     ↪ mutation_results, vector<int> breakpoints) { // modified line
...
   for (int i=0; i<2; ++i) {
      for (int j=1; j<mutation_results[i].size(); ++j) {
         if (mutation_results[i][j] > 0) {
            sequences[i].push_back(mutation_results[i][j]);
            sort(sequences[i].begin(), sequences[i].end()); // new line
         }
         else
            remove_allele_by_position(i, -1 * mutation_results[i][j]);
      }
   }

   if (breakpoints.size() > 0) // new line
      resolve_crossover( breakpoints ); // new line
}
```

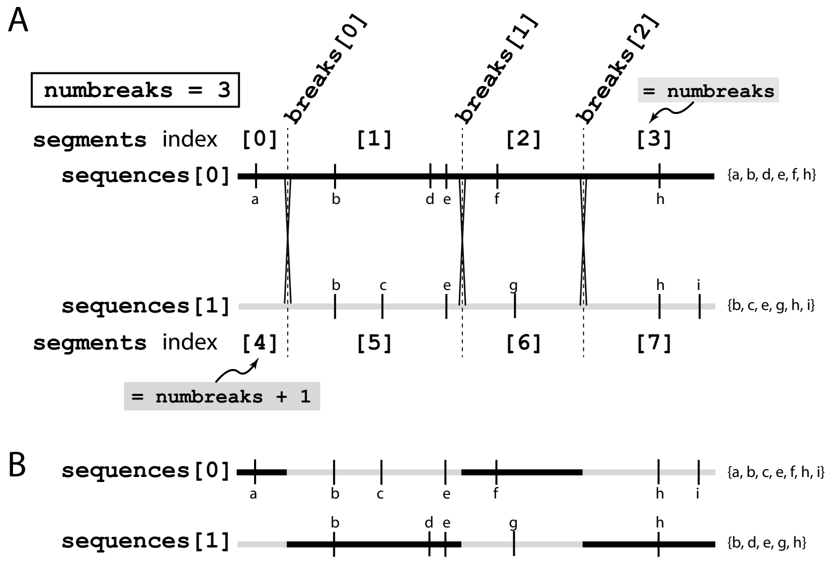


Fig. 5.9 Resolution of cross-over events between the two transmitted `sequences` that define the genetic variation of a newly instantiated object of class `Individual`. (**a**) Break point positions are stored in `vector breaks`. Lower-case letters represent positions of derived alleles on the prerecombination `sequences`. Function `resolve_crossover()` first breaks each of the `sequences` into `segments` that hold the derived allele positions of each segment; for example, `segments[1]` would hold the integers corresponding to the positions of derived alleles *b*, *d*, and *e*. (**b**) The recombined chromosomes transmitted to the new object of class `Individual` are built segment-by-segment, with every other segment switching places between `sequences`

5.3.4 *Window-Based Summary Statistics and Calculating the Number of Haplotypes K in* `summarystats.h`

File `summarystats.h` receives a new function—`get_windowStats()`—that calculates summary statistics for each window defined by the `extern` parameters `windowSize` and `windowStep`. In addition, we will now calculate a new summary statistic—the number of unique haplotypes, *K*—for each window. Because haplotypes are specific combinations of alleles along a sequence and crossing-over shuffles these combinations, it stands to reason that *K* is at least partially a function of crossing-over.

Function `get_windowStats()` is declared on line 2 and defined on lines 5–47. The function is called by a `Population` object and returns a `map` that keys the name of a specific summary statistic (as a `string`) to the values of that statistic for each window as a `vector<double>`. Line 6 declares the variable `stats` that will be returned, followed by the declaration of three variables internal to the function: (1) the `map haplotypes`, which stores the haplotype of each window as a `string` and has *sequence number as key*; (2)

`windowBitSample`, which stores the haplotypes of each window as a `vector` of `bitset` and has *window number as key*; and (3) `windowS`, which stores the value of S for each window.

Next, the function iterates through the sample of sequences passed by reference to `get_windowStats( )` (lines 10–29). Lines 11 and 12 convert the `bitset` representation of the entire sequence to a `string` and trim the string to the size of the number of segregating sites in the sample, S, respectively. Variable `haps` (line 13) temporarily stores the haplotypes of each window for the sequence under consideration and `count` (line 14) will keep track of window number. With these preliminaries out of the way, the `for` loop on lines 15–27 iterates through each window. Variable `wl` is the lower limit of the window and controlled by the `for` loop (line 15), while variable `wu` is the upper limit on the window (line 16). Lines 17–18 use `upper_bound()` to find the iterators pointing to the first entry in `positions` that is greater than `wl` and `wu`, respectively. To make sense of these lines, recall that the variable `positions` passed by reference to `get_windowStats()` is a `vector<int>` holding the *ordered* positions of segregating sites in the sample. The iterators are then converted to integers, which thanks to zero-based indexing correspond to the positions in the substring `b` (line 12) that contain segregating sites within the currently considered window (lines 19–20). The `string`-formatted haplotype for this window is added to `haps` and the `bitset` version is added to `windowBitSample` (lines 22–23). Because S for each window is constant across the population, window-specific S is only calculated on the first pass through the loop (lines 25–26). Finally, the `vector<string>` of haplotypes for this sequence's windows is assigned to the `map haplotypes`.

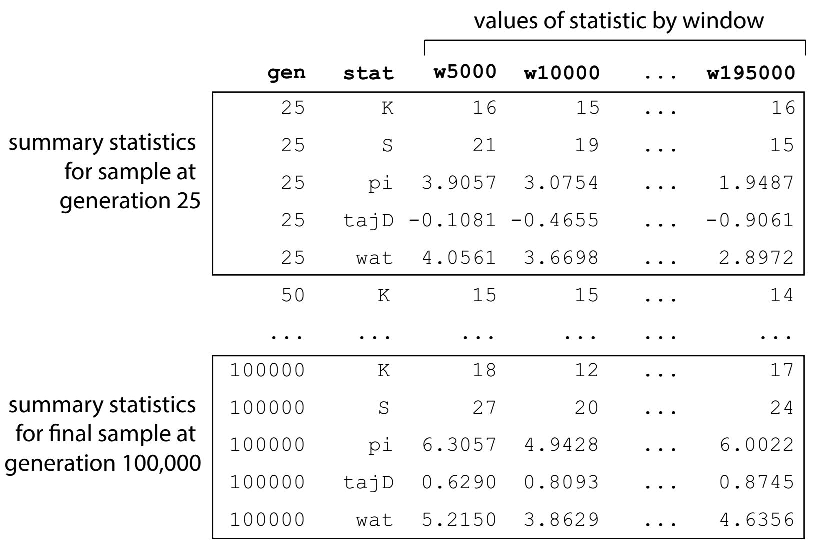
On line 32, a new `map`, `haplotype_counts` is declared, which is used in the subsequent `for` loop to count the frequency of each unique haplotypes for a given window. This frequency count is not used in this chapter, but sets the stage for its use in subsequent chapters. The `for` loop (lines 33–45) first declares a `map`, `counts_this_window`, that takes a `string` representation of a haplotype whose value is the number of times this haplotype is counted in the sample and is populated on lines 35–36. Note that `haplotypes[j][i]` is the `string`-based haplotype of sequence j and window i. Lines 38–40 allow window-specific calculation of π and θ_W, while lines 30, 37, and 41–43, populate the `map`, `stats`, with the results that are passed back to the `Population` object (line 46) for printing to the output file.

summarystats.h: additions for this chapter

```
1  // new function declaration
2  map<string, vector<double> > get_windowStats( vector<int> &positions,
       ↪ vector<bitset<bitlength>> &sample, int S );
3
4  // new function definition
5  map<string, vector<double> > get_windowStats( vector<int> &positions,
       ↪ vector<bitset<bitlength>> &sample, int S ) {
6    map<string, vector<double> >stats;
```

```
    map<int, vector<string> > haplotypes;
    map<int, vector<bitset<bitlength> > > windowBitSample;
    vector<double> windowS;
    for (int i = 0; i < sample.size(); ++i) {
        string a = sample[i].to_string();
        string b = a.substr(bitlength-S, S);
        vector<string> haps;
        int count = 0;
        for (int wl = 0; wl+windowSize<= seqlength; wl += windowStep) {
            int wu = wl + windowSize;
            vector<int>::iterator lowindex = upper_bound(positions.begin(),
                ↪ positions.end(), wl);
            vector<int>::iterator upindex = upper_bound(positions.begin(),
                ↪ positions.end(), wu);
            int l = lowindex - positions.begin();
            int u = upindex - positions.begin();
            haps.push_back(b.substr(l, u-l));
            bitset<bitlength> btemp (b.substr(l,u-l));
            windowBitSample[count].push_back(btemp); // by window #
            count++;
            if (i == 0)
                windowS.push_back(u-l);
        }
        haplotypes[i] = haps;
    }
    stats["S"] = windowS;
    int windowCount = haplotypes[0].size();
    map<int, map<string, int> > haplotype_counts; // key is window number,
        ↪ internal map consists of string and its counts for this window
    for (int i = 0; i<windowCount; ++i) {
        map<string, int> counts_this_window;
        for (int j=0; j<sample.size(); ++j)
            counts_this_window[haplotypes[j][i]]++;
        stats["K"].push_back(counts_this_window.size());
        double pi, wat;
        pi = get_pi(windowBitSample[i]);
        wat = get_watterson(windowS[i]);
        stats["pi"].push_back(pi);
        stats["wat"].push_back(wat);
        stats["tajD"].push_back(get_tajimas_d(pi, wat, windowS[i]));
        haplotype_counts[i] = counts_this_window;
    }
    return (stats);
}
```

It is important to note that the output file will contain several contiguous rows with results from the same sampling generation. A portion of the output from a simulation run for which window-based summary statistics were calculated is shown in Fig. 5.10. Each sample produces five lines of results in the output file—namely, window-specific values of the summary statistics K, S, π, Tajima's D, and θ_W. It is convenient to have one output file



```
                        values of statistic by window
                        ┌────────────────────────────────────┐
   gen    stat   w5000   w10000    ...   w195000
    25       K      16       15    ...        16
    25       S      21       19    ...        15
    25      pi  3.9057   3.0754    ...    1.9487
    25    tajD -0.1081  -0.4655    ...   -0.9061
    25     wat  4.0561   3.6698    ...    2.8972
    50       K      15       15    ...        14
   ...     ...     ...      ...    ...       ...
100000       K      18       12    ...        17
100000       S      27       20    ...        24
100000      pi  6.3057   4.9428    ...    6.0022
100000    tajD  0.6290   0.8093    ...    0.8745
100000     wat  5.2150   3.8629    ...    4.6356
```

summary statistics for sample at generation 25
summary statistics for final sample at generation 100,000

Fig. 5.10 Annotated partial output file that demonstrates the structure of window-based summary statistic reports output to the file `sumstats` by FORTUNA

and, when processing the results in R, it is simple to split the overall data set into separate objects for each statistic based on the `stat` column.

5.3.5 *Results*

I ran three distinct forward simulations for comparison: (1) recombination hotspot at 80–82 kbp along a 200 kbp segment using the parameter values of the `parameters` file listed in Sect. 5.3.1; (2) uniform recombination rate, implemented by changing parameter `useHotRec` to 0 in the `parameters` file; and (3) no recombination, implemented by changing parameter `useRec` to 0 in the `parameters` file.

5.3.6 *Effect of Recombination on the Number of Unique Haplotypes,* K

The presence of a recombination hotspot greatly increases the number of unique haplotypes (K) locally but not globally. Figure 5.11a shows the distribution of K for overlapping 10 kbp windows across the 4000, $n = 100$ samples drawn from a simulated population each 25 generations over a 100,000-generation span. Throughout the simulation, the windows span-

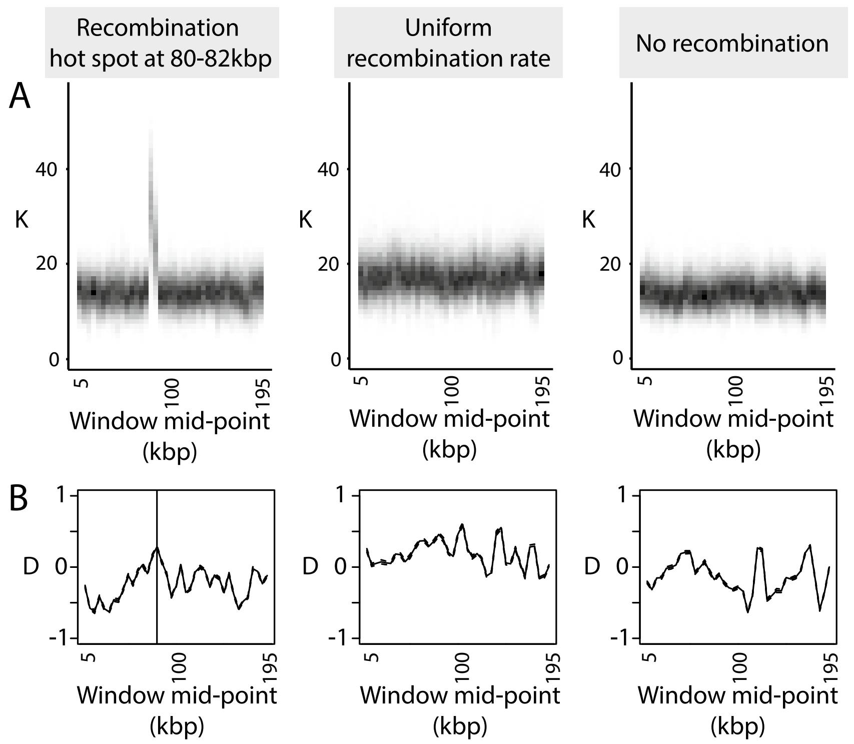


Fig. 5.11 *K* and Tajima's *D* for overlapping 10 kbp windows under three different scenarios: recombination hotspot, uniform recombination rate, and no recombination. (**a**) The distribution of *K* at each window across the 4000 samples taken every 25 generations. (**b**) The mean value of Tajima's *D* at each window across the 4000 samples. About 95% confidence intervals on the mean estimates are shown as dashed lines but are very narrow

ning base pairs 75–85 kbp and 80–90 kbp possess roughly two times or more haplotypes than other windows. These are the two windows that contain the simulated recombination hotspot at 80–82 kbp. Uniform recombination rate increases *K* globally relative to sequences simulated in the absence of recombination, although the increase is rather slight (cf. two rightmost panels of Fig. 6.10a). Before moving on to panel B of Fig. 5.11, we take a moment to detail an R script that uses window-based output from FORTUNA in the file `sumstats` to generate plots of the type shown in Fig. 5.11a.

5.3.7 Visualizing the Distribution of a Summary Statistic Across Simulations and by Window

The output file provides us with a time series of data for multiple windows within the simulated sequence. This is good in that we have a multitude of data to contemplate and analyze. However, the very first step in our exploration of the data—producing effective visualizations of the data—is a difficult one. Our decision regarding the type of visualization we produce depends most on the message we are attempting to convey. In the case of Fig. 5.11a, I hoped to show as holistic a summary of the results as possible. I did not want to simply graph the mean value of K across all 4000 replicates versus mid-window position. Rather, I wanted to show variance across replicates, which meant I needed to display a *distribution*. Notice the difference between panels A and B of Fig. 5.11. In Fig. 5.11b, I do graph the mean of a statistic; it is distinctly less informative than panel A in my opinion. I chose to use a tiled heat map. The column of colored tiles corresponding to a given window on the x-axis visualizes the distribution of K values across the 4000 replicate simulations. The tiles are colored according to a white-black color gradient, with light colors indicating low density and dark colors indicating high density.

The following R script includes one function whose parameter is the name of summary statistics output file named `sumstats`.

heatmap.r

```r
library(ggplot2)
library(cowplot)
library(reshape2)

heatmap <- function(datafile="sumstats") {
  d <- read.table(file = datafile, header = T);
  dd <- split(d, d$stat)
  size <- dim(dd$K)
  numcol = size[2] - 2
  kmat <- dd$K[,3:size[2]]
  kmat2 <- matrix(0, nrow = max(kmat)+5, ncol = numcol)
  for (i in 1:size[1]) {
    for (j in 1:numcol) {
      kmat2[kmat[i,j],j] <- kmat2[kmat[i,j],j] + 1
    }
  }
  kmat2.melted <- melt(kmat2)
  ggplot(kmat2.melted, aes(x = Var2, y = Var1, fill = value)) +
      ↪ geom_tile() + coord_equal() + scale_fill_gradient(low="white",
      ↪ high="black")
}
```

After reading in the data from `sumstats`, the function `heatmap( )` splits the data by type of summary statistic. Although the output file is a mixture of results for different summary statistics (Fig. 5.10), the base `split()` function allows us to quickly partition the data into statistic-specific data frames. For example, the object `dd` (line 7) can be used to access the data for Tajima's *D* and *K* by using `dd<currencydollar>tajD` and `dd<currencydollar>K`, respectively. In this case, we focus on `dd<currencydollar>K`, as *K* is the statistic of interest. On line 10, the first two columns of `dd<currencydollar>K` are removed because these hold the generation number and statistic type, which are not needed to generate the heat map of *K* alone. Next, on line 11, we create a matrix of zeroes with column number equal to five greater than the maximal value of *K* found across all samples and windows and row number equal to window number. The nested `for` loops on lines 12–16 populate this matrix, `kmat2`, which we subsequently "melt" on line 17. `kmat2.melted` produces a data frame with three columns: (1) row number, i.e., *K*; (2) column number, i.e., window number *w*; and (3) value, i.e., how many times *K* unique haplotypes were found at window *w*. Note that in the `ggplot` function, we use window number (the second column of `kmat2.melted` as the *x*-axis variable because we want sequence position to run along the *x*-axis. To utilize this function:

- open R
- set the working directory to the location of this script and the output file `sumstats`
- load the function using `source("heatmap.r")`
- run `heatmap( )`

5.3.8 *Effect of Recombination on Tajima's* D *and Simulation as Exploration*

For the same set of three simulations—recombination with hotspot, uniform recombination, and no recombination—recombination alone does not yield statistically significant values of Tajima's *D* on average (Fig. 5.11b). However, close examination of these plots shows that (1) mean values of *D* for 10 kbp windows are, on average, greater in the presence of uniform recombination than in the case of no recombination and (2) the greatest mean value of *D* in the presence of a recombination hotspot coincides with the position of the hotspot. Given that the numerator of Tajima's *D* is $\pi - S/a_n$, these two points suggest that nucleotide diversity (π) is positively correlated with recombination rate. Empirical data from the honey bee *Apis mellifera*—a species with globally high recombination rate and great intragenomic variation in recombination rate—corroborate this correlation (Beye et al. 2006; Liu et al. 2015).

This is a good example of the way in which simulations can be used to move theoretical considerations forward. Empirical genomic data are the end result of numerous (often confounding) evolutionary factors. In the simulations presented in this section, we have eliminated the influence of demographic change and natural selection to focus on the effects of recombination in isolation, which turn out to be increased K and slightly increased values of π and Tajima's D. This is not a strong result. It has not been verified through comprehensive exploration of parameter space. It may have no bearing on the theory of evolution. But I hope you can see that toying experimenting with some very basic code allows us to begin exploring ideas that intrigue us but are difficult or impossible to assess with pure mathematical analysis.

Ultimately, our ability to add and remove different evolutionary factors and to vary the parameter values of those factors we do include provides us with a rich, multidimensional space to explore. Simulation frees us a bit to explore somewhat further reaches of the model and parameter spaces. In turn, the hope is that insights gained from these computational experiments can be applied to and contextualized by empirical data from nature. The toy example presented here of course only hints at the deeper possibilities.

References

Beye M, Gattermeier I, Hasselmann M, Gempe T, Schioett M, Baines J, Schlipalius D, Mougel F, Emore C, Rueppell O, Sirvio A, Guzman-Novoa E, Hunt G, Solignac M, Page R (2006) Exceptionally high levels of recombination across the honey bee genome. Genome Res 16:1339–1344

Hedrick P (1987) Gametic disequilibrium measures: proceed with caution. Genetics 117:331–341

Lewontin R (1964) The interaction of selection and linkage. i. general considerations, heterotic models. Genetics 49:49–67

Lichten M, Goldman ASH (1995) Meiotic recombination hotspots. Annu Rev Genet 29:423–444

Liu H, Zhang X, Huang J, Chen JQ, Tian D, Hurst L, Yang S (2015) Causes and consequences of crossing-over evidenced via a high-resolution recombinational landscape of the honey bee. Genome Biol 16:15

6 Population Structure and Migration

There were other social consequences of the plague. After each successive epidemic wave had passed, the gene flow between classes increased in intensity. Cities found themselves depopulated and lowered their standards for citizenship. Venice, normally very closed to foreigners, now granted free citizenship to anyone who settled there for a year. Social mobility increased, as surviving elites needed to replenish their ranks with fresh blood. Relationships among cities altered because of the enormous demographic shifts wrought by the plague. The eventual emergence of Venice as the core of the network system was in no small measure a consequence of those demographic changes.[1]

– Manuel De Landa, *A Thousand Years of Nonlinear History*

6.1 Background and Theory

To this point we have modeled and simulated a single, *panmictic* population. When devising a model to simulate, however, we frequently encounter the need to model nonrandom mating due to mate choice based on phenotype, geographical barriers to effective migration, ecological barriers to dispersal, and many other causes. Imagine a room in which a solitary individual is smoking a cigarette in one corner. Let intensity of cigarette smoke odor be analogous to allele frequency. Due to diffusion of the odorant molecules throughout the room—a process analogous to random mixing of mates throughout a population—in short order all points in the room will smell of cigarette smoke at similar intensity (Fig. 6.1a). Similarly, given sufficient generations of random mating, allele frequencies will equilibrate at values

[1] Quoted with permission. 2000, Zone Books, Brooklyn, NY.

that hold across the entire population, i.e., there will not be pockets of individuals or sub-regions of the population range in which allele frequencies differ substantially from those of other social, ecological, or geographical stratifications of the population.

To put it in the terms of evolutionary genetics, a panmictic population does not show *population genetic structure*, more commonly referred to simply as *population structure*. In a panmictic population, you expect estimated allele frequencies to be the same regardless of the physical position within the population from which the sample was taken. If a population is not panmictic, genetic heterogeneity (i.e., structure) begins to emerge. Lack of genetic population structure in empirical data supports a null hypothesis of random mating and vice-versa. However, we must remember that population structure can emerge for a diversity of reasons.

Now consider two adjacent but separate rooms, one with a cigarette smoker and one in which 100 lemons are thrown and smashed against a wall. Because there is no flow of air between the rooms, the two smells diffuse to equilibrium within their respective rooms but there is no mixing of odors (Fig. 6.1b). This is analogous to *isolation by barrier*, which commonly leads to the fixation of distinct alleles (cigarette or citrus odor) in isolated populations (rooms). Now imagine you repeat the same scenario many times, progressively enlarging an opening in the wall between the two rooms. When the connection between the two rooms is very small (Fig. 6.1c), analogous to a modicum of gene flow, you detect a hint of cigarette smoke in the lemon room and vice versa. Once the connection between the two rooms becomes sufficiently large, however, the two rooms are as one (Fig. 6.1d); without additional information, you would be hard pressed to determine which room initially contained the smoker and which the lemons. These trials are analogous to increasing *gene flow* (aka, effective migration) between the two populations (rooms). Finally, consider a cigarette smoker smoking outside. From a sufficient distance, no smell of cigarette is detected. However, as you approach the smoker, the intensity of cigarette smell becomes stronger and stronger (Fig. 6.1e). This is analogous to *isolation by distance* (IBD), in which the frequency of an allele is autocorrelated with distance. This translates to a simple definition of IBD: individuals close to one another are more genetically similar to each other than to those farther away.

Classical models of discrete populations (or subdivisions of populations called *demes*), include *island models* as well as stepping-stone models of effective migration.[2] The key parameters in both cases are (1) the number of populations or demes, d, and (2) the migration rate m. The latter parameter is defined as the fraction of a deme's individuals this generation that are

[2] Throughout this text (and in the extended version of FORTUNA developed in this chapter), I will use the term deme and population interchangeably; my justification for this sloppy nomenclature is that you could model networks of populations (metapopulations) *or* semi-isolated subdivisions of one population (demes) using different parameter values in the same program, FORTUNA.

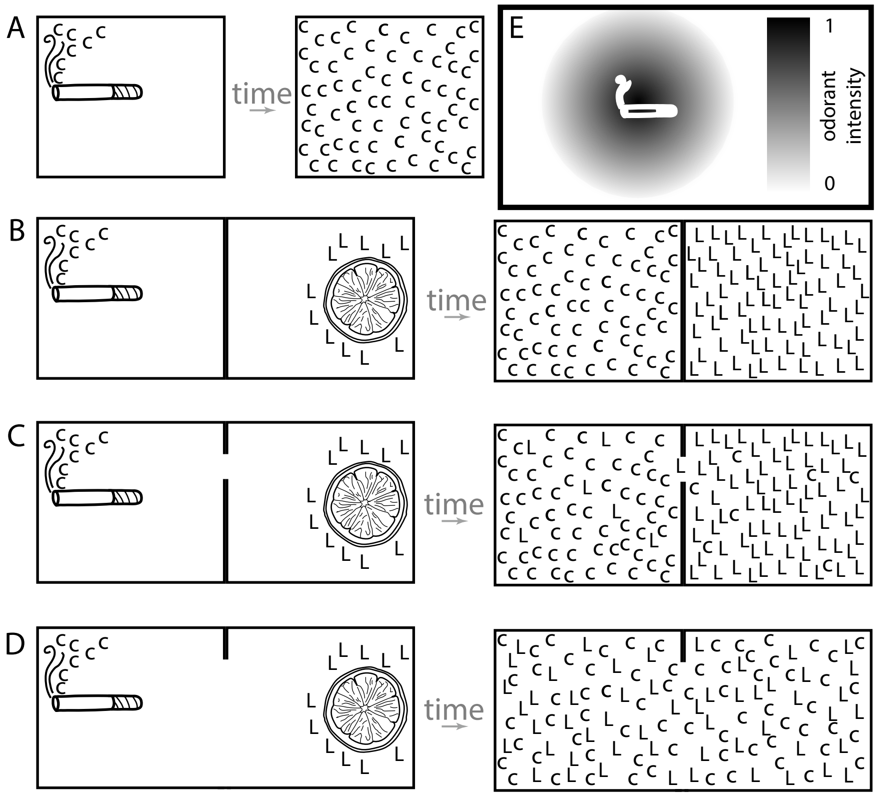


Fig. 6.1 Odorant diffusion as an analogy for gene flow. A burning cigarette releases C odorants, analogous to one allele, while smashed lemons release L odorants, analogous to an alternative allele at the same locus. (**a**) Analogy to a panmictic population. Given sufficient time, C odorants become evenly distributed within the enclosed room. (**b**) Analogy to isolation by barrier. In this case, an air-tight wall prevents the mingling of alleles/odorants. (**c**) Analogy to minimal gene flow. A small break in the barrier allows a small number of alleles/odorants to pass from one room to the other. (**d**) Analogy to high gene flow. Despite the separate origins of cigarette smoke and lemon alleles/odorants, given sufficient time, diffusion effectively eliminates the distinction between the rooms. With sufficient gene flow, the two rooms/demes become a single, panmictic population. (**e**) Analogy to isolation by distance. If a cigarette is smoked outside on a calm day, the intensity of the allele/odorant increases with decreasing distance to the source

offspring of parents present in another deme in the previous generation; another way to state this is that m is the fraction of a deme's population that emigrated from another deme. An island model is defined as a set of demes in which each deme is connected to all other demes by some, perhaps equal (or symmetric), value of m (Fig. 6.2a). A stepping-stone model is defined as a set of demes in which only those demes adjacent to each other may exchange migrants (Fig. 6.2b). Furthermore, for both the island and stepping-stone model, we can make the distinction between *conservative* and *proportional migration*. In the conservative case, demes exchange the

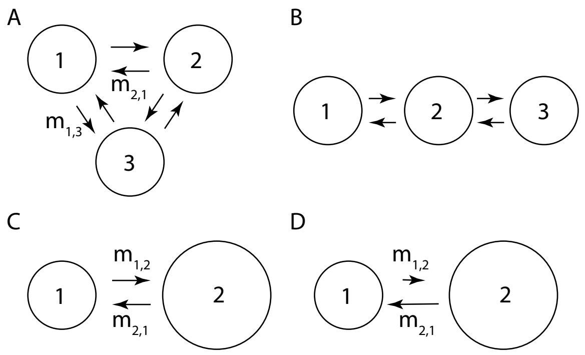


Fig. 6.2 Discrete models of migration. (**a**) A symmetric island model in which migration rates between all demes are equal. For example, $m_{1,3} = m_{2,1}$, where $m_{x,y}$ is the proportion of offspring this generation in population y derived from parents present in population x the previous generation. (**b**) A stepping stone model. Only adjacent demes—e.g., 1 and 2 as well as 2 and 3—exchange migrants. (**c**) Conservative migration in which migration rates are equal between demes even when N_e, proportional to the size of the circle denoting the deme, is unequal. (**d**) Proportional migration in which the number of emigrants from a deme to another is proportional to N_e of the source deme

same number of migrants, regardless of their population sizes (Fig. 6.2c); the model is conservative in the sense that this migration pattern will not change N_e of either population. In the proportional case, emigrants are released from a population as a function of N_e (Fig. 6.2d). Thus, a small population receives a proportionally greater number of emigrants than a large population. Given sufficient time, N_e of the two populations will become equal.

One of the more influential and intuitive models for thinking about population structure is the *structured coalescent*, (e.g., Nordborg 1997; Rice 2004, Fig. 6.3). In this case, demes might be viewed as semi-permeable containers where the genetic uniqueness of demes is negatively correlated with their porosity to effective migration (i.e., gene flow). When considering two random genes from the sample, there are now three possible events that could happen to either of the lineages in the previous generation (i.e., looking backwards): mutation, migration from one deme to another, or coalescence. Note that coalescence requires both lineages exist in the *same* deme at the *same* time. Interestingly, the formulas that identify f_{ii}—the probability that two genes sampled from the same deme are identical by descent—and f_{ij}—the probability that two genes randomly sampled from separate demes are identical by descent—take the form of moment-generating functions for the random variable of time (T). This means we can calculate the expected time to coalescence for genes sampled from the same deme (T_s) and those sampled from different demes (T_d) as well as the variances on these random variables.

In the case of symmetric migration where d demes are connected by the same rate of migration, m, and each population is of size N (Wakeley 2008):

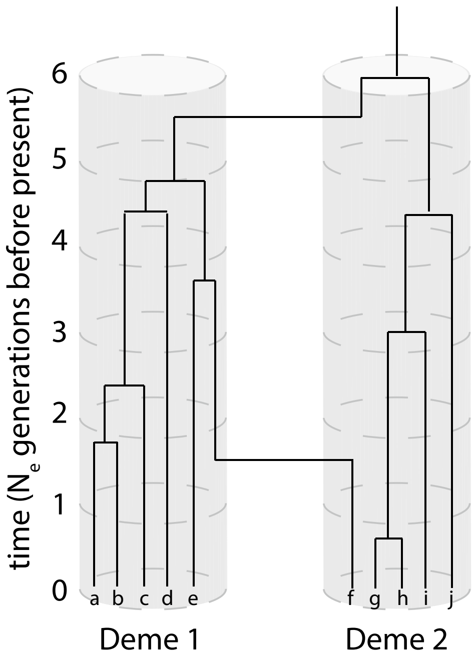


Fig. 6.3 A simple illustration of the structured coalescent modified from Fig. 3.13 of Rice (2004). In present time, genes $a - e$ are sampled from deme 1 and genes $f - -j$ are sampled from deme 2. In the illustrated case, lineage f moves/migrates from deme 1 to deme 2 roughly $1.5N_e$ generations in the past, while the lineage leading to the MRCA of genes $a - -f$ migrates to deme 2 roughly $5N_e$ generations ago. Theory underlying the structured coalescent provides the probability that (1) two randomly sampled genes from either deme 1 ($f_{1,1}$) or deme 2 ($f_{2,2}$) are identical by descent and (2) two randomly sampled genes, one from each deme, are identical by descent ($f_{1,2}$). As shown in this illustration and embodied in Eqs. 6.1 and 6.2, on average genes sampled from the same deme have shorter coalescent times (T_s) than genes sampled from separate demes (T_d). However, this illustration also shows that T_s for any pair of sequences sampled from deme 2 that includes gene f (e.g., f and i) is much greater ($6N_e$ generations) than the expected value of $E[T_s] = 2N_e d$, which equals $4N_e$ generations in this case with two demes

$$E[T_s] = 2Nd \tag{6.1}$$

$$E[T_d] = 2Nd + \frac{d-1}{2m}, \tag{6.2}$$

$$\mathtt{var}(T_s) = 4N^2d^2 + 2N\frac{(d-1)^2}{m}. \tag{6.3}$$

Similar, though often more complex, expectations may be found in the case of asymmetric migration.

6.2 Forward Simulation of Two Demes

In this chapter, we focus on the simulation of discrete demes separated from each other by user-specified migration rates, m_{ij}, which specify the fraction of individuals in deme j that are immigrants from deme i this generation.

6.2.1 Two Formerly Independent Demes Begin Exchanging Migrants

We start with a simple example, in which two demes of equal population size are generated by coalescent simulation. Then, beginning at the first generation of the forward simulation the two demes begin exchanging migrants at user-specified rates. We will use simulation of this example to corroborate the often stated rule of thumb that *if* $N_e m > 1$, *the two demes will act as one panmictic population* (Spieth 1974). $N_e m$ is an important compound parameter that has an intuitive interpretation as the *number of migrants* to the deme each generation. Consider Fig. 6.1b–d once again. We would like to confirm that one migrant between demes is the threshold at which the analogy illustrated in Fig. 6.1d manifests, i.e., an underwhelming barrier to gene flow leads to a panmictic population. Surely, $N_e m$ of 1 is too small to affect the genetic makeup of either deme.

Although the modeled scenario is simple, we must now simulate two objects of the `Population` class at once. In order to make this more manageable in the long run, we (1) create a `Metapopulation` class that holds a vector of pointers to the individual demes (each an object of class `Population`); (2) substantially revise *parameters* and *params.cc*; and (3) create a `Matrix` class that efficiently stores and accesses the matrix of migration rates between all demes. As in previous chapters, the changes will be explicitly documented, but I encourage you to visit this volume's website for the full source files used in this chapter as this time some rather radical changes are being made to the previously documented code. This may be particularly true of this chapter as the changes and additions are substantial.

6.2.1.1 Reading Parameters for More Than One Population

We begin with a full listing of the new form of the *parameters* file.

parameters // new format w/ global and deme-specific parameters

```
mutrate 1e-08
useRec 1
useHotRec 1
recrate 1e-09
```

```
 5 hotrecrate 1e-06
 6 hotrecStart 80000
 7 hotrecStop 81999
 8 seqlength 2e05
 9 sampsize 100
10 sampfreq 25
11 getWindowStats 1
12 windowSize 10000
13 windowStep 5000
14 pop_num 2
15 modelMigration 1
16 migration_rates 0. 0.00001 0.00001 0.
17 runlength 5000
18 DEME   /// 0
19 popsize 10000
20 demography 0
21 birthgen 0
22 extinctgen 25000
23 dem_parameter 0
24 dem_start_gen 0
25 dem_end_gen 100000
26 carrying_cap 0 0 0
27 useMS 1
28 mscommand ./ms 20000 1 -t 80 >ms_output
29 DEME   /// 1
30 popsize 10000
31 demography 0
32 birthgen 0
33 extinctgen 25000
34 dem_parameter 0
35 dem_start_gen 0
36 dem_end_gen 100000
37 carrying_cap 0 0 0
38 useMS 1
39 mscommand ./ms 20000 1 -t 80 >ms_output
```

Lines 1–17 are `extern` variables, global in the sense that these parameter values apply to all demes. While lines 1–13 and the parameter values they specify are of the same name and meaning as previously used, we now add *pop_num* to specify the number of demes simulated (line 14), the `bool modelMigration` that can switch simulation of migration off (0) or on (1) (line 15), *migration_rates* to populate the migration rate matrix (line 15), and *runlength* to specify the number of generations to run the simulation (line 17). Previously, this was entered as an argument to program execution. As shown, the parameter value of `migration_rates` will generate a migration matrix stored as an object of `Matrix` class and this form:

$$\begin{bmatrix} m_{0,0} & m_{0,1} \\ m_{1,0} & m_{1,1} \end{bmatrix} = \begin{bmatrix} 0 & 0.0001 \\ 0.0001 & 0 \end{bmatrix}$$

In other words, the first two numbers of `migration_rates` specify the first row of migration rates (from deme 0 to all demes), and the second two numbers the second row.

The remainder of `parameters` specifies a block of deme-specific parameter values for each deme, with each block led by a line that contains the uppercase word `DEME` (lines 18 and 29). Only the first `pop_num` demes specified will be considered by the program. Thus far, the only new parameters in these blocks are `birthgen` and `extinctgen`. `birthgen` is set to zero for both demes because simulation begins with both demes active/extant. Moreover, because we want to simulate each population for the duration of the program, `extinctgen` needs to be set to a value greater than or equal to `runlength`. However, these two parameters provide us with the capacity to introduce or eliminate populations intrasimulation. Note that `useMS` is set to 1 for both demes because we begin with coalescent-simulated populations in each case. Below, we will also create demes through population splitting or merger. We *could* set `mscommand` differently for the two populations with the warning that `popsize` must be adjusted accordingly and that `mutrate` is the same for both populations once the forward-in-time simulation begins; this means that the `-t` flag should also be adjusted appropriately in the `mscommand`. As an explicit example, if we wished to model deme 1 as consisting of 5000 rather than 10,000 diploid individuals, the appropriate `mscommand` would be `./ms 10000 1 -t 40 >ms_output`, which reflects the requirement of just 10,000 sequences and $\theta_{sequence} = 4N_e\mu = 4 \times 5000 \times 10^{-8} \times 200{,}000 = 40$. Furthermore, we would set `popsize` to 5000.

In order to intake parameter values from the newly formatted `parameters` file, we need to modify `params.cc` as well. First, we look at modifications to the function `read_parameters_file()`. Recall that this function simply reads each line of the `parameters` file and stores each line in a `map`, where the key is the parameter name and the value is a `string` of the parameter value(s).

params.cc: read_parameters_file( ) update; modifications to **params.h**

```
map<int, map<string, string> > read_parameters_file(const string &parameters_fn)
{
  map<int, map<string, string> > params_by_block;
  map<string,string> params;
  ifstream paramfile(parameters_fn.c_str());
  string line;
  int block = -1;
  regex query("DEME");
  while(getline(paramfile, line)) {
    istringstream iss(line);
    string key, nextone, value;
    iss >> key;
    if (regex_search(key, query)) { // check if entering next deme block
      params_by_block[block] = params;
      params.clear();
      ++block;
17        } else {
18           while (iss >> nextone)
19              value += nextone + " ";
20           params[key] = value;
21        }
22     }
23     params_by_block[block] = params; // assign values for final deme
24     return params_by_block;
25  }
26  map<int, map<string, string> > p = read_parameters_file("parameters");
27
28  //params.h
29  extern vector<string> mscommand;
30  extern vector<bool> useMS;
31  extern map<int, vector<int> > pop_schedule;
32  extern bool trackAlleleBirths;
33  extern int diploid_sample;
34  extern int printhapfreq;
```

Much of this code remains the same as before and is included solely for context. The changes are motivated by our need to account for the blocks of deme-specific parameters in the `parameters` file. To this end, we declare a new variable `block` and define it as -1 (line 7). This indexing variable will assign parameters read to separate blocks, where -1 is the block of "global" variables and blocks `0 ... n` are blocks of variables specific to demes 0 through *n*. Each block of read-in parameters is again stored as a `map<string, string>`, which is then indexed by `block` and returned in the form of a `map<int, map<string, string> >` container (line 1).

Recall that lines in the `parameters` file that include the word `DEME` indicate subsequent lines that specify parameter values specific to a deme. Thus, we need to define a `regex` query to search for this word in each line (line 8) and, as we move through the lines of the `parameters` file (lines 9–22), do the following each time the query is matched (line 13): (1) add the accumulated `map<string, string> params` to the container `params_by_block` using the current value of `block` as the key; (2) clear `params`; and (3) increment `block` (lines 14–16). The function call of line 26 then stores the returned `map` as variable `p`. Note that the new parameters added to this chapter are also declared in `params.h` (lines 29–34).

The other major set of changes we need to make to `params.cc` are to declare (without defining) all `extern` variables (lines 2–7) and write a new function that will process the `map` named `p` returned by `read_parameters_file()`. Our need to create containers of deme-specific variables (indexed by deme number) motivates the creation of this function, `process_parameters( )` (lines 9–44).

params.cc: process_parameters()

```
1  // new and updated parameter declarations
2  int pop_num, runlength;
```

```
 3 vector<double> m;
 4 vector<string> mscommand;
 5 vector<bool> useMS;
 6 vector<int> birthgen, extinctgen;
 7 map<int, vector<int> > pop_schedule;
 8
 9 int process_parameters() { // replaces old version of function
10    for (auto iter = p.begin(); iter!=p.end(); ++iter ) {
11       map<string, string> parameters = iter->second;
12       if (iter->first == -1) { // block of global parameters
13          mutrate = atof(parameters["mutrate"].c_str());
14          recrate = atof(parameters["recrate"].c_str());
15          hotrecrate = atof(parameters["hotrecrate"].c_str());
16          useRec = atoi(parameters["useRec"].c_str());
17          useHotRec = atoi(parameters["useHotRec"].c_str());
18          hotrecStart = atoi(parameters["hotrecStart"].c_str());
19          hotrecStop = atoi(parameters["hotrecStop"].c_str());
20          sampsize = atoi(parameters["sampsize"].c_str());
21          seqlength = atof(parameters["seqlength"].c_str()); // covernsion
               ↪ using atof() enables use of e notation in parameters file
22          sampfreq = atoi(parameters["sampfreq"].c_str());
23          getWindowStats = atoi(parameters["getWindowStats"].c_str());
24          windowSize = atoi(parameters["windowSize"].c_str());
25          windowStep = atoi(parameters["windowStep"].c_str());
26          pop_num = atoi(parameters["pop_num"].c_str());
27          runlength = atoi(parameters["runlength"].c_str());
28          m = get_multi_double_param("migration_rates", parameters);
29       } else { // block of deme parameters
30          popsize = atoi(parameters["popsize"].c_str());
31          demography = get_multi_int_param("demography", parameters);
32          dem_parameter = get_multi_double_param("dem_parameter",
               ↪ parameters);
33          dem_start_gen = get_multi_int_param("dem_start_gen", parameters);
34          dem_end_gen = get_multi_int_param("dem_end_gen", parameters);
35          carrying_cap = get_multi_int_param("carrying_cap", parameters);
36          birthgen.push_back( atoi(parameters["birthgen"].c_str()) );
37          extinctgen.push_back( atoi(parameters["extinctgen"].c_str()) );
38          useMS.push_back( atoi(parameters["useMS"].c_str()) );
39          mscommand.push_back( parameters["mscommand"] );
40          pop_schedule[iter->first] = create_pop_schedule();
41       }
42    }
43    return 1;
44 }
45
46 int good_parameters = process_parameters();
47 double* a = &m[0]; // Matrix takes array argument
48 Matrix<double> mig(pop_num, pop_num, a);
```

Lines 1–8 declare all variables listed in params.h. Because the values of deme-specific parameters are potentially different among demes for useMS, mscommand, birthgen, extinctgen, and pop_schedule, we store them in containers. In the case of the first four, we use a vector of the appropriate type (lines 5–7, 36–39) and the values within each vector are indexed by deme number. For pop_schedule, the by-generation population size is stored in a vector<int> that is the value of a map whose key is the population number (lines 7, 40). The function is called at line 46 and the migration matrix mig is instantiated as a Matrix object[3] (lines 47–48) using the migration rates listed in the parameters file (lines 47–48).

6.2.1.2 Class Metapopulation

We now introduce a new class, Metapopulation, which coordinates the simulation of multiple objects of the class Population—thought of as demes *or* populations depending on the simulated model. Before proceeding to the class definition however, we take care of some modifications necessary to the main source file:

fortuna.cc // modifications

```
 1      ...
 2       #include "metapopulation.h"
 3       #include "matrix.h"
 4      ...
 5      mt19937 engine2(time(0));
 6      Metapopulation::f = engine2;
 7       ...
 8       cout << "MIGRATION matrix:" << endl;
 9      mig.print_matrix();
10      Metapopulation meta;
11
12      for (int i =0; i < runlength; ++i) {
13        meta.reproduce_and_migrate(i);
14        if (i % 25 == 0) { cout << "gen " << i << endl;}
15      }
16      meta.close_output_files();
17      return 0;
18  }
19
20  // additional static variables for metapopulation class
21  mt19937 Population::e;
22  mt19937 Metapopulation::f;
```

We include the Metapopulation and Matrix class header files (lines 2–3) and create a separate Mersette pseudo-random number generator f for use in the Metapopulation object (lines 5–6, 22). The migration matrix is printed to STDOUT (lines 8–9) and a Metapopulation object is instantiated in line 10. Because we now specify a parameter runlength in the parameters file, the for loop from lines 12–15 invokes this variable. Any arguments to the pro-

[3] matrix.h defines Matrix objects and the functions used to access and change their cell values; see online code for the file if interested in its details.

gram call will be ignored. Each iteration of this loop calls the `Metapopulation` member function `reproduce_and_migrate()`, which takes the current generation as its argument (line 13) and the current generation is printed to STDOUT every 25 generations (line 14). Note that this program can still be used to simulate a single population; you need to only change parameter `pop_num` to 1 and any DEME blocks *beyond the first one* will be ignored.

The name of the class does not imply that we are necessarily modeling a metapopulation in the technical, biological sense.

metapopulation.h

```
#ifndef METAPOPULATION_H
#define METAPOPULATION_H

class Metapopulation {

private:
vector<Population*> populations;
uniform_real_distribution<double> random01;

public:
  void reproduce_and_migrate(int gen) {
    // reproduction within all extant demes
      // AND check for birth/extinction of population
    for (int i=0; i<pop_num; ++i) {
      if ((*populations[i]).get_extant()) {
        if (extinctgen[i] == gen)
          (*populations[i]).set_extinct();
        else
          (*populations[i]).reproduce(gen);
      }
      else {
        if ( birthgen[i] == gen ) {
          (*populations[i]).set_extant();
          (*populations[i]).reproduce(gen);
        }
      }
    }

    // migration among all demes
    for (int i=0; i<pop_num; ++i) {
      if ( (*populations[i]).get_extant() ) {
        for (int j=0; j < pop_num; ++j) {
          if ( (*populations[j]).get_extant() ) {
            double Nm = mig[i][j] * pop_schedule[j][gen];
            if (Nm < 1) {
              if(random01(f) < Nm)
                Nm = 1;
              else
                Nm = 0;
            }
            else
              Nm = floor(Nm);
```

```

                  for (int k=0; k<Nm; ++k) {
                     vector<vector<int>> v1 =
                          ↪ (*populations[i]).get_sequences(k);
                     (*populations[j]).add_immigrant( v1 );

                     // check for new alleles introduced to population j
                     for (int m = 0; m < 2; ++m) {
                        vector<int> v2 =
                             ↪ (*populations[j]).get_allele_positions();
                        vector<int> diff;
                        set_difference(v1[m].begin(), v1[m].end(),
                             ↪ v2.begin(), v2.end(), inserter(diff,
                             ↪ diff.begin()));
                        for (auto q:diff)
                              (*populations[j]).insert_new_allele(
                                   ↪ (*populations[i]).get_allele_info(q) ) ;
                     }
                  }
                  (*populations[i]).remove_emigrants(Nm);
               } else continue;
            }
         } else continue;
      }
      if ( (gen+1) % sampfreq == 0)
         for (int i=0; i<pop_num; ++i)
            if ((*populations[i]).get_extant())
               (*populations[i]).sample(gen);
   }

   void close_output_files() {
      for (auto iter = populations.begin(); iter != populations.end();
           ↪ ++iter)
         (*iter)->close_output_files();
   }

   Metapopulation() {
      for (int i=0; i < pop_num; ++i) {
         if (birthgen[i] != 0 )
            populations.push_back( new Population(i, 0) );
         else
            populations.push_back( new Population(i, 1) );
      }
      random01.param(uniform_real_distribution<double>::param_type(0.,1.));
   }

   ~Metapopulation() {}

   static mt19937 f;
};

#endif
```

The `Metapopulation` constructor (lines 73–81) creates `pop_num` new `Population` objects. As noted above, you might specify ten demes in the `parameters` file, but only the first `pop_num` demes' (`Population` objects) will be created. For each newly created `Population` object, a pointer to the object is added to the end of the `private` variable `populations` (line 7). In addition, the `Population` constructor now takes a second argument (see details below), which specifies whether the deme is currently extant or not—1 and 0, respectively. When the forward simulation begins at generation 0, only those demes that have a `birthgen` of 0 will be extant (lines 75–76). Demes that come into existence later in the simulation are still instantiated (lines 77–78), but they will not be activated until their `birthgen` generation. Finally, the constructor instantiates the random number generator `random01` (line 8) as a random number generator on the range [0,1] (line 80).

Two `public` functions are defined. Function `close_output_files( )` (lines 68–71)—is analogous to the function of the same name in the `Population` class, with the difference that this function causes the output files of *each* deme to be closed. Function `reproduce_and_migrate( )` (lines 11–66)—controls the simulation of all demes as time moves forward. A `for` loop (lines 14–27) cycles through all `Population` objects and first queries whether the deme in question is extant (line 15). If so, a second check is made to see if the current generation is the `extinctgen` of the deme under consideration (line 16). If so, the deme is inactivated by a call to `Population` function `set_extinct()` (see details below). Otherwise, the `reproduce()` function is called on the current deme (lines 18–19), which acts as before for this deme, i.e., reproduction, mutation, recombination, and sampling are all carried out for this deme. If the deme is not currently active, as indicated by a `false` return to the `if` statement on line 15, we check if the current generation is the `birthgen` of this currently inactive deme (line 22). If so, we activate the deme and immediately call `reproduce( )` on it (lines 23–24). In Sect. 6.3, we will introduce splits and mergers of demes, but for now we assume all demes are initiated by coalescent simulation.

Following reproduction in all active demes, we then determine which individuals, *if any*, migrate from one deme to another and afterwards update the `alleles` variable of each deme (lines 30–61). We begin with migration, where variable `i` of the outer `for` loop (begins line 30) represents the deme number from which emigrants may leave and variable `j` of the inner `for` loop (begins line 32) represents the deme number that may receive migrants. If deme `i` is not extant (`false` returned at line 31), it cannot yield migrants and so we skip to the next deme (via `continue` statement; line 60). Assuming deme `i` is extant, we then check to see if deme `j` is extant (line 33) and again skip the deme if not (line 58). If both demes `i` and `j` are active, we then proceed with migration by first determining the number of migrants `Nm` by multiplying the fraction of individuals in deme `j` that are immigrants from deme `i` each generation (`mig[i][j]`) by the current population size of deme `j` (line 30). If `Nm` is less than zero, we then determine if one or zero individuals migrate from `i` to `j` using the random number generator `random01` (lines 35–40).

For each immigrant to deme `j`, we obtain the sequences of this individual currently in deme `i`, store it in variable `v1`, and add it to population `j` using the function `add_immigrant( )` described below (lines 44–46). For each new immigrant to deme `j`, we then check to see if any alleles previously absent from deme `j` are now present due to the immigration of an individual from another deme (lines 49–55). To do this, we retrieve the list of allele positions in deme `j` (line 50) and use the standard library functions `set_difference( )` and `inserter( )` to check for any allele positions present in the sequences of the immigrant but absent from the population and create new `Allele` objects for deme `j` if this is the case (lines 51–54). Upon considering all immigrants to deme `j`, we then remove the emigrant from deme `i` and free the memory occupied by its pointer using the function `remove_emigrants( )` (line 57) detailed below. Figure 6.4 illustrates the collective actions of the `add_immigrant( )` and `remove_emigrants( )` functions for a simple case of *symmetric* migration.

Finally, evolution of the current generation is not complete until after reproduction *and* migration have occurred. In previous iterations of the FORTUNA program, the `reproduce()` function of class `Population` called `get_sample()` automatically during sampling generations. However, we must now wait to sample until after migration has occurred. This necessitates the creation of a separate function in class `Population`, which we name `sample()` (see below for details). Lines 62–65 call the `sample()` function for each deme that is extant (lines 63–65) conditional on the current generation being a sampling generation (line 62).

6.2.1.3 Modifications to Class `Population`

In order for a `Metapopulation` object to perform properly, several minor additions and changes are required to `population.h`. The following listing summarizes these changes.

population.h // additions and modifications

```
1  ...
2  private:
3  int popn;
4  bool extant;
5  ...
6  void update_alleles(const int &gen) {
7  ...
8      if (current_count == pop_schedule[popn][gen]*2) { // add [popn] index
9  ...
10 public:
11 inline int get_popnum() { return popn;}
12 inline bool get_extant() {return extant;}
```

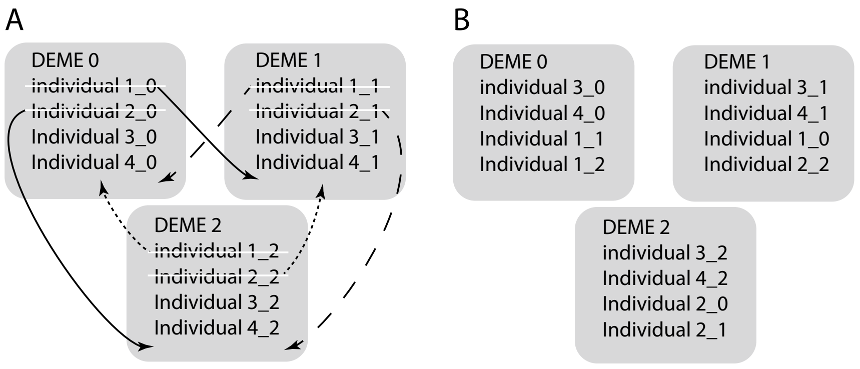


Fig. 6.4 Collective actions of the `add_immigrants()` and `remove_immigrants()` functions. (**a**) Black lines indicate immigration of an individual from one deme to another. New immigrants are pushed to the back of the `vector` that holds `individuals` in the receiving deme. For simplicity we assume $N_e = 4$ and $m_{i,j} = 0.25$ for all i and j. On average, then, one individual immigrates from each deme to each other. After all migration is accounted for, the white lines indicate that the emigrants from each deme are removed. (**b**) The outcome of immigration. Although the original labels of each individual are retained to emphasize the action of migration, understand that each individual in each deme is now a true member of that deme. For example, the individual labeled 2_2 in DEME 1 is now a member of DEME 1 and not DEME 2 from which it emigrated

```
inline void set_extant() {extant = 1;}
inline void set_extinct() {extant = 0;}
inline vector<vector<int>> get_sequences(int indnum) { return
    ↪ (*individuals[indnum]).get_sequences();}
inline void add_immigrant(vector<vector<int>> ses)
    ↪ {individuals.push_back( new Individual(ses) ); }

void remove_emigrants(int Nm) { // new function
  for (auto iter = individuals.begin(); iter != individuals.begin() + Nm;
      ↪ ++iter)
    delete *iter;
  individuals.erase(individuals.begin(), individuals.begin()+Nm);
}

vector<int> get_allele_positions() { // new function
  vector<int> v;
  for (auto iter=alleles.begin(); iter!=alleles.end(); ++iter)
    v.push_back(iter->first);
  return v;
}

vector<int> get_allele_info(int s) { // new function
  vector<int> v = {s};
  v.push_back( (*alleles[s]).get_birthgen() );
  v.push_back( (*alleles[s]).get_originating_population() );
  return v;
}
```

```
37
38 void insert_new_allele(vector<int> v) { // new function
39   alleles.insert( { v[0] , new Allele( v[0], v[1], v[2] ) } );
40 }
41 ...
42 void reproduce(int gen) { // add migration and variable N
43   randomind.param(uniform_int_distribution<int>::param_type(0,pop_schedule
        ↪ [popn][gen]-1));
44   int N = pop_schedule[popn][gen]; // N is number of individuals to
        ↪ produce
45   if (modelMigration) {
46     int n_imm = 0;
47     int n_emi = 0;
48     for (int i=0; i<pop_num; ++i) {
49       n_emi += mig[popn][i] * pop_schedule[i][gen];
50       n_imm += mig[i][popn] * N;
51     }
52     N += n_emi;
53     N -= n_imm;
54   }
55 ‘ for (int i=0; i< N; ++i) {
56     ... // random parents drawn as before, producing N individuals
57   }
58   for (auto iter = individuals.begin(); iter != individuals.end() - N;
        ↪ ++iter)
59     delete *iter;
60   individuals.erase(individuals.begin(), individuals.end() - N);
61 }
62
63 void sample(int gen) { // new function
64   update_alleles(gen);
65   random_shuffle(individuals.begin(), individuals.end() ) ;
66   get_sample(gen + 1);
67 }
68
69 Population (int popnum, int eextant):popn(popnum), extant(eextant) { //
      ↪ added parameters to constructor
70 ...
71   randomind.param(uniform_int_distribution<int>::param_type(0,pop_schedule
        ↪ [popn][0] - 1));
72 ...
73   if (useMS[popn]) {
74    ...
75    system(mscommand[popn].c_str());
76    ...
77    for (int i=0; i<pop_schedule[popn][0]; ++i) {
78    ...
79 }
```

Two new `private` variables are added to the class: (1) popn (line 3), which is the population number and (2) `extant` (line 4), of type `bool` and set to 1 if the deme is currently active within the simulation or 0 if it is not. In addition, several helper (get or set) functions are defined on lines 11–16. Of particular interest is `get_sequences()`, which returns a vector of both sequences for an individual and is used during migration (line 45 of `metapopulation.h` listing above). Conversely, the function `add_immigrant()` creates a new individual and pushes a pointer to this object to the back of `immigrants` (line 16). The function `remove_emigrants()` takes as an argument the number of individuals that have emigrated from the deme and subsequently deletes the object and frees the memory assigned to the pointer to this object (lines 18–22).

The function `reproduce( )` (lines 42–61) is modified to determine to work out the calculus of migration—making sure that offspring of parents currently in the deme are generated after accounting for the gain of immigrants and loos of emigrants. Then, N is modified to reflect the number of offspring the deme needs to produce from within. The initial value of N is set to the population size associated with current generation as detailed in the `pop_schedule` for the deme in question (line 44). As an example, consider lines 45–54 and then imagine a deme that is scheduled to consist of 5000 individuals after this round of reproduction and migration. If the number of immigrants from all sources (`n_imm`) are 400 and the number of emigrants to all other demes (`n_emi`) are 100, then `N = population_schedule[popn][gen] + n_emi - n_imm` (lines 42, 52–53), or `N` = 5000 + 100 − 400 = 4700 in this case. In other words, only 4700 individuals need to be produced, as 100 of these individuals will emigrate to other demes, but 400 immigrants from other demes will bring population size up to the required 5000. Of note, this works for both symmetric and asymmetric migration.

Next, the `N` individuals are produced as before (lines 55–57); the upper bound of the `for` loop is simply changed to the calculated `N`. Lines 58–60 `delete` the pointers to individuals of the previous generation and `erase` the orphaned pointers postdeletion. The upper bound of the `iterator` is set to `individuals.end() - N` in each case because this will remove all but the `N` newly minted individuals from the `vector` that holds them. As mentioned in the previous subsection, it is necessary to create a new function, named `sample()`, for use with migration (lines 45–49).

Lines 24–40 introduce three short new functions that allow increased capability to manage alleles' histories and frequencies in the context of multiple demes. The constructor (beginning on line 34) now takes a second argument, which is the initial value of the variable `extant`. The remaining changes account for the fact that several of the `extern` variables defined in `params.h` are now indexed by deme (i.e., population number), which requires the addition of a `[popn]` index to several variables (e.g., line 8).

6.2.1.4 Results of Simulating Two Demes Connected by Different Rates of Migration

After all of those changes, we are now ready to simulate the scenario proposed at the beginning of Sect. 6.2 and attempt to corroborate the theoretical result that $N_e m > 1$ between two demes is essentially synonymous to a single panmictic population. I performed three simulations in which parameter values were identical to those listed in 6.2.1.1, with the exception that parameter `migration_rates` was set to one of the following in each separate simulation:

- 0. 0.00001 0.00001 0.
- 0. 0.0001 0.0001 0.
- 0. 0.001 0.001 0.

As both demes were set to `popsize` of 10,000 diploid individuals and all migration rates are symmetric, these `migration_rates` settings were equivalent to 0.1, 1, and 10 migrants between the two populations each generation, respectively. In each simulation, each deme began with genetic variation generated by independent coalescent simulations. At generation zero, migration ensued between the two demes and each simulation was run for 5000 generations. To assess genetic similarity between the two demes, I plotted the values of Tajima's D. This seemed a good summary statistic to use as it incorporates both the number of segregating sites and nucleotide diversity. However, I note that the results are qualitatively similar for single summary statistics. In Sect. 6.3, we will calculate F_{ST} to assess the genetic similarity between demes.

In short, the results of these simulations confirm that $N_e m = 1$ appears to be the threshold at which the two demes no longer appear genetically divergent. When $N_e m = 0.1$, values of Tajima's D for the two demes are highly dissimilar (Fig. 6.5). Conversely, whether $N_e m = 1$ or $N_e m = 10$, the values of Tajima's D for both populations are highly coincident (Fig. 6.5).

6.3 Forward Simulation of *n* Demes

In the last section, we spent considerable effort to provide FORTUNA with the functionality to simulate any number of demes, using this added functionality to simulate the simple scenario of two demes connected by variable magnitudes of migration. In this section, we add further functionality to FORTUNA, allowing us to: (1) split and merge demes rather than initiating each deme with coalescent-simulated genetic diversity; (2) output full haplotypes at specified time points; (3) choose between haploid and diploid sampling; (4) calculate observed and expected heterozygosity at all segregating sites; (5) calculate single- and multilocus F_{ST} for each pair of demes;

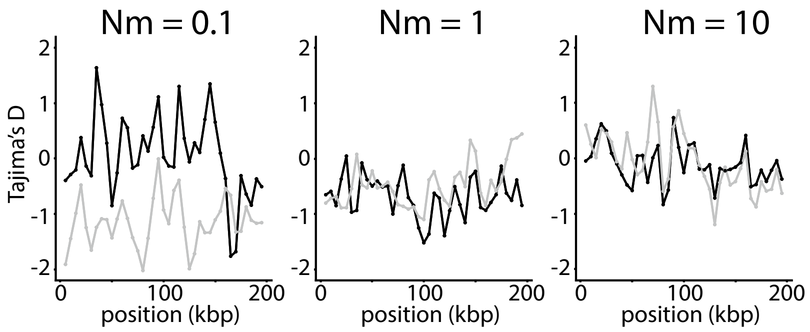


Fig. 6.5 $N_e m \geq 1$ provides sufficient mixing between two demes to make them appear as one panmictic population. Values of Tajima's *D* are plotted for windows of 10 kbp with step size of 5 kbp for the two demes as gray and black lines

and (6) simulate a true metapopulation scenario in which some demes are sources and others are sinks. Note that (2)–(4) provide us with functionality that is not specific to simulations that include migration.

6.3.1 Deme Splitting and Merger

To facilitate the splitting or merger of currently extant demes, each DEME block in the `parameters` file includes two new parameters: `splitgenesis` and `mergergenesis`. Before we dig into the use of these parameters, consider that the *n* demes alluded to in the section title may be *coeval* and/or *allochronic*. Here, we simulate a scenario that involves both deme splitting and merger (Fig. 6.6). By definition, the splitting or merger of a deme leads to two or one descendant deme(s), respectively. Therefore, both coeval and allochronic demes will be simulated (e.g., Fig. 6.6).

The parameter `splitgenesis` is defined for each deme block as a vector of three integers. As an example, the values `1 0 40` indicate that the deme originates via splitting (1 in the first position) of deme 0 (0 in the second position) and *begins* with 40% of the individuals from deme 0. Parameter `mergegenesis` is also defined by a vector of three integers. For example, the values `1 2 3` specify the deme originates via merger (1 in the first position) and that it is the amalgam of demes 2 and 3 (the final two integers). If the `useMS` parameter of the DEME block is set to 1, these parameters are ignored, as the deme is initiated by coalescent simulation. Clearly, if `useMS` is set to zero, the first parameter argument of either `splitgenesis` or `mergegenesis` must be set to one. That is, each new deme must come from somewhere—either initiated with coalescent simulation, the split of an ancestral deme, or the merger of two ancestral demes.

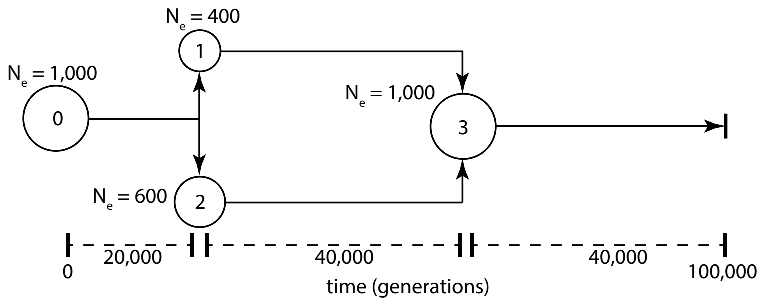


Fig. 6.6 Scenario simulated in this section. Genetic variation of deme 0 at generation 0 is produced by coalescent simulation. This deme is forward simulated for 20,000 generations at which point a population split yields demes 1 and 2, which receive 40% and 60% of the individuals in deme 0, respectively. Both demes then evolve for 40,000 generations with minimal gene flow between them, after which a population merger yields deme 3 ($N_e = 1000$) that goes on to evolve for 40,000 more generations. Only demes 1 and 2 are coeval, while all other pairs of demes are allochronic

Because we have covered in detail the changes to `params.cc` and `params.h` associated with the addition of new parameters to the `parameters` files in this and other chapters, I ask the reader to look at the Chap. 6 code online and consider the modifications required to these files. From now on, however, I will assume we have seen enough of these mundane changes to warrant skipping explicit instruction. Instead, let us look at the DEME blocks of the `parameters` file that will allow us to simulate the scenario sketched in Fig. 6.6.

`parameters` file // values used to simulate model in Fig. 6.6

```
DEME   /// 0
popsize 1000
demography 0
birthgen 0
extinctgen 20000
...
useMS 1
mscommand ./ms 2000 1 -t 8 >ms_output
splitgenesis 0 0 0
mergegenesis 0 0 0
DEME   /// 1
popsize 400
demography 0
birthgen 20000
extinctgen 60000
...
useMS 0
mscommand ./ms
splitgenesis 1 0 40
mergegenesis 0 0 0
```

```
21  DEME   /// 2
22  popsize 600
23  demography 0
24  birthgen 20000
25  extinctgen 60000
26  ...
27  useMS 0
28  mscommand ./ms
29  splitgenesis 1 0 60
30  mergegenesis 0 0 0
31  DEME   /// 3
32  popsize 1000
33  demography 0
34  birthgen 60000
35  extinctgen 100001
36  ...
37  useMS 0
38  mscommand ./ms
39  splitgenesis 0 0 0
40  mergegenesis 1 1 2
```

The ellipses in the previous listing skip the demographic parameters of each DEME block, as we are not modeling population expansion or contraction in any of the four demes. Deme 0 is initiated via coalescent simulator MS (lines 7–8); 2000 sequences are generated corresponding to the `popsize` of 1000 diploid individuals (line 2). It is the only extant deme for the first 20,000 generations of the simulation (lines 4–5).

At generation 20,000, demes 1 and 2 originate (their `birthgen` parameter is set to 20,000) via splitting of deme 0 (lines 19 and 29). Note that the third number of the `splitgen` parameter is 40 and 60 for demes 1 and 2, respectively. This means 40% of the individuals in deme 0 will be randomly assigned to new deme 1, while the remaining 60% of individuals will be assigned to new deme 2. Importantly, these split percentages must add up to 100 at most. If you want a new deme to increase in size following the split, you should specify its demographic parameters appropriately (Chap. 4). It is fine if the two split percentages add to less than 100. The "unused" individuals from the ancestral deme are simply lost. It is also important to set the `popsize` of each deme resulting from a split appropriately; in this case, for deme 1 `popsize` = $1000 \times 0.4 = 400$ and for deme 2 `popsize` = $1000 \times 0.6 = 600$ (lines 12 and 22).

`extinctgen` is set to 60,000 for demes 1 and 2, while `birthgen` of deme 3 is set to 60,000 because demes 1 and 2 merge to become deme 3 at 60,000 generations (Fig. 6.6; lines 15, 25, and 34). The `mergegenesis` parameter for deme 3 tells the program that it is the product of merger of demes 1 and 2 (line 40).

Now, we consider the code that implements the splits and mergers of populations. This requires us to examine the necessary changes to `population.h` and `metapopulation.h`, which include a new `inline` function as well as modifications to the functions `set_extant()` (Population class; previously implemented as a simple, `inline` function) and `reproduce_and_migrate()` (Metapopulation class).

population.h and metapopulation.h // additions and modifications

```
// population.h, new public functions
inline int get_current_popsize(int gen) {return pop_schedule[popn][gen];}
vector<int> set_extant() {
  extant = 1;
  vector<int> i;
  if (splitgenesis[popn][0] > 0) {
    i.push_back(1);
    i.push_back(splitgenesis[popn][1]); // source population
    i.push_back(splitgenesis[popn][2]); // percent
  } else if (mergegenesis[popn][0] > 0 ) {
    i.push_back(2);
    i.push_back(mergegenesis[popn][1]); // first source population
    i.push_back(mergegenesis[popn][2]); // second source population
  } else
    i.push_back(0);
  return(i);
}

// metapopulation.h, add to function reproduce_and_migrate( )
void reproduce_and_migrate(int gen) {
  for (int i=0; i<pop_num; ++i) {
    if ((*populations[i]).get_extant()) {
      ...
    } else {
      if ( birthgen[i] == gen ) {
        vector<int> change = (*populations[i]).set_extant();
        map<int, int> alleles_to_add;
        if (change[0] == 1) { //split
          int N = (double) (change[2]) / 100 * (*populations[
              ↪ change[1] ]).get_current_popsize(gen);
          for (int k = 0; k<N; ++k) {
            vector<vector<int>> v1 = (*populations[ change[1]
                ↪ ]).get_sequences(k);
            (*populations[i]).add_immigrant( v1 );
            for (int m=0; m<2; ++m)
              for (auto iter=v1[m].begin(); iter!=v1[m].end(); ++iter)
                ++alleles_to_add[*iter];
          }
          (*populations[ change[1] ]).remove_emigrants(N); // so that
              ↪ the other deme populated by the split doesn't receive
              ↪ same individuals
        }  else if (change[0] == 2) { //merger
          vector<int> N;
          N.push_back( (*populations[ change[1]
              ↪ ]).get_current_popsize(gen) );
          N.push_back( (*populations[ change[2]
              ↪ ]).get_current_popsize(gen) );
          for (auto q:N) {
```

```
43              for (int k = 0; k<q; ++k) {
44                vector<vector<int> > v1 = (*populations[ change[1]
                     ↪ ]).get_sequences(k);
45                (*populations[i]).add_immigrant( v1 );
46                for (int m=0; m<2; ++m)
47                  for (auto iter=v1[m].begin(); iter!=v1[m].end();
                       ↪ ++iter)
48                    ++alleles_to_add[*iter];
49              }
50            }
51          }  // else built from MS
52          if (change[0] > 0) // deme not built from MS
53              for (auto iter=alleles_to_add.begin(); iter !=
                   ↪ alleles_to_add.end(); ++iter) // populate alleles
                   ↪ in new deme
54                (*populations[i]).insert_new_allele( (*populations[
                     ↪ change[1] ]).get_allele_info(iter->first) );
55          (*populations[i]).reproduce(gen);
56        }
57      }
58    }
59     ...
60  }
```

Line 2 codes a function allowing us to obtain the current population size of the deme in question, which is particularly important when the deme is undergoing demographic change. In lines 3–17, we expand the function `set_extant()`, first introduced in the previous section, such that it not only sets the Population class `private` variable to true (line 4) but also returns a `vector<int>` that specifies the manner in which the deme should be activated. The `if-else if-else` control structure (lines 6–15) determines whether the new deme is the result of a split, merger, or coalescent-simulation data. This determination is the first entry of the returned `vector<int>` (0, 1, or 2 for coalescent, split, and merger, respectively). For a split or merger additional relevant values are pushed to the returned `vector<int>` (lines 8–9 and lines 12–13).

Lines 20+ demonstrate the modifications to the Metapopulation class function `reproduce_and_migrate()`, which now invokes the Population class functions detailed in the previous paragraph. As before, the function cycles among all populations (extant or not; line 21). If the population is not extant, but its `birthgen` is the current generation, the modified `set_extant()` function is called, returning the `vector<int>` called `change` (lines 25–26). Any deme originating from split or merger initially holds no Allele objects; we therefore create a `map<int, int>` (`alleles_to_add`) to temporarily store the new alleles (line 27). Based on the value of `change[0]` returned from `set_extant()`, we either carry out a split (lines 28–37), a merger (lines 38–51), or do nothing special in the case of a deme originating from coalescent simulation. Regardless, the current deme, if extant, will call `reproduce()`

(line 55) and participate in migration (ellipsis at line 59, documented in Sect. 7.2).

In the case of a split, we first determine the size of the daughter deme (`N`), which requires us to account for the percentage of the parent deme that ends up in the new deme as well as the current population size of the parent deme (line 29). Using a `for` loop, we then populate the new deme with the first `N` individuals from the parent deme (lines 31–32), add the alleles from each individual to `alleles_to_add` (line 33–35), and remove the `N` individuals used to create the new deme from the parent deme (line 37). This last step ensures that we do not add the same individuals' sequences to the other daughter deme.

In the case of a merger, we first determine the current population sizes of the two merging demes (lines 39–41). The `for` loop spanning lines 42–50 then adds all individuals from each of the merging demes and populates `alleles_to_add` based on the sequences of these individuals.

Assuming the focal deme originates from a split or merger (line 52 returns `true`), the `map alleles_to_add` will be populated. The values of this `map` (how many times the derived allele at each position was observed in the new deme) are irrelevant to the next step. However, the keys of this `map` are the positions of all derived alleles in the population ordered from least to greatest by `map` index. For each key (i.e., polymorphic site), an Allele object is then created, a pointer to which is added to the deme's `private` variable `map<int, Allele*> alleles` (lines 53–54).

6.3.2 *Distributional and Longitudinal Visualization of Summary Statistics*

In Chap. 5, we used a heat map to plot the distribution of haplotypes (*K*) by window over the course of the simulation. In this subsection, we introduce two R functions for visualization. One is an expansion of the function `distributional_heatmap( )` covered in Chap. 5, which allows us to plot distributional heat maps of any discrete or continuous summary statistic by window. The second function—`longitudinal_heatmap( )`—plots a longitudinal heat map of any summary statistic; that is, the *evolving* value of a summary statistic is shown for each window for each sampling generation. Both functions are written to accept the `sumstats` output file generated by FORTUNA. We will use this function to plot the evolution of nucleotide diversity for demes 0 and 1 in the model pictured in Fig. 6.6.

The following listing provides the code for both functions.

heatmaps.r

```
library(ggplot2)
library(cowplot)
library(reshape)

distributional_heatmap <- function(inputfile, stat, discrete = T,
    ↪ interval = 0.5) {
  d <- read.table(file = inputfile, header = T)
  dd <- split(d, d$stat)
  size <- dim(dd[[stat]])
  numcol = size[2] - 2
  mat <- dd[[stat]][,3:size[2]]
  numrow = numeric()
  high = numeric()
  low = numeric()
  brks = vector()
  if (discrete) {
    numrow <- max(mat) + 1
  }  else {
    for (i in 1:numcol) {
      minnie <- min(mat[,i], na.rm=T)
      maxie <- max(mat[,i], na.rm=T)
      if (i==1) {
        low <- minnie
        high <- maxie
      } else {
        if (minnie < low) {low <- minnie}
        if (maxie > high) {high <- maxie}
      }
    }
    low = floor(low)
    high = ceiling(high)
    brks = seq(low, high, interval)
    numrow = length(brks) - 1
  }
  mat2 <- matrix(0, nrow = numrow, ncol = numcol)
  if (discrete) {
    for (i in 1:size[1]) {
      for (j in 1:numcol) {
          mat2[mat[i,j],j] <- mat2[mat[i,j],j] + 1
      }
    }
  } else {
    for (i in 1:numcol) {
      h = hist(mat[,i], breaks = brks, plot = F)
      mat2[,i] = h$counts
    }
  }
  mat2.melted <- melt(mat2)
  r = 1 # ratio for creating squares in heatmap
  if (!discrete) {
    for (i in 1:length(mat2.melted[,1])) {
      mat2.melted[i,1] = brks[mat2.melted[i,1]]
      }
      r = 1 / interval
    }
    ggplot(mat2.melted, aes(x = X2, y = X1, fill = value)) + geom_tile() +
        ↪ coord_equal(ratio = r) + scale_fill_gradient(low = "white",
        ↪ high = "black")
}

longitudinal_heatmap <- function(inputfile, stat="K", timeflow="down",
      ↪ scalelim=vector(), timelim=vector(), lowcol=1, fix=0) {
    d <- read.table(file = inputfile, header = T)
    dd <- split(d, d$stat)
    size <- dim(dd[[stat]])
    q <- dd[[stat]][,c(1,3:size[2])]
    m <- melt(q, id.vars=c("gen"))

    d<- ggplot(m, aes(x=variable, y=gen, fill=value)) + geom_tile()

    if (length(timelim) != 0) {
      if (timeflow == "down") {
        d <- d + ylim(timelim[2], timelim[1])
      } else {
        d <- d + ylim(timelim[1], timelim[2])
      }
    } else {
      if (timeflow == "down") {
        d <- d + ylim(max(m$gen), min(m$gen))
      }
    }

    if (length(scalelim) != 0) {
      if (lowcol) { # use white as low-value color
        d <- d + scale_fill_gradient(low="white", high="black",
             ↪ limits=scalelim)
      } else {
        d <- d + scale_fill_gradient(low="black", high="white",
             ↪ limits=scalelim)
      }
    } else {
      if (lowcol) { # use white as low-value color
        d <- d + scale_fill_gradient(low="white", high="black")
      } else {
        d <- d + scale_fill_gradient(low="black", high="white")
      }
    }

    if (fix != 0) {
      d <- d + geom_hline(yintercept=fix, linetype=2)
    }

    d <- d + geom_vline(xintercept = (size[2] - 2) / 2, linetype = 2)
    d
}
```

We begin with explanation of the `distributional_heatmap()` function. Unlike the version covered in Chap. 6, this function is now generalized in the sense that it can plot any continuous or discrete summary statistic. Two arguments must be supplied while the other two have default values that can be changed if desired. Function parameter `inputfile` must be specified and is simply the name of a summary statistics output file from a FORTUNA run (remember to surround it in double quotation marks. The `stat` argument can take the values `"K"` (haplotypes), `"pi"`, `"S"`, `"tajD"`, and `"wat"` (unless you have expanded the program to calculate additional summary statistics, in which case there would be more options). If the summary statistic is discrete (`K` or `S`), those are the only parameters that require our attention, as `discrete` is set to `TRUE` by default and the value of `interval` is irrelevant. On the other hand, if the summary statistic is continuous (i.e., `pi`, `tajD`, or `wat`), we must set `discrete` to `FALSE` and choose an `interval` of choice. In the discrete case, lines 17–33, 41–46, and 49–54 are used to compute a histogram for each window, where the break points of the histogram's bins are `interval` apart. For example, the top panel of Fig. 6.7 was generated using the call `distributional_heatmap("testrun.0", stat="tajD", discrete=F, interval=0.5)`, where *testrun.0* is the imaginary name of a summary statistic output file. The lower panel of Fig. 6.7 required the same call, with the exception that `interval=0.2`.

Calls to function `longitudinal_heatmap()` require a greater number of arguments. Again, the `inputfile` and `stat` of interest must be specified. Three other arguments affect the look of the resulting plot. First, `timeflow` can be set to `"down"` (the default) or `"up"`. When `timeflow` is down, the top cells correspond to the oldest time points; in other words, time flows forward as we move *down* the y-axis. Clearly, the opposite is true of `timeflow="up"`. In this case, the lowest cells correspond to the oldest time points and time flows forward *up* the y-axis. Second, we can manually set the scale of the third dimension of these heat maps—color coding corresponding to the value of the summary statistic—by providing a low and high value as a vector for the argument to `scalelim`—e.g., `scalelim = c(-3,3)`. If `scalelim` is not specified, the scale will be set automatically to include the highest and lowest values of the summary statistic in the data set. One motivation for specifying the scale of the heat map's third dimension is to guarantee that the same color applies to the same value of summary statistic across multiple graphs drawn from independent replicates or separate demes. An example of this is shown in Fig. 6.8 where much less genetic variation, as measured by nucleotide diversity, is present in deme 1 than in deme 0. By setting `scalelim = c(0,3.5)` when plotting *both* graphs, each shade of gray means the same thing in both and the lesser diversity in deme 1 becomes quite evident.

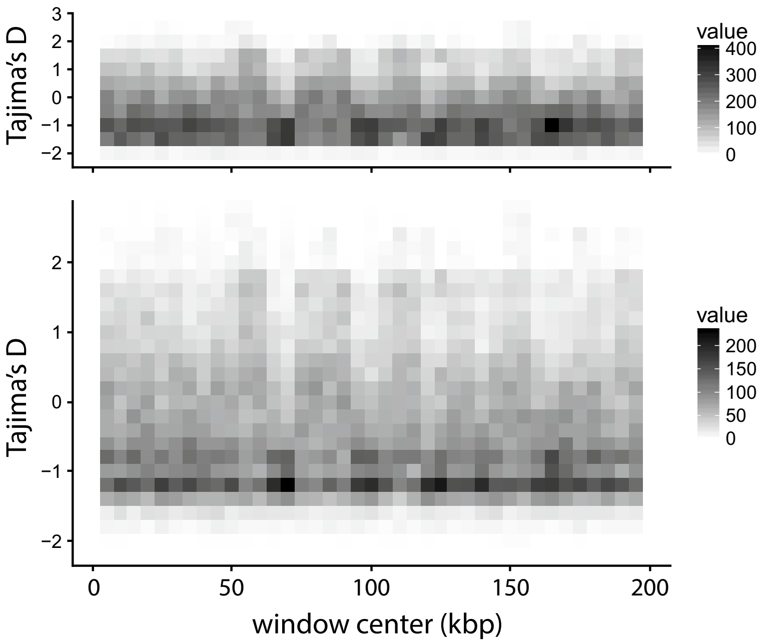


Fig. 6.7 Visualization of Tajima's *D* for deme 0 (following the model shown in Fig. 6.6) as a distribution of values across all 400 replicates at each of the windows. Both figures were generated using the `distributional_heatmap()` function, with `interval=0.5` (top panel) or `interval=0.2` (bottom panel)

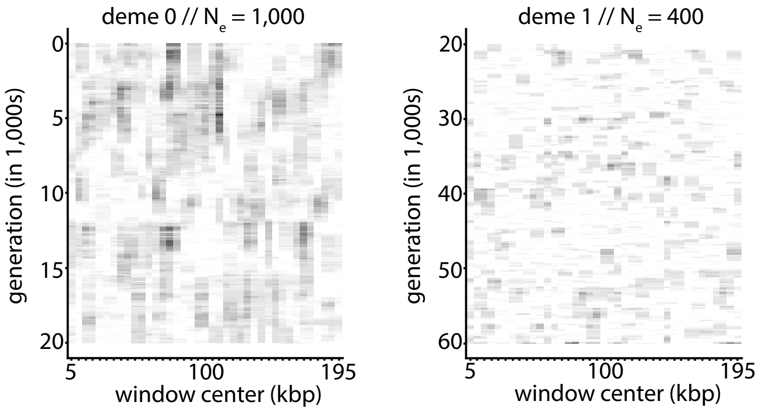


Fig. 6.8 Evolution of nucleotide diversity (π) along the 200,000 bp simulated sequence for demes 0 and 1 depicted in Fig. 6.6. Figures were generated using the `longitudinal_heatmap()` function with manually set limits of 0 and 3.5. Thus, black corresponds to $\pi = 3.5$, while white corresponds to $\pi = 0$. Window size is 10,000 bp and step size is 5000 bp. Note that the duration of time documented is twice as long for deme 1. Because the default `timeflow` of "down" was used, earlier generations are found at the top of the *y*-axis

6.3.3 *Diploid Sampling and Outputting Full Haplotypes at Specified Time Points*

There are several motivations for recording a sample of haplotypes every n generations. First, we can return to the actual sequence data and calculate whatever summary statistics we like at a later date (e.g., F_{ST}, Sect. 6.3.4). Second, we can use the sequences as input for analysis in a wide variety of other programs. Third, we can return to a critical time point in the simulation and use these sequence data as the starting point for many simulation replicates to quantify the frequency with which a certain outcome is achieved from these initial conditions.

To this point, sequence samples have consisted of a single haplotype from n individuals where n = `sampsize`. In most empirical contexts, however, we sample both haplotypes from each individual sampled. In this subsection, we therefore also add the ability to sample both haplotypes from each sampled individual. In this case if `sampsize=100`, both haplotypes will be sampled from 50 randomly selected individuals. The following adjustments to the code will allow us to (1) choose between haploid and diploid sampling and (2) print haplotypes sampled from the population to file every `printhapfreq` generations.

`modifications to add haplotype printing functionality`

```
\\ parameters file (global block)
printhapfreq 5000
diploid_sample 1

\\ params.h file
extern int printhapfreq;
extern int diploid_sample;

\\ params.cc file
int process_parameters() {
  ...
       if(iter->first == -1) {
          ...
          printhapfreq = atoi(parameters["printhapfreq"].c_str());
          diploid_sample = atoi(parameters["diploid_sample"].c_str());
          }
}


\\ population.h file
void get_sample(int gen) {
  ofstream sequencefile; //only used if gen % printhapfreq == 0
  string ofname = "deme" + to_string(popn) + "_" + to_string(gen);
  bool printhap = false;
  if (gen % printhapfreq == 0) printhap = true;
  if (printhap) sequencefile.open(ofname.c_str());
   ...
  int additional = sampsize;
28      if (diploid_sample)
29        additional /= 2;
30      for (auto iter = individuals.begin(); iter !=
            ↪ individuals.begin()+additional; ++iter) {
31        vector<int> haplotype = (**iter).get_sequence(0);
32        for (auto iter2 = haplotype.begin(); iter2 != haplotype.end();
              ↪ ++iter2)
33          ++allele_counts[*iter2];
34
35        if (diploid_sample) {
36          vector<int> haplotype = (**iter).get_sequence(1);
37          for (auto iter2 = haplotype.begin(); iter2 != haplotype.end();
                ↪ ++iter2)
38            ++allele_counts[*iter2];
39        }
40      }
41       ...
42      // print column headers
43      if (printhap) {
44        for (auto iter = positions.begin(); iter != positions.end(); ++iter)
45          sequencefile << "nt" << to_string(*iter) << " ";
46        sequencefile << endl;
47      }
48
49      for (auto iter = individuals.begin(); iter !=
            ↪ individuals.begin()+additional; ++iter) {
50        for (int h=0; h<diploid_sample+1; ++h) { // modified loop
51          vector<int> haplotype = (**iter).get_sequence(h);
52          sort(haplotype.begin(), haplotype.end());
53          string hap;
54          for (auto iter = allele_counts.begin(); iter !=
                ↪ allele_counts.end(); ++iter)
55            if ( binary_search (haplotype.begin(), haplotype.end(),
                  ↪ iter->first)) {
56              hap += "1";
57              if (printhap) sequencefile << "1 ";
58            } else {
59              hap += "0";
60              if (printhap) sequencefile << "0 ";
61            }
62          sample.push_back(bitset<bitlength> (hap));
63          if (printhap) sequencefile << endl; // new line
64        }
65      }
66       ...
67       if (printhap) sequencefile.close(); // new line
68    }
```

We begin with the addition of two global parameters and the necessary code for reading them into memory (lines 1–17):

- printhapfreq, which specifies how often sequence samples are printed to file

- `diploid_sample`, set to 1 if both haplotypes are to be sampled from each randomly selected diploid individual and 0 otherwise

Because the program will only check for printing of haplotypes within the `get_sample()` function, which is called every `sampfreq` generations, calculation `sampfreq` / `printhapfreq` must have a remainder of zero to ensure that the code to print haplotype files is executed.

Indeed, the remaining coding changes and additions involve the `get_sample()` function of class `Population`. An output stream and file name, specific to the deme in question, are created each time `get_sample()` is called (lines 21–22). However, the stream will only be opened if the current generation divided by `printhapfreq` yields a remainder of zero, which is determined in lines 24–25 and also sets the `bool` variable `printhap` to true if the remainder is zero. Lines 27–29 are used to set the value of `additional` and the following `for` loop (lines 30–40) thereby cycles through the appropriate number of `individuals` to obtain the sample: either `sampsize individuals` in the case of haploid sampling or `sampsize` / 2 `individuals` in the case of diploid sampling. Although true the `iterator` of this loop causes us to sample the first `additional individuals` in the vector, recall that we `random_shuffle( )` to mix the contents of the `vector` prior to sampling (Sect. 3.4.3.2). In this way, a random sample of `individuals` is obtained. Within the `for` loop, the first is analyzed and used to set the `allele_counts` variable local to the `get_sample)()` function. If a diploid sample is required, the same happens with the second haplotype of the individual (lines 35–39).

The next block of new code (lines 42–47) prints the column headings of the sequence output file, which are simply the positions of the segregating sites. This makes use of the `vector<int>` called `positions`, which holds the positions of all segregating sites (first defined within `get_sample( )` in Sect. 5.3.2). The outer `for` loop covering lines 49–65 again cycles through the sampled `individuals`. The nested `for` loop (lines 50–64) is either traversed once (haploid sampling) or twice (diploid sampling). Each traversal obtains the haplotype (line 51). Some of this code is, as before, used to populate the `vector<bitset<bitlength> >` named `sample`. New code, however, prints the haplotypes to the file. Importantly, note that if diploid sampling is used, the two haplotypes sampled from a single individual are printed consecutively.

6.3.4 *Calculating Multilocus* F_{ST}

A common statistic used to measure the genetic (dis)similarity of deme pairs is Wright's F_{ST} (Wright 1951), sometimes referred to as the fixation index. F_{ST} ranges from 0 to 1. Consider a sequence alignment where sampled members of both demes are included. There are three possible types of segregating sites in this alignment:

1. the derived allele is not fixed in either deme, i.e., the SNP is polymorphic in both demes.
2. one deme is fixed for either the ancestral or derived allele while the site is polymorphic in the other deme.
3. one deme is fixed for the ancestral allele and the other for the derived allele.

Another way of thinking about the third category of segregating site is that every individual in each deme is homozygous at the site. Many formulas for calculating F_{ST} therefore compare expected heterozygosity of a sample both within and across demes at a site that shows variation within the alignment of sequences from both demes, which is true of any of the three categories listed above. The bottom line is that $F_{ST} = 0$ at a single site when allele frequencies are identical in both demes and $F_{ST} = 1$ at a single site of the third type. These two extremes indicate a *general* interpretation of F_{ST}: Highly similar demes, with highly similar allele frequencies, will have F_{ST} values close to zero while highly dissimilar demes will have F_{ST} values near one.

F_{ST} can therefore be used to quantify the "connectivity" of two demes, given that we expect high levels of gene flow between two demes to maintain very similar allele frequencies among the demes and vice-versa. Indeed, Wright (1931) showed that for an island model F_{ST} for small values of m can be used to estimate $N_e m$, the absolute number of migrants flowing from one deme to another each generation:

$$F_{ST} = \frac{1}{4N_e m + 1} \Rightarrow \hat{N_e m} = \frac{1}{4}\left(\frac{1}{F_{ST}} - 1\right) \tag{6.4}$$

Note, however, this estimator is only reliable under quite exacting conditions (Whitlock and McCauley 1999).

F_{ST} was initially defined with reference to a single-segregating site. Given that we can now gather data from very large numbers of segregating sites, how do we scale up to a multilocus F_{ST}? One widely used solution is the multilocus measure of F_{ST} derived in Reich et al. (2009). For details on the derivation, see the Supplementary Information of this article. Implementing the calculation in code is quite simple and is provided in the following listing of `calc_fst.r`.

calc_fst.r

```
calc_fst <- function(fn1, fn2)
{
  p1 = read.table(file = fn1, header = T)
  p2 = read.table(file = fn2, header = T)

  diff12 = setdiff(names(p1), names(p2)) # in p1 but not p2
  diff21 = setdiff(names(p2), names(p1)) # in p2 but not p1

  p1[,diff21] <- 0
  p2[,diff12] <- 0
11   p1 <- p1[,order(names(p1))]
12   p2 <- p2[,order(names(p2))]
13
14   a1 = colSums(p1)
15   n1 = length(p1[,1])
16   a2 = colSums(p2)
17   n2 = length(p2[,1])
18   nn1 = n1 * (n1-1)
19   nn2 = n2 * (n2-1)
20   q1 = a1/n1
21   q2 = a2/n2
22   N = (q1 - q2)^2 - (( ( a1*(n1-a1) ) / nn1 ) / n1) - (( (a2*(n2-a2) ) /
         ↪ nn2 ) / n2)
23   D = N + ( (a1*(n1-a1)) / nn1 ) + ( (a2*(n2-a2)) / nn2 )
24   fst = sum(N) / sum(D)
25
26   print(fst)
27 }
```

A call to the function `calc_fst()` requires two arguments: the filenames of two diploid haplotype files whose production was detailed in the previous section. Because it is likely that some or many of the polymorphic sites in one deme are monomorphic in the other deme, we must identify these category 2 sites so we can add a placeholder column to the deme lacking polymorphism (lines 6–7). To this end, we append a column of zeroes to indicate there is no variation at this site in the deme (lines 9–10). After supplementing the two data frames (`p1` and `p2`) with the necessary monomorphic data, we then reorder both data frames so that the indexing of the columns (SNPs) is the same (lines 11–12). Lines 14–24 then use these reordered data frames to calculate the multilocus value of F_{ST} detailed in Reich et al. (2009), which is printed to screen (line 26).

Although `calc_fst()` should technically be used with a set of fully unlinked SNPs, we will apply it using SNPs present in simulated 200,000 bp sequences. To do so, we modify the simulation scenario presented in Fig. 6.6, by only simulating the population split and significantly shortening the timescale such that we sample demes 1 and 2 for only 900 generations following their split from deme 0. Effective population sizes remain the same. Three replicate simulations each were run for $m = 0.01$ and $m = 0.00001$. As expected, low levels of gene flow lead to rapid genetic divergence of the two populations and vice-versa (Fig. 6.9). Note that even when m is as low as 0.00001, F_{ST} appears to approach an asymptotic value in ¡1000 generations.

6.4 Printing Allele History File

Now we leverage the explicit, time-stamped record generated by simulation to provide us with insight into the history of alleles segregating in one or

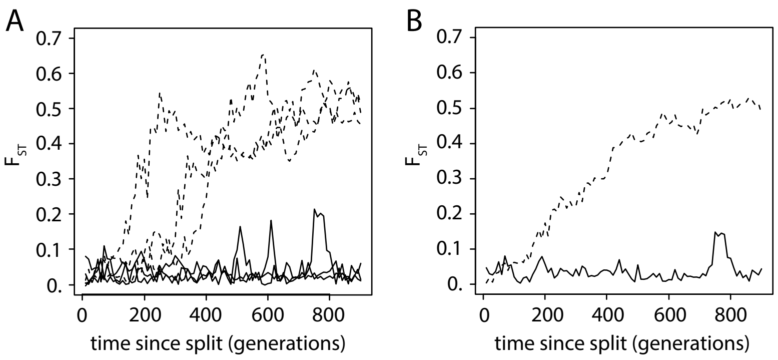


Fig. 6.9 Calculated F_{ST} between demes 1 and 2 descended from ancestral deme 0 as picture in Fig. 6.6. (**a**) Calculated values of F_{ST} for three independent replicates each where $m = 0.01$ (solid lines) or $m = 0.00001$ (dashed lines). (**b**) The mean values of F_{ST} over the three replicates

more demes. These histories are not something we can assess with certainty in the natural world. Perhaps more importantly in the context of this volume, these histories are also unavailable to us from coalescent simulation. Although forward-time simulations are much slower than coalescent simulations by virtue of the fact that all individuals and all alleles are accounted for, this computational cost provides us with additional information that motivates simulation experiments that require greater complexity. The following listing provides the means to document the origins of alleles emerging in all simulated demes. I then detail a problem the reader can solve to find the proportion of alleles in deme x that originated in another deme y for different values of $N_e m$.

Modifications to multiple FORTUNA source files to implement documentation of allele history

```
\\ parameters file (global block)
trackAlleleBirths 1

// params.h file
extern bool trackAlleleBirths;

// params.cc file
int process_parameters() {
   ...
    if (iter->first == -1) {
      ...
      trackAlleleBirths = atoi(parameters["trackAlleleBirths"].c_str());
    }
}

// population.h file
private:
```

```
...
ofstream abf;
...
void get_sample(int gen) {
   ...
    if (printhap) {
      ...
      for (auto iter = positions.begin(); iter != positions.end(); ++iter)
         sequencefile << (alleles[*iter]) -> get_originating_population() << " ";
      sequencefile << endl;
      for (auto iter = positions.begin(); iter != positions.end(); ++iter)
         sequencefile << (alleles[*iter]) -> get_birthgen() << " ";
      sequencefile << endl;
      ...
   }
   ...
}
...
vector<vector<int> > mutate(const vector<int> &parents, const int &gen) {
   ...
   for (int i=0; i<2; ++i) {
      for (int j = 0; j < mutnum[i]; ++j) {
         ...
         if (alleles.find(position) == alleles.end()) { // new mutation to a derived allele in the population
            alleles.insert({position, new Allele(position, gen, popn)}); //updated with popn
             if (trackAlleleBirths) // new statement
               abf << "nt" << position << "\t" << gen << "\t" << popn << endl;
            ...
            }
         ...
      }
}
...
public:
...
void close_output_files() {
    ...
    if(trackAlleleBirths) abf.close(); // new line
    ...
}
...
Population (int popnum, int eextant):popn(popnum), extant(eextant) {
   ...
   if (useMS[popn]) {
       ...
        while(getline(ms_output, ms_line)) {
            if (regex_search(ms_line, query)) {
                ...
                while(iss >> s) {
                    ...
68              alleles.insert( { position , new Allele(position,-1,popn) }
                     ↪ ); //updated line with popn
69            }
70         }
71      ...
72      }
73      ...
74   }
75   ...
76   // Print new alleles to abf, if applicable
77    string ofname = "deme" + to_string(popn) + "_allele_births";
78   if (trackAlleleBirths)
79      abf.open(ofname.c_str());
80   ...
81 }
82 
83 // allele.h file
84 ...
85 private:
86     int originating_population;
87 public:
88   inline int get_originating_population() { return
         ↪ originating_population; } // new inline function
89    // modify constructor to include additional parameter
90   Allele (int pos, int gen, int op): position(pos), birthgen(gen),
         ↪ originating_population(op) {
91      ...
92   }
```

Lines 1–14 illustrate how the `extern bool` variable `trackAlleleBirths` is read from the `parameters` file. A value of 1 indicates that we will keep track of allele births. In the `population.h` file, we add the additional `private` variable `ofstream abf`, which provides us with a buffer to which allele births will be printed (line 19). Lines 51–57 in the class constructor create a name for the output file and open the `abf` buffer with this name. Within the `mutate()` function of `population.h`, for each new mutation we add lines 43–44, which print the position of the mutation, current (origin) generation of the allele, and the deme number of the originating deme to the allele history file. If haplotype files are generated, we now add an additional two header lines to the haplotype file; the first line lists the nucleotide position of each segregating site among the sampled haplotypes, while the second lists the generation of origin of the segregating site in each column. Note that this change requires us to slightly modify the `calc_fst()` R function detailed in the previous subsection, as the first two lines will not consist of haplotypes but, rather, metadata. See online code for the requisite change.

Next, we write an R function that allows us to identify the origins of alleles from haplotype files. Recall that the names of these files take the form of `deme1_500`, which is the `sequencefile` (actually haplotypes) for deme 1 at generation 500.

analyze_alleles.r

```
 1  library(ggplot2);
 2  library(reshape2);
 3  library(cowplot);
 4
 5  analyze_allele_origins <- function(timepoints, focaldeme, origindemes,
       ↪ imagefile="allele_origins.pdf")
 6  {
 7    q = matrix(nrow=length(timepoints), ncol = length(origindemes)+1); # 1
        ↪ extra for generation
 8    for (i in 1:length(timepoints)) {
 9      t <- read.table (file = paste("deme", focaldeme, "_", timepoints[i],
            ↪ sep = ""), header = T);
10      q[i,1] = timepoints[i];
11      for (j in 1:length(origindemes)) {
12        q[i,1+j] = sum(t[1,] == origindemes[j]);
13      }
14    }
15    n <- c("gen");
16    for (i in 1:length(origindemes)) {
17      n <- c(n, paste("d",origindemes[i],sep=""));
18    }
19    q <- as.data.frame(q);
20    names(q) <- n;
21    mq <- melt(q, id.vars="gen");
22    gg <- ggplot(mq, aes(gen)) + geom_point(aes(y=value, colour=variable));
23    gg <- gg + geom_line(aes(y=value, colour=variable));
24    #gg + scale_colour_manual(values = c("snow2", "snow4", "black"));
25    gg;
26    ggsave(imagefile);
27    return(q);
28  }
```

The parameters of the function `analyze_alleles()` (line 5) include:

- `timepoints`: a `seq( )` (e.g., `seq(500, 5000, 500)`) of the generations when haplotype files were generated
- `focaldeme`: number of the focal deme
- `origindemes`: a `vector( )` of possible demes from which alleles may have originated by mutation
- `imagefile`: the name (inducing the .pdf extension) of the file to save the plot produced by the function; file name is `allele_origins.pdf` by default

Note that I have commented out line 24 because you may wish to simply allow `ggplot2` to assign default colors to each curve. If you do specify colors of your choice using line 24, however, you must make sure the number of listed colors are less than or equal to the length of the `origindemes` vector.

Modify the `parameters` file to simulate an ancestral deme of size N_e = 10000 (`useMS=1`), which splits into demes x and y, each with N_e = 5000, and allows both demes to evolve for 2000 generations. Run multiple simulations, changing the value of m each time, to obtain results across a wide range of gene flow. Use R to read results files and produce plots of the proportion of polymorphic sites in deme x whose derived allele originated in deme y versus the value of m. Try plotting the x-axis on a logarithmic scale. Do the results confirm your expectations? Practice explaining why the results are as they are.

6.4.1 Results and Validation

We begin with three simulations. In all three, deme 0 (N_e = 10000) is simulated using `mscommand ./ms 20000 1 -t 80 -r 80 200001 >ms_output`. Note that recombination is explicitly modeled and a sequence of length 200,000 bp is simulated. This deme remains whole for 100 generations, after which it splits into demes 1 and 2 evenly, i.e., each begins with N_e = 5000 randomly assigned individuals. Two of the simulations are replicates of a scenario of symmetrical migration in which $m_{1,2} = m_{2,1} = 0.005$ and the *per-site* recombination rate of the simulated 200,000 bp sequence is $r = 10^{-9}$. We sample haplotypes every 500 generations and set `trackAlleleOrigins` to 1 and `runlength` to 50001 (generations). The output haplotype files have the names `deme1_X` and `deme2_X`, where `X` is the sampled generation. We then apply the R function `analyze_alleles` detailed in the previous section to generate a graph for each deme:

- `d1 = analyze_allele_origins( seq(500, 50000, 500), 1, origindemes = c(0,1,2), imagefile="deme1.pdf"`
- `d2 = analyze_allele_origins( seq(500, 50000, 500), 2, origindemes = c(0,1,2), imagefile="deme2.pdf"`

The objects `d1` and `d2` store the table used to generate the two PDFs that are each saved to file. Note that the first argument, `timepoints`, is given a sequence running from the generation number of the first haplotype sample (generation 500) to the last haplotype sample (generation 50,000) by steps of 500.

The results of the two replicates are shown in Fig. 6.10a, b. Because $4N_em = 20{,}000 \times 0.005 = 1000 >> 1$, we expect the two demes to exchange genes at a sufficiently high rate for the two demes to become one panmictic population. This is confirmed by the fact that alleles originating by mutation in demes 1 and 2 are found in roughly equal numbers in both demes in both replicates (dark gray and black lines). As we would expect, alleles that originated in ancestral deme 0 decline over time as they are randomly lost from both demes (Fig. 6.10a, b; light gray lines). However, the step-like pattern of this decline in deme-0 origin alleles may seem curious. For example, the last 20,000 generations of Fig. 6.10a show no decline in deme-0 origin alleles. Upon examining individual haplotype files, I found that there were two haplotypes of deme-0 origin alleles faithfully transmitted from generation to generation and between demes 1 and 2. However, this pattern is much different than the jagged increase in deme-1 and deme-2 origin alleles shown in Fig. 7.10a, b). Furthermore, upon running a simulation in which r for the forward simulation was increased by an order of magnitude to 10^{-8}, the decline in deme-0 origin alleles is smoother (Fig. 7.10c).

Observations such as these in our simulation results should give us some pause. *Most often, patterns that strike your intuition as odd are not indicative of a newly discovered population genetic phenomenon; rather they are indicative of something overlooked in the simulation process.* We might question, for example: Is the difference (jagged increases vs. step-like declines) due to the fact that deme-0 alleles were primarily derived from coalescent simulation, while deme-1 and deme-2 origin alleles were exclusively the product of mutations in these demes during the forward-time portion of the simulation? It is worth validating the "reality" of a seemingly anomalous observation whenever possible.

My first thought was that perhaps a per-site r of 10^{-9} was simply too low to break up the haplotypes present after the coalescent simulation. Inspection of haplotype files confirmed "stubborn" blocks of tightly linked alleles that were by chance not mixed by the process of crossing over. Still, curious. Shouldn't both demes be purged of all deme-0 origin alleles after 50,000 generations? And why do the increases in deme-1 and deme-2 origin alleles oscillate so much?—indicating that haplotypes of these alleles are never given the chance to evolve.

In Fig. 6.10, each panel shows that the increase in number of derived alleles originating from deme 1 and 2 is similar but not even. Consider Fig. 6.10a (left). It shows us that in deme 1, there are always slightly more alleles originating from deme 1 than deme2. The opposite of this is true in Fig. 6.10a (right). Is this more likely:

- **a signal that the two demes are *not quite* a panmictic population?**
- **or, simply due to what must be a relatively large number of exceedingly rare alleles recently produced in the focal deme?**

Use results of a simulation to determine if the hypothesis in point 2 is actually true. For example, in deme 1 is there an excess of rare alleles that originated in deme 1?

In this case, I reasoned that if the step-like decline of alleles derived from an ancestral population is real, we should see a similar pattern in a subsequent

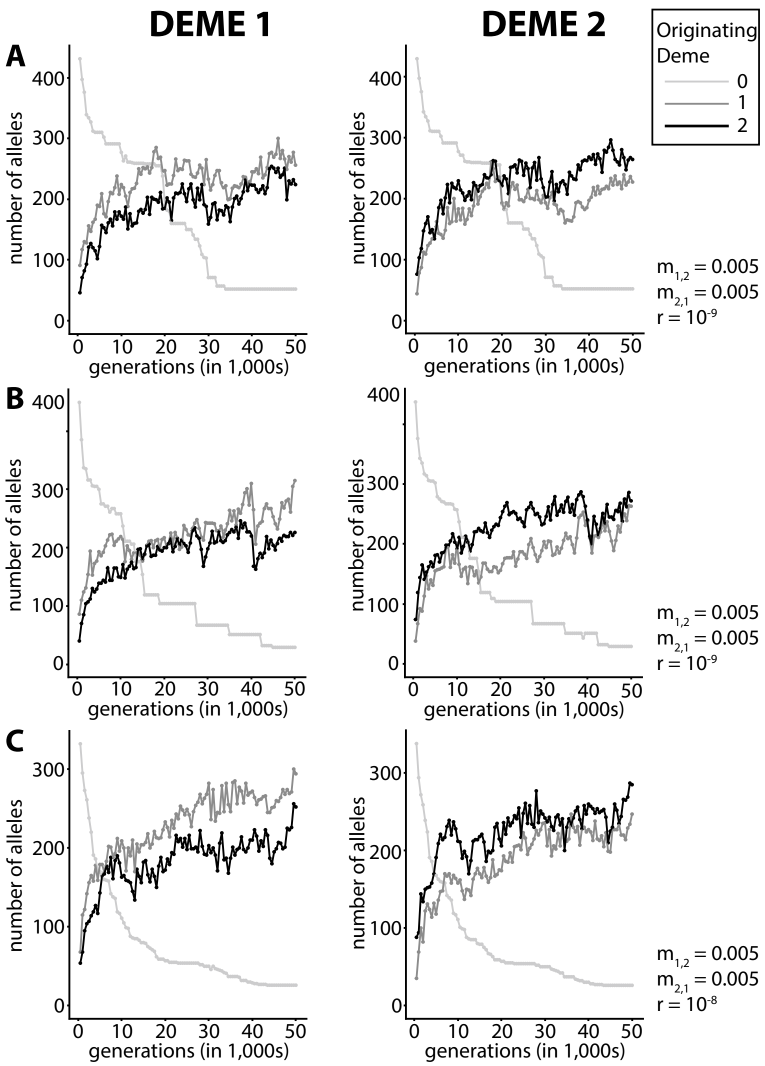


Fig. 6.10 Symmetric migration between two demes. Each **row** shows the results of a separate simulation; deme 1 results are in the left column and deme 2 results are in the right column. In each graph, the three lines document the number of derived alleles in the focal deme that originated by mutation in deme 0, 1, or 2 (see legend). The data to generate these graphs were read from the allele information file, whose construction is detailed at the beginning of Sect. 6.4

split of either deme 1 or deme 2. Therefore, I altered the `parameters` file such that two additional demes (3 and 4) were $N_e = 2500$ descendants of deme 1 splitting in two at generation 50,000. Deme 2 was allowed to go extinct at generation 50,001 and demes 3 and 4 were simulated for an additional 50,000 generations (Fig. 6.11). The results validated the step-like decline of alleles originating in ancestral demes (solid dark gray and black lines in the bottom row of graphs in Fig. 6.11), thereby seeming to rule out the possibility that this pattern was a result of coalescent simulation.

With this certainty in mind, we can move on to explaining the difference between jagged increases and step-like declines in allele counts from ancestral and extant demes. The jagged increase in allele counts originating in extant demes is due to the fact that new mutants necessarily begin at a frequency of $1/2N_e$, which is also the probability that each will rise to fixation. The effect is that the vast majority of new alleles are lost rapidly. Alleles originating in extant demes are mostly appearing and then blinking out. On the other hand, alleles inherited from an ancestral deme that hang around for an appreciable number of generations were likely at a high frequency when the ancestral population splits in two. They decline in number over time because no new alleles from extinct demes are entering the extant demes. Yet the high-frequency alleles that are present in descendant demes are tightly linked to each other and must be eliminated together—though at a rate contingent on recombination frequency. This takes a long time, particularly as these ″sub-haplotypes″ of ancestrally derived alleles are ping-ponging back and forth between the extant demes connected by gene flow.

Next, we look briefly at the results of three simulations in which the symmetrical migration rate is low ($N_em = 1$; Fig. 6.12a) or migration rates are asymmetrical (Fig. 6.12b, c). Symmetrical migration where $N_em = 1$ shows a noisy but overall increasing number of deme-1 origin alleles in deme 2 and vice-versa (Fig. 6.12a). Extrapolating this trend—i.e., given sufficient time—we would expect both demes to become essentially panmictic. However, this contrasts sharply with the results shown in Fig. 7.10a, b, where we see an instantaneous equal sharing of alleles between demes 1 and 2.

In the cases where migration flows in *only* the deme 1 to deme 2 direction (Fig. 6.12b) or is two orders of magnitude greater in the deme 1 to deme 2 direction (Fig. 6.12c), we find the rather remarkable result that deme-1 origin alleles vastly outnumber deme-2 origin alleles in deme 2. Again, we need to assess everything we are doing, including the parameter values simulated and the code itself, to make sure this is not an artificial pattern. In this case, if you will trust me, the code checks out. We must therefore form a hypothesis for this counter-intuitive result.

Note that the counts of deme-1 origin alleles are very similar over time in both demes. One way to interpret these patterns is that new alleles are ″bred″ in deme 1 and soon shared with deme 2. Another way of putting this is that both demes are panmictic for deme-1 origin alleles, but not deme-2 origin alleles because the latter are not ″shipped″ to deme 1. Also of note is the fact that deme-2 origin alleles (black lines) appear to be at equilibrium from the very first sample at 500 generations (400 generations after the split between

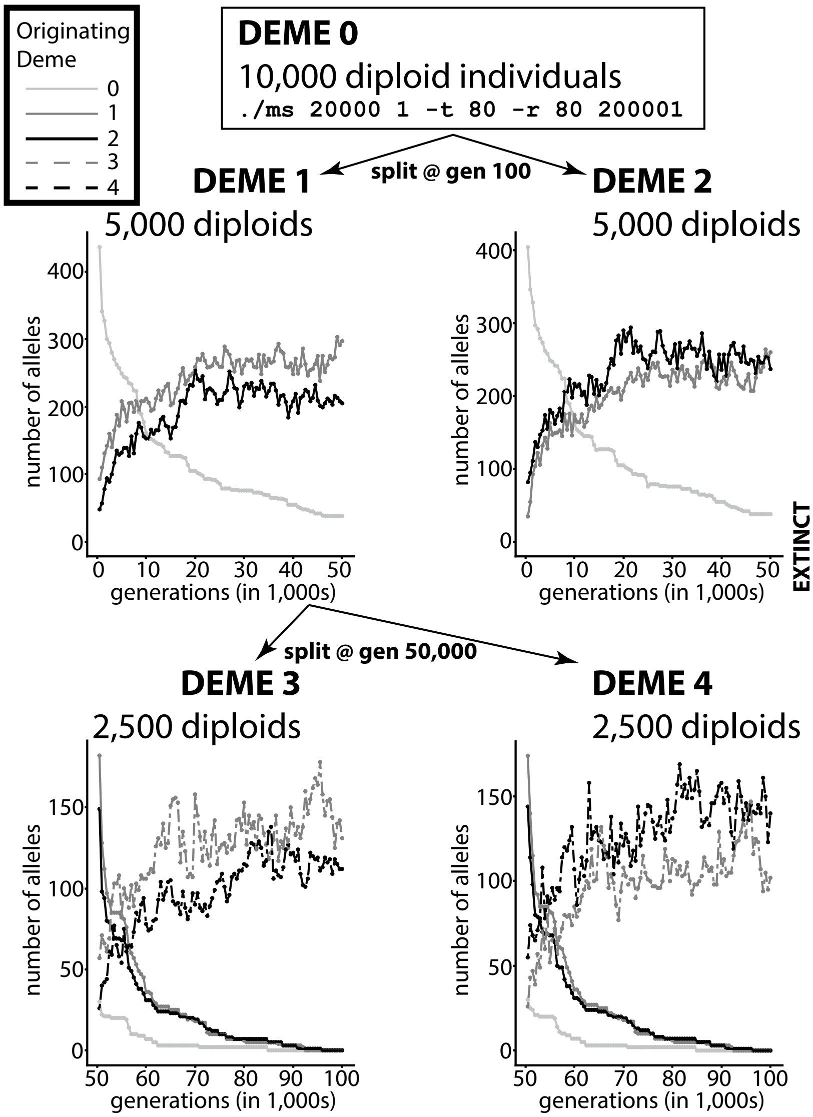


Fig. 6.11 Validation of the step-like decline of alleles derived from ancestral demes. The schematic of the simulation is shown, in which deme 0 is generated by coalescent simulation, splitting into demes 1 and 2 at generation 100. At generation 50,000, deme 2 goes extinct and demes 3 and 4 are the result of an even split of deme 1. Given their history, demes 3 and 4 include a mixture of alleles originating from all five demes simulated. Importantly, alleles originating from the ancestral demes 0, 1, and 2 all show step-like declines over the course of 50,000 generations

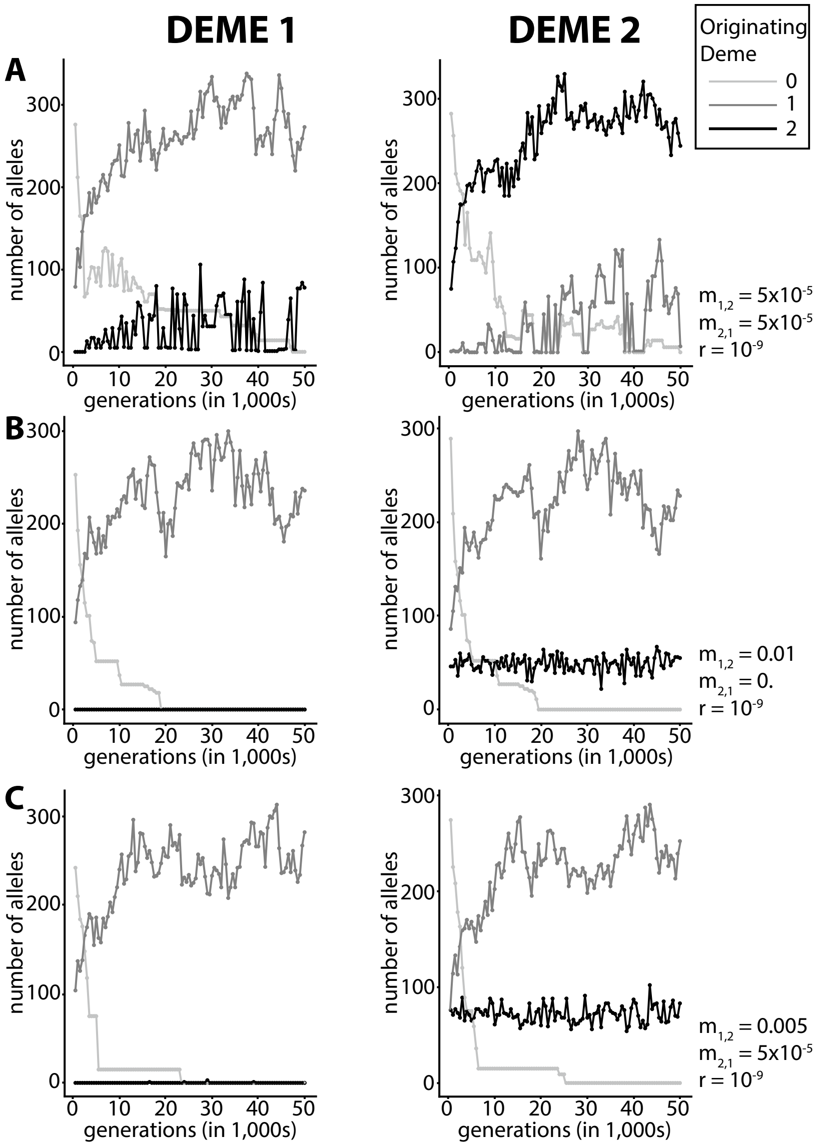


Fig. 6.12 Minimal or asymmetric migration between two demes. Details of the graphs are the same as in Fig. 6.10

demes 1 and 2) and that the equilibrium number of deme-2 origin alleles is much lower. We can explain this result using the logic that deme-2 origin alleles are sequestered in their evolution within a smaller population of 5000 individuals, while deme-1 origin alleles are essentially evolving within a population of 10,000 individuals (deme 1 + deme 2). Finally, note that asymmetric migration results in much more rapid die-off of deme-0 origin alleles in deme 1 which receives zero or minimal numbers of migrants from deme 2 (cf. Deme 1 panels of Fig. 6.12b and c).

References

Nordborg M (1997) Structured coalescent process on different time scales. Genetics 146:1501–1514

Reich D, Thangaraj K, Patterson N, Price A, Singh L (2009) Reconstructing Indian population history. Nature 461:489–494

Rice SH (2004) Evolutionary theory: mathematical and conceptual foundations, 1st edn. Sinauer Associates, Sunderland

Spieth PT (1974) Gene flow and genetic differentiation. Genetics 78:961–965

Wakeley J (2008) Coalescent theory: an introduction. Roberts and Company Publishers, Greenwood Village

Whitlock MC, McCauley DE (1999) Indirect measures of gene flow and migration: $f_{ST} \neq 1/(4nm+1)$. Heredity 82:117–125

Wright S (1931) Evolution in mendelian populations. Genetics 16:97–159

Wright S (1951) The genetical structure of populations. Ann Eugenics 15:323–354

7 Natural Selection

Around her aged loins, she gathered a whole nation of humble ancient deaths, shades long silent ... This body of ours, this disguise put on by common jumping molecules, is in constant revolt against the abominable farce of having to endure.[1]
– Louis-Ferdinand Céline, *Journey to the End of the Night*

7.1 Background and Theory

Beginning biology students often place an identity sign between natural selection and biological evolution. This is a case of mistaken identity. After all, we have already encountered numerous other determinants of molecular and, therefore, phenotypic evolution. These include mutation, sampling variability due to finite population size, recombination of linked genetic loci that can generate novel haplotypes in a population, and effective migration between semi-isolated demes.

One critical difference between natural selection and other evolutionary factors is that natural selection can act as a true *driver* of evolution. For example, unlike the stochastic action of genetic drift, positive directional selection drives the frequency of an adaptive allele *up*. Conversely, purifying selection drives the frequency of a deleterious allele *down*. Yet, the line that connects random mutation to adaptive phenotype is so conceptually bright that it tempts us into *perhaps* overly simplistic accounts of biological evolution, i.e., just-so stories (a pejorative term for a simplistic, pat hypothesis of adaptation)

[1] JOURNEY TO THE END OF THE NIGHT, copyright 1934, 1952 by Louis-Ferdinand Celine, translation copyright 1983 Ralph Manheim. Reprinted by permission of New Directions Publishing Corp.

R. J. Haasl, *Nature in Silico*, https://doi.org/10.1007/978-3-030-97381-0_7

of the general form: mutation x fixed because phenotype y to which it maps is adaptive in environment z. When true, it is a beautiful discovery to find clear adaptive connections of this form. In the absence of strong, empirically based evidence supporting adaptation, however, we must remember that the quantity and character of extant genetic variation are functions of many genetic and (in the case of complex traits) environmental factors.

Storytelling focused on the natural history, evolution, and defining adaptations of a species is central to dissemination of knowledge in evolutionary biology. The impulse to tell stories of evolution is strong enough that it is acceptable to use teleological phrases such as "Wings are for flight" to help explain the adaptive significance of a trait despite the general antipathy of evolutionary biologists toward teleology (Okasha 2018). Often missing from these stories is the death correlated with adaptation. Differential fitness implies loss of life, the "humble ancient deaths" of those who were slightly less-fit and, therefore, not as long-lived. Death of the less-fit is the morbid but inevitable *cost of selection*.

Another reason natural selection holds so much sway over our minds is the concept's association with the most famous opening act of evolutionary biology: Charles Darwin, M.A., F.R.S., F.G.S., &c. The illustrious pedigree of the concept sometimes makes it feel impossible to overstate the importance of natural selection to evolution. Important historical counterpoints to the assumed predominance of natural selection include Gould and Lewontin (1979) and the (semi-)neutral theory of Motoo Kimura and Tomoko Ohta. I personally believe it *is* possible to overstate the importance of natural selection. But of course natural selection *is* important. Supporting evidence? I devote this chapter *and the next* to the topic.

7.1.1 Natural Selection as Optimization?

The branch of computing called evolutionary computation uses algorithms that mimic organic evolution. In a standard algorithm of this kind, fragments of code play the part of alleles and are subject to recombination, mutation, and natural selection during each iteration of a `for` loop, i.e., generation. Based on a fitness function that quantifies the success of the evolving code, the code self-optimizes to perform a specific task. Computer code of greater fitness is passed on to the next iteration of the algorithm with greater probability, thereby improving the efficiency and/or efficacy of the next "generation" of code fragments.

The danger of equating natural selection with an optimization process is that it may cause some to envision biological evolution as a process that leads to ever-better "product"—with the implication that generation t is better equipped to handle the dangers and insults of the world than generation $t-1$, just as *it* was better prepared than generation $t-2$. Sewall Wright's famous and forever-debated metaphor of an adaptive landscape is one manifestation of this view, in which populations are imagined climbing hills of the adaptive landscape whose altitude is measured in mean population fitness ($\bar{w}$, see Sect. 7.1.2.1) (Wright 1932). Ronald Fisher's contemporaneous "fundamental theorem of natural selection" predicts the change in $\bar{w}$ due to the effects of natural selection alone (Fisher 1930). The fundamental theorem of natural selection has, on occasion, also been interpreted as a mathematical result that implies mean population fitness always increases and thus a core statement in favor of the optimizing capacity of natural selection. However, the idea of ever-increasing fitness is not true and it was not Fisher's intent (Edwards 2002).

The most obvious problem with treating natural selection as an organic optimization algorithm is that the environment is in constant flux; the optimal combination of genotypes is therefore ever-shifting. The situation reminds me of the eponymous Luke played by Paul Newman in *Cool Hand Luke* (1967). Luke is instructed to dig a hole and upon doing so is asked why the hole is empty. Then, after completing the refilling, Luke is asked what all that dirt is doing in the hole. In a similar way, a population "strives" toward the current, genotypic ideal only to have that ideal cruelly changed from time to time.

In his thorough examination of fitness optimization by natural selection, Samir Okasha (2018) identifies two additional reasons; the idea is controversial. First, populations only bear a limited number of *available* genotypes and, therefore, phenotypes and adaptations. Thus, if natural selection is a process of optimization, it is a conditional optimization; the total set of *conceivable* phenotypes are not within the reach of any population. Second, in addition to the vulnerability of an adaptive allele at low frequency, it is important to remember that there are forms of selection that are not expected to lead to the fixation of a particular allele. Examples include negative frequency-dependent selection (Sect. 7.2.2) and overdominance (Sect. 7.2.3).

It seems safe to conclude that the classic, Darwinian view of natural selection in which an adaptive variant emerges by mutation and rises to fixation is a type of optimization. In addition, despite reasonable philosophical objections, I find the adaptive landscape metaphor to be a useful conceptual and pedagogical tool as long as we are honest about its limitations. If humans could visualize more than three or four (with color-coding) dimensions, complaints about misleading labeling of axes on the adaptive landscape would be moot. Both the adaptive landscape and the view of natural selection as an optimization process can prove valuable. We must simply work carefully.

7.1.2 Fitness

Fitness is the mean reproductive output of an individual with a given genotype or set of genotypes. In population genetics, fitness is often defined with reference to genotype at a single locus. In this case, fitness of each genotype at the focal locus is implicitly averaged across all genetic backgrounds. In a real sense, this is not a satisfactory model of evolution; an individual's true fitness is determined by variation at *all* loci determinative of phenotypes that affect the viability or fecundity of an individual. However, most empirical studies do focus on a single or small set of loci that impact fitness. It is therefore reasonable to consider fitness as a function of this small number of loci, treating the fitness of each genotype as a fitness marginalized over all possible genetic backgrounds. In practice, we do not explicitly marginalize fitness; we just do not have the information regarding genetic backgrounds to make this calculation. Conceptually, however, it is important to keep in mind that the fitnesses of genotypes at a given locus isolate the effect of genetic variation at this locus from the genetic background as a whole.

The following discussion of the different "faces" of fitness and their roles in deriving Eqs. 7.6 and 7.7—which describe the effects of one generation of positive natural selection on frequency of an adaptive allele—follows Rice (2004). I find this derivation logical and intuitive.

The most direct measure of the fitness of a genotype is *absolute fitness*, which is expressed in terms of the average number of offspring produced by individuals of each genotype. Consider a locus with two alleles—A and a—in a diploid species. Although empirical determination of absolute fitness is laborious, imagine that a research team has determined the following absolute fitnesses of the three possible genotypes:

genotype	**absolute fitness**
A/A	10
A/a	9.5
a/a	9

Although absolute fitness is an intuitive measure, it is more instructive to express fitness in relative terms. *Relative fitness* is obtained by dividing each absolute fitness value by the maximum absolute fitness. In this way, the table of absolute fitnesses is converted to the following table of relative fitnesses, which has the attractive advantage of normalizing all fitness values to a maximum relative fitness of 1.0:

genotype	**relative fitness**
A/A	$w_{A/A} = 10/10 = 1.0$
A/a	$w_{A/a} = 9.5/10 = 0.95$
a/a	$w_{a/a} = 9/10 = 0.9$

A *selective regime* consists of the biotic and abiotic factors that collectively determine the expected outcome of natural selection. I now introduce a simple two-parameter model of relative fitness values that, however indirectly, is meant to reflect the reproductive consequences of the selective regime on individuals with distinct genotypes at a locus. It is important for the reader to understand that *I will refer to the set of relative fitness values as a selective regime throughout*. I do this for ease of communication. Despite this bow to facility, remember that the modeled relative fitness values are a locus-specific *consequence* of the active selective regime and, therefore, a *reflection* of the selective regime—and not the selective regime itself, as commonly defined. The two parameters of our model are (1) the selection coefficient, s, and (2) the dominance coefficient, h. At a diallelic locus, the relative fitness values of genotypes, from most- to least-fit, are then 1., $1-hs$, and $1-s$. Using parameter values $h = 0.5$ and $s = 0.1$, we recover the values of relative fitness shown in the previous table:

genotype	**relative fitness**
A/A	$w_{A/A} = 1.0$
A/a	$w_{A/a} = 1. - hs = 0.95$
a/a	$w_{a/a} = 1. - s = 0.9$

With respect to simulation, the definition of a selective regime in terms of h and s offers great flexibility. A dominant selective regime holds when $h = 0.$, and an additive selective regime holds when $h > 0$. Meanwhile, s controls the strength of selection regardless of the value of h. Of course, there are many situations—e.g., overdominance—that cannot be captured by an unaltered version of this model no matter the chosen values of h and s.

7.1.2.1 Marginal Fitness and Mean Population Fitness

In words, mean population fitness is defined as the average relative fitness of an individual in a population. Mathematically, it is easiest to express this quantity $\bar{w}$ as a function of allele frequencies and their values of *marginal fitness*.

Informally, marginal fitness can be thought of as the average fitness of an allele in a population. *Marginal* is used in the same sense as in a marginal probability distribution, where the probability of a random variable is calculated (marginalized) over the probabilities of all other random variables under consideration. Clearly marginal fitness is an abstraction in diploid organisms, as natural selection acts upon differences in phenotype that are dependent on genotype rather than a single allele. However, calculation of marginal fitnesses does possess some heuristic value and makes calculation of $\bar{w}$ easier.

To see how marginal fitness is calculated, again consider a diallelic locus with alleles A and a. Thus, there are three possible genotypes—A/A, A/a, and a/a—and each allele is found in the context of two of the three genotypes. To obtain the marginal fitness of allele A, we first ask the question: How probable is it that an A allele finds itself in a homozygous genotype (i.e., A/A)? The answer is simply the probability that the other allele of the genotype is also A, which is equal to the frequency of A, or p. Second, we ask: How probable it is that an A allele finds itself a member of the heterozygous genotype A/a? The answer is simply the probability that the other allele of the genotype is a, which is equal to the frequency of a, or q. Finally, we weight the relative fitness of genotypes A/A and A/a by the probability of the other allele in the genotype A/_—which is simply p or q—to obtain the marginal fitness of allele A:

$$w_A = pw_{A/A} + qw_{A/a} \tag{7.1}$$

Following similar logic, the marginal fitness of allele a is:

$$w_a = pw_{A/a} + qw_{a/a} \tag{7.2}$$

With marginal fitnesses in hand, the mean fitness of individuals in the population is easy to calculate:

$$\bar{w} = pw_A + qw_a \tag{7.3}$$

Here, we weight the marginal fitness of each allele by its frequency in the population.

Marginal fitnesses can also be expressed in terms of our basic model of single-locus selection as functions of p, q, s, and h:

$$w_A = pw_{A/A} + qw_{A/a} = p(1) + (1-p)(1-hs) = 1 - hsq \tag{7.4}$$

$$w_a = pw_{A/a} + qw_{a/a} = p(1-hs) + (1-p)(1-s) = 1 - hsp - qs \tag{7.5}$$

7.1.2.2 Deterministic Changes to Allele Frequencies

We now calculate *deterministic* changes to the frequency of the adaptive allele, p. These changes are deterministic in the sense that the stochastic effects of finite population size are ignored. Even a highly adaptive allele is often lost in stochastic simulations (and, presumably, in nature) simply because it emerges on a single chromosome in a diploid population at a frequency of $1/2N$. In the deterministic simulation that follows, we isolate the effects of natural selection, and the rise in frequency of the adaptive allele will be identical between runs whenever the simulation is run with the same parameter values.

The frequency of the favored A allele in the next generation is:

$$p_{t+1} = p_t \frac{w_A}{\bar{w}}, \tag{7.6}$$

where p_{t+1} is the frequency of the A allele after one generation of natural selection. Note that the term $\frac{w_A}{\bar{w}}$ controls whether the allele increases or decreases in frequency. If this quantity is > 1, the current frequency of A (p_t) will increase and if this quantity is < 1, the current frequency of allele A will decrease. This should make intuitive sense. If the marginal fitness of A alleles is greater than mean population fitness $\bar{w}$, we expect A alleles to be passed on to the next generation with greater frequency than a alleles. In other words, the term $\frac{w_A}{\bar{w}}$ measures the expected bias (in this case, positive) of transmission of allele A to the next generation due to natural selection.

Finally, the deterministic *change* in the frequency of an allele in one generation can be expressed as:

$$\Delta p = p_{t+1} - p_t = \frac{p_t w_A}{\bar{w}} - \frac{p_t \bar{w}}{\bar{w}} = \frac{pq(w_A - w_a)}{\bar{w}} \tag{7.7}$$

The following R script simulates the deterministic trajectory of a favored allele under positive natural selection for the model discussed in Sect. 7.1.2 and parameterized by user-input values of N, s, and h.

deterministic_selection.r

```
deterministic_selection <- function(N, s, h, epsilon)
{
  timecount <- 0
  p <- 1 / (2*N)
  fitness <- c(1-s, 1-h*s, 1)
  marg_fitness <- calc_marginal_fitness(fitness, p)

  gen <- c(timecount)
  frequency <- c(p)
  popfit <- c(calc_population_fitness(p, marg_fitness))


  while (p <= 1-epsilon) {
    gen <- c(gen, timecount <- timecount + 1)
    marg_fitness <- calc_marginal_fitness(fitness, p)
    wbar <- calc_population_fitness(p, marg_fitness)
    frequency <- c(frequency, p <- p * ( marg_fitness[2] / wbar ))
    popfit <- c(popfit, wbar)
  }

21   d <- data.frame("gen" = gen, "frequency" = frequency, "popfit" = popfit)
22   return(d)
23 }
24
25 calc_marginal_fitness <- function(fitness, p)
26 {
27   q <- 1-p
28   mfit <- c(q*fitness[1] + p*fitness[2], p*fitness[3] + q*fitness[2])
29   return(mfit)
30 }
31
32 calc_population_fitness <- function(p, mfit)
33 {
34   q <- 1-p
35   wbar <- q*mfit[1] + p*mfit[2]
36 }
```

In addition, to N, s, and h, the function `deterministic_selection()` takes the argument ϵ, which determines how long the function will run (line 13). The motivation for this detail is that under deterministic conditions it will take a very large number of generations to obtain fixation of the favored allele even though fixation is *approached* after a much smaller number of generations.

Run `deterministic_selection()` using the following arguments:

- `deterministic_selection(10000, 0.01, 0., 0.003)`
- `deterministic_selection(10000, 0.01, 0.5, 0.000001)`
- `deterministic_selection(10000, 0.1, 0., 0.003)`
- `deterministic_selection(10000, 0.1, 0.5, 0.000001)`

The first and third commands are for a dominant selective regime, while the second and fourth commands are for an additive selective regime in which the heterozygote (A/a) has a relative fitness centered exactly between the relative fitnesses of the two homozygotes (A/A and a/a) because $h = 0.5$.

Before examining the deterministic results, however, we pause to consider the Hardy-Weinberg law, which helps explain the differing trajectories of an increasing p (the frequency of the favored allele) between additive and dominant selective regimes. The Hardy-Weinberg law for two alleles is expressed as the quadratic equation

$$p^2 + 2pq + q^2 = 1, \tag{7.8}$$

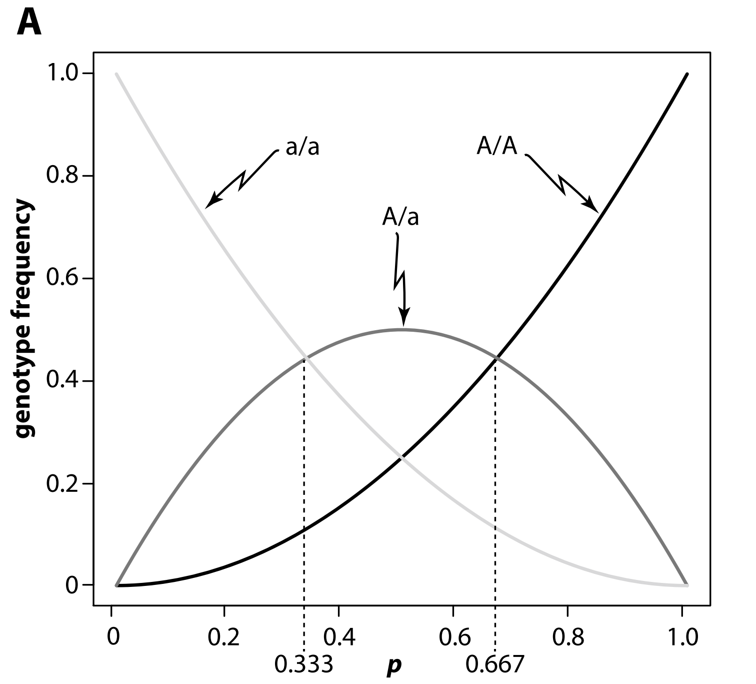
where each of the terms on the left-hand side provides the expected frequency of the three possible genotypes. Specifically, $f_{A/A} = p^2$, $f_{A/a} = 2pq$, and $f_{a/a} = q^2$. Figure 7.1a shows the genotype frequencies expected for different values of p, from 0 to 1. Two key values of p are indicated. First, when $0 \geq p < \frac{1}{3}$, a/a genotypes outnumber A/a genotypes. Second, when $\frac{2}{3} < p \leq 1$, A/A genotypes outnumber A/a genotypes. Most importantly, the fraction of A alleles found in A/a genotypes is greater than the fraction of A alleles in

A/A genotypes for all $p < 0.5$, while the fraction of a alleles in A/a genotypes is greater than the fraction of a alleles in a/a genotypes for all $p > 0.5$.

It follows that for *very* low values of p, we expect *all sampled* A alleles to be found in heterozygotes. Consider that if $p = 1/20{,}000$, then $p^2 = 2.5 \times 10^{-9}$; in a population of a billion individuals, you would only expect to find two or three A/A homozygotes. Conversely, for values of p approaching 1, the vast majority of a alleles are found in heterozygotes. The importance of this lies in the difference in fitness of heterozygotes specified by the dominant and additive selective regimes. Under a dominant selective regime, both homozygotes for the favored allele (A/A) and heterozygotes have the same, maximal relative fitness of 1.0. On the other hand, under an additive selective regime, in which h is nonzero, the heterozygotes possess a lower fitness than homozygotes for the favored allele (A/A).

As a result of these facts, we find that $w_{A/a}$ impacts differences in the slope of the adaptive allele's frequency between additive and dominant selective regimes during two critical phases of its increase: when p is very low or very high. For low values of p and an *additive selective regime* under which $w_{A/a} < w_{A/A}$, *the A allele struggles to rise in frequency* because there are likely no A/A genotypes in the population. Thus, noticeable increases in p begin much later under an additive selective regime regardless of selective strength s (Fig. 7.1b,c). Second, *fixation of the A allele* ($p = 1$) *takes much longer to accomplish under a dominant selective regime*. To consider the effect of s on these dynamics, again consider Fig. 7.1b,c. In both panels, results from a dominant ($h = 0$) and additive ($h = 0.5$) selective regimes are shown (black and gray lines, respectively). Both plots look identical, with the exception that the length of time shown on the x-axis is an order of magnitude different: 0–500 generations for $s = 0.01$ (Fig. 7.1b) and 0–5000 generations for $s = 0.1$ (Fig. 7.1c), i.e., s influences the quantitative more than the qualitative dynamics of the adaptive allele's ascent under a deterministic model.

Differences in the value of h, on the other hand, lead to noticeable distortions of dp/dt. Focusing on the case of $s = 0.01$ (Fig. 7.1b), the only difference in relative fitnesses is for the heterozygotes: $w_{A/a} = 1 - hs = 1 - (0)(0.01) = 1.$ for the dominant selective regime and $w_{A/a} = 1 - hs = 1 - (0.5)(0.01) = 0.995$ for the additive selective regime. Thus when p is very small, under a dominant selective regime the only genotype with a selective deficit is a/a. As a result, a alleles are more rapidly eliminated and the dominant selective regime is associated with an earlier onset in the rise of $p = f_A$. Conversely, when p is very large, a alleles are able to "hide out" in heterozygotes under a dominant selective regime, because heterozygotes do not suffer a fitness deficit. This results in the much longer time needed to purge the population of the deleterious a alleles associated with a dominant selective regime. On the other hand, because nearly all a alleles in the population are found in heterozygotes for large p, the fitness deficit of heterozygotes under an additive selective regime facilitates the rapid elimination of a alleles from the population.



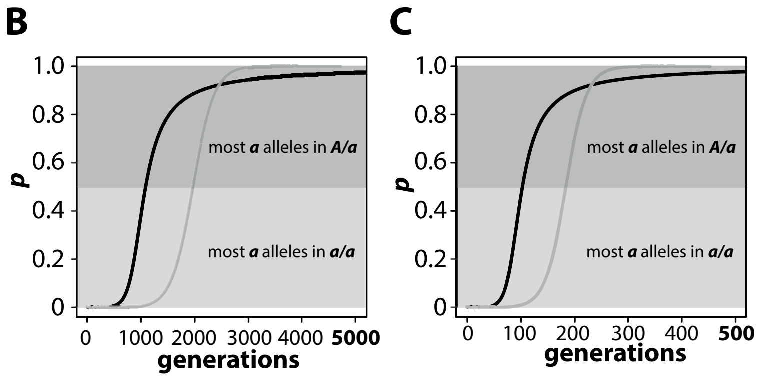


Fig. 7.1 Deterministic changes to the frequency of the favored allele A with $p = f_A$. (**a**) The relationship between p and genotype frequencies under the Hardy-Weinberg law. Two values of p are singled out. First, at $p \geq 0.333$, the frequency of A/a genotypes is greater than the frequency of a/a genotypes. Second, at $p \geq 0.667$, the frequency of A/A genotypes is greater than the frequency of A/a genotypes. (**b** and **c**) Deterministic trajectories of p for dominant (black lines, $h = 0$) and additive (gray lines, $h = 0.5$) for both $s = 0.01$ (**b**) and $s = 0.1$ (**c**). Note that although the dominant and additive curves are proportionally identical in (**b**) and (**c**), the timescale is an order of magnitude greater in (**b**) than (**c**)

In the next section, we use simulation to address the complication of finite population size. In other words, we take into account the fact that random fluctuations in allele frequencies still occur despite the directional pressure on allele frequencies provided by natural selection.

7.2 Stochastic Simulation of the Selected Variant Only

In Chap. 8, we will consider the effect of natural selection on linked variation, requiring us to simulate full sequences bracketing the selected variant. For now, we limit our attention to the selected variant itself. To this end, we cover a simple simulation program that includes reproduction in addition to user-input values of selection parameters. Inclusion of reproduction mimics a more realistic situation in which natural selection and sampling error due to genetic drift jointly influence evolution of the selected variant. In short, we now simulate a stochastic process.

7.2.1 Frequency-Independent Selection

Program `stochastic_selection.cc` is a standalone program that requires the user to input values of N, a string for the output file suffix, s, and h on the command-line in that order. It is assumed that an *adaptive* allele has just arisen by mutation and therefore starts at a frequency of $1/2N$. Due to the randomness provided by reproduction, it is very common for the favored, derived allele to be lost from the population early on. To account for this, the boolean varible `trigger` is used to control the loop that runs from lines 34–79. In this way, the simulation starts over each time the favored allele is lost and the program will not complete until a run of the simulation results in fixation of the favored allele. If *the nonindented*, commented-out lines are activated, the program will print the current frequency of the favored allele to screen as well as the number of tries (variable `numtries`; lines 32, 35, and 81) required for the favored allele to successfully achieve fixation. Output from this program is the frequency of the favored allele and $\bar{w}$ for each generation.

`stochastic_selection.cc`

```
1 #include <random>
2 #include <fstream>
3 #include <string>
4 #include <map>
5 #include <vector>
6 #include <iostream>
7
```

```
using namespace std;

int main(int argc, char *argv[]) {

  // command line arguments
  double N = atof(argv[1]);
  string suffix = argv[2];
  double s = atof(argv[3]);
  double h = atof(argv[4]);

  map<int, int> pop;
  vector<double> fitness;
  fitness.push_back(1.-s);
  fitness.push_back(1.-(h*s));
  fitness.push_back(1.);

  default_random_engine engine(time(0)); //initialize the random engine
  map<int, vector<double>> data;
  uniform_int_distribution<int> randind(0,N-1);
  uniform_real_distribution<double> randd(0.,1.);

  bool trigger = true;
  int gen;
// int numtries = 0;

  while(trigger) {
// numtries++;
    // populate the population with N-1 homozygotes for ancestral allele
    // ... and 1 heterozygote for the derived, beneficial allele
    for (int i=0; i<N-1; i++)
      pop[i] = 0;
    pop[N-1] = 1;
    gen = 0;
    double p = 1/(2*N);

    while (p != 0. && p != 1.) {
      gen++;
      vector<int> survivor_indices;
      map<int, int> nextpop;
      for (int i=0; i<N; i++) {
        if (randd(engine) <= fitness[pop[i]])
          survivor_indices.push_back(i);
      }
      uniform_int_distribution<int> randind(0,survivor_indices.size());

      for (int i=0; i<N; i++) { // constant population size
        double offspring = 0.;
        for (int j=0; j<2; j++) {
          switch (pop[survivor_indices[ randind(engine) ] ] ) {
            case 0: break;
            case 1: if (randd(engine) <= 0.5) { offspring+=1.; }
                break;
            case 2: offspring+=1.;
          }
```

```
            }
            nextpop[i] = offspring;
         }
         pop = nextpop;
         p = 0.;
         for (int i = 0; i<N; i++) p += pop[i];
         p /= (2*N);
         data[gen].push_back(p);
         double q = 1. - p;
         double popfit = p*(p*fitness[2]+q*fitness[1]) +
             ↪ q*(p*fitness[1]+q*fitness[0]); // mean population fitness
         data[gen].push_back(popfit);
// cout << p << endl;
      }
      if (p == 1.) trigger = false;
      else data.clear();
// cout << "*****************************" << endl;
   }

// cout << "Number of tries: " << numtries << endl;

   // output data held in the map, data
   string fname = "stochastic_selection_resultsN" + suffix;
   ofstream output;
   output.open(fname.c_str());
   output << "gen\tfrequency\tpopfit" << endl;
   for (int i=1; i<data.size(); i++)
      output << i << "\t" << data[i][0] << "\t" << data[i][1] << endl;
   output.close();

   return 0;
}
```

Representative results from this simulation are shown in Fig. 7.2. These simple results from a simple program inform our growing intuition regarding positive selection targeting a new allele just arisen from mutation. First, when selective pressure is low, the random influence of genetic drift on allele frequencies is quite evident. When $s = 0.01$ (Fig. 7.2a-c), the trajectory of the increase in $p = f_A$ is jagged. Conversely, when $s = 0.1$ (Fig. 7.2e), p increases smoothly under both selective regimes, as selection is sufficiently strong to dampen the random fluctuations due to genetic drift. In other words, strong selection results in a *quasi*-deterministic increase in p provided the favored allele survives its tenuous existence when it first emerges via mutation.

Second, under a dominant selective regime, mean population fitness ($\bar{w}$) climbs at a rapid rate despite the protracted time it takes to purge the population of all less-fit alleles (Fig. 7.2d,f). A close comparison of the p to $\bar{w}$ under a dominant selective regime (cf. black lines in Fig. 7.2c vs. d and Fig. 7.2e vs. f) shows that once $p \sim 0.75$, $\bar{w}$ is very nearly 1. Thus, the lengthy time required for fixation of the A allele under the dominant selective regime may be of little real consequence. This result is easily explained by the fact

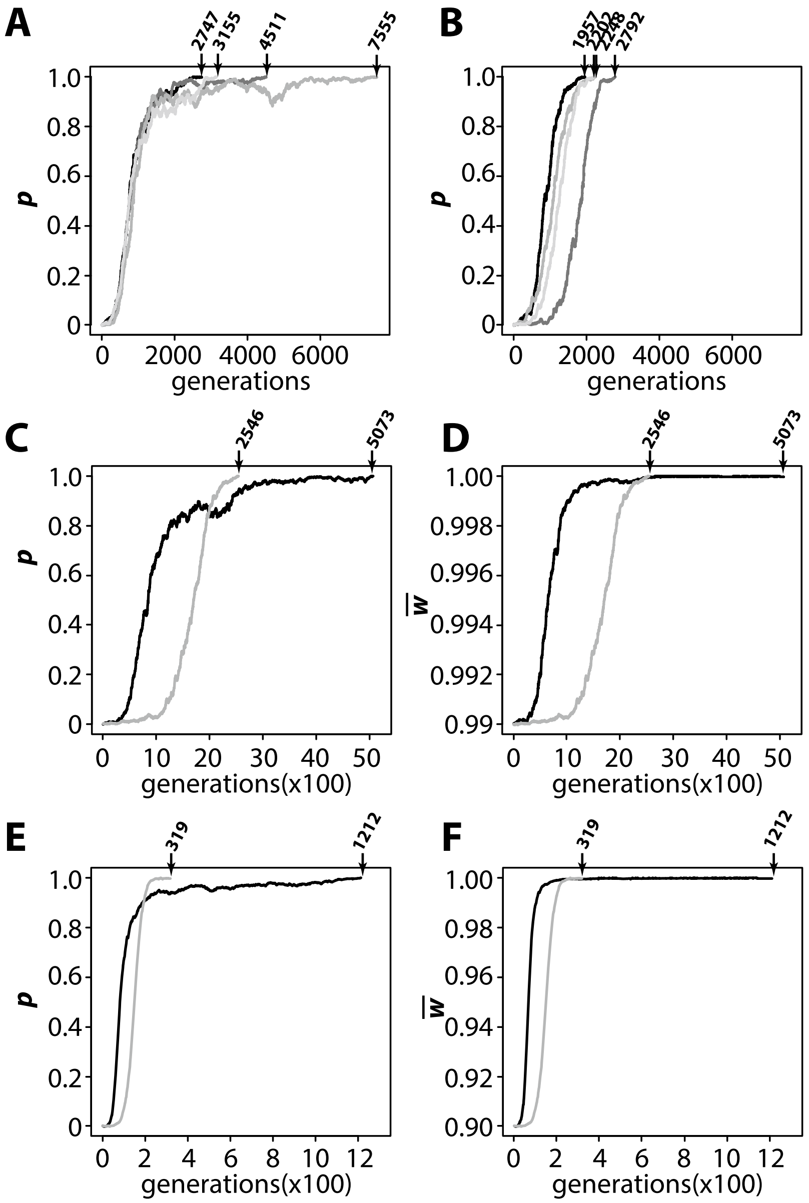


Fig. 7.2 Stochastic selection, which accounts for both the force of positive selection and finite population size. (**a**,**b**) Four replicate simulations each where $s = 0.01$ for both dominant [(**a**), $h = 0$] and additive [(**b**), $h = 0.5$] selective regimes. (**c**) $s = 0.01$ and $h = 0$ (black curve) or $h = 0.5$ (gray curve). (**d**) Mean population fitness $\bar{w}$ for the same simulations as in (**c**). (**e**) $s = 0.1$ and $h = 0$ (black curve) or $h = 0.5$ (gray curve). (**f**) $\bar{w}$ for the same simulations as in (**e**)

that once p is substantial, there are relatively few a/a homozygotes, which is the only genotype with relative fitness < 1. In other words, although a fairly large number of a alleles may still be segregating in the population, most of them are found in heterozygotes and do not debit mean population fitness. Figure 7.2d,f shows that, conversely, the slower up-tick in p under an *additive* selective regime ($h = 0.5$) has the consequence of the population spending a longer period with low $\bar{w}$ relative to a dominant selective regime.

7.2.2 Negative Frequency-Dependent Selection

In the previous Sect. 7.2.1, we assumed relative fitness is independent of current allele frequencies. However, this is not always true, in which case we refer to natural selection as frequency-dependent. We specifically focus on *negative* frequency-dependent selection because it can lead to the important and interesting situation in which polymorphism is maintained despite selection. Under negative frequency-dependence, the relative fitness of a homozygous genotype is greater when its component allele is rare. For example, $w_{A/A}$ is greatest when $p = f_A$ is small and $w_{a/a}$ is greatest when $q = f_a$ is small.

Before introducing a model of negative frequency-dependent selection, we consider the expected dynamics of allele frequencies in response to this type of selection. The R script that follows—`analyze_selective_regimes.r`—calculates the one-dimensional vector field and $\bar{w}$ for a given set of relative fitness functions that may be dependent or independent of p. The vector field visualizes $\dot{p} = \frac{dp}{dt}$, which is the rate of change in p on the domain [0,1]. A variety of one-locus, two-allele selective regimes can be analyzed using these functions. The utility of these functions is first shown by applying them to two frequency-*independent* selective regimes.

`analyze_selective_regime.r`

```
analyze_selective_regime <- function(s, h, relfitAA, relfitAa, relfitaa)
{
  p <- seq(0,1,0.001)
  q <- 1-p
  relfit <- list()

  relfit[[1]] <- eval(substitute(relfitAA))
  relfit[[2]] <- eval(substitute(relfitAa))
  relfit[[3]] <- eval(substitute(relfitaa))

  wA <- p*relfit[[1]] + q*relfit[[2]]
  wa <- p*relfit[[2]] + q*relfit[[3]]
  wbar <- p*wA + q*wa

  pt <- p * wA / wbar
```

```
   change <- pt-p
   vf <- data.frame("p" = p, "pdot" = change)
   popfit <- data.frame("p" = p, "wbar" = wbar)
   q <- list()
   q[[1]] <- vf
   q[[2]] <- popfit
   return(q)
}

plot_vector_field <- function(q)
{
   plot(q[[1]]$p, q[[1]]$pdot, type = "l", xlab = "p", ylab = "dp/dt")
   abline(h = 0., lty = 2)
}

plot_wbar <- function(q)
{
   plot(q[[2]]$p, q[[2]]$wbar, type = "l", xlab = "p", ylab = "wbar")
   m <- q[[2]][q[[2]]$wbar == max(q[[2]]$wbar),][1]
   print(m)
   abline(v = m, lty = 2)
}
```

Arguments passed to function `analyze_selective_regime()` include *s*, *h*, and the relative fitness of each genotype—`relfitAA`, `relfitAa`, and `relfitAa`; the latter are entered as expressions in terms of *s*, *h* (assuming two parameters are required to express relative fitness), as well as *p* and *q* in cases of frequency-*dependent* selection. We begin by calculating the vector field and mean population fitness for dominant and additive selective regimes of frequency-independent selection:

```
> source("analyze_selective_regime.r")
> dom <- analyze_selective_regime(0.05, 0, 1, 1-h*s, 1-s)
> add <- analyze_selective_regime(0.05, 0.5, 1, 1-h*s, 1-s)
```

Note the expressions used to represent relative fitness of each genotype. Next, we plot (1) the vector field and (2) $\bar{w}$ versus *p* for the two frequency-independent selective regimes:

```
> plot_vector_field(dom)
> plot_wbar(dom)
> plot_vector_field(add)
> plot_wbar(add)
```

The resulting plots are shown in Fig. 7.3. Although plots of *p* versus time (e.g., Figs. 7.1b,c and 7.2) showed that a dominant selective regime is characterized by an early increase *p* and then a protracted climb toward fixation, vector fields provide a more explicit, high-resolution depiction of these facts. Examining the vector fields for the dominant (Fig. 7.3a) and additive (Fig. 7.3b) selective regimes, we first notice that $\dot{p}$ is greater than zero for all values of

p other than zero and one, which are absorption points in the absence of mutation. Thus, if an adaptive allele is not lost in the early phases of positive natural selection, this consistent drive toward fixation leads to fixation with near-certainty. Second, notice that $\dot{p}$ varies depending on the value of p. Remember this we are currently considering frequency-independent selection, by which we mean relative fitness is not a function of allele frequencies. Yet, dp/dt does change with the value of p; the slope shows a definite peak at different values of p for the two selective regimes (Fig. 7.3a,b). Finally, as compared to the additive model, dp/dt shows a rapid decline for $p > 0.5$ under the dominant selective regime (Fig. 7.3a).

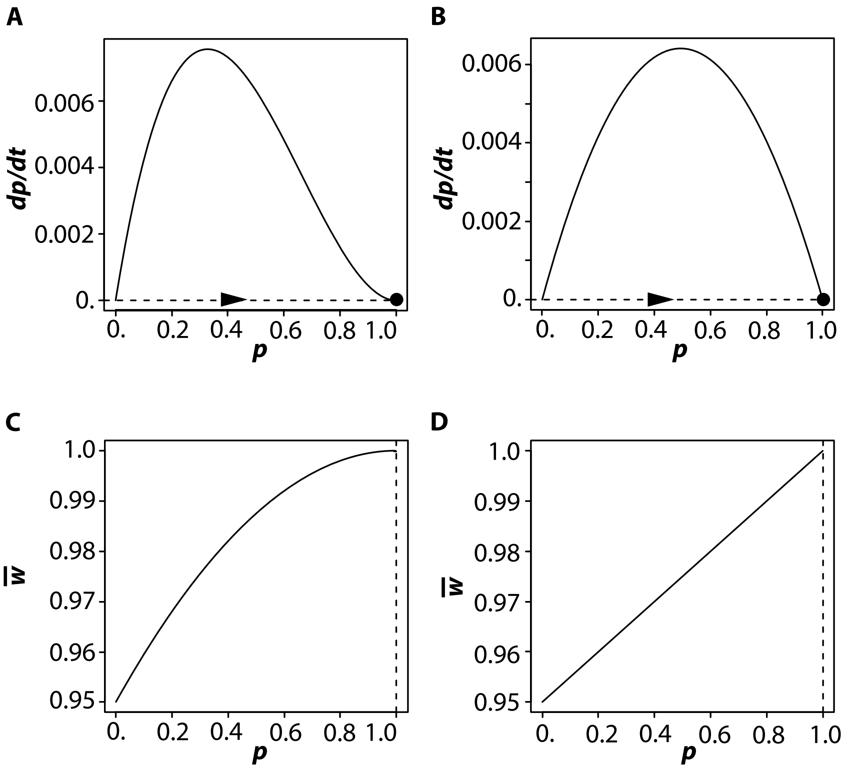


Fig. 7.3 One dimensional vector fields (**a**-**b**) and mean population fitness (**c**-**d**) for $p \in [0,1]$ under frequency-*independent* natural selection. (**a** and **c**) Dominant selective regime $s = 0.05$ and $h = 0$. (**b** and **d**) Additive selective regime $s = 0.05$ and $h = 0.5$. Arrowhead indicates direction of p over time implied by dp/dt. Filled circle indicates a fixed point. For example, in (**a**), the vector field indicates that for any starting value of p, p will increase until it fixes at a value of 1. In the absence of mutation and drift, fixation of p by positive natural selection is inevitable. Vertical dashed lines in (**c**,**d**) indicate the value of p that maximizes $\bar{w}$. **Note that the R function `plot_vector_field()` does not plot the arrowheads or fixed point symbols. These were added when designing the figure**

We could find the curves shown in Fig. 7.1b,c without ever running a simulation. (A) How would you map the information in Fig. 7.3a to produce the *deterministic* p versus t plots of Fig. 7.1b,c. (B) What effect, if any, does the value of s have on the one-dimensional vector field of additive and dominant selective regimes?

When we discussed Fig. 7.2b,f, we noted that $\bar{w}$ was generally greater for the dominant selective regime than the additive selective regime. Comparing Fig. 7.3c to d, this observation is corroborated. For all values of p, the dominant selective regime possesses greater $\bar{w}$, which is most pronounced for intermediate values of p. $\bar{w}$ is maximized when $p = 1$, fully replacing the less-fit allele.

We next perform graphical analysis of two scenarios of negative frequency-*dependent* (NFD) natural selection: (1) a dominant case in which $w_{A/A} = wA/a$ and the relationship between p and relative fitness is linear (Fig. 8.4a); (2) an additive case where the relationship between p and relative fitness is quadratic (Fig. 8.4b). To contextualize these selective regimes, consider a toy example in which a single selected locus determines the outward appearance of a prey species. As predators hunt these prey species, they learn a search image that corresponds to the most frequent outward appearance of the prey species. Thus, individuals of the prey species bearing a genotype that maps to a rare outward appearance stand a better chance of surviving. Over time, the situation reverses as the formerly rare phenotype becomes common and the relative fitnesses of the different causative genotypes flip rank.

Under the additive selective regime, the heterozygote genotype maps to a third outward appearance (phenotype), somewhat complicating the dynamics of the two allele frequencies in the population (cf. Fig. 7.4a and b).

As an example of how to use `analyze_selective_regime()` in this context, we encode a dominant NFD selective regime that is linear in form and defined in terms of just s and p:

genotype	**relative fitness**
A/A	$w_{A/A} = 1.0 - sp$
A/a	$w_{A/a} = 1.0 - sp$
a/a	$w_{a/a} = 1.0 - s + sp$

As our second example, we encode an *additive* NFD selective regime that is quadratic in form and uses two selective parameters (s and h) as well as the frequency of the A allele $p = 1 - q$:

genotype	**relative fitness**
A/A	$w_{A/A} = 1.0 - sp^2$
A/a	$w_{A/a} = 1.0 - 2shpq$
a/a	$w_{a/a} = 1.0 - sq^2$

First, note that the relative fitness of each genotype incorporates its expected frequency under the Hardy-Weinberg law. Second, note that h is again used as a dominance coefficient that affects the magnitude of fitness deficit accorded to heterozygotes. You—and Nature!—could of course specify many other NFD selective regimes. You alone can use `analyze_selective_regimes.r` to graphically analyze your creations.

We do so next for our two example scenarios defined in the tables above. We assume $s = 0.05$ and, for the additive NFD regime, $h = 2$:

```
> nfd_dom <- analyze_selective_regime(0.05, 0, 1-s*p, 1-s*p, 1-s+s*p)
> plot_vector_field(nfd_dom)
> plot_wbar(nfd_dom)
> nfd_add <- analyze_selective_regime(0.05, 2., 1-s*p^2, 1-2*s*h*p*q,
    ↪ 1-s*q^2)
> plot_vector_field(nfd_add)
> plot_wbar(nfd_add)
```

The vector fields and plots of $\bar{w}$ versus p for both NFD selective regimes are shown in Fig. 7.4c–f. In the case of dominant NFD, a single stable point is found at $p = 0.5$ where $dp/dt = 0$ (Fig. 7.4c). When A is the minor allele ($p < 0.5$), positive values of dp/dt drive p upwards toward the stable point. Conversely, when a is the minor allele, negative values of dp/dt drive p downwards toward the stable, fixed point. In the absence of mutation, $p = 0.$ and $p = 1.$ are absorption points. They would not be stable points were mutation included.

Things are somewhat more complicated (and interesting?) under the additive NFD selective regime (Fig. 7.4d). Excluding the absorption points at 0 and 1, $dp/dt = 0$ for three values of p: 0.276, 0.5, and 0.724. The first and last of these points are stable because they are bracketed by dp/dt of opposing direction, while the intermediate point is unstable as any perturbation will lead p away from this point. Importantly, under both NFD selective regimes, the stable points are at values of p that are neither zero nor one. In other words, as long as p is neither equal to zero or one, NFD is one means of preserving polymorphism at a locus.

Next, we compare the stable points of each selective regime to its point of maximal $\bar{w}$. Under the dominant NFD selective regime, $max(\bar{w}) = 0.392$ (Fig. 7.4e). However, the stable point, which is an attractor of p is at 0.5 (Fig. 7.4c). We will see in subsequent simulations that $p = 0.5$ is indeed an attractor. An important conclusion drawn from this example is that natural selection may **not** always maximize fitness even when environmental conditions are stable and the values of s and h do not change.

In the additive NFD case, the two stable points at $p = 0.276$ and $p = 0.724$ do correspond to the *two* values of $max(\bar{w})$ (cf. Fig. 7.4d and f). Although population fitness is expected to be maximized by natural selection in this case, the actual value of p found in a population maximizing $\bar{w}$ is uncertain as it depends on what the value of p was at the onset of natural selection (Fig. 7.4d).

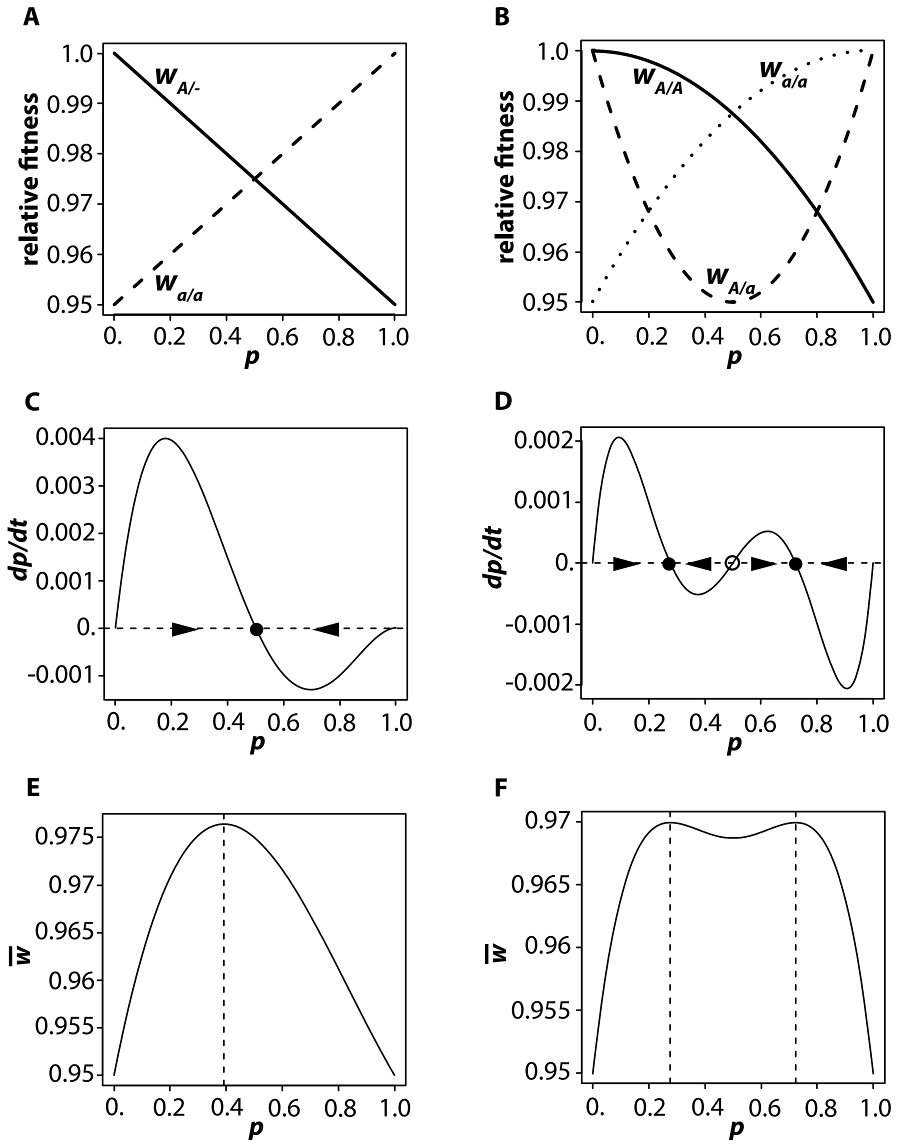


Fig. 7.4 Two selective regimes of negative frequency-dependence (NFD). The left-hand column (**a**,**c**,**e**) corresponds to a dominant form of NFD in which heterozygotes bear the same relative fitness as homozygotes for the allele A, whose frequency is denoted p. The right-hand column (**b**,**d**,**f**) corresponds to an additive form of NFD in which all three genotypes bear different relative fitness. In the case of the dominant NFD, maximal $\bar{w}$ (**e**) does not coincide with the stable point at $p = 0.5$ (**c**). The points of maximal $\bar{w}$ for the additive NFD (**f**) do coincide with the two stable points (**d**)

The next step is to write a forward simulation program that simulates evolution at a single locus under an NFD selective regime. As you might expect, the simulation is very similar to that for frequency-independent selection. I do not reproduce the program here, but it is available online among Chap. 7 materials (`stochastic_selection_negfreq.cc`). Command-line inputs to this program include N_e, s, h (the last of which is meaningless in the case of the dominant NFD selective regime that only needs the one selection parameter s), and the number of generations to simulate. This last parameter is necessary because unlike frequency-independent selection, we expect polymorphism to persist. There is no fixation of p and therefore no natural point at which to terminate the program. By default, the program begins with $p = 1/2N_e$; as before, the program starts over if p is absorbed at $p = 0$.

For both NFD selective regimes detailed above, I ran the program for 20,000 generations for both $N_e = 10{,}000$ and $N_e = 1000$. In the dominant case (Fig. 7.5a,b), we observe maintenance of polymorphism at the single attractor at $p = 0.5$, which does not coincide with the frequency of the A allele that would maximize $\bar{w}$. In addition, as population size decreases the variance on p around the attractor increases.

In the additive case (Fig. 7.5c,d), decreasing population size produces a more interesting effect. When $N_e = 10{,}000$, because we begin the simulation with the lowest possible nonzero value of $p = 1/2N_e$, p approaches and remains on the lower of the two attractors at $p = 0.276$ (Fig. 7.5c). However, when we decrease N_e to 1000, the population oscillates between periods in which p is focused on the smaller attractor and periods focused on the greater attractor at $p = 0.724$ (Fig. 7.5d). This can be explained by the fact that sampling error associated with small population size somewhat frequently perturbs p sufficiently for it to be attracted to another stable point. Again, note that unlike the dominant NFD selective regime, both stable points correspond to the two values of p that maximize $\bar{w}$ (Figs. 7.4f and 7.5d).

7.2.3 *Overdominance*

Overdominance (heterozygote advantage) refers to a selective regime in which the heterozygote bears greater relative fitness than either of the two homozygotes. Here is a two-parameter model of overdominance:

genotype	**relative fitness**
A/A	$w_{A/A} = 1.0 - s$
A/a	$w_{A/a} = 1.0$
a/a	$w_{a/a} = 1.0 - t$

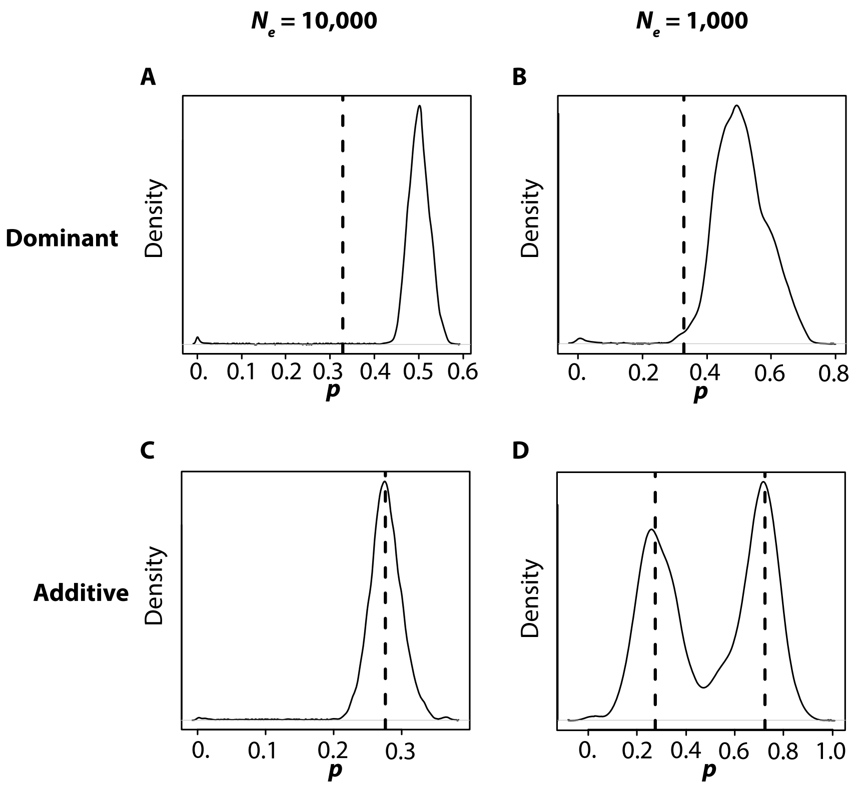


Fig. 7.5 Kernel density estimates of p over 20,000 generations of forward, stochastic simulations of the selective regimes summarized in Fig. 7.4a and b. In all simulations, p began at a frequency of $1/2N_e$ and s was set to 0.05. **Top row**: Results from simulation of the dominant NFD selective regime summarized in Fig. 7.4a. **Bottom row**: Results from simulation of the additive NFD selective regime summarized in Fig. 7.4b. $\mathbf{N_e}$: 10,000 in (**a**, **c**); 1000 in (**b**,**d**). Vertical dashed lines indicate value(s) of p at which $\bar{w}$ is maximized for the selective regime

s and t may be equal or not. We now examine an overdominant selective regime in which t is twice the value of s. This corresponds to a hierarchy of relative fitnesses: $w_{A/a} > w_{A/A} > w_{a/a}$.

Heterozygote advantage, almost by definition, implies maintenance of polymorphism, and equal values of s and t should lead to maintenance of both alleles near frequencies of 0.5. However, the greater relative fitness of A/A homozygotes than a/a homozygotes that we explore here suggests the frequency of the A allele will be attracted to a value greater than $p = 0.5$. For two different sets of s and t values that satisfy $s = 2t$, vector fields do in fact correspond to a stable point at $p = 0.667$ (Fig. 7.6a). Vector fields and plots of $\bar{w}$ versus p suggest two additional conclusions: (1) different sets of $\{s,t\}$, that share identical proportions of s and t, share the same stable point of p,

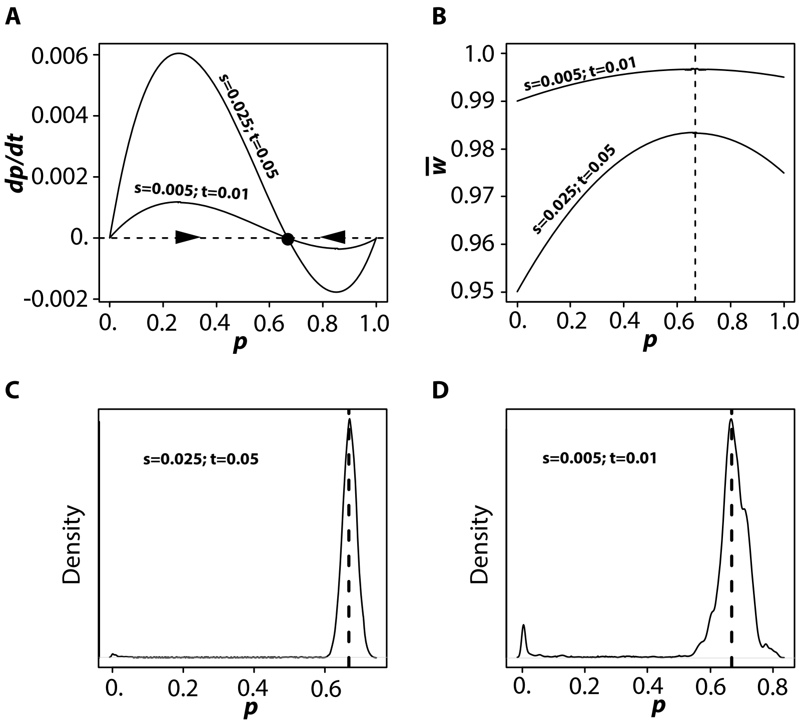


Fig. 7.6 Evolution of overdominant selective regimes. (**a**) Vector fields for two overdominant selective regimes. In both cases, the $t = 2s$, meaning reduction in relative fitness of the A/A homozygote is half that of the a/a homozygote, where $w_{A/a} = 1$. However, the strength of selection matters. Higher values of s and t lead to more rapid convergence on the stable attractor at $p = 0.667$ and maintain a tighter spread around the stable attractor as evidenced in (**c**). Again, $p = 0$ and $p = 1$ are absorption points in the absence of mutation. (**b**) $\bar{w}$ for the two selective regimes. (**c** and **d**) Kernel density estimates of p across 20,000 generations of simulation of the two selective regimes. In both simulations, $N_e = 10{,}000$ and p began at a frequency of 10^{-4} (one allele only). The slower approach of p to the attractor at 0.667 under the weaker selective regime explains the noticeable peak in density for small values of p; during this stage of the simulation, p was only weakly drawn toward the attractor and therefore oscillated at lower values of p for some time

although lower values of s dampen the rate of approach (Fig. 7.6a), and (2) the stable point coincides with maximal $\bar{w}$ (Fig. 7.6b).

Although the overdominant selective regime defined in the previous paragraph is not frequency-dependent, we use the program of the previous subsection (`stochastic_selection_negfreq.cc`) to simulate overdominance because fixation is not expected for either allele. Forward simulation of these selective regimes confirms $p = 0.667$ as an attractor (Fig. 7.6c,d) and that weaker selection leads to greater variance over time around this stable point (Fig. 7.6d).

7.3 Selection at Two Linked Sites

We now turn our attention to the case of natural selection targeting two linked loci. The main complicating factor in this analysis is that fitness of an organism is dependent upon the combinatorial state of two genotypes derived from two haplotypes.

7.3.1 *Simulation of Two-Locus Selection with Recombination*

We consider a commonly investigated situation of two-locus selection: balancing selection in which polymorphism is maintained at both loci. The following relative fitness matrix defines this form of selection. Two assumptions are inherent: (1) there are just two alleles at each locus, and (2) both forms of double heterozygotes bear equal relative fitness, i.e., $w_{AB/ab} = w_{Ab/aB}$. The latter assumption is reasonable as long as we do not have reason to suspect that one locus has a cis-regulatory effect on the other locus (Rice 2004).

$$\begin{array}{cccc}
 & \text{A/A} & \text{A/a} & \text{a/a} \\
\hline
\text{B/B} & 1-s_A-s_B-e_{AB} & 1-s_B & 1-t_A-s_B-e_{aB} \\
\text{B/b} & 1-s_A & 1 & 1-t_A \\
\text{b/b} & 1-s_a-t_B-e_{Ab} & 1-t_B & 1-t_A-t_B-e_{ab}
\end{array}$$

Before turning to the definitions of the terms in this matrix, note the column and row headings of the matrix: relative fitnesses are defined at the intersection of the genotypes at both loci. It makes sense to define the bi-locus fitnesses in terms of genotypic combinations because absent cis-effects, the genotypes at each locus , are what determine phenotype and therefore fitness.

Our calculations will require us to obtain the marginal fitnesses of *haplotypes*. Ignoring recombination for the moment, this reduces the problem to selection at a *single* site with four variants—haplotypes AB, Ab, aB, and ab. This sleight of hand means we can apply many of the same analytical steps as outlined earlier in the chapter to two-locus selection. However, we cannot realistically ignore recombination. Because the two simulated loci are linked, we must account for the fact that crossing-over among doubly heterozygous *haplotypes* generates recombinant haplotypes and, thereby, affects linkage disequilibrium as measured by D.

We now turn to the meaning of the new parameters included in the relative fitness matrix above. Because we assume balancing selection, each of the two loci *individually* is overdominant. Thus for each locus, any individual with a homozygous genotype A/A or B/B is reduced by the quantity s_A and/or s_B, respectively. Similarly, any individual with a homozygous genotype a/a or b/b is reduced by the quantity t_A and/or t_B, respectively. In addition to recombination, consideration of *two* loci also requires us to account for potential gene-gene (epistatic) interactions among the individuals who are homozygous at both loci. For example, AB/AB double homozygotes may suffer a fitness disadvantage due to a deleterious interaction among A and B alleles, which is quantified by the epistatic coefficient e_{AB}. The relative fitnesses of the three other *double* homozygotes are quantified by e_{aB}, e_{Ab}, and e_{ab}. Fitness of the double heterozygote is set to 1.

For brevity in the following discussion, from here on we refer to haplotypes AB, A-b, a-B, and a-b as haplotypes 1, 2, 3, and 4, respectively. Furthermore, we denote their frequencies as h_i and marginal fitnesses as w_i. Then, the marginal fitness of each haplotype is calculated as:

$$w_i = \sum_{j=1}^{4} h_j w_{ij}, \tag{7.9}$$

where w_{ij} is the relative fitness of the combination of haplotypes i and j as defined in the fitness matrix above. For example, an individual with haplotypes 1 and 2 (i.e., a joint genotype of AB/Ab) corresponds to the cell that intersects genotypes A/A and B/b, or $w_{12} = 1 - s_A$. The summation weights the relative fitness of each possible combination of haplotype i and a second haplotype by the frequency of the second haplotype, h_j. See Eqs. 7.1 and 7.2 of this chapter, obtained by the same logic in a one-locus case; the distinction is that here we are treating each *haplotype* as an allele.

Figure 7.7 shows which of the four cells in the relative fitness matrix are relevant to the calculation of the each haplotype i's *marginal fitness*. Here is an explicit calculation of the marginal fitness of haplotype 1 (AB), which makes use of the relative fitnesses associated with the submatrix highlighted in gray in Fig. 7.7:

$$w_1 = (h_1 \times w_{AB/AB}) + (h_2 \times w_{AB/Ab}) + (h_3 \times w_{AB/aB}) + (h_4 \times w_{AB/ab}),$$

where each w_{ij} is the relative fitness of cells [1,1], [2,1], [1,2], and [2,2] in the fitness matrix, respectively.

Mean population fitness is then calculated as the sum of each haplotype's relative fitness weighted by its frequency:

$$\bar{w} = \sum_{i=1}^{4} h_i w_i \tag{7.10}$$

Finally, the frequency of a given haplotype in the next generation may be calculated as:

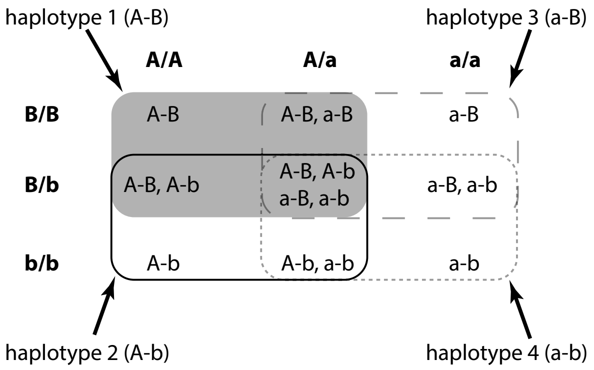


Fig. 7.7 Submatrices of the relative fitness matrix for combinations of genotypes at two loci relevant to the calculation of the marginal fitnesses of haplotypes 1, 2, 3, and 4. The four cells highlighted for each haplotype are the cells that contain the haplotype in question. For example, the four cells highlighted in gray all contain the haplotype A-B. The set of haplotypes shown in each cell include all haplotypes found in at least one haplotype pair that yields the single-locus genotypes corresponding to the cell's row and column. For example, cell [2,1] lists haplotypes A-B and A-b because only this pair of haplotypes yields an A/a *and* B/b genotype

$$\begin{aligned}
h_1' &= \frac{1}{\bar{w}}(h_1 w_1 - rDw_{1,4}) \\
h_2' &= \frac{1}{\bar{w}}(h_2 w_2 + rDw_{1,4}) \\
h_3' &= \frac{1}{\bar{w}}(h_3 w_3 + rDw_{1,4}) \\
h_4' &= \frac{1}{\bar{w}}(h_4 w_4 - rDw_{1,4}),
\end{aligned} \tag{7.11}$$

where $w_{1,4}$ is the relative fitness of the double heterozygote (dihybrid) in cis: AB/ab. Looking at Eq. 7.11, we find that the next generation's frequency of each haplotype is a function of both (1) fitness effects ($h_i w_i$) and (2) the generation or loss of haplotype i due to crossing-over ($\pm rDw_{1,4}$). Again, the derivation of these equations assumes that $w_{1,4} = w_{2,3}$, where 2,3 is the dihybrid in trans: Ab/aB. Because both types of dihybrids are assumed equally fit, we can use one or the other in Eq. 7.11. Note, however, the signs of subtraction and addition of $rDw_{1,4}$ are specific to the $w_{1,4}$ case; recombination between loci A and B in a 1,4 dihybrid reduces the number of AB and ab haplotypes (haplotypes 1 and 4, respectively; Fig. 7.7) and increases the frequency of the aB and Ab haplotypes. In other words, we *subtract* $rDw_{1,4}$ in the h_1' and h_4' calculations because crossing-over between haplotypes 1 and 4 will reduce the number of 1 and 4 haplotypes in the population. Conversely, such recombination will yield *new* 2 and 3 haplotypes; thus the sign of addition. If we used $w_{2,3}$ as the reference combination, signs would be reversed.

7.3.1.1 Deterministic Simulation

The R function `deterministic_twolocus_selection()` requires a large number of parameters, including selection coefficients, epistatic parameters, the starting frequencies of the four haplotypes, and the number of generations to run the simulation.

deterministic_twolocus_selection.r

```r
deterministic_twolocus_selection <- function(r, sA, sB, tA, tB, eAB, eAb,
    ↪ eaB, eab, h1, h2, h3, h4, gens)
{
  fit <- matrix(nrow=3, ncol=3, c(1-sA-sB-eAB, 1-sA, 1-sA-tB-eaB, # col 1
                                  1-sB, 1, 1-tB,            # col 2
                                  1-tA-sB-eAb, 1-tA, 1-tA-tB-eab)) # col 3
  print(fit);

  # initial haplotype frequencies
  h <- c(h1, h2, h3, h4) # f_A-B, f_A-b, f_a-B, f_a-b

  w <- vector(); #vector for marginal fitnesses
  anchor <- list(a1 = c(1,1), a2 = c(2,1), a3 = c(1,2), a4 = c(2,2)) #
      ↪ upper-left cell of each submatrix
  d <- setNames(data.frame(matrix(ncol = 9)), c("gen", "h1", "h2", "h3",
      ↪ "h4", "pA", "pB", "wbar", "D")) # initialize data matrix

  for (gen in 1:gens) {
    # calc marginal fitness of each haplotype
    for (i in 1:4) {
      subfit <- fit[ anchor[[i]][1]:(anchor[[i]][1]+1),
          ↪ anchor[[i]][2]:(anchor[[i]][2]+1)]
      subfit <- c(subfit[,1], subfit[,2])
      w[i] <- sum(h * subfit)
    }

    # calc mean population fitness and D
    wbar <- sum(h*w)
    D <- h[1] - (h[1]+h[2]) * (h[1]+h[3])

    # calc new frequencies of haplotypes
    for (i in 1:4) {
      if (i ==2 | i == 3) {
        h[i] <- ( (h[i]*w[i]) + (r*D*fit[2,2]) ) / wbar
      } else {
        h[i] <- ( (h[i]*w[i]) - (r*D*fit[2,2]) ) / wbar
      }
    }
    d[gen,] = c(gen, h[1], h[2], h[3], h[4], h[1]+h[2], h[1]+h[3], wbar, D)
  }
  return(list("data" = d, "fitness_matrix" = fit))
}
```

The parameterized fitness matrix for each combination of genotypes is defined on lines 3–5 and returned as the second element of a list (line 38). Remember that in R, matrices are defined column by column, which is distinct from our implementation of matrices in C++, where matrices are defined row by row. Vector `h` (line 9) holds the current frequencies of the four haplotypes. It is of course critical that the values of these frequencies entered as arguments to the function sum to unity. Vector `w` (line 11) holds the marginal fitness of each haplotype and the list `anchor` defines the upper-left cell of each submatrix as shown in Fig. 7.7. The `for` loop on lines 15–36 is run each generation, beginning with the calculation of the four marginal fitnesses according to Eq. 7.9. Initially `subfit` holds the 2x2 submatrix corresponding to one of the four haplotypes (line 18). We then combine the two columns of this submatrix into one vector and multiply by each of the four appropriate haplotype frequencies (lines 19–20). Calculation of mean population fitness (Eq. 7.10, line 24) and the measure of linkage disequilibrium D (line 25) are straightforward. Finally, the next generation haplotype frequencies are calculated according to Eq. 7.11 on lines 28–34; again, the reference combination of haplotypes (1 and 4) used here and quantified by `fit[2,2]` is arbitrary. The function returns a list of (1) the time series data and (2) the relative fitness matrix (line 37).

We first run `deterministic_twolocus_selection()` for two different parameter sets:

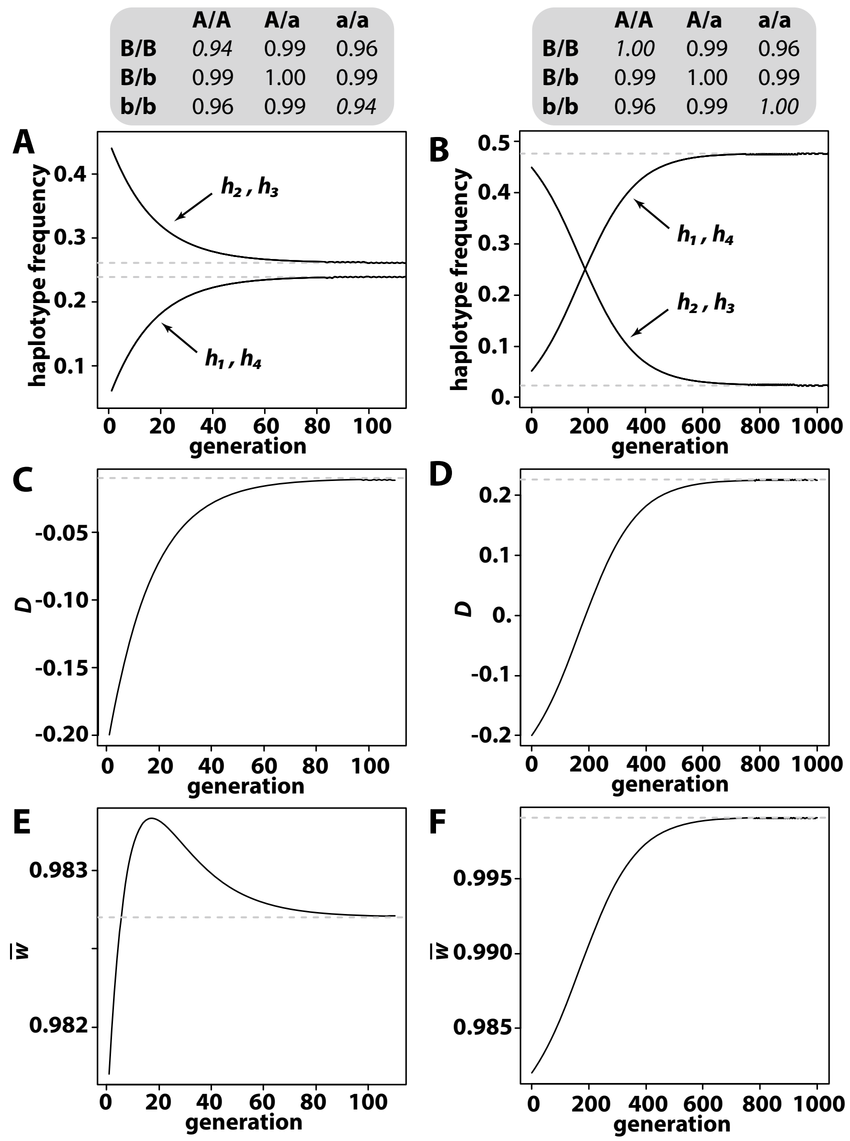
Two-locus parameter set 1
- $r = 0.05$
- $s_A = s_B = t_A = t_B = 0.01$
- $e_{AB} = e_{ab} = 0.04$
- $e_{Ab} = e_{aB} = 0.02$

Two-locus parameter set 2
- $r = 0.001$
- $s_A = s_B = t_A = t_B = 0.01$
- $e_{AB} = e_{ab} = -0.02$
- $e_{Ab} = e_{aB} = 0.02$

In both cases, starting haplotype frequencies were set to $h_1 = 0.05$, $h_2 = 0.45$, $h_3 = 0.45$, and $h_4 = 0.05$. Figure 7.8 shows the evolution of haplotype frequencies, $f_A = p_1$, $f_B = p_2$, $\bar{w}$, and D for both parameter sets.

Note that there are two differences between parameter sets 1 and 2. First, recombination between the target loci is 50 times more common under parameter set 1. Second, haplotypes 1 (AB) and 4 (ab) are disfavored relative to haplotypes 2 (Ab) and 3 (aB) in parameter set 1 ($e_{AB} = e_{ab} = 0.04$), while the opposite is true for parameter set 2 ($e_{AB} = e_{ab} = -0.02$).

These two differences in the modeled genetic systems result in substantially different values of all metrics monitored by the R function. First, comparing Fig. 7.8a and b, haplotypes 1 and 4 equilibrate at frequencies lower



	A/A	A/a	a/a
B/B	*0.94*	0.99	0.96
B/b	0.99	1.00	0.99
b/b	0.96	0.99	*0.94*

	A/A	A/a	a/a
B/B	*1.00*	0.99	0.96
B/b	0.99	1.00	0.99
b/b	0.96	0.99	*1.00*

Fig. 7.8 Results of deterministic two-locus selection for parameter set 1 (left column) and parameter set 2 (right column). The resulting fitness matrix for each parameter set is shown at the top of each column. In all panels, dashed gray lines indicate the values of the given quantity associated with the internal equilibrium found by grid search. Furthermore, in each run of `deterministic_twolocus_selection()`, starting haplotype frequencies were $h_1 = h_4 = 0.05$ and $h_2 = h_3 = 0.45$. (**a**,**b**) Evolution of haplotype frequencies. (**c**,**d**) Evolution of the measure of linkage disequilibrium D. (**e**,**f**) Evolution of mean population fitness $\bar{w}$

than those of haplotypes 2 and 3 due to differences in relative fitness under parameter set 1 and vice-versa for parameter set 2. However, the frequency differences between these two sets of haplotypes are much greater in the case of parameter set 2 due to the very low incidence of recombination between the two selected loci. Second, while the high recombination frequency associated with parameter set 1 results in near linkage equilibrium ($D = 0$), the very low recombination frequency associated with parameter set 2 results in near-maximal linkage disequilibrium ($D > 0.2$; cf. Fig. 7.8c and d). Finally, mean population fitness rises rapidly under parameter set 1 and then falls to a sub-optimal value of ~ 0.981 while $\bar{w}$ nearly reaches unity under parameter set 2, presumably because the near absence of recombination allows the most fit haplotypes of AB and ab to nearly eliminate the less-fit haplotypes of Ab and aB (cf. Fig. 7.8e and f). Equilibrium values of haplotype frequencies, D, and $\bar{w}$ agree with those identified by grid search (boxed problem below; dashed gray lines in Fig. 7.8).

Changes to the starting frequencies of the two haplotype classes—(1) AB and ab versus (2) Ab and aB—can drastically alter the frequency equilibria, even in a deterministic framework. Figure 7.9a and b shows the results of running the two parameters sets again but starting with flipped frequencies of the two haplotype classes, i.e., setting $h_1 = h_4 = 0.45$ and $h_2 = h_3 = 0.05$. Comparing these results to those of Fig. 7.8a, we see that identical haplotype frequency equilibria are attained; the path to those equilibria is simply different because of the different starting conditions. Allowing the starting frequencies of the haplotypes within each haplotype class to be slightly different from one another can cause dramatic changes. For example, starting with $h_1 = 0.04$, $h_4 = 0.05$ and $h_2 = 0.46$, $h_3 = 0.45$, haplotype frequencies eventually attain predicted equilibrium frequencies, but at a slower pace (cf. Figs. 7.8c and 7.9c, note scale of the generation axis). More dramatically, starting with the same haplotype frequencies, the greater starting frequency of haplotype 4 (ab) coupled with its fitness advantage ultimately results in the loss of polymorphism at both loci as haplotype 4 fixes (Fig. 7.9d). Flipping the starting frequencies within each haplotype class results in reciprocal results (Fig. 7.9e,f); notably, the higher starting frequency of haplotype 1 leads to fixation of this haplotype—rather than haplotype 4, as in Fig. 7.9d—under parameter set 2 (Fig. 7.9f). The results in Fig. 7.9d and f indicate that when two loci are tightly linked, it can be relatively easy for balancing selection to fail to preserve polymorphism. They also suggest that small perturbations in allele frequency due to finite population size may lead to unpredictable results when stochastic simulation is employed (Sect. 7.3.1.2)

Focusing on the results using parameter set 2, even when the frequencies of the favored 1 and 4 haplotypes start 0.002 apart, the result is the same. While starting all haplotype frequencies at 0.25 leads to predicted haplotype frequencies (Fig. 7.10a), making the minor adjustment of setting $h_1 = h_2 = 0.25$, $h_3 = 0.249$, and $h_4 = 0.251$ leads to the fixation of haplotype 4 despite its truly modest head start (Fig. 7.10b; $h_4 = 0.251$ vs. $h_1 = 0.249$). Again

	A/A	A/a	a/a
B/B	*0.94*	0.99	0.96
B/b	0.99	1.00	0.99
b/b	0.96	0.99	*0.94*

	A/A	A/a	a/a
B/B	*1.00*	0.99	0.96
B/b	0.99	1.00	0.99
b/b	0.96	0.99	*1.00*

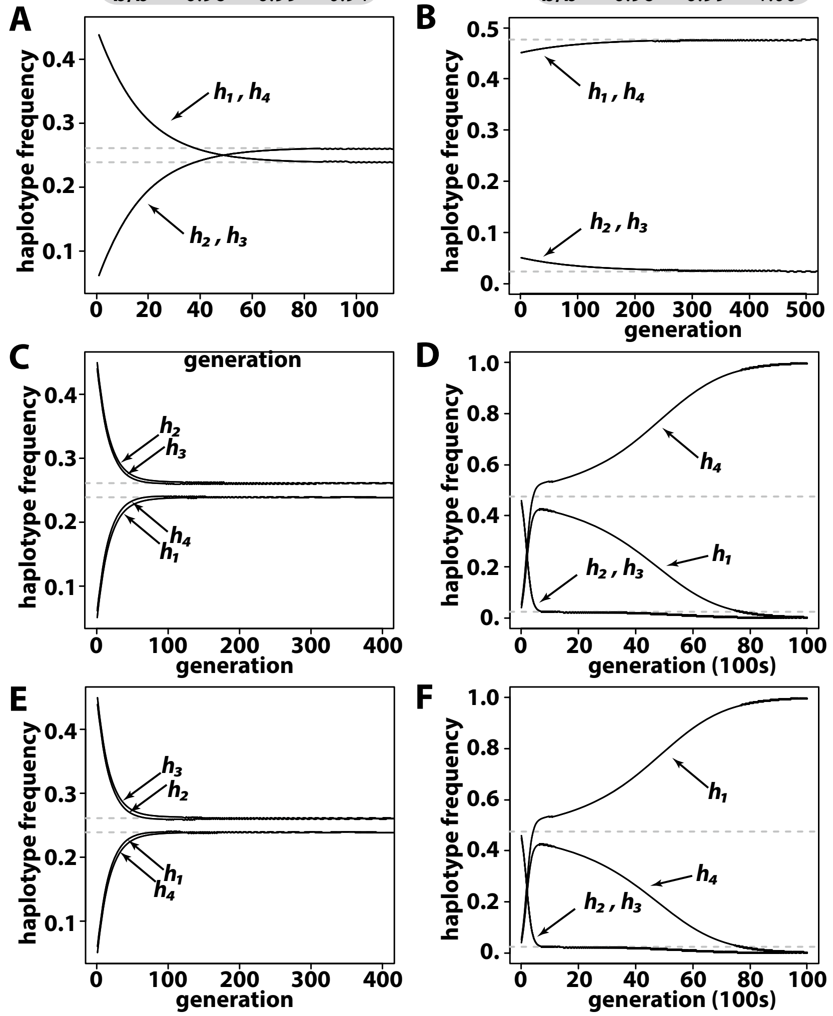


Fig. 7.9 Evolution of haplotype frequencies under deterministic two-locus selection for parameter set 1 (left column) and parameter set 2 (right column). The resulting fitness matrix for each parameter set is shown atop each column. In all panels, the dashed gray lines indicate the values of haplotype frequencies associated with the internal equilibrium found by grid search. Each row shows the results for different starting frequencies of the four haplotypes: (**a**,**b**) $h_1 = h_4 = 0.45$ and $h_2 = h_3 = 0.05$; (**c**,**d**) $h_1 = 0.04$, $h_2 = 0.46$, $h_3 = 0.45$, and $h_4 = 0.05$; (**e**,**f**) $h_1 = 0.05$, $h_2 = 0.45$, $h_3 = 0.46$, and $h_4 = 0.04$

demonstrating the substantial effect of recombination rate r on equilibrium frequencies, increasing recombination by one order of magnitude (e.g., from $r = 0.001$ to $r = 0.01$) dramatically decreases the equilibrium frequency of the favored haplotype class (1 and 4, or AB and ab) and increases the equilibrium frequency of the disfavored haplotype class (2 and 3, or Ab and aB; Fig. 7.11). Interestingly, the implication is that at least for some selection regimes even modest recombination can substantially lower the equilibrium frequencies of the favored haplotypes. In short, these results suggest that evolution of linked, two-locus systems is easily perturbed by any number of relevant parameters, in many cases leading to sub-optimal equilibrium frequencies of haplotypes.

In this subsection as well as Figs. 7.8, 7.9, 7.10, and 7.11, I referred to expected equilibrium values of haplotype frequencies, $\bar{w}$, and D via *grid search*. However, I did not provide details of a grid search. As Figs. 7.8, 7.9, 7.10, and 7.11 show, expectations of key values derived from a different source than a simulation program allow us to validate that our program is performing correctly. In this situation, a grid search is a brute force approach in which we test as many sets of haplotype frequencies as possible to find at least an approximate equilibrium point. Write a program that performs a grid search of haplotype frequencies to identify equilibrium haplotype frequencies. Here are some things to consider when planning the program:

- Equilibrium haplotype frequencies must sum to one.
- Frequencies must be on the domain [0,1], but the search space becomes exponentially larger as the precision of the equilibrium frequencies becomes finer—for example, from 0.01 to 0.001 to 0.0001 to 0.00001. Test different levels of precision.
- You will need some numerical criterion whose maximum or minimum value (depending on the criterion) is associated with the set of haplotypes that provide the closest approximation to equilibrium values.
- Your inputs should include r, D, selection coefficients, and epistatic coefficients.

7.3.1.2 Stochastic Simulation

We now consider a program written in C++ that enables stochastic simulation of a two-locus selection system.

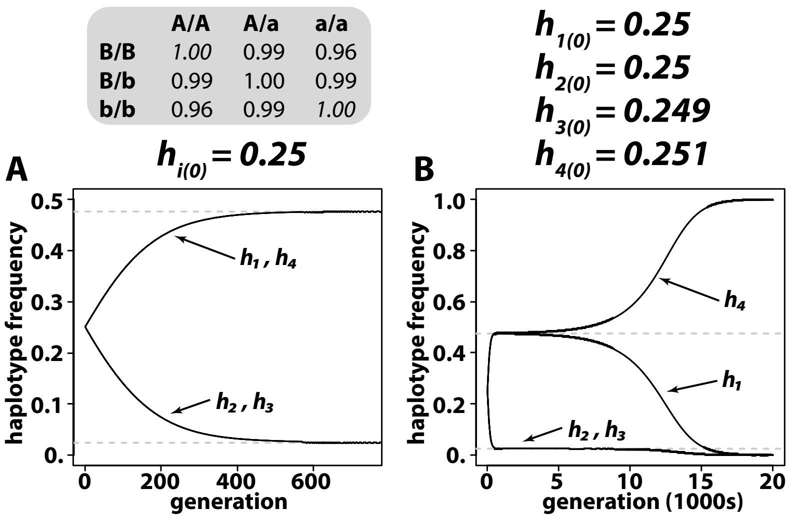


Fig. 7.10 Evolution of haplotype frequencies under deterministic two-locus selection for parameter set 2. In all panels, the dashed gray lines show the equilibrium haplotype frequencies identified by grid search. (**a**) All starting haplotype frequencies are 0.25. (**b**) Starting haplotype frequencies as shown above the panel

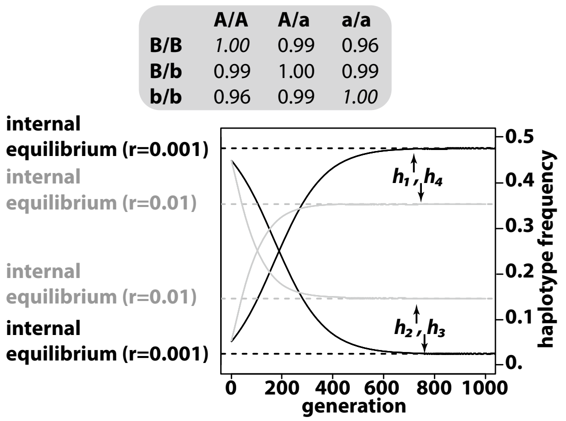


Fig. 7.11 Recombination rate r affects the internal equilibrium under two-locus selection. Here, results for two different values of r are shown: $r = 0.001$ (black lines, parameter set 2) and $r = 0.01$ (gray lines, parameters identical to those of parameter set 2, except for r). Note that the fitness matrix remains the same regardless of r. Dashed lines show the equilibrium haplotype frequencies identified by grid search. In both cases, starting haplotype frequencies were $h_1 = h_4 = 0.05$ and $h_2 = h_3 = 0.45$

stochastic_twolocus_selection.cc

```
#include <random>
#include <fstream>
#include <map>
#include <vector>
#include <algorithm>
#include "matrix.h"
using namespace std;

vector<int> genos_from_haps(const vector<int> &haps);
vector<int> reproduce(const vector<vector<int> > &parents, const double
     ↪ &r, const vector<double> &prerand);

int main(int argc, char *argv[]) {

  // command line arguments
  double r = atof(argv[1]); // recombination rate r on the range (0.,0.5)
  double s_A = atof(argv[2]); // selection coefficients
  double s_B = atof(argv[3]);
  double t_A = atof(argv[4]);
  double t_B = atof(argv[5]);
  double h0 = atof(argv[6]); // initial haplotype COUNTS of ... A-B
  double h1 = atof(argv[7]); // ... A-b
  double h2 = atof(argv[8]); // ... a-B
  double h3 = atof(argv[9]); // ... a-b
  double eAB = atof(argv[10]); // epistatic parameter for AB/AB
  double eAb = atof(argv[11]); // ... Ab/Ab
  double eaB = atof(argv[12]); // ... aB/aB
  double eab = atof(argv[13]); // ... ab/ab
  int gens = atoi(argv[14]);

  uniform_real_distribution<double> randomnum(0.,1.);
  mt19937 engine(time(0));

  double p_A, p_B, w0, w1, w2, w3, D; // allele frequencies, marginal
       ↪ fitnesses and D

  // calculate popuation size and fitness matrix
  int numhaplotypes = h0+h1+h2+h3;
  int N = numhaplotypes/2;
  vector<double> hapcounts = {h0, h1, h2, h3};
  vector<double> fitness;
  fitness.push_back(1-s_A -s_B-eAB);
  fitness.push_back(1-s_B);
  fitness.push_back(1-t_A-s_B-eAb);
  fitness.push_back(1-s_A);
  fitness.push_back(1);
  fitness.push_back(1-t_A);
  fitness.push_back(1-s_A-t_B-eaB);
  fitness.push_back(1-t_B);
  fitness.push_back(1-t_A-t_B-eab);
  double* f = &fitness[0];
  Matrix<double> fit(3, 3, f); //fitness matrix

  //print fitness matrix
  for (int i=0; i<3; ++i) {
    for (int j=0; j<3; ++j)
      cout << fit[i][j] <<"\t" ;
    cout << endl;
  }

  // create vector of available haplotypes
  vector<int> haplotypes;
  for (int i=0; i<4; ++i)
    for (int j=0; j<hapcounts[i]; ++j)
      haplotypes.push_back(i);

  // randomly pair haplotypes to individuals
  map<int, vector<int> > individuals;
  random_shuffle(haplotypes.begin(), haplotypes.end());
  for (int i=0; i<numhaplotypes; i+=2) {
    individuals[i/2].push_back(haplotypes[i]);
    individuals[i/2].push_back(haplotypes[i+1]);
    individuals[i/2].push_back(0);
    individuals[i/2].push_back(0);
  }

  // add A and B locus genotypes of each individual as [2] and [3] entries
  for (int i=0; i<N; ++i){
    vector<int> haps = {individuals[i][0], individuals[i][1]};
    vector<int> genos = genos_from_haps(haps);
    individuals[i][2] += genos[0];
    individuals[i][3] += genos[1];
  }

  ofstream datafile("twolocus_selection_data");
  datafile << "gen\th1\th2\th3\th4\tp1\tp2\tD"<< endl;

  for (int gen=0; gen<gens; ++gen) {
    // calc and store current haplotype frequencies
    vector<double> haplotypeFreqs;
    for (int i=0; i<4; ++i)
      haplotypeFreqs.push_back(hapcounts[i]/numhaplotypes);

    // calc and store current allele frequencies
    p_A = haplotypeFreqs[0] + haplotypeFreqs[1]; // A-B and A-b
    p_B = haplotypeFreqs[0] + haplotypeFreqs[2]; // A-B and a-B

    // calculate D using the A-B haplotype and print
    D = haplotypeFreqs[0] - p_A * p_B;
    datafile << gen << "\t" << haplotypeFreqs[0] << "\t" <<
        ↪ haplotypeFreqs[1] << "\t" << haplotypeFreqs[2] << "\t" <<
        ↪ haplotypeFreqs[3] << "\t" << p_A << "\t" << p_B << "\t" << D
        ↪ << endl;

    // cull individuals that don't make fitness test
    vector<int> remainder; // stores index of surviving individuals
    for (int i=0; i<N; ++i) {
      if (randomnum(engine) <= fit[ individuals[i][3] ][
          ↪ individuals[i][2] ] ) //
        remainder.push_back(i);
    }

    uniform_int_distribution<int> randomint(0, remainder.size()-1);
    map<int, vector<int> > nextgen;

    for (int i=0; i<N; ++i) {
      vector<vector<int> > parents;
      parents.push_back( individuals[remainder[randomint(engine)]] );
      parents.push_back( individuals[remainder[randomint(engine)]] );
      vector<double> pr = {randomnum(engine), randomnum(engine),
          ↪ randomnum(engine)};
      nextgen[i] = reproduce(parents, r, pr);
    }

    individuals = nextgen;
    hapcounts = {0,0,0,0};
    for (int i=0; i<N; ++i) {
      hapcounts[individuals[i][0]]++;
      hapcounts[individuals[i][1]]++;
    }
  }
  datafile.close();
  return(0);
}

vector<int> genos_from_haps(const vector<int> &haps) {
  vector<int> vecky = {0,0};
  if (haps[0] == 2 || haps[0] == 3)
    vecky[0]++;
  if (haps[1] == 2 || haps[1] == 3)
    vecky[0]++;
  if (haps[0] == 1 || haps[0] == 3)
    vecky[1]++;
  if (haps[1] == 1 || haps[1] == 3)
    vecky[1]++;
  return(vecky);
}

vector<int> reproduce(const vector<vector<int> > &parents, const double
    ↪ &r, const vector<double> &prerand) {
  vector<int> v = {-1,-1,0,0};
  for (int i=0; i<2; ++i) {
    if (parents[i][0] + parents[i][1] == 3) { // then double
        ↪ heterozygote (AB/ab or Ab/aB)
                              // need to check for recombination ...
      if (prerand[2] <= r) {
        if (parents[i][0] == 0 || parents[i][1] == 0) {
          if (prerand[i] <= 0.5)
            v[i]=1;
          else
152               v[i]=2;
153           } else {
154              if (prerand[i] <= 0.5)
155                 v[i]=0;
156              else
157                 v[i]=3;
158           }
159        }
160     }
161     if(v[i]<0) {
162        if (prerand[i] <= 0.5)
163           v[i]=parents[i][0];
164        else
165           v[i]=parents[i][1];
166     }
167  }
168
169  vector<int> haps = {v[0], v[1]};
170  vector<int> genos = genos_from_haps(haps);
171  v[2] = genos[0];
172  v[3] = genos[1];
173  return(v);
174 }
```

Lines 9–10 declare two functions defined on lines 129–140 and 147–179. Although it is a bit cumbersome, all parameters of the simulation are read as command-line arguments; you could see. They include selection coefficient, initial haplotype *counts*, epistatic parameters, and the number of generations to run the simulation. Parameter values must be entered on the command-line in the same order as they are read on lines 15–28. These parameters are then used to determine population size (lines 36–37) and initialize the fitness matrix (lines 40–50), which are printed to standard output (lines 53–57).

Lines 60–63 initialize the `vector` variable `haplotypes`, which holds a number of zeroes equal to the *count* of haplotype 1 (AB), a number of ones equal to the *count* of haplotype 2 (Ab), &c (admittedly, zero-based indexing makes this code a bit awkward). N `individuals` are then initialized as a `map`, where the key is an integer corresponding to an individual and the value is a vector of the two haplotypes *and* two genotypes it carries (line 66). The haplotypes are obtained by shuffling the `haplotypes vector` and then assigning every two haplotypes to the next individual (lines 67–73). Genotypes are inferred from the haplotypes and assigned to the second and third indices of the `individuals` vector (lines 76–81) using the function `genos_from_haps()` (lines 129–140). The final bit of setup is to create an output data file that will track the frequencies of all four haplotypes, the frequencies of the A and B alleles at the two loci, and the linkage disequilibrium measure D (lines 83–84).

Simulation of the selection on the two-locus system is coded in the `for` loop spanning lines 86–127. Haplotype frequencies (lines 88–90), allele frequencies (lines 93–94), and D (line 97) of the current generation are calculated and printed to the output file (line 98). Viability selection is then imposed by eliminating individuals that fail a selection test and survivors indices are stored in the `vector<int> remainder` (lines 101–105). Next, a random number generator is created to randomly select parents from among `remainder` (line 107) and a temporary `map` named `nextgen` is used to store the offspring of mating (line 108). In the `for` loop spanning lines 110–116, an `individual` of the next generation is generated by passing two randomly selected parents and a vector of three randomly generated numbers on the domain [0,1] to the function `reproduce()` (see next paragraph). After N offspring are generated in this manner, the temporary `nextgen` is assigned to `individuals` and the new `hapcounts` are calculated (lines 118–123).

The function `reproduce()` (lines 142–174; called at line 115) first determines if an individual parent is a double heterozygote (conditional on line 145). If so, it is necessary to test whether or not crossing-over between the two selected loci occurs (line 147). If it does, the haplotype passed on to the offspring form , the parent currently under consideration, depends on the type of double heterozygote (AB/ab or Ab/aB) and a random fifty-fifty chance as to which recombinant haplotype is passed on (lines 148–158). If the parent is not a double heterozygote—or it is, but crossing-over does not occur—one of the two nonrecombinant haplotypes is passed on to the offspring (lines 161–166). After considering both parents, the genotypes are assigned by a call to `genos_from_haps()` and the vector of haplotypes and genotypes is returned (lines 169–173).

Figure 7.12 shows the results of stochastic simulation of parameter sets 1 and 2 (introduced in the previous Sect. 7.3.1.1). As suggested by grid search and deterministic simulation, parameter set 1 rapidly leads to a situation in which haplotypes 2 and 3 maintain a slight frequency advantage over haplotypes 1 and 4. The stochastic nature of the simulation is indicated by the moderate up-and-down changes in haplotype frequencies (Fig. 7.12a). In the case of parameter set 2, we already saw (in a deterministic context, Fig. 7.9d,f) that even marginal differences between the starting frequencies of favored haplotypes 1 and 4 caused the haplotype of greater starting frequency to fix. The stochastic simulation results shown in Fig. 7.12b are drawn from a simulation in which the favored haplotypes began at equal frequencies of 0.05. This leads my mind to an intuition that the two favored haplotypes would achieve equal, mid-range frequencies as in the deterministic context (Fig. 7.9b). The representative stochastic simulation results for parameter set 2 (Fig. 7.12b) of course provide an empirical counterpoint to any similar intuition you may have held with me. In this representative example, h_1 *randomly* gains a sufficient frequency advantage around generation 400 that ultimately drives it to fixation and the other three haplotypes (including the favored haplotype 4) to extermination. Yet again, we see that chance determines an outcome that diverges from expectations based on the driving action of natural selection alone. The random number generators of stochastic simulation mimic the noise of natural processes.

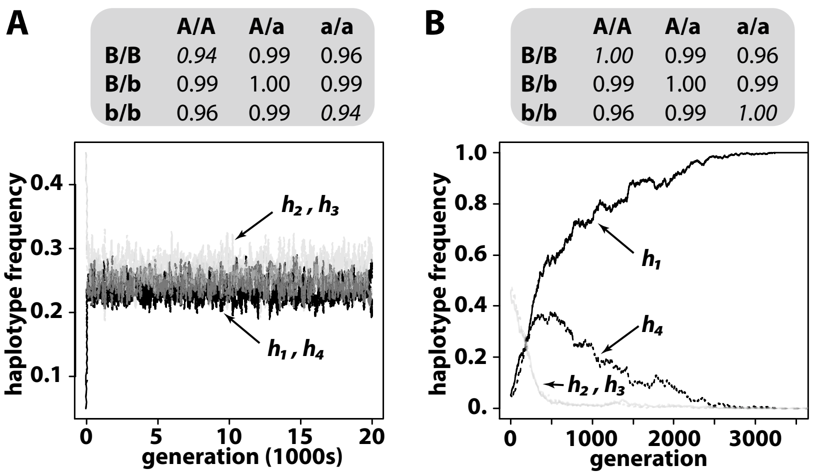


Fig. 7.12 Stochastic simulation of selection on two linked loci. Representative examples for (**a**) parameter set 1 and (**b**) parameter set 2. Note that although h_1 is shown to fix in (**b**), there were even chances of either h_1 or h_4 ultimately fixing in the simulated population. In both cases, the starting haplotype frequencies were $h_1 = h_4 = 0.05$ and $h_2 = h_3 = 0.45$

References

Edwards AWF (2002) The fundamental theorem of natural selection. Theor Popul Biol 61:335–337

Fisher R (1930) The genetical theory of natural selection. Clarendon Press, Oxford, UK

Gould SJ, Lewontin RC (1979) The spandrels of san marco and the parglossian paradigm: a critique of the adaptionist programme. Proc Roy Soc B Biol Sci 205:581–598

Okasha S (2018) Agents and goals in evolution. Oxford University Press, Oxford, UK

Rice SH (2004) Evolutionary theory: mathematical and conceptual foundations, 1st edn. Sinauer Associates

Wright S (1932) The role of mutation, inbreeding, crossbreeding and selection in evolution. Proc Sixth Annu Congress Genet 1:356–366

8
Effects of Selection on Linked Variants

Space and perception generally represent, at the core of the subject, the fact of his birth, the perpetual contribution of his bodily being, a communication with the world more ancient than thought.[1]
– Maurice Merleau-Ponty, *Phenomenology of Perception*

8.1 Modeling Natural Selection and Linked Polymorphism

In this chapter, we return to amending the forward simulation program FORTUNA. We begin with numerous additions and changes to existing program files. These changes allow us to model and simulate natural selection within FORTUNA and track its effects on **linked sequence diversity**. In this section, we focus on the changes to program files that enable the simulation of natural selection. Section 8.2 then explores the use of the updated program to simulate different types of *positive* selection, including selection on new or standing adaptive variants, overdominance, and negative-frequency dependence. In Sect. 8.3, we investigate purifying selection and background selection using the updated program. Note that the following listings contain some code related to frequency-dependent and purifying selection that I will address more fully in Sects. 8.2 and 8.3.

fortuna.cc // additions and modifications

```
#include <numeric> // for iota() function
int main() {
  ...
   bool simulate = true;

```

[1] Quoted with permission from the translation by Colin Smith, 2002, Francis & Taylor Group.

R. J. Haasl, *Nature in Silico*, https://doi.org/10.1007/978-3-030-97381-0_8

```
 6     while (simulate) {
 7       Metapopulation meta;
 8       for (int i =0; i < runlength; ++i) {
 9         if (i % 5 == 0) cout << i<< endl;
10         bool test = meta.reproduce_and_migrate(i);
11         if (! test) break;
12         if (i == runlength - 1) simulate=false;
13       }
14       meta.close_output_files();
15     }
16   }
```

Modification to the main() function consists of wrapping the instantiation of the Metapopulation object within a while loop (lines 6–15) conditioned on the Boolean variable simulate initially set to true (line 4). This variable only becomes false when the simulation has run for runlength generations (line 12), after which the program is set to terminate. Lines 10–11 allow the simulation to restart if the selected variant is lost from a population, which happens frequently when the adaptive variant begins at very low frequency. A modified Metapopulation function—reproduce_and_migrate() (see metapopulation.h listing this section) only returns true (see subsequent listings) when the selected variant has a frequency > 0. If the selected variant is lost and test therefore returns false (line 10), program execution falls out of the for loop (line 11) and output files are closed (line 14). Because simulate still equals true, the return to line 7 instantiates a new Metapopulation, output files are overwritten, and a new simulation begins.

parameters, **params.h**, **params.cc** // additions and modifications

```
 1  // parameters file
 2  ... // in each DEME block
 3  sellocus 500000 1 0.05 0.025 1
 4  possel 0. 0. 0.
 5  nfdsel 0.5 3.
 6  negsel 0. 0. 275000 325000 100
 7  ...
 8
 9  // params.h
10  extern map<int, vector<double> > sellocus, possel, nfdsel, negsel;
11
12  // params.cc
13  map<int, vector<double> > sellocus, possel, nfdsel, negsel;
14  int process_parameters() {
15    if (iter->first == -1) {
16      ...
17     } else {
18      ...
19      sellocus[iter->first] = get_multi_double_param("sellocus",
          ↪ parameters);
20      possel[iter->first] = get_multi_double_param("possel", parameters);
21      nfdsel[iter->first] = get_multi_double_param("nfdsel", parameters);
22        negsel[iter->first] = get_multi_double_param("negsel", parameters);
23     }
24  }
```

Four new, multi-part parameters specific to populations (and therefore listed in the DEME block of each simulated population in file `parameters`) facilitate simulation of qualitatively different forms of natural selection (lines 3–6, `parameters`). Lines 9–24 show the code necessary for declaring and defining these parameters using the same mechanics as in previous chapters. The parameter values shown on lines 3–6 are representative only. In this section, we only discuss the meanings of `sellocus` and `possel`. Meanings of `nfdsel` and `negsel` will be addressed in Sects. 8.2.5 and 8.3, respectively.

Parameter `sellocus` first specifies the location of the selected locus (index [0]) and the starting *count* of the selected variant (index [1]). If the selected variant begins at a count of greater than 1, we need to identify a polymorphic site generated by MS simulation that most closely matches the position and starting count specified by the first two elements of `sellocus`. Required precision of this match is controlled by the next two elements of the parameter (indices [2] and [3]). Respectively, they specify the maximum *distance* (as a fraction of `seqlength`) the MS generated variant can lie on either side of the preferred location and the maximum *frequency difference* from the desired frequency that will be tolerated. For example, if `seqlength` is set to 1,000,000, `popsize` is set to 10,000, and `sellocus` is set, as shown, to `500000 250 0.05 0.025 1`, then the program will search for an MS generated variant that is between the position 450,000bp and 550,000bp and at a frequency between 0 and 0.05.

The final entry of parameter `sellocus` is set to 0 or 1. This controls whether the loss of the selected variant causes the simulation to begin again. This is not particularly useful when only one population is simulated. However, when multiple populations are simulated, it may be desirable to allow certain populations to lose the selected variant without terminating the simulation. The fifth (index [4]) entry of the `sellocus` parameter should be set to zero for any population where loss of the selected variant is *allowed*.

Parameter `possel` specifies the selective regime under positive natural selection. Respectively, the three entries specify the coefficients s, t, and h, where s and h are the selection and dominance coefficients as discussed in Chap. 7 and t is a second selection coefficient used when simulating overdominance (Sect. 8.2.4).

`metapopulation.h` // additions and modifications

```
public:
  int reproduce_and_migrate(int gen) { // return variable changed to int
    bool fixtest = true; // new variable
    ...
    if ( gen==0 || (gen+1) % sampfreq == 0) {
      for (int i=0; i<pop_num; ++i) {
```

```
7           if ((*populations[i]).get_extant()) {
8              if (possel[(*populations[i]).get_popnum()][0] != 0 &&
                   ↪ sellocus[(*populations[i]).get_popnum()][4] == 1 )
9                 fixtest = (*populations[i]).sample(gen);
10             else
11                int dummy = (*populations[i]).sample(gen);
12             if (! fixtest )
13                break;
14          }
15       }
16    }
17    return (fixtest);
18 }
```

Within the modified function `reproduce_and_migrate()`, variable `fixtest` (line 3) remains `true` as long as the selected variant is at a frequency > 0. Furthermore, `fixtest` is now the return variable for the function `reproduce_and_migrate()` to the `main()` function (line 2; see `fortuna.cc` listing above). The value of `fixtest` determines whether the simulation is restarted due to loss of the selected variant—in which case the variable evaluates as `false` upon returning from the call to `sample()` on line 9. If a population is extant this generation (line 7), we check to see if (1) positive selection is active in this population and (2) if so, whether simulation restart is required upon loss of the selected variant in this population (line 8). If both of these conditions hold, we then take a sample from the current population and set `fixtest` based on whether or not the adaptive variant is still present in the currently considered population. If one or neither of the conditions tested in line 8 hold, we still sample the population and update allele frequencies in the population. However, the return value of function `sample()` is irrelevant and ignored (lines 10–11). Finally, if `fixtest` returns `false` (line 12)—because we *do* require the selected variant to survive in the currently considered population and it has not—we immediately `break` the `for` loop (lines 6–15).
The majority of changes necessary to incorporate simulation of natural selection are to `population.h`:

population.h // additions and modifications

```
private:
...
ofstream sinfo;
bool activeselection{}, standingvar{}, newvariant{}, nfdselection{},
   ↪ negselection{};
int keypos=-999;
vector<double> fitness;
double nfd_s, nfd_h;
vector<int> purifying_sites;
...
void update_selected_freqAndFit(const int &gen) {
  double count = pop_schedule[popn][gen]*2; // start count at ALL alleles
      ↪ derived
  vector<double> genotype_freqs = {0.,0.,0.}; // holds frequency of A/A,
      ↪ A/a, a/a, where A is derived allele
  for (auto iter = individuals.begin(); iter != individuals.end();
      ↪ ++iter) {
    int num_ancestral_alleles = (**iter).get_genotype(keypos);
    count -= num_ancestral_alleles;
    genotype_freqs[num_ancestral_alleles]++;
  }
  double p = count/(pop_schedule[popn][gen]*2); // frequency of derived
      ↪ allele
  double q = 1-p;
  for (int i=0; i<3; ++i)
    genotype_freqs[i] /= pop_schedule[popn][gen];
  sinfo << gen << "\t" << p << "\t" << genotype_freqs[0] << "\t" <<
      ↪ genotype_freqs[1] << "\t" << genotype_freqs[2] << "\t";

  if (nfdselection) {
    /* fitness[0] = 1 - (nfd_s*p*p);
    fitness[1] = 1 - (2*nfd_s*nfd_h*p*q);
    fitness[2] = 1 - (nfd_s*q*q);*/

    double max_fit = -999;
    fitness[0] = 1 - nfd_h*genotype_freqs[1] + nfd_h*genotype_freqs[2];
    if (fitness[0] > max_fit) max_fit = fitness[0];
    fitness[1] = 1 - nfd_s*genotype_freqs[1];
    if (fitness[1] > max_fit) max_fit = fitness[1];
    fitness[2] = 1 - nfd_h*genotype_freqs[1] + nfd_h*genotype_freqs[0];
    if (fitness[2] > max_fit) max_fit = fitness[2];
    for (int i=0; i<3; ++i)
      fitness[i] /= max_fit;
  }
  sinfo << fitness[0] << "\t" << fitness[1] << "\t" << fitness[2] << endl;
}

bool update_alleles(const int &gen) { return value change: void to bool
  bool fixtest = true;
  ...
  for (auto iter = alleles.begin(); iter != alleles.end(); ++iter) {
    ...
      if (current_count == 0) {
       ...
         if ( iter->first == keypos) fixtest = false;
      }

    if (current_count == pop_schedule[popn][gen]*2) { // derived allele
        ↪ FIXED
       if ( (activeselection && iter->first != keypos) ||
           ↪ !activeselection) {
         to_remove.push_back(iter->first);
         for (auto iter2 = individuals.begin(); iter2 !=
             ↪ individuals.end(); ++iter2)
           (**iter2).remove_fixed_allele(iter->first);
           }
        }
          ...
      }
      return (fixtest);
}

public:
void reproduce(int gen) {
    ...
     // removal of potential parents by selection (all new)
    vector<int> recode_parents;
    if (activeselection)
       for (int i=0; i<individuals.size(); ++i)
          if (randomnum(e) <= fitness[
              ↪ (*individuals[i]).get_genotype(keypos) ] )
              ↪ recode_parents.push_back(i);
    if (negselection)
       for (int i=0; i<individuals.size(); ++i)
          if (randomnum(e) <= fitness[
              ↪ (*individuals[i]).get_negsel_genotype(purifying_sites) ] )
              ↪ recode_parents.push_back(i);
     if (activeselection || negselection)
       randomind.param(uniform_int_distribution<int>::param_type(0,
            ↪ recode_parents.size() - 1) );

     for (int i=0; i<N; ++i) {
       vector<int> parents;
       if (activeselection || negselection) {
          parents.push_back(recode_parents[randomind(e)]);
          parents.push_back(recode_parents[randomind(e)]);
       } else {
          parents.push_back(randomind(e));
          parents.push_back(randomind(e));
       }
         ...
     }
     ...
    if (gen % sampfreq == 0 && (!activeselection && !negselection)) \\
        ↪ added last set of conditionals
       update_alleles(gen);

    if ( gen != 0 && gen % sampfreq == 0 && (!activeselection &&
        ↪ !negselection)) { // added last conditionals
       random_shuffle(individuals.begin(), individuals.end() ) ;
       get_sample(gen);
    }
    if (activeselection) update_selected_freqAndFit(gen);
}

bool sample(int gen) {
    bool fixtest = update_alleles(gen);
    if (!fixtest && sellocus[popn][4]==1) {return fixtest;}
    random_shuffle(individuals.begin(), individuals.end() ) ;
```

```
   get_sample(gen);
   return(fixtest);
}

void close_output_files () {
   ...
    if (activeselection) sinfo.close();
}

Population (...) {
   if (possel[popn][0]!=0 || nfdsel[popn][0]!=0 || negsel[popn][0]!=0) {
         if (nfdsel[popn][0] != 0)
            nfdselection = true;
        if (negsel[popn][0] != 0) {
            negselection = true;
            fitness = {1, 1-(negsel[popn][0]*negsel[popn][1]),
                 ↪ 1-negsel[popn][0]};
            vector<int> possible_sites(negsel[popn][3]-negsel[popn][2]+1);
            iota(possible_sites.begin(), possible_sites.end(),
                 ↪ negsel[popn][2]);
            random_shuffle(possible_sites.begin(), possible_sites.end());
            auto iter = possible_sites.begin();
            purifying_sites.assign(iter, iter+negsel[popn][4]);
            sort (purifying_sites.begin(), purifying_sites.end());
        } else
            activeselection = true;

        if (activeselection) {
            keypos = sellocus[popn][0];
            if (sellocus[popn][1] > 1)
               standingvar=true;
            else
               newvariant=true;
            double s = possel[popn][0];
            double t = possel[popn][1];
            double h = possel[popn][2];
            nfd_s = nfdsel[popn][0];
            nfd_h = nfdsel[popn][1];
            if (t == 0) fitness = {1, 1-(h*s), 1-s}; // dominant or additive
            else fitness = {1-s, 1, 1-t}; // overdominant
        }
   }
    ...
   if (activeselection) {
      ofname = "deme" + to_string(popn) + "_selectioninfo";
      sinfo.open(ofname.c_str());
   }

    if (useMS[popn]) {
      ...
      if (activeselection && !standingvar)
            alleles.insert( { keypos, new Allele(keypos, -1, popn) } );

      while(getline(ms_output, ms_line)) {
        if (regex_search(ms_line, query)) { // reading in allele positions
            ...
          while (iss >> s) {
              int position = seqlength * atof(s.c_str());
              allele_positions.push_back(position);
              if (standingvar || position != keypos )
                alleles.insert( { position , new Allele(position,-1,
                    ↪ popn) } );
          }
             ...
        }
        if (trigger) {
          vector<int> s1, s2;
          for (int i=0; i < ms_line.length(); ++i)
            if (ms_line[i] == '1') {
              if (allele_positions[i] != keypos)
                s1.push_back(allele_positions[i]);
              if (standingvar)
                (*(alleles[allele_positions[i]])).increment_count();
            }
          getline(ms_output, ms_line);
          for (int i=0; i < ms_line.length(); ++i)
            if (ms_line[i] == '1') {
              if (allele_positions[i] != keypos)
                s2.push_back(allele_positions[i]);
              if (standingvar)
                (*(alleles[allele_positions[i]])).increment_count();
            }
          if (activeselection && !standingvar && newvariant) {
            s1.push_back(keypos);
            newvariant = false;
            sort(s1.begin(), s1.end());
          }
          ...
       }
     }

    if (activeselection && standingvar) { // determine keypos
      int low=keypos-floor(sellocus[popn][2]*seqlength);
      if (low < 0 ) low = 0;
      int high=keypos+floor(sellocus[popn][2]*seqlength);
      if (high >= seqlength) high = seqlength-1;
      vector<int> current_best{-999, 999999,-999};
      int acceptablediff =
           ↪ floor(pop_schedule[popn][0]*2*sellocus[popn][3]);
      int target_low = sellocus[popn][1] - acceptablediff;
      int target_high = sellocus[popn][1] + acceptablediff;
      for (auto iter=alleles.begin(); iter != alleles.end(); ++iter) {
        if (iter->first >= low && iter->first <= high) { // in range
          int ct = (*(iter->second)).get_count();
          if (ct >= target_low && ct <=target_high) {
            int diff = abs(sellocus[popn][1]-ct);
            if (diff < current_best[1] ) {
              current_best[0] = iter->first;
```

```
208                   current_best[1] = diff;
209                   current_best[2] = ct;
210                }
211             }
212          } else {
213             if (iter->first > high) break;
214             else continue;
215          }
216       }
217       if (current_best[0] != -999) {
218          keypos = current_best[0];
219          cout << "Standing variant identified at base pair " << keypos <<
               ↪ " with a starting allele count at time 0 of " <<
               ↪ current_best[2] << endl;
220       } else
221          throw("suitable standing variant not identified");
222
223       if (nfdselection)
224          update_selected_freqAndFit(0);
225
226       sinfo << "gen\tp\tf_AA\tf_Aa\tf_aa\tw_AA\tw_Aa\tw_aa" << endl;
227    }
228    ...
229 }
```

We first declare several `private` variables required to implement various types of natural selection within the program (lines 3–8). Note that variable `keypos` holds the position of the selected variant in cases where natural selection is active and only one site is the target of selection (line 5).

The function `update_selected_freqAndFit()` (lines 10–40) is only required for implementation of negative frequency-dependent selection (NFD)—where relative fitness of genotypes requires calculation of current allele frequencies. The function updates the frequency of alleles at the site targeted by selection and updates `vector<double> fitness` when NFD is simulated. However, we call this function when any form of selection is simulated because (1) updating the frequency of a single site costs next-to-nothing computationally (unlike updating all segregating sites in the population), and (2) consistent updating of the selected variant's frequency allows us to identify the specific generation when, for example, a positively selected variant reaches fixation.

Function `update_selected_freqAndFit()` first declares and defines `count` as the number of chromosomes present in the diploid population (line 11). Then, `genotype_counts` is declared and list-defined with each of the three possible genotypes set to an initial frequency of 0 (line 12). The `for` loop from lines 13–17 determines the genotype of each `individual` through a call to a new function `get_genotype()` of the `Individual` class (see next listing). Genotype is recorded as the number of ancestral alleles, i.e., a genotype of 0 bears two *derived* alleles. On line 15, `count`—initially set to $2N_e$ but ultimately holds the number of derived alleles in the population—is debited by the number of ancestral alleles of the individual under consideration. On line 16, the relevant genotype is incremented. Frequencies of the derived and ancestral alleles (*p* and *q*, respectively) are calculated on lines 18–19 and genotype frequencies are calculated by dividing the genotype counts held in genotype_freqs by the total number of chromosomes in the population (lines 20–21). Results are then printed to the selection information file (handle sinfo; line 22). The remainder of the function is specific to the case of negative frequency-dependent selection (lines 24–39). In the case of NFD, relative fitnesses may either be calculated in terms of expected genotype frequencies (lines 25–27) or empirical genotype frequencies (lines 29–37). As shown in this listing, the former method is commented out, forcing the program to calculate the three relative fitnesses based on empirical genotype frequencies. Recalculated relative fitness values are then printed to the selection information file (line 39).

Several additions are made to the previously established private function update_alleles(). This function is called whenever a sample is drawn from the population via the function sample(). Its essential, computationally intensive, role is to update the frequencies of all alleles. The return type is changed to bool (line 42) and the return variable fixtest is declared and defined as true (line 43). If an allele is found to have a count of zero in the population (line 47) and the position of the allele is keypos, fixtest is set to false (line 49); remember, a false value of fixtest indicates the advantageous allele has been lost from the population. Conversely, if only the derived allele is left at a previously polymorphic site (conditional test on line 52), the Allele class object is removed from the population. However, we want a fixed, adaptive allele to remain listed as an allele in the population. Otherwise, each individual will suddenly be absent the advantageous allele and another, artificial selective sweep will impact the population . To facilitate this, monomorphic alleles are only removed from the population if one set of conditions on line 53 is met: (1) the variable activeselection (see below) is false, meaning we are not simulating *positive* selection or (2) activeselection is true but the position of the allele is not keypos.

Viability selection is implemented through additions to the previously described public function reproduce(). First, vector<int> recode_parents is declared (line 68). This container holds the indices of individuals that survive selection and therefore become potential parents of the next generation. In the case of positive selection, the variable activeselection is true. In this case, an individual is added to recode_parents if a random number uniformly distributed on the domain [0,1] is $\leq$ the fitness of the individual (lines 69–71). When simulating purifying selection, negselection is true and an individual is added to recode_parents if a random number is less than the fitness determined by function get_negesel_genotype() specific to purifying selection (lines 72–74; see Sect. 8.3). If either type of selection is operative, the randomind uniform random number generator is subsequently reset to select integers from 0 to one less than the number of individuals contained by `recode_parents` (lines 75–76). The `for` loop that generates individuals of the next generation (lines 78–88) is now modified to draw `parents` from the surviving `recode_parents` when natural selection is active (lines 80–83).

It is important to understand why removal of potential parents in this manner is a means of modeling viability selection. Fitness deficits under viability selection are manifested as the inability to reach sexual maturity and therefore produce offspring. In code, failing the "fitness test" of lines 71 and 74 results in the removal of the individual from the set of possible parents. Although we do not do so here, it would be simple to modify this aspect of function `reproduce()` to, for example, model **fecundity selection**. In this case, a fitness deficit would manifest as a lower mean number of offspring.

Consider how you would implement fecundity selection as well as **sexual selection** in code. Note that both forms of selection require an implementation of the sex of each `individual`.

Finally, we add an extra set of conditionals to previous code (lines 90 and 93) in order to prevent running `update_alleles()` (line 91) and `get_sample()` (line 95) twice when selection is active. This is necessary, because the function `sample()` is called when selection is active, and this function (discussed next) itself calls `update_alleles()` and `get_sample()`. Thus, lines 91 and 95 should only be executed if selection is not active.

Within the `public` function `sample()` (lines 100–106), the value of the Boolean variable `fixtest` is derived from the call to `update_alleles()` (line 101). If `fixtest` is found to be `false`, meaning none of the adaptive alleles remain in the population, `sample()` is prematurely terminated (line 102) as it takes time to generate the sample and the simulation will start over anyway. If neither of the conditions of line 102 are met, a sample is generated as before and the value of `fixtest` is returned to its calling function, `reproduce_and_migrate()` in the class `Metapopulation`.

The remainder of the changes to `population.h` are to the constructor. The `if` block on lines 114–143 is used if any form of natural selection is called for by the parameters, i.e., one of the conditionals on line 114 evaluates `true`. Depending on the type of selection, the relevant Boolean variable `nfdselection` (line 116), `negselection` (line 118), or `activeselection` (line 127) is set to `true`. Note that in the case of NFD, *both* `nfdselection` and `activeselection` are ultimately set to `true`. The initial declarations of these `private` variables with trailing brackets (line 4) ensure these variables are set to `false` by default. We will discuss lines 119–125 in Sect. 8.3, where we discuss purifying selection.

Lines 129–142 are invoked when dealing with positive selection or overdominance. First, `keypos` is set based on the `possel` parameter. Next, we determine whether positive selection acts on standing variation or a new, adaptive variant (lines 131–130). Then we read the selection parameters from `possel` and `nfdsel` (lines 135–139). In the case of positive selection, we also define a constant dominant or additive selective regime if selection coefficient `t == 0` (line 140) or overdominant selective regime otherwise (line 141).

A selection information file is created and opened on lines 145–148 if positive selection (NFD or otherwise) is simulated. A history of the adaptive allele frequency when simulating straight positive selection or one of the two alleles evolving under NFD is recorded in this file, as well as the relative fitness of each genotype—which is unchanging in cases other than NFD.

When MS is used to generate the starting sequences in the population (lines 150–190), a number of modifications are required for simulating different forms of natural selection. In the case of positive selection on a new variant, we create a new `Allele` object at this position (lines 152–153) because we know we will begin with *at least* one derived allele at that position. The `if` loop on lines 156–165 first reads the line of the MS output file that lists the decimal positions of each of the segregating sites. If simulating standing variation (i.e., the frequency of the adaptive allele is $> 1/2N_e$) *or* the position under consideration does not equal `keypos` we create a new object of class `Allele` at the current position (lines 161–162).

The `if` block on lines 166–189 interprets two consecutive lines/sequences of the MS output file to populate the `individuals` vector. For both sequences, we add the position of a derived allele to the sequence `vector s1` or `s2` as appropriate, *if* the position is not `keypos` (lines 170–171 and lines 178–179). In addition, when we simulate positive selection on standing variation, we need to calculate the frequencies of derived alleles at all segregating sites so that we can identify a site that closely matches the characteristics specified in the `sellocus` parameter. Thus, when simulating selection on standing variation, we increment the derived allele count of each segregating site when necessary (lines 172–173 and lines 180–181).

In the case of positive selection on a new variant, we add `keypos` to the first `s1` sequence built and switch `newvariant` to `false` (lines 183–187). The latter action has the effect of preventing the addition of `keypos` to any other sequences due to the `newvariant` condition on line 183; thus, only one sequence has a derived allele at the `keypos` initially. We also sort the derived allele positions in `s1` so that `keypos` is properly positioned (line 186). Failure to do so would make recombination ineffective.

Lines 192–216 comprise an `if` block that attempts to identify a segregating site from MS output that meets the starting constraints of a standing variant specified in parameter `sellocus`. If a site is found, `keypos` is set to the site and its position and starting frequency are printed to standard output (line 218). It is possible that a segregating site meeting the specified constraints does not exist, in which case the program terminates with an error message printed to the screen (line 221).

Finally, relative fitness values are calculated/updated if simulating NFD (lines 223–224) and the header line of the selection information file is printed (line 226).

individual.h // additions

```
 1  public:
 2  ...
 3  int get_genotype(int pos) { // returns number of ancestral alleles (i.e.,
        ↪ 0s)
 4    int geno = 2;
 5    if ( find( sequences[0].begin(), sequences[0].end(), pos) !=
          ↪ sequences[0].end() ) --geno;
 6    if ( find( sequences[1].begin(), sequences[1].end(), pos) !=
          ↪ sequences[1].end() ) --geno;
 7    return geno;
 8  }
 9
10  int get_negsel_genotype(vector<int>& sites) { // returns number of seqs
        ↪ with a deleterious allele
11    int geno = 0;
12    for (int i=0; i<2; ++i) {
13      vector<int> intersectionality;
14      set_intersection(sites.begin(), sites.end(), sequences[i].begin(),
            ↪ sequences[i].end(), back_inserter(intersectionality));
15      if (intersectionality.size() >0) ++geno;
16    }
17    return geno;
18  }
```

Two new `public` functions are added to `individual.h`, both of which return a relevant genotype in the qualitatively distinct cases of positive and purifying selection. Function `get_genotype()` (lines 3–8) returns the number of ancestral alleles present at the `pos` position passed to the function (line 3). The genotype begins with its maximum value of 2 (line 4). Then, both `sequences` of the `Individual` object are interrogated for the presence of the position, and, if found, the genotype value is decremented (lines 5–6). A value of 2 is assigned to a genotype with both ancestral, less-fit alleles because the relative fitness values calculated for each `Population` object index the most fit genotype as [0] and the least fit genotype as [2]. In the case of negative selection, the function `get_negsel_genotype()` (lines 10–18) returns the number of sequences with at least one deleterious, derived allele. Again this number may range from 0 to 2. Setting the genotype to zero initially (line 11), we use the standard library function `set_intersection()` in order to search to populate the `vector<int> intersectionality` with the intersection between the positions of the queried sequence and the positions of sites that are subject to purifying selection (line 14; see Sect. 8.3). If the size of `intersectionality` is greater than 0, the sequence contains at least one deleterious allele and the genotype is increased by one (line 15). Note that the `vector` of deleterious positions is a parameter of the function (line 10). We will return to the determination of these sites in Sect. 8.3.

8.2 Positive Selection

Positive natural selection—which acts to *increase* the frequency of a favored allele—has characteristic effects on linked, neutral genetic diversity. These effects become most evident when positive selection seizes upon a *new*, adaptive variant derived from mutation. Temporally, these *signatures of selection* tend to be most evident near the time of fixation of the adaptive allele. Characteristic signatures of selection include: (1) an excess of high-frequency, derived alleles; (2) numerous low-frequency alleles; (3) a general elimination of linked diversity, assessed by declines in summary statistics such as the number of unique haplotypes K, nucleotide diversity π, and the number of segregating sites S; and (4) perturbation of linkage disequilibrium among neutral variants on either side of the site targeted by natural selection. The general decline in linked variation is called a **selective sweep**, while the increase in the frequency of derived alleles linked to the adaptive variant is named **genetic hitchhiking**.

As linked genetic diversity declines in response to positive natural selection both S and π decline. As genetic diversity begins to recover via new mutation, however, increases in S and π occur at different rates. Each new variant captured in a sample of sequences adds one to S but very little to the value of π because this statistic is averaged over all pairs of sequences—most of which will not contain the sequence difference. We focus on the test statistic Tajima's D, which captures this disparity between S and π; as a rule of thumb, regions with a value less than -2 suggest the region contains or is closely linked to a variant targeted by selection.

8.2.1 Selection on a New Variant

Selection on a **new variant** produced by mutation is easier to detect than selection on a **standing variant**—i.e., an allele that is present as more than one copy when selection begins. An adaptive variant produced by new mutation necessarily arises on a single haplotypic background. This is important because as the adaptive variant rises in frequency, linked variants are also transmitted to the next generation; they hitchhike, thereby reducing linked genetic diversity and (Fig. 8.1a) and pushing the allele frequency spectrum off neutral equilibrium. Changes in the pattern of genetic variation are assessed using a variety of summary and test statistics, including the aforementioned K, π, S, and Tajima's D. Now consider how things change when a *previously*

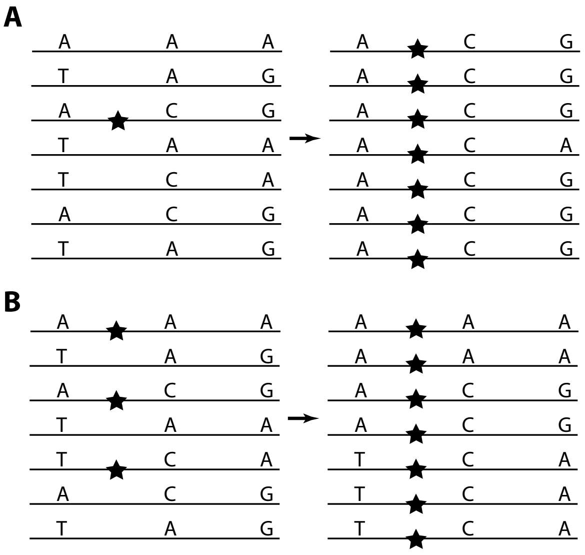


Fig. 8.1 Illustration of how positive selection on a new (**a**) or standing (**b**) variant affects linked polymorphism. (**a**) The adaptive variant (star symbol) arises from mutation on one specific haplotype, A-C-G. The right-facing arrow indicates the passage of time. Upon fixation, nearly all tightly linked polymorphism has been lost. In other words, a selective sweep has occurred. The one adenine at the third SNP is the result of crossing-over during the march to fixation of the adaptive variant. (**b**) Here, the adaptive variant existed as a neutral variant before it became a target of positive natural selection. As a result, it is found on multiple haplotypic backgrounds. As the adaptive variant rises to fixation, genetic hitchhiking of linked variants still occurs. However, which variants hitchhike depends on the chromosome—e.g., A-A-A, A-C-G, or T-C-A in this scenario. The end result of this starting difference is a less comprehensive selective sweep in which polymorphism is easily retained at linked sites

neutral, standing variant at a frequency of, say, 0.3 suddenly becomes a target of natural selection due to changes in the environment. The adaptive variant finds itself embedded within a variety of haplotypic backgrounds (Fig. 8.1b). This means that as adaptive variant frequency rises toward fixation there is not one specific set of neutral variants at linked polymorphic sites that hitchhike with the selected variant. In other words, the decline in diversity is not nearly as pronounced (Fig. 8.13b). In these cases, our summary and test statistics will have considerably less statistical power to detect the action of positive natural selection.

As an example of the effect on genetic polymorphism brought about by selection on a new variant, consider the results of a simulation with the following relevant parameter values:

Natural selection scenario

```
mutrate 1e-08
useRec 1
useHotRec 0
recrate 1e-08
...
seqlength 1e06
sampfreq 25
windowSize 10000
windowStep 5000
pop_num 1
runlength 10001
...
DEME /// 0
popsize 10000
useMS 1
mscommand ./ms 20000 -t 400 -r 400 1000000 >ms_output
...
sellocus 500000 1 0. 0. 1
possel 0.05 0. 0.5
```

In summary, we simulate a 1Mbp sequence where mutation and recombination rates both equal 1×10^{-8} in a single population of size $N_e = 10{,}000$. The new, adaptive variant at base pair 500,000 evolves under a selective regime in which $s = 0.05$ and $h = 0.5$. Figure 8.2, which was created using the R function `longitudinal_heatmap()` introduced in section X.X.X, shows the evolution of the summary statistic π. A clear reduction in π is observed in and around the selected site beginning shortly before fixation of the adaptive allele. This "footprint" of selection gradually narrows for thousands of generations until near-equilibrium values of π are regained. Note that the 10,000bp samples here have an expected value of $\pi = 4N_e\mu \times \texttt{seqlength} = 4\times10^4\times10^{-8}\times10^4 = 4$.

We next introduce a new visualization function in the R file `heatmaps.r`. Function `longitudinal_discrete_heatmap()` allows discretization of continuous summary statistic values into bins whose bracketing values are defined by the user.

heatmaps.r // add function longitudinal_discrete_heatmap()

```
longitudinal_discrete_heatmap <- function(inputfile, stat="K",
    ↪ brks=vector(), timeflow="down", timelim=vector(), fix = 0) {
  dat <- read.table(file = inputfile, header = T);
  ddat <- split(dat, dat$stat);
  size <- dim(ddat[[stat]]);
  q <- ddat[[stat]][,c(1,3:size[2])];
  m <- melt(q, id.vars=c("gen"));
  m$value <- cut(m$value, breaks = c(-Inf, brks, Inf));
  cols <- gray.colors(length(brks)+1, start = 0., end = 1., gamma = 2)

  d <- ggplot(m, aes(x=variable, y=gen, fill=value)) + geom_tile();

  if (length(timelim) != 0) {
    if (timeflow == "down") {
```

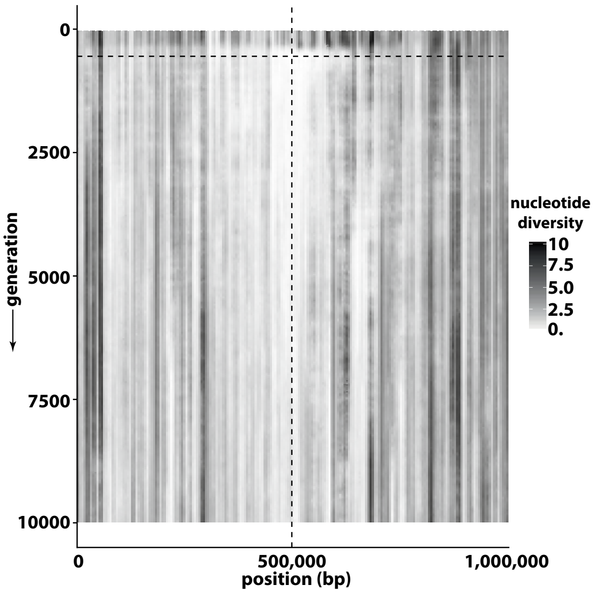


Fig. 8.2 Continuous heat map of nucleotide diversity (π) plotted using the R function `longitudinal_heatmap()` detailed in section X.X and argument `lowcol=1`. The simulation visualized by π was run with $s = 0.05$, $h = 0.5$, $N_e = 10{,}000$, and a single favored variant at generation 0. Dashed horizontal line shows the approximate generation of fixation of the favored variant, while the dashed vertical line shows the position of the favored variant. Values were sampled every 25 generations, with sample size $n = 500$

```
            d <- d + ylim(timelim[2], timelim[1]);
        } else {
            d <- d + ylim(timelim[1], timelim[2]);
        }
    } else {
        if (timeflow == "down") {
            d <- d + ylim(max(m$gen), min(m$gen));
        }
    }

    d <- d + scale_fill_manual(values = cols);

    if (fix != 0) {
        d <- d + geom_hline(yintercept=fix, linetype=2);
    }
    d;
}
```

The arguments `inputfile`, `stat`, `timeflow`, `timelim`, and `fix` all have the same meaning as in the non-discrete version of this function. A majority of the code is in fact similar or identical to the non-discrete function. The key difference is that we `cut` data by the points specified by the function parameter `brks` (lines 1 and 7) and, dependent on the number of resulting bins, assign a grayscale color to each bin in which the lowest bin is associated with black and the highest bin is associated with white (line 8). Note the polarity of this colorization scheme. For example, if you focus on the number of segregating sites S in each window and specify `brks = c(1,3,5,7)`, six bins will be created descending in associated color from black to white along the grayscale: $(-\infty,1]$, $(1,3]$, $(3,5]$, $(5,7]$, and $(7,+\infty)$. Of course, we are dealing with a finite summary statistic in S whose lower bound is zero. Therefore, the first interval is better thought of as $[0,1]$ and the last interval as > 7.

Figure 8.3a shows a low resolution discretization of K for the natural selection scenario described above created using `brks = c(10, 20, 30, 40)`, while Fig. 8.3b shows a higher resolution discretization of K values using `brks=c(5, 10, 15, 20, 25, 30, 35, 40)`. The higher resolution plot (Fig. 8.3b) makes it more clear that K declines in windows in and around the selected site at and following the time it fixes. Furthermore, it is more evident in Figs. 8.15b than a that an initial broad reduction in K gradually narrows in scope after fixation. This example shows that some experimentation is required with any output data in order to find the most helpful values of `brks`.

Next, we focus on the test statistic Tajima's D, which is sensitive to changes in the allele frequency spectrum caused by natural selection and/or other evolutionary factors such as demographic change. Remember the rule of thumb that values of Tajima's D less than -2 may indicate positive natural selection. We first consider two scenarios that are identical to the scenario listed above with the exception that selective strength is varied. Namely, the simulation whose results are shown in Fig. 8.4a was run with the parameter `possel 0.01 0. 0.5` ($s = 0.01$) while the simulation on which Fig. 8.4b is based was run with the parameter `possel 0.05 0. 0.5` ($s = 0.05$). Because $h = 0.5$, both are simulations of an additive selective regime. Comparison of Fig. 8.4a and b clearly demonstrates the positive correlation between the strength of selection as measured by s and the selective footprint—the width of sequence affected by the action of natural selection. Not surprisingly this comparison also shows that fixation occurs more quickly when the strength of selection is greater. Finally, and perhaps most interestingly, note that in the case of $s = 0.01$, values of Tajima's D in some windows adjacent to the selected site are less than -2 hundreds of generations before the final fixation of the advantageous allele (Fig. 8.4a). This is likely due to the fact that by the point Tajima's D begins to crash the frequency of the adaptive allele is already close to fixation and has therefore eliminated most variation in the immediate vicinity. Because s is relatively small, it then takes a rather long time to fix the adaptive allele after which significant values of Tajima's D begin to erode.

Results for a dominant selective regime ($h = 0$) are shown in Fig. 8.5. Results shown in Fig. 8.5a are from a simulation with `possel 0.01 0. 0.` while those in Fig. 8.5b are from a simulation with `possel 0.05 0. 0.`. In

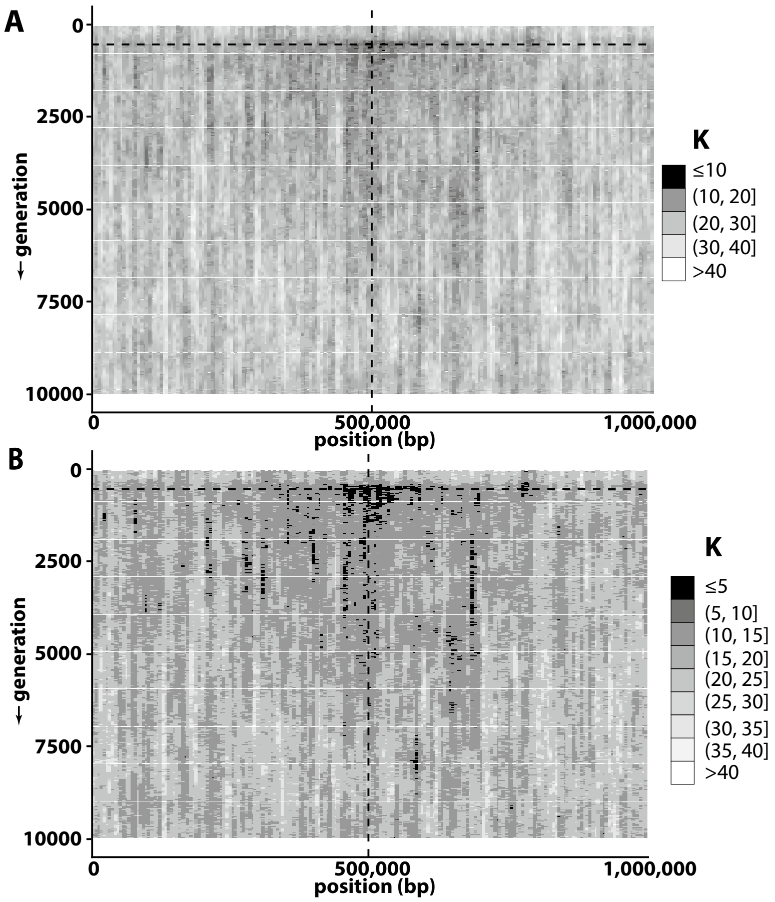


Fig. 8.3 The number of unique haplotypes K by window. Simulation and sampling conditions identical to those in Fig. 8.2. The difference between panels (**a**) and (**b**) is the number and value of breaks passed to parameter `brks` in function `longitudinal_discrete_heatmap()`. The vertical dashed line marks the position of the adaptive allele while the horizontal dashed line marks the time of fixation

other words, these two simulations of a dominant selective regime differ only in the magnitude of selective strength, s.

Under the dominant selective regime, we still see a positive correlation between s and fixation time as well as between s and the width of the selective footprint. In the dominant case, however, the decrease in Tajima's D to significant levels (less than -2) occurs long before fixation regardless of selective strength. In fact, we see a near-immediate decline in Tajima's D when $s = 0.05$. This can be explained by the dynamics of the adaptive allele frequency under a dominant selective regime. Recall that the frequency of an adaptive allele rises rapidly to near fixation under a dominant selective

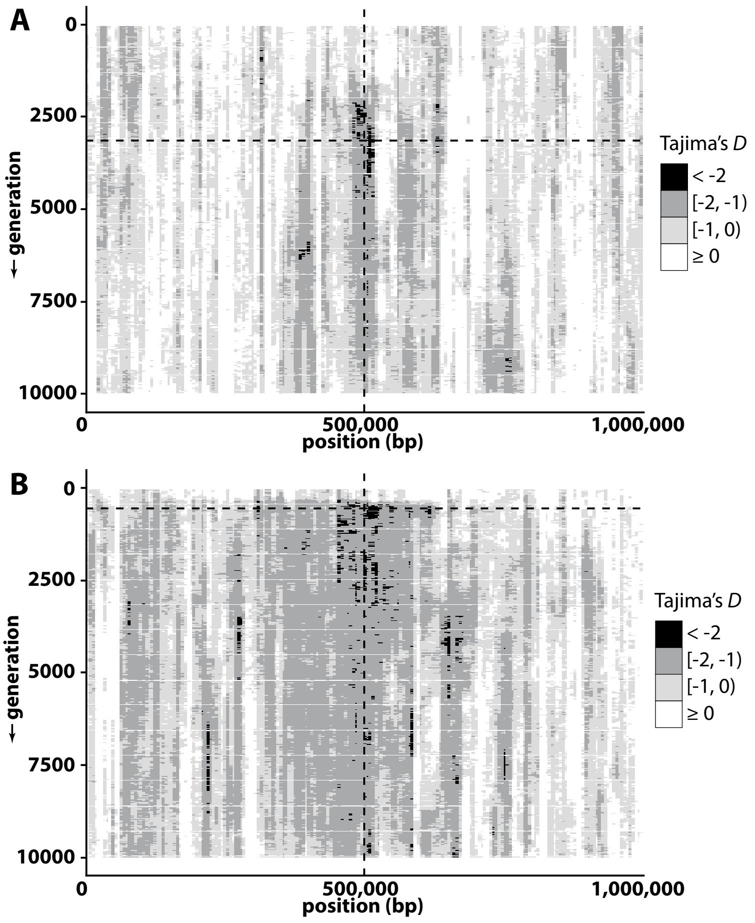
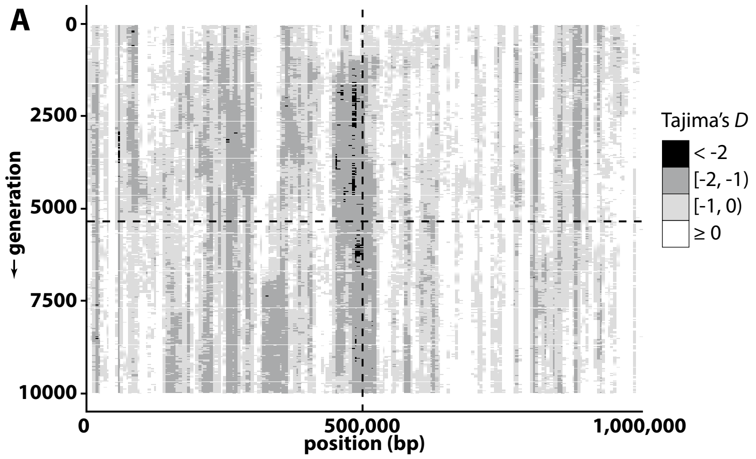


Fig. 8.4 Discrete, longitudinal heat maps of Tajima's *D* for two distinct simulations of positive selection for a single advantageous allele embedded within a 1Mbp sequence under an **additive** selective regime. **In both cases** $h = 0.5$**.** (**a**) $s = 0.01$. (**b**) $s = 0.05$. Horizontal, dashed lines indicate the approximate generation when the advantageous alleles fixed, while vertical, dashed lines indicate the position of the advantageous variant

regime, *but* true fixation of the adaptive allele may take a very long period of additional time to achieve because heterozygotes bearing a copy of the less-fit allele have relative fitness equal to that of individuals homozygous for the adaptive allele. In short, most of the selective sweep is accomplished quickly, driving Tajima's *D* downward, where it remains for some time following true fixation of the adaptive allele.

Although longitudinal heat maps, continuous or discrete, are quite helpful to obtain a broad view of the evolution of a sequence, we often want to plot the values of a statistic across the simulated sequence at a specific time



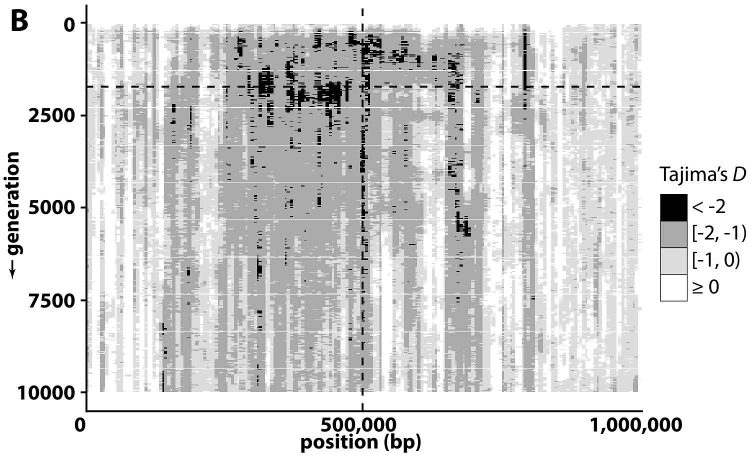


Fig. 8.5 Discrete, longitudinal heat maps of Tajima's *D* for two distinct simulations of positive selection for a single advantageous allele embedded within a 1Mbp sequence following a **dominant** selective regime. **In both cases** $h = 0$. (**a**) $s = 0.01$. (**b**) $s = 0.05$. Horizontal, dashed lines indicate the approximate generation when the advantageous alleles fixed, while vertical, dashed lines indicate the position of the advantageous variant

point. I now introduce the R function `stat_at_timepoint()` to facilitate this need. Even though this function does not produce a heat map, I add it to `heatmaps.r` for convenience.

heatmaps.r // add function stat_at_timepoint()

```
stat_at_timepoint <- function(inputfile, stat="tajD",
    ↪ windows=seq(5000,995000,5000), timepoint, mvgavg=0)
{
  d <- read.table(file = inputfile, header = T);
  dd <- split(d, d$stat);
```

```
   size <- dim(dd[[stat]]);
   q <- dd[[stat]];
   qq <- q[q$gen==timepoint, 3:size[2]];
   plot(windows, qq, type = "b", lwd=0.25);

   if (mvgavg >0 & (mvgavg %% 2 == 1) ) {
      pos<-vector();
      mavg<-vector();
      abvbel = floor(mvgavg/2);
      for(i in (abvbel+1):(size[2]-2-abvbel)) {
         avg<-0;
         pos<-c(pos, windows[i]);
         for(j in (i-abvbel):(i+abvbel)) {
            avg <- avg+qq[j];
         }
         mavg<-c(mavg, avg/mvgavg);
      }
      lines(pos, mavg, col = "black", lwd =2);
   }

   abline(h=0, lty = 1);
   abline(h=-2, lty = 2);
   abline(v=500000, lty = 3);
}
```

This function plots a given statistic (Tajima's *D*, by default) from a FORTUNA summary statistic output file (e.g., `sumstats0`). It is required that you change the value of the `windows` parameter if the window configuration sampled differs from the default. For example, if you simulated a 100,000bp sequence and sampled 10kbp windows with no overlap, you would enter `windows=seq(10000, 90000, 10000)`. You must also provide a value for the argument `timepoint`, which is the generation number of the sample to be plotted. By default, argument `mvgavg` is set to zero, in which case only the statistic values of each window will be plotted. If `mvgavg` is set to a positive and odd integer, however, a solid line will be superimposed on the raw data. For example, if `mvgavg` is set to 5, the value of the superimposed line will be the average value of the current window and the two windows before and after. Plotting this moving average may make it easier to spot trends in the data.

To display the use of the function, we graph Tajima's *D* at four time points for a representative simulation in which all parameters are the same as before except for the `possel` parameter. Figure 8.6 shows the plot of Tajima's *D* from a simulation with parameter `possel 0.01 0. 0.5`, while Fig. 8.7 shows the plot of Tajima's *D* from a simulation with parameter `possel 0.10 0. 0.5`. In both cases, the time points considered are (1) 25 generations after mutation to the adaptive allele, (2) 100 generations before fixation of the adaptive allele, (3) at fixation, and (4) 1000 generations post-fixation. At generation 25 of both simulations, we observe a neutral pattern of Tajima's *D*, in which values oscillate in, to all appearances, a random manner. At

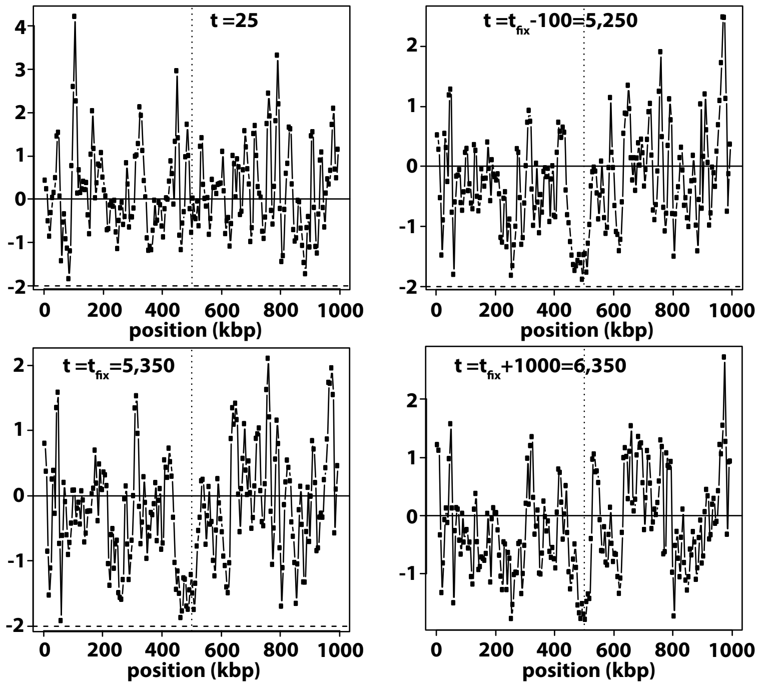


Fig. 8.6 Values of Tajima's *D* at four different time points for a simulation of positive selection on a single, new variant at position 500kbp where $h = 0.5$ and $\mathbf{s} = \mathbf{0.01}$. The time points shown are (upper, left) the first sample at $t = 25$, (upper, right) 100 generations before fixation at $t = 5250$, (lower, left) fixation at $t = 5350$, and (lower, right) 1000 generations post-fixation at $t = 6350$. **Note that the scale of the y-axis changes between panels**

all other time points, however, a distinct trough in the value of Tajima's *D* is observed adjacent to the selected site at 500kbp; by trough, I mean that multiple contiguous windows show a depressed level of Tajima's *D* that approaches or is less than −2. As expected, weaker selection produces a narrower trough (Fig. 8.6, where $s = 0.01$) compared to simulation of ten-fold stronger selection ($s0.10$) on which Fig. 8.7 is based. Another detail to note in these plots is that each trough appears to be centered to the side of the actual target of selection (upstream of 500kbp in both cases illustrated). Simulation studies of selective sweeps repeatedly demonstrate this off-center signal of selective sweeps (Kim and Nielsen 2004). Similar skewing is observed in the longitudinal plots (e.g., Fig. 8.5). However, the main guidance offered by these results is that a lone significant value of Tajima's *D* or other test statistic should not be taken as evidence of natural selection or other population genetic processes. Rather, we expect a sustained trend in the test statistic across multiple windows.

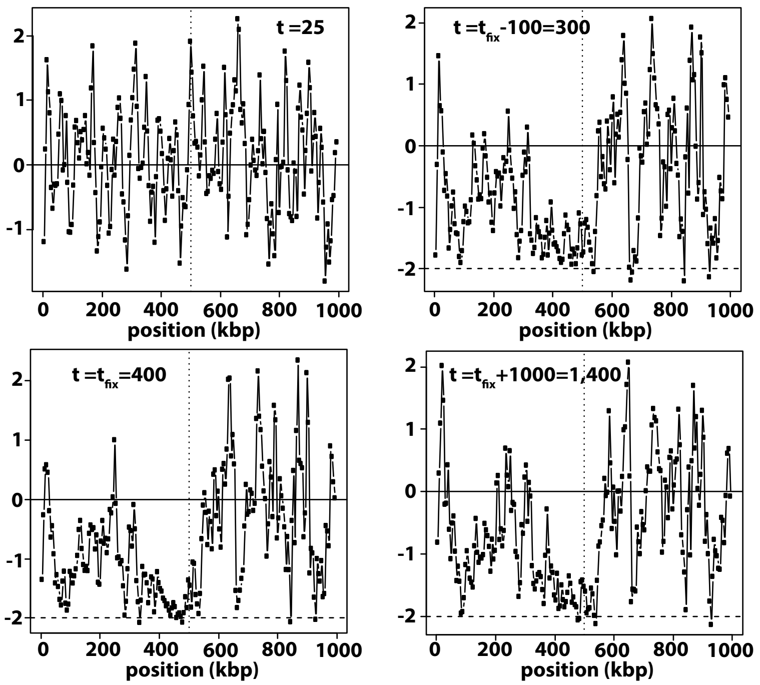


Fig. 8.7 Values of Tajima's D at four different timepoints for a simulation of positive selection on a single, new variant at position 500kbp where $h = 0.5$ and $\mathbf{s} = \mathbf{0.10}$. The time points shown are (upper, left) the first sample at $t = 25$, (upper, right) 100 generations before fixation at $t = 300$, (lower, left) fixation at $t = 400$, and (lower, right) 1000 generations post-fixation at $t = 1400$. **Note that the scale of the y-axis changes between panels**

8.2.2 *Selection on Standing Variation*

As suggested by the conceptual illustration in Fig. 8.1, positive selection on a standing variant is not expected to eliminate linked variation with the same efficiency as selection on an adaptive variant produced by new mutation. We can model positive selection on a standing variant by altering the `sellocus` parameter in any or all deme blocks of the `parameters` file. The results of these simulations are shown in Fig. 8.8; in all cases parameter `possel 0.05 0. 0.5`. The differences between the simulations whose results are shown in Fig. 8.8 were as follows:

- `sellocus 500000 1 0. 0. 1` // Fig. 8.8a
- `sellocus 500000 1000 0.01 0.01 1` // Fig. 8.8b
- `sellocus 500000 10000 0.01 0.01 1` // Fig. 8.8c

With the exception of `possel` and `sellocus` all parameter values were the same as in natural selection scenario listing in Sect. 8.2.1. Recall that the first two entries to the `sellocus` parameter are (1) position of the selected variant and (2) starting number of derived (in this case adaptive) alleles. Because the initial genetic variation is drawn from an MS simulation, there is almost no chance one of the neutrally evolved SNPs will happen to be at the desired position with the desired count of derived alleles. The third and fourth entries of `sellocus` help us identify a random SNP from the MS output that is *close* to the first two parameter values; in particular, they allow us to control what counts as close enough. For example, `sellocus 500000 1000 0.01 0.01 1` asks the program to find a SNP from the MS output that (1) is within 0.01×1Mbp = 10,000bp on either side of site 500,000 *and* (2) shows a derived allele frequency within 0.01 of 1000/20,000 = 0.05—i.e., between 0.04 and 0.06. Recall that the final value of `sellocus` takes either a 0 or 1; 1 specifies that the program will start over if the derived allele is lost from the deme—a nearly impossible event when we begin with 1000 or 10,000 adaptive alleles in the deme.

Figure 8.8a shows selection on a new variant and is included for the sake of comparison. Note that selection for a *standing* variant beginning at a frequency of ~ 0.05 (Fig. 8.8b) still lowers the values of Tajima's D. However, in comparison to Fig. 8.8a, the signal is much weaker. When the standing variant begins at a frequency of ~ 0.5, the signal of selection is nearly non-existent. We would be hard pressed to flag this region as a putative target of selection if Fig. 8.8c was based on an empirical data set.

8.2.3 *No Recombination*

Crossing-over during the course of a selective sweep has the effect of separating the adaptive allele from initially linked variants over time. Because recombination rate increases with the physical distance between the adaptive and linked alleles, it is the more distantly linked variants that are first separated from the adaptive allele. Crossing-over therefore explains why the trough of Tajima's D observed in Figs. 8.6 and 8.7 is *localized* to the immediate vicinity of the adaptive allele. It also helps explain why the width of this trough is positively correlated with selective strength. Stronger selection fixes the adaptive allele more rapidly, thus giving crossing-over less time to separate adaptive and linked variants. Think of the adaptive allele as the bottom of a river valley and the strength of selection as the magnitude of kinetic energy exerted by the river's flow. A mighty river carves a wider valley into the earth.

There are, of course, sequences that experience little or no crossing-over. For example, (1) the majority of the Y- and W-chromosome sequences in X-Y and Z-W sex determination systems are subject to minimal crossing-over, and (2) maternally inherited mitochondrial DNAs and chloroplast DNAs are only rarely subject to crossing-over. Therefore, we next consider a simulation of natural selection with no recombination. This serves two purposes, the

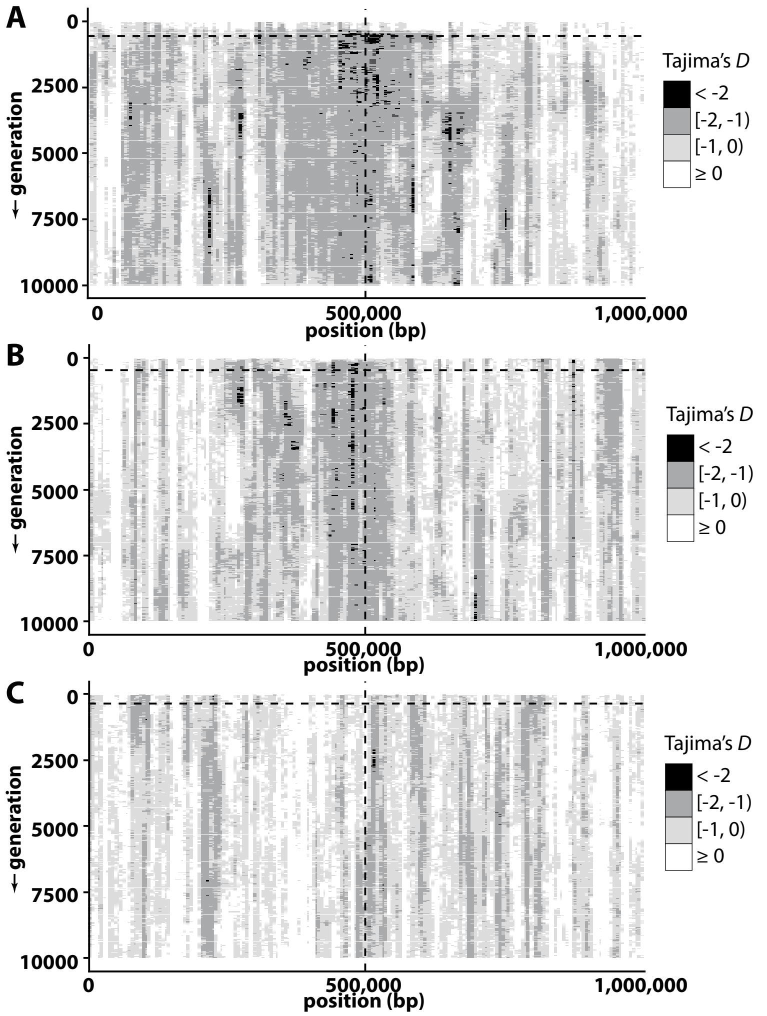


Fig. 8.8 Weakening of the selective footprint associated with selection on standing variants. The three panels show the distribution of Tajima's *D* for three separate simulations; in all cases, $s = 0.05$, $h = 0.5$, and $N_e = 10{,}000$. The number of favored variants in the population differs between the three simulations: (**a**) new variant, $p = 1/20{,}000 = 0.00005$; (**b**) standing variant when selection begins at frequency $p = 1000/20{,}000 = 0.05$, and (**c**) standing variant when selection begins at frequency $p = 10{,}000/20{,}000 = 0.5$

first of which is technical. We hope to see that our simulation program is properly integrating the effects of selection, recombination, and mutation. In the absence of recombination, we expect a *global* rather than *local* selective footprint along the simulated sequence, as crossing-over is not available to break up the association of the adaptive allele to variants linked to it at the onset of selection. If we turn off recombination, by setting program parameter `useRec` to 0 and find the same patterns of natural selection as seen earlier, we would know that our code has a problem. The second purpose for simulating positive selection with no recombination is to mimic the evolution of non-recombinant DNA.

Figure 8.9a shows the discrete longitudinal distribution of Tajima's D based on simulation of positive selection on a new variant in the absence of recombination. Parameter values are as before with the following exceptions:

- `useRec 0`
- `possel 0.1 0. 0.5`

Given that $s = 0.1$, this is rather strong selective pressure and because $h = 0.5$ (an additive selective regime), we expect rapid fixation of the adaptive allele. As expected in the absence of recombination, the entire length of the sequence shows a marked decrease in Tajima's D and recovery following fixation of the adaptive allele—morphing back to lighter shades of gray indicative of greater values of Tajima's D—occurs relatively evenly across the sequence.

Simply "turning off" recombination in our simulation does not faithfully represent the evolution of Y-DNA or mtDNA. What real, biological elements of uniparental inheritance are missing when we just set `useRec` to 0? Can we incorporate these elements into our simulation by changing `popsize`? After all, Y-DNA has 1/4 the effective population size of an autosome in a diploid species. Or, do we again need to consider explicitly modeling the sex of each `individual` in simulated populations?

8.2.4 *Overdominance*

At an overdominant locus, the heterozygous genotype is of higher fitness than either of the homozygous genotypes. We now consider whether there are any characteristic patterns of genetic variation associated with overdominance. Theory suggests that selection at an overdominant locus does affect levels of diversity at linked, neutral sites. Specifically, we expect overdominant selection to *increase* measures of diversity such as π and to bring about

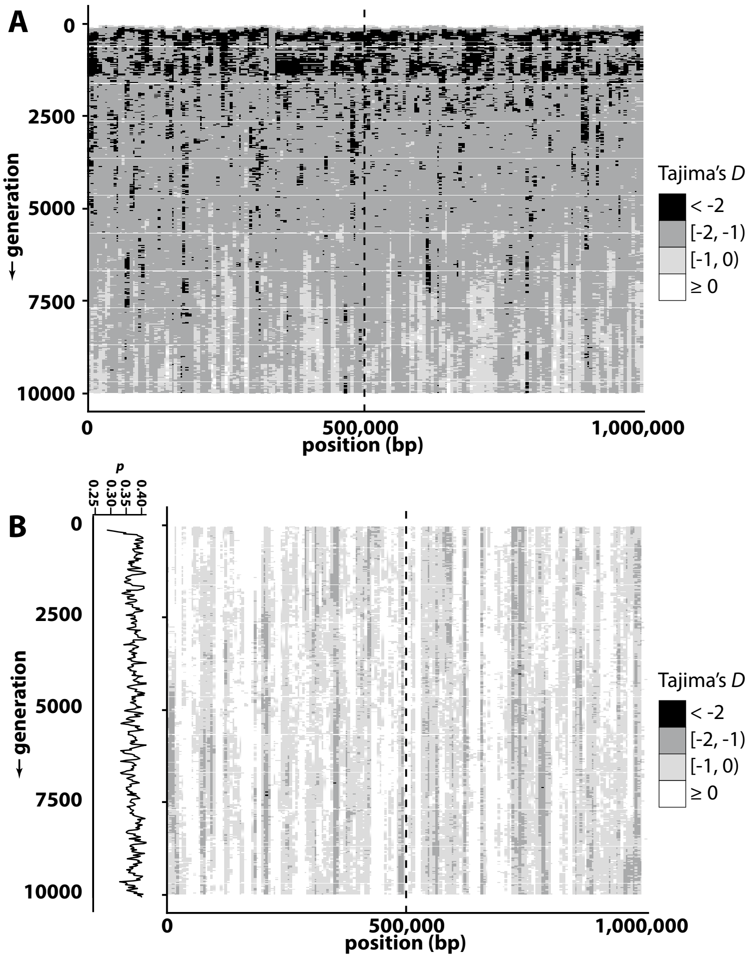


Fig. 8.9 (**a**) Across a simulated 1Mbp non-recombinant sequence, the selective footprint looks similar at all points. Here we simulate strong selection with $s = 0.1$, $h = 0.5$, and $N_e = 10{,}000$. (**b**) Simulation of overdominant selection with $N_e = 10{,}000$, $s = 0.05$, and $t = 0.03$. The latter two parameters define a selective regime of $w_{A/A} = 0.95$, $w_{A/a} = 1.0$, and $w_{a/a} = 0.97$. The simulation began with $f_A = p = 0.25$ and the first sample was taken 25 generations after the beginning of selection. The rotated graph on the left-hand side of the panel tracks the frequency of the A allele, p, while the longitudinal heat map graphs values of Tajima's D

a pattern called **associative overdominance** (Ohta and Kimura 1970): neutral polymorphisms tightly linked to the targeted locus will also show high levels of observed heterozygosity.

Both of these patterns—increased overall diversity and associative overdominance—are difficult to discern as anomalous in scans of empirical data. The utility of these signals to identify overdominant loci under selection is therefore questionable. Nevertheless, one motivation for simulation is build intuition and assess whether our preconceptions are corroborated. To simulate selection at an overdominant locus, the second entry of the `possel` parameter is set to a non-zero value. Figure 8.9b is based on the results of a simulation of overdominant selection where the *two* selective coefficients $s = 0.05$ and $t = 0.03$ were specified using parameter `possel 0.05 0.03 0.`. This represents the selective regime

- w_A/A = $1 - s$ (homozygous derived allele)
- w_A/a = 1
- w_a/a = $1 - t$ (homozygous ancestral allele),

In addition, we set the starting frequency of the derived allele (A, as represented above) to 0.25 by setting `popsize` to `10000` and the second entry of parameter `sellocus` (the initial count of the derived allele) to `2500`. Overdominant selection produces no discernible anomaly in the pattern of Tajima's D in the proximity of the locus under balancing selection (Fig. 8.9b).

To find positive if ephemeral evidence of associative overdominance, we monitor observed heterozygosity, H_{obs}—the observed fraction of individuals with a heterozygous genotype—at each polymorphic site in the simulated sequence. In addition, we monitor expected heterozygosity H_{exp} for comparison; expected heterozygosity is calculated using the Hardy-Weinberg expectation of heterozygote frequency, $2pq$, where p and q are the frequencies of the balanced alleles. I leave it to the reader to implement calculation of these summary statistics.

Using the parameter setting of $s = t = 0.05$ (`possel 0.05 0.05 0.`) and focusing on the time point 1000 generations after the onset of balancing selection, we find evidence of both associative overdominance and increased diversity at and near the targeted locus. First, a number of loci tightly linked to the overdominant target of selection show observed heterozygosity (H_{obs}) > 0.5. Given that *expected* heterozygosity (H_{exp} at a diallelic locus is maximized at 0.5 when $p = q$, this result is by definition unexpected and indicative of associative overdominance (Fig. 8.10a).

Because H_{exp} is a function of current allele frequencies, it is instructive to look at the difference $H_{obs} - H_{exp}$ rather than simply looking for a signal of maximized H_{obs}. For the same simulation, we see a peak in $H_{obs} - H_{exp}$ directly adjacent to the site targeted by selection that rapidly falls to negative values in both the downstream and upstream directions (Fig. 8.10b). In other words, there is a narrow region surrounding the targeted site that shows higher-than-expected genetic diversity as measured by heterozygosity. I reiterate

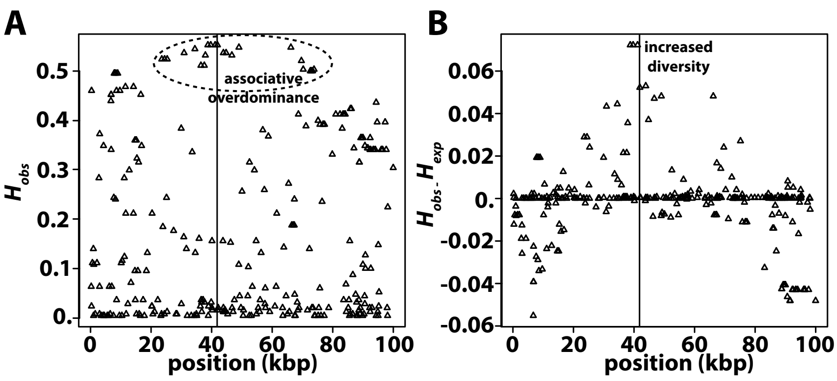


Fig. 8.10 Associative overdominance and increased genetic diversity resulting from overdominant selection. In both cases $s = t = 0.05$, $N_e = 10{,}000$, the initial frequency of the derived allele was 0.25. Each triangle represents a SNP in a sample of size $n = 500$ *at 1000 generations following the onset of selection*. Vertical lines indicate the position of the overdominant locus at base pair 41,820. (**a**) A large number of loci linked to the selected locus show $H_{obs} \geq 0.5$. In other words linked loci "act" as overdominant loci despite their neutrality. (**b**) Increased diversity at and surrounding the overdominant locus as measured by $H_{obs} - H_{exp}$. Positive values of this metric are indicative of greater diversity than expected based on allele frequencies

that these patterns are often transient and therefore not particularly useful in the context of inference. Still, exploring parameter space via simulation may allow us to identify conditions in which associative overdominance is more sustainable.

Identify a means for calculating H_{obs} and H_{exp} either through post-simulation calculation or by modifying FORTUNA code to calculate and record these quantities during a simulation. Next, explore the dynamics of associative overdominance. How does recombination rate affect maintenance of this signal? I referred to associative overdominance as a transient pattern. Do simulation results confirm this?

8.2.5 Negative Frequency-Dependent Selection

In Sect. 7.1.2, we examined the dynamics of allele frequencies at a site subject to negative frequency-dependent selection (NFD). We now focus on an interesting case of NFD examined by Rice (2004); in addition to documenting the dynamics of the targeted site, we investigate whether or not this form of NFD has an effect on linked diversity.

We begin by explicating the relevant code in the `private` function `update_selected_freqAndFit()` found on lines 10–40 of the updated `population.h` file (Sect. 8.1). This function is required to accurately simulate NFD selection because fitness of each genotype is dependent on current allele frequencies. The variable `count` holds the total number of derived alleles in the deme, which we again label A, while the `vector<double> genotype_freqs` holds the frequencies of the genotypes A/A, A/a, and a/a, respectively. We begin by defining `count` as the total number of alleles in the deme (lines 11); we will decrement this count as *ancestral* alleles are encountered. The genotype frequencies are all defined as 0 (line 12) initially. The `for` loop on lines 13–17 iterates through the `individuals` of the deme and uses the number of ancestral alleles returned by function `get_genotype()` to modify both `count` and `genotype_freqs` (lines 15 and 16, respectively). The frequencies p and q are then calculated (lines 18–19) as are updated `genotype_freqs` (lines 20–21). After printing these frequencies to the selection information file (line 22) a block on lines 24–38 is encountered. This is only executed if `nfdselection` is true, which will be the case if the first entry of parameter `nfdsel` of the `parameters` file is **not** equal to zero.

The two entries to parameter `nfdsel` are read in as `nfd_s` and `nfd_h`, respectively. These coefficients are used to calculate the fitness of each genotype under NFD in one of two ways, either using (1) the commented-out lines 25–27, which use *expected* genotype frequencies or (2) lines 29–37, which use *empirical* genotype frequencies. To implement the first means, uncomment the relevant lines, comment out lines 29–37, and recompile FORTUNA. Here, however, we focus on the second, more complex method in which relative fitness values are functions of updated genotype frequencies. Lines 29, 31, 33, and 35 are used to find the current maximum of relative fitness among the genotypes (variable `max_fit`), which is needed to normalize the fitness values (lines 36–37). Finally, relative fitness values are printed to the selection information file (line 39).

The NFD selective regime encoded in lines 29–35 of the `population.h` listing from Sect. 8.1 follows an example of NFD briefly discussed by Rice (2004) in which the relative fitness values are: $w_{A/A} = 1 - hf_{A/a} + hf_{a/a}$, $w_{A/a} = 1 - sf_{A/a}$, and $w_{a/a} = 1 - hf_{A/a} + hf_{A/A}$. The relative fitness of the heterozygote contains parameter s (`nfd_s`). The relative fitness of heterozygotes decreases with increasing s. Therefore the slope of $\frac{dp}{dt}$ vs. p increases with s around $p = q = 0.5$ and, as we will see, large values of s manifest as chaotic jumps in the value of p through time. The relative fitness values of the homozygotes contain the parameter h (`nfd_h`), which we set to 3 in both cases illustrated below.

Figure 8.11 shows results from a representative simulation of the NFD selective regime described in the previous paragraph, using the parameters `possel 500000 250 0.05 0.005 1` and `nfdsel 2 3` while all other parameters were set as listed in the baseline natural selection scenario (Sect. 8.2.1). We use the settings in `possel` to aim for a selected site near the middle of the 1Mbp simulated sequence with a standing variant frequency beginning on the domain $250/10{,}000 \pm 0.005 = [0.02, 0.03]$, where 10,000 is popsize. The values assigned to nfdsel correspond to $s = \mathtt{nfd}_\mathrm{s} = 2$, $h = \mathtt{nfd}_\mathrm{h} = 3$.

The vector field $\frac{dp}{dt}$ vs p (Fig. 8.11a) was generated with the following R code:

```
source("analyze_selective_regime.r");
d <- analyze_selective_regime(2, 3, 1-(h*2*p*q)+(h*q*q), 1-(s*2*p*q),
    ↪ 1-(h*2*p*q)+(h*p*p));
plot_vector_field(d);
```

Note that in the function analyze_selective_regime() we use Hardy-Weinberg expectations for the frequencies of the genotypes (line 2, arguments 3-5). The vector field in Fig. 8.11a is superimposed by actual output (dots) from the results of the simulation described in the last paragraph. Clearly, results from stochastic simulation hew to the expected dynamics of the system. In Fig. 8.11b, the next-generation frequency of the derived allele (p_{t+1}) is plotted against the current frequency of the derived allele (p_t), and each move from p_t to p_{t+1} is denoted by a connecting gray line. The chaotic single-generation changes to the value of p are evident in this graph. This example of NFD selection eventually results in large global losses of diversity across the 1Mbp sequence (Fig. 8.11c).

The panels of Fig. 8.12 show results for a system where the only parameter change relative to the simulation results summarized in Fig. 8.11 is to set s (nfd_s) to 1 rather than 2. Comparing panel A of Figs. 8.11 and 8.12, we find an evident reduction in magnitude of the slope in the region surrounding $p = 0.5$ that results from the lesser value of parameter s. We also see that observed p in the stochastic simulation is found in only two clusters at roughly $p = 0.35$ and $p = 0.65$ (Fig. 8.12a). The evolution of this system from a low starting frequency of p rapidly reaches a point where the derived allele frequency reliably ping-pongs between these two frequencies generation after generation (connecting gray lines in Fig. 8.12b). Rather interestingly, the effect on linked diversity is also quite different from the case of $s = 2$. In the windows immediately surrounding the site targeted by NFD, genetic variation begins to accumulate in a slowly widening region surrounding the polymorphism under selection while a tightly linked region of very low diversity emerges and then begins to dissipate (Fig. 8.12c).

Although it is questionable whether the extreme NFD selective regimes simulated here correspond to selection on any real phenotype in nature, these results are interesting and remind us that simulation allows us to test all sorts of possibilities that may or may not be found within the confines of the natural world. The very different effects of NFD selection on linked diversity when $s = 1$ and $s = 2$ are particularly interesting and worth taking a moment to consider and at least partially explain. We start by taking a closer look at the dynamics of both p and observed heterozygosity ($f_{A/a}$). Figure 8.13a and b show the generation-to-generation changes in p for $s = 2$

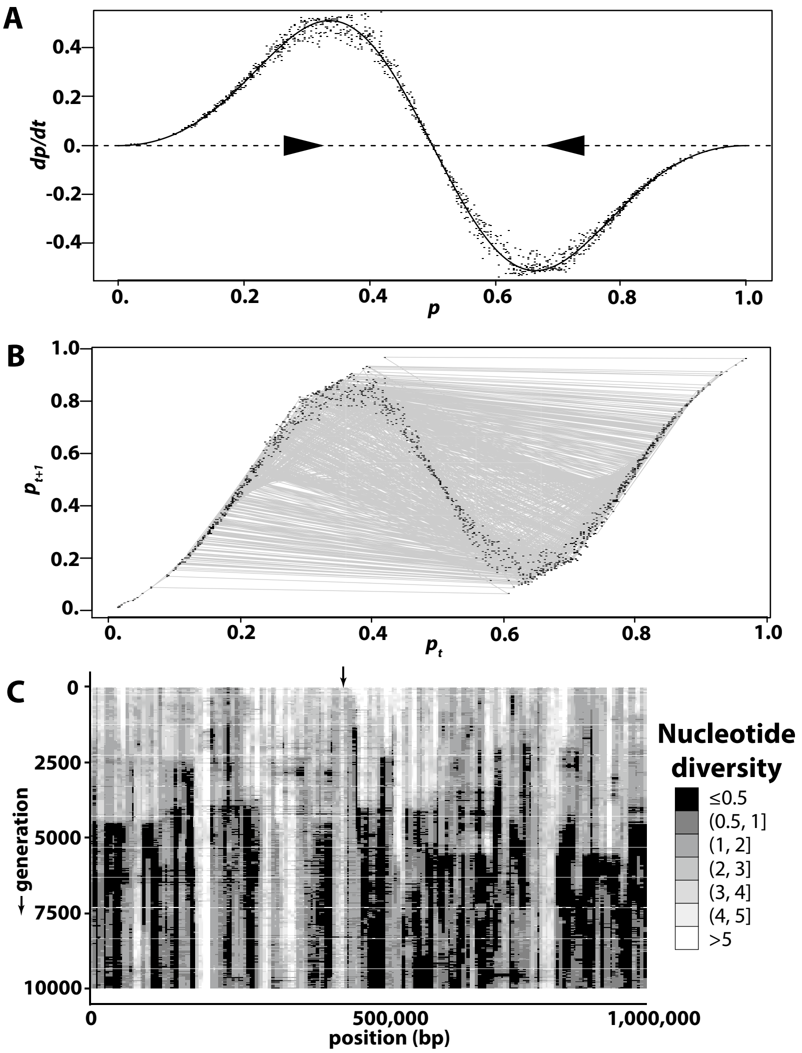


Fig. 8.11 The effects of negative frequency-dependent selection on selected and linked sites. Relative fitness of genotypes were simulated under the general model of $w_{A/A} = 1-3f_{A/a}+3f_{a/a}$, $w_{A/a} = 1-sf_{A/a}$, and $w_{a/a} = 1-3f_{A/a}+3f_{A/A}$ with $s = 2$ in the underlying simulation. (**a**) Vector field for the selected site, where $p = f_A$, the frequency of the derived allele. Dots correspond to simulated data. (**b**) The value of p in the next generation (p_{t+1}) vs. the current generation (p_t). Gray lines connect adjacent points in the time series, showing the chaotic changes in p. (**c**) Discrete heat map of nucleotide diversity (π). The site targeted by selection was found at the point indicated by the arrow (454,500bp), where p began at a frequency of 0.0218. The most striking pattern here is the steadfast depauperization of genetic diversity across the entire 1Mbp sequence. Sample size of $n = 500$

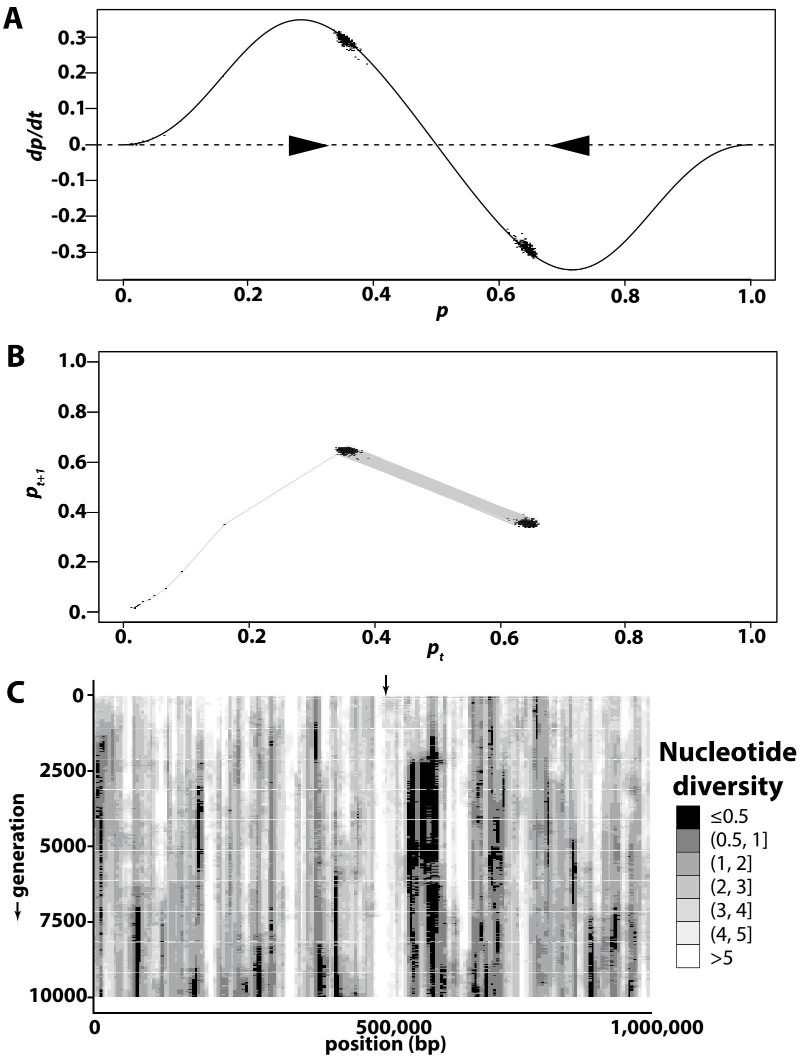


Fig. 8.12 The effects of negative frequency-dependent selection on selected and linked sites. Simulation parameters identical to the results documented in Fig. 8.11 with the exception that s (nfd_s is set to 1 rather than 2. (**a** and **b**) This change results in oscillation between $p \sim 0.35$ and $p \sim 0.65$. (**c**) A clear increase in π is observed in the region immediately surrounding the selected site, although flanked by an upstream region of substantially decreased nucleotide diversity. The site targeted by selection was found at the point indicated by the arrow (523,200bp), where p began at a frequency of 0.0203

and $s = 1$, respectively. Comparing these two panels, the first thing to note is that when $s = 1$ the changes from generation to generation are large but highly predictable; if p is currently ~ 0.35, it will swing upward to ~ 0.65 the next generation and vice-versa (Fig. 8.13b). On the other hand, the changes to p when $s = 2$ (Fig. 8.13a) are less predictable. There are still large swings between low and high values of p. In fact they are frequently of greater magnitude than those seen in the case of $s = 1$ (e.g., the change marked by the arrow in Fig. 8.13a). However, there are smaller jumps interspersed and no repeatable pattern is discernible.

This difference in the dynamics of p between the two selective regimes is dramatic enough for us to suspect it underlies the very different effects on linked diversity. Because heterozygosity—the fraction of individuals in a sample with a heterozygous genotype—is associated with greater genetic variation, in Fig. 8.13c we plot $f_{A/a}$ at the selected site for the two distinct NFD selective regimes. The comparison reveals a striking difference. While observed heterozygosity (i.e., $f_{A/a}$) is essentially static at ~ 0.45 when $s = 1$ (Fig. 8.13c, dashed line), it oscillates dramatically from generation to generation when $s = 2$ (Fig. 8.13c, solid line). This suggests that when $s = 1$ the generation-to-generation shifts in p are driven by the loss of the *homozygote* A/A or a/a. Conversely, when $s = 2$, declines in p are to a large degree driven by the loss of heterozygotes, as we might expect given that s is inversely proportional to the fitness of heterozygotes. Frequent loss of heterozygotes, linked to a larger variety of alleles at linked sites, eventually has the affect of removing linked variation (Fig. 8.11c). However, when $s = 1$, the great and steady retention of heterozygotes means that closely linked sites also retain variation—i.e., the widening region of increased nucleotide diversity seen in Fig. 8.12c reflects associative overdominance. Note that the increase in nucleotide diversity is localized. Conversely, the *decline* in nucleotide diversity when $s = 2$ is global in effect. I suggest this is because the frequent removal of variation across the sequence enacted by the removal of heterozygotes at the selected site outpaces the ability of mutation to replace the lost variation.

8.3 Purifying Natural Selection and Background Selection

Most new mutations are neutral or deleterious, rather than adaptive. Selection *against* deleterious alleles is referred to as negative or **purifying natural selection**. A relatable if extreme example of a deleterious variant subject to purifying selection is an allele that causes a disease that is invariably terminal. Returning to the basic equations of natural selection from Sect. 7.1.2, a lethal allele such as this is modeled by a selection coefficient of $s = 1$. Then, $w_{d/d} = 1 - s = 0$, which reflects the absence of offspring from individuals with the d/d genotype.

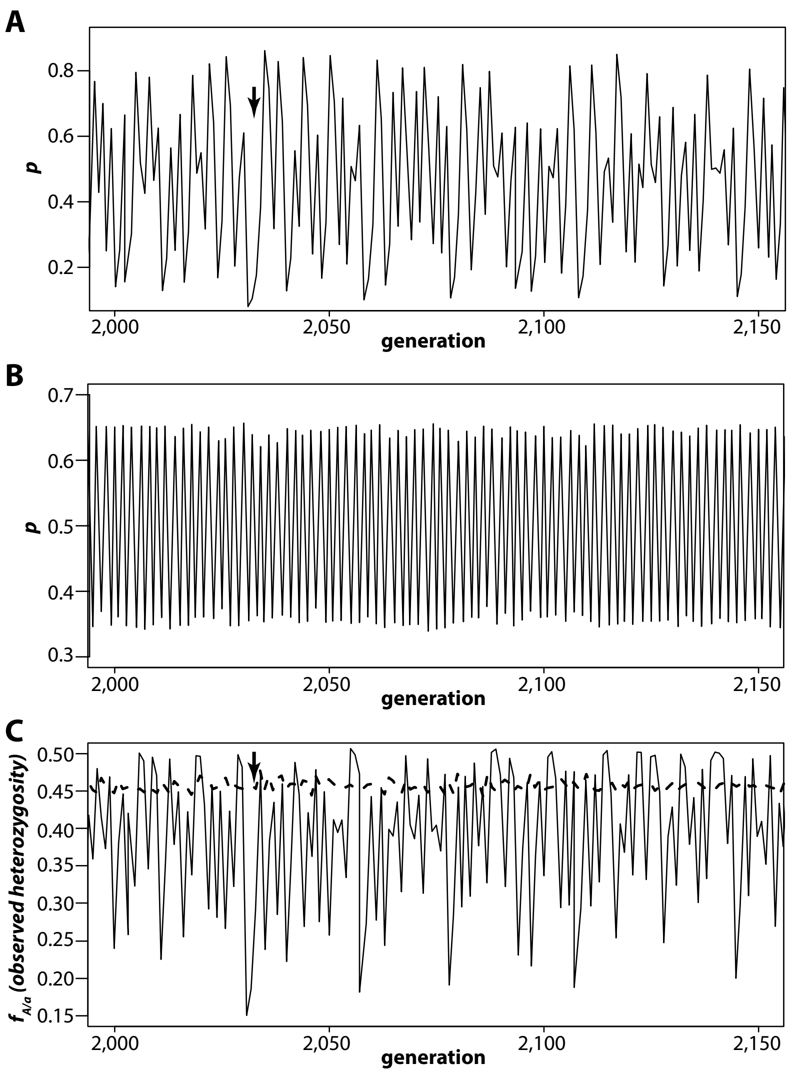


Fig. 8.13 Extreme NFD selection and resulting changes in allele frequency and observed heterozygosity. The data shown here are drawn from the same simulations underlying those shown in Figs. 8.11 and 8.12. (**a** and **b**) The frequency of the derived allele from generation to generation during a representative time interval from generation 2000 to 2150 from the same simulation represented in Fig. 8.11 (**a**; i.e., $s = 2$) and Fig. 8.12 (**b**; i.e., $s = 1$). (**c**) Observed heterozygosity ($f_{A/a}$) for the $s = 2$ (solid line) and $s = 1$ (dashed line) simulations. The arrows in (**a**) and (**c**) point to the same time point, where there is a particularly large swing in both p and $f_{A/a}$

Now consider a sequence interval within the genome where new mutations at many different sites are likely to have a deleterious effect. An obvious example of such an interval is an exon or entire coding sequence, where mutations at non-synonymous sites affect protein function to varying, but potentially devastating, degrees. Such sequences accrue deleterious mutations each generation in a population of any considerable size. Their incessant removal by purifying selection, generation after generation, has the effect of also removing linked variation. The decline in genetic diversity associated with repeated purifying selection is named **background selection** (Charlesworth et al. 1993, 1995).

Before turning to examples of purifying selection simulations, we add an additional means of visualizing polymorphism along a sequence to `heatmaps.r`. The function `sequence_boxplot()` takes the summary statistic output file from a FORTUNA run (e.g., `sumstats0` for deme 0) and produces a box plot of the specified summary statistic for each window. The function also allows the user to turn off the plotting of outlier values by setting `outlier=F` and to specify a range for the y-axis by setting, for example, `y=c(0,2)`. The set of box plots shown in Fig. 8.14c,d were created using this specification of the y-axis range, which facilitates direct comparison between the results from two separate simulations. The sequence interval covered by the box plot will simply be the width of the sampled window as specified by the parameter `windowSize`.

heatmaps.r // add function sequence_boxplot()

```
sequence_boxplot <- function(inputfile, stat="pi", outlier=T, y=c())
{
   dat <- read.table(inputfile, header = T);
   ddat <- split(dat, dat$stat);
   q <- ddat[[stat]];
   q <- q[,3:length(q[1,])];
   boxplot(q, outline=outlier, ylim = y);
}
```

To simulate purifying selection, we use the `negsel` parameter and set the first entries of the `possel` and `ndfsel` parameters set to zero. We now model a large, 1500bp exon in the middle of a simulated sequence that is 26,500bp long. To set up this model, we use the parameter values `negsel 1 0.1 12500 13999 1000`. The first two values specify values of the coefficients s and h under a standard selective model: thus, $w_{D/D} = 1$, $w_{D/d} = 1 - hs = 0.9$, and $w_{d/d} = 1 - s = 0$, where d stands for the deleterious allele. In words, the deleterious d allele is a lethal recessive, but heterozygotes suffer a considerable but lesser fitness deficit. The next two parameter values indicate that this selective regime applies to at least some of the nucleotides in the region between base pair 12,500 and 13,999 of the simulated sequence, our exon. Operating under the rather coarse assumption that two of every three nucleotides are subject to the specified selection regime while the remaining

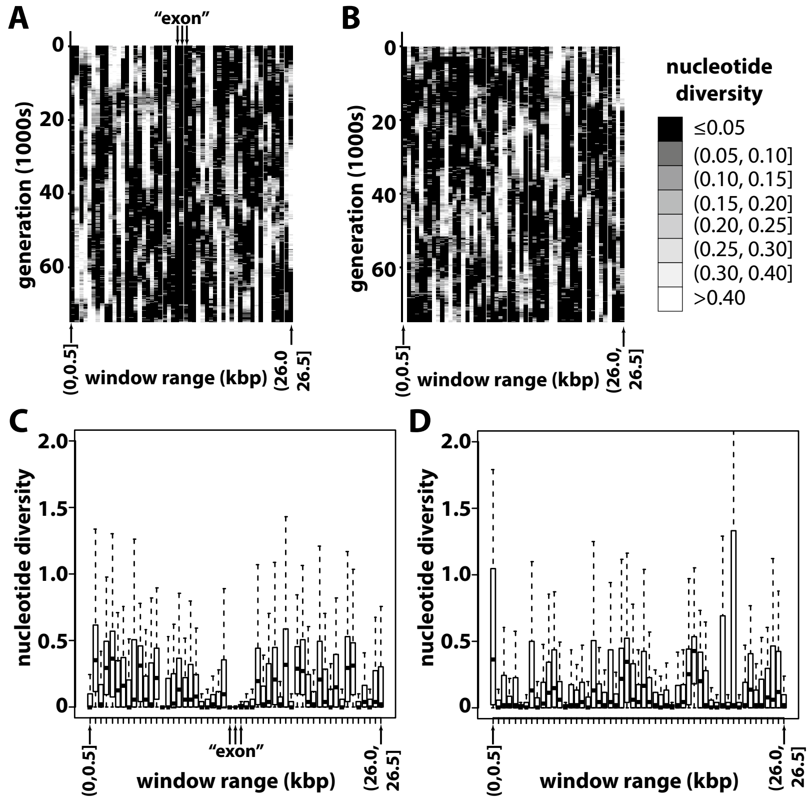


Fig. 8.14 The effect of purifying selection. (**a** and **c**) Parameter `negsel` was set to `0.1 1 12500 14000 1000` on a 26,500bp simulated sequence. Samples ($n = 100$) were drawn every 100 generations over the course of 75,000 simulated generations. The discrete heat map (**a**) and boxplots (**c**) are partitioned into non-overlapping windows of 500bp. In both cases, the three windows that span the "exonic" sequence show sustained, near-absence of genetic diversity as measured by nucleotide diversity (π). (**b** and **d**) Results of a simulation in which there was no selective pressure on any nucleotide in the 45,000 bp sequence, also visualized as both a discrete heat map and set of boxplots. Given that both simulations were performed using point mutation and recombination rates $\mu = r = 1 \times 10^{-8}$ and $N_e = 10{,}000$, under equilibrium conditions the expected value of π for each 500bp window is 0.2

"third codon positions" are neutral, the final parameter value specifies that 1000 of the 1500 nucleotides in the specified interval will be targets of purifying selection. These 2 of 3 nucleotides are selected randomly by the program.

Figure 8.14a and b compare results from the simulated 26,500bp sequence with (a) and without (b) purifying selection. Note that window size is only 500bp in these simulations. Therefore, expected nucleotide diversity $\pi_{window} = \theta_{window} = 4N_e\mu \times 500 = 4 \times 10^4 \times 10^{-8} \times 5 \times 10^2 = 0.2$. The comparison of longitudinal plots shows a clear and persistent lack of genetic diversity at the three windows covering the simulated "exon" (Fig. 8.14a), while the sim-

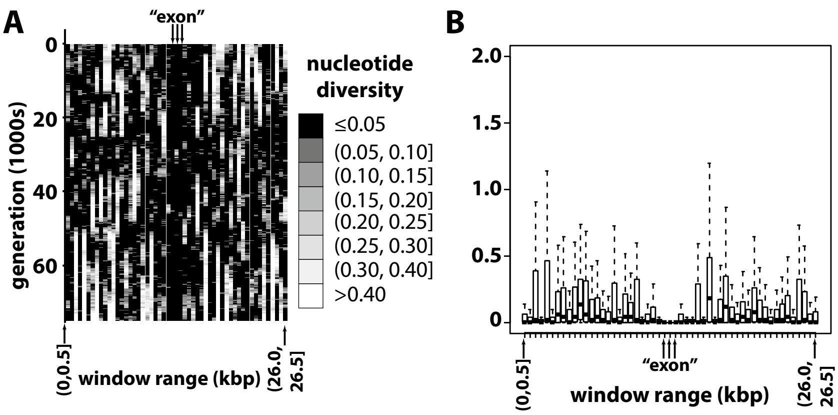


Fig. 8.15 Lower recombination rate leads to a wider selective footprint. Results shown here are from a simulation in which all parameters except `recrate` are the same as in Fig. 8.14. Namely, `recrate 1e-10` was used instead of `recrate 1e-08`. (**a**) The longitudinal plot of nucleotide diversity shows consistent absence of variation in the three middle "exonic" windows, but noticeable declines in genetic diversity are also evident in adjacent "intronic" windows. (**b**) Boxplots show the range of nucleotide diversity across the 75,000 simulated generations. The wider selective footprint is particularly evident here when compared to Fig. 8.14c. Moreover, median and maximum values of π are noticeably lower than in Fig. 8.14c

ulation of a neutral sequence shows no region that is consistently low or high in nucleotide diversity over the 75,000 simulated generations (Fig. 8.14b). Using function `sequence_boxplot()` to summarize nucleotide diversity across the samples over the 75,000 simulated generations, the difference is made more obvious. Variation in the three 500bp "exon" windows is basically absent (though a few outlier samples not shown contained some variation), while the surrounding windows show a wide range of genetic variation over the course of the 75,000-generation simulation (Fig. 8.14c). In the absence of purifying selection on an identically sized sequence, we observe some windows that show less variation in nucleotide diversity during the simulation. However, they are randomly scattered along the sequence (Fig. 8.14d).

These results confirm expectation; over the entire simulated sequence, genetic diversity is depressed as measured by π. At any given point in time (consider a thin, horizontal slice in Fig. 8.14a or b), most windows show nucleotide diversity less than the neutral expectation of 0.2. However, outside the three "exonic" windows of Fig. 8.14a, the distribution of low-diversity windows changes over time—i.e., what was once a window with a value of π below the neutral expectation eventually transitions to a window with above-neutral π and vice versa. Another way to say this is that the region showing persistent background selection is rather narrow.

The footprint of background selection becomes wider in regions of low recombination, which is also true of the footprint of *positive* selection. To see

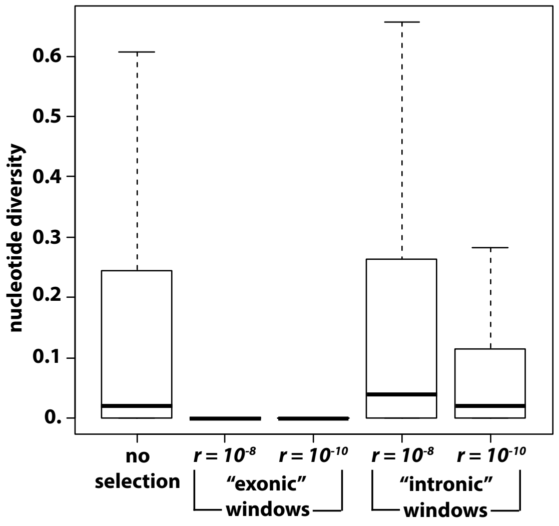


Fig. 8.16 Background selection and recombination rate. Each box plot summarizes nucleotide diversity over all windows meeting the labeled definition across all samples over the course of a single representative simulation. "Exonic" nucleotide diversity is (nearly) zero whether simulations used $r = 10^{-8}$ or $r = 10^{-10}$. However, neutral "intronic" windows show a noticeably lower median and range of nucleotide diversity when $r = 10^{-10}$ than when $r = 10^{-8}$. In fact, "intronic" windows show a noticeably smaller range of nucleotide diversity than neutral windows in a neutral simulation (the *no selection* box plot). This decline in diversity relative to the neutral expectation at sites linked to a region undergoing repeated purifying selection is the pattern associated with background selection

evidence of this, we repeat the previous simulation of purifying selection at an exon with one change: we set `recrate` to `1e-10` ($r = 1\times10^{-10}$) rather than `1e-08` ($r = 1\times10^{-8}$). This change leads the *persistent* decline in genetic diversity to bleed into sequence adjacent to the exonic windows (Fig. 8.15). Although background selection is most obvious in the "intronic" windows tightly linked to the "exonic" windows (Fig. 8.15b), the median and range of nucleotide diversity across *all* intronic windows declines when the lower recombination rate is simulated (Fig. 8.16). This is evidence of background selection: repeated loss of genetic diversity due to selection against variants in a small region of sequence leads to declines in linked, neutral diversity.

References

Charlesworth B, Morgan MT, Charlesworth D (1993) The effect of deleterious mutations on neutral molecular variation. Genetics 134:1289–1303

Charlesworth D, Charlesworth B, Morgan MT (1995) The pattern of neutral molecular variation under the background selection model. Genetics 141:1619–1632

Kim Y, Nielsen R (2004) Linkage disequilibrium as a signature of selective sweeps. Genetics 167:1513–1524

Ohta T, Kimura M (1970) Development of associative overdominance in through linkage disequilibrium in finite populations. Genetics Research 16:165–177

Rice SH (2004) Evolutionary theory: mathematical and conceptual foundations, 1st edn. Sinauer Associates

9 Quantitative Traits

Finest of all the things I have left is the light of the sun,
Next to that the brilliant stars and the face of the moon,
Cucumbers in their season, too, and apples and pears.

– Praxilla, *Adonis*

9.1 Background and Theory

In this chapter, we expand the functionality of FORTUNA by adding code that keeps track of alleles at multiple genes. Collectively, the genotypes of these genes help determine the value of a **quantitative trait**, which is characterized by continuous rather than discrete variation. Familiar examples of quantitative traits are adult height in humans, crop yield, and resistance to pathogens or insecticides. In addition to polygenic variation, environment also affects quantitative trait value. The branch of evolutionary genetics focused on quantitative traits is, not surprisingly, named **quantitative genetics**.

We first focus on basic theory of quantitative genetics and model how genetic variation and environment collectively determine the distribution of quantitative trait values. Most experimental studies of quantitative genetics use controlled crosses in an attempt to identify **quantitative trait loci** (QTL)—those multiple loci in the genome that show some correlation with the quantitative phenotype in question and harbor the actual, causative genes.

Quantitative traits important to the livelihood of humans (e.g., crop yield) have been the subjects of **artificial selection** for millennia, which shows that knowledge of the underlying genetic variation at causative loci is not necessary to meaningfully push the mean quantitative trait value of a herd or field in the desired direction generation after generation. Numerous Assyrian reliefs (ninth Century BC) depict priests and "winged genies" hand-pollinating

R. J. Haasl, *Nature in Silico*, https://doi.org/10.1007/978-3-030-97381-0_9

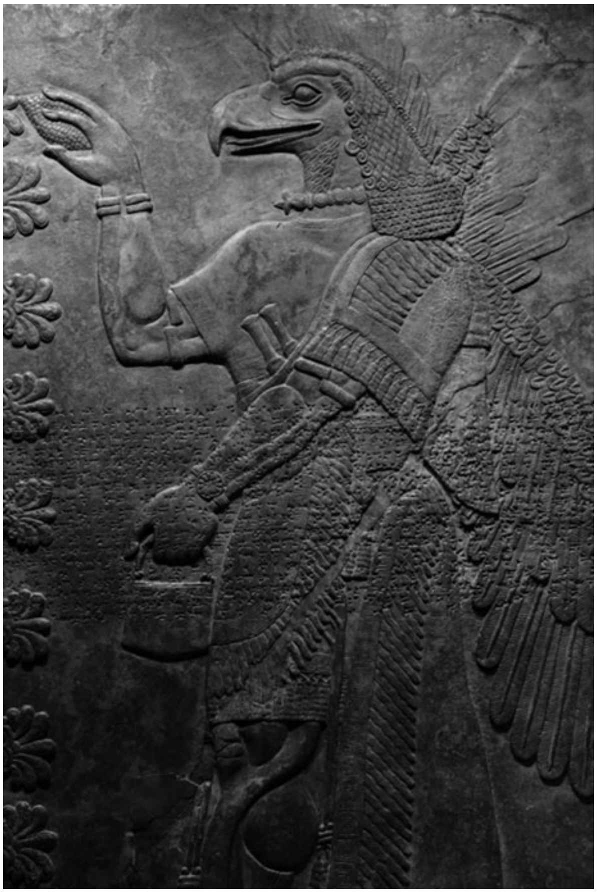


Fig. 9.1 Assyrian relief of an eagle-headed winged genie from the palace of the king of Assyria, Ashur-nasir-pal II (883–859 BC). Photo: In the public domain, designated CC0

female date palm flowers through application of pollen from the male flower (Fig. 9.1). This is the near-magic of artificial selection enshrined.

Because quantitative traits vary continuously, we model the distribution of quantitative trait values in a population as a normal distribution with population mean μ and variance V_P, the latter of which is the sum of two other variances:

$$V_P = V_G + V_E. \tag{9.1}$$

V_G is genetic variance and V_E is environmental variance. This equation embodies the nature vs. nurture debate—which asks whether biology or environment is of greater importance to phenotype—with V_G representing "nature" and V_E representing "nurture." The relative importance of genes and environment to determination of phenotype varies from trait to trait. The relative importance of genetic determinants is the quantitative trait's *heritability*—the fraction of variance in phenotype (V_P) due to genetic variation. Similarity between parent and offspring phenotypes increases with heritability of the quantitative trait.

Broad-sense heritability places all sources of genetic variance in the numerator:

$$H^2 = \frac{V_G}{V_P} \tag{9.2}$$

Note that H^2 is the *symbol* for broad-sense heritability; the "2" superscript does *not* imply $H = (V_G/V_P)^{1/2}$. For animal and plant breeders, greater practical importance is found in the **narrow-sense heritability** of a trait:

$$h^2 = \frac{V_A}{V_P}, \tag{9.3}$$

where V_A is additive variance, one component of total genetic variance V_G.

Additive variance is of greater utility to animal and plant breeders because it derives from QTL showing **additive gene action** in which the trait value of the heterozygous genotype is at the midpoint between the trait values associated with the two homozygous genotypes. Let the *allele effect* of an allele be a. Conceptualize allele effect as the amount by which trait value increases for each copy of the allele in the genotype (Fig. 9.2a). In contrast, at QTL showing **dominant gene action**, the single copy of the focal allele F in the heterozygote adds $a+d$ to the phenotype associated with individuals homozygous for the non-focal allele, where d is the *dominance effect* (Fig. 9.2b)—i.e., one allele shows partial or complete dominance over the other. Dominance effects limit the effectiveness of **truncation selection**, a tactic used by animal and plant breeders to force mean phenotype in one direction or another each generation. It is easiest to intuit the confounding influence of dominance effects by considering an instance of artificial selection such as that detailed in Fig. 9.3.

Gene-gene interaction contributes to the final component of V_G. The most direct demonstrations of these interactions, termed **epistasis**, are those in which variation at one gene *completely masks* the phenotypic influence of another locus. One such trait is coat color in Labrador Retrievers. In reality, Labrador coat color is controlled by many genes, but the most common phenotypic variants—black, brown, and yellow—can be explained by the effects of just two genes. The first, traditionally referred to as the B locus, determines the darkness of the pigment eumelanin produced by these dogs. B/B and B/b dogs produce black eumelanin while b/b dogs produce lighter, brownish eumelanin; this variation explains the phenotypic difference black and brown Labs. The second locus, traditionally referred to as the E locus, determines whether or not the eumelanin produced in the melanosomes of the skin are further deposited in the fur. Deposition of eumelanin in the

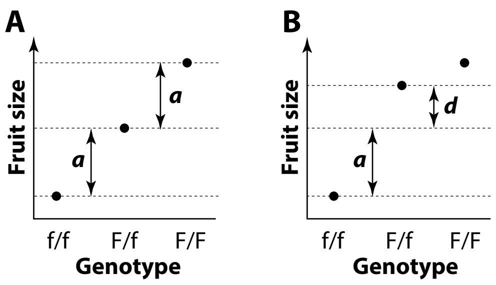


Fig. 9.2 The difference between additive and dominant gene action. Imagine fruits of a species range from small to large size and are determined by variation at a single gene with two alleles, F and f. (**a**) If the causative locus shows additive gene action, the heterozygote bears a fruit size intermediate to that of the two homozygotes (F/F = big and f/f = small). *a* is the allele effect—in this case, how much an F allele adds to phenotype. (**b**) If the causative locus shows dominant gene action, then the heterozygote shows a phenotype that differs from the midpoint by dominance effect *d*. If $d = a$, then a locus shows *complete dominance* and both the F/f and F/F genotypes are associated with the same phenotype. As illustrated, $d < a$, $a + d < 2a$ and thus the locus shows partial dominance

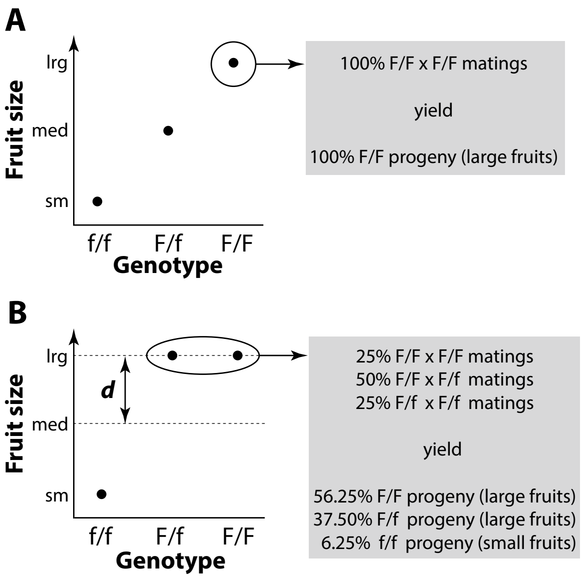


Fig. 9.3 Additive genetic variance is more helpful to animal and plant breeders. Consider the same single-locus trait of fruit size discussed in the caption to Fig. 9.2. Moreover, imagine you are a plant breeder wishing to *increase* the fruit size of plants in the next generation. The intuitive tactic is to choose parents who show the most desirable phenotype—i.e., individuals with large fruits. (**a**) If gene action is additive, all parents selected for large fruit size are F/F and all progeny are therefore F/F. (**b**) If gene action is dominant and we assume that half the parents chosen due to large fruit size are F/f and half are F/F then 6.25% of progeny will, on average bear small fruit. Note that complete dominance ($d = a$) is shown in this case. Also consider that if the majority of large-fruit plants in our stock are heterozygotes, the percentage of small-fruit progeny will be greater than 6.25%

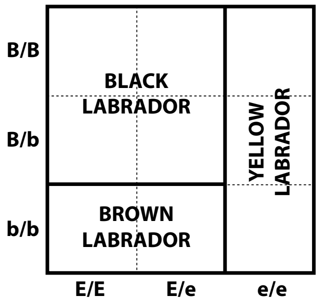


Fig. 9.4 Epistatic determination of coat color in Labrador Retrievers. See text for discussion

fur occurs in E/E and E/e individuals, but not e/e individuals. The type of eumelanin produced by e/e yellow Labradors is masked due to its lack of deposition to the fur. Although there is no direct interaction between the protein products of the two loci, this is still epistasis in the sense that the final phenotype is determined by the variation present at both loci. If you are told an unseen Labrador is B/b you cannot know whether it is a black or yellow Labrador due to your lack of knowledge regarding the E locus (Fig. 9.4).

Technically speaking, the Labrador coat color example is a form of *recessive epistasis*, in which a homozygous recessive genotype at what I will call the *epistatic locus* masks the effects at what I will call the *main locus*. Note that the *B* locus is also subject to dominant gene action—complete dominance, in this case.

Figure 9.5 shows the **genotype-phenotype landscape** of some two-locus interactions. In Fig. 9.5a,b, the epistatic locus completely masks the contribution to the phenotype of the main locus in a recessive or dominant fashion. When the masking epistatic locus is recessive (Fig. 9.5a), only the E_1/E_1 genotype at the epistatic locus (E) masks additions to phenotype associated with the M locus. Otherwise, M acts additively. When the E_1 allele is dominant, just one E_1 allele is sufficient to mask the effects of the M locus (Fig. 9.5b). Figure 9.5c shows the joint additive effect of two loci in the *absence* of epistasis, with the added complication that the allele effects of loci A and B—X_A and X_B, respectively—have different values.

Although it is possible to model higher-order interactions, we now focus on the four possible types of interactions between pairs of loci: additive x additive (AxA), additive x dominance (AxD), dominance x additive (DxA), and dominance x dominance (DXD). Here, if a locus is labeled additive, it shows only additivity—regardless of the genotype at the other, interacting locus. If a locus is labeled dominance, it shows some form of departure from additivity. Here, dominance refers to a locus that disrupts the additivity of alleles at the other locus.

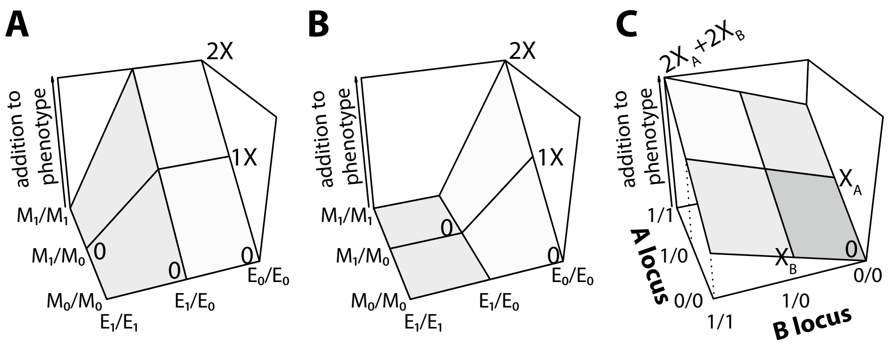


Fig. 9.5 Some genotype-phenotype landscapes of two-locus epistasis. The effect of the epistatic allele E_1 in (**a**) and (**b**) is to completely mask the influence of the main locus on phenotype. (**a**) The masking epistatic allele E_1 is recessive. It only masks the effect of the main allele genotype when the individual is homozygous for E_1. X is the allele effect of allele M_1. (**b**) The masking epistatic allele E_1 is dominant. It is only when both epistatic alleles are E_0 that the purely additive nature of the M locus becomes apparent. Note that dominant refers to the masking allele E_1, not a dominance effect. (**c**) The contribution of two purely additive loci to quantitative phenotype. X_A and X_B represent the allele effects of the "1" allele at loci A and B, respectively. The greatest addition to phenotype is achieved when both loci are homozygous for the "1" allele. The uneven appearance of the plane is due to the fact that $X_A > X_B$

Figure 9.6 shows the results of interaction between pairs of main and epistatic loci for each type of two-locus epistasis.[1] When looking at Fig. 9.6a (AxA epistasis), imagine standing in front of the M_1/M_0 position on the M axis and looking forward through the graph; you would first see a straight line above you slanting downward to your right, followed by a straight line with zero slope and, in the back, a straight line sloping downward to your left. The straightness of these lines shows additive effects associated with the M locus in each case. The difference in slope between the lines, however, is a function of the E locus genotype. In other words, the E locus genotype modulates the additivity of the M locus, but it does not *eliminate* additivity of the M locus. Figure 9.6a is an example of AxA epistasis because you can carry out the same thought experiment as before, but this time standing in front of the E_1/E_0 point on the E-axis and looking through the graph. Again, E-locus genotypes show additivity regardless of the M-locus genotype, but the slope of the phenotypic changes depending on the M-locus genotype.

Figures 9.6b,c show AxD and DxA epistasis, respectively, and are simply mirror images of each other. We focus on the AxD epistasis shown in Fig. 9.6b. Here, main locus M shows purely additive effects, though the slope of additive effects is a function of E-locus genotype. The difference from AxA epistasis is made clear, however, when we fix M-locus genotype

[1] I stress that labeling one locus *main* and the other *epistatic* is not something you will find in the literature; these are labels of convenience for constructing FORTUNA only.

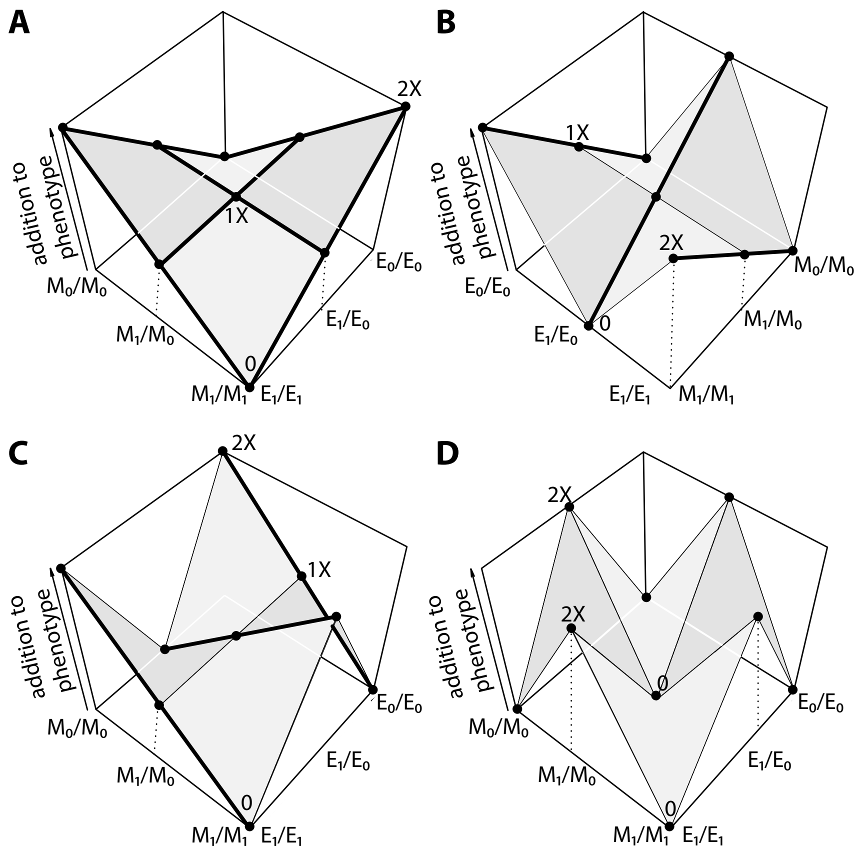


Fig. 9.6 Genotype-phenotype landscapes of two-locus epistasis. Heavy lines indicate additivity as you move along the parallel axis, while light lines indicate dominance (departures from straight lines and, therefore, additivity) as you move along the parallel axis. For several of the joint genotypes (dots), the addition to phenotype is indicated in terms of *X*, the additive effect specified for the main locus *M* in the FORTUNA `parameters` file (see next Sect. 9.2.2). (**a**) AxA epistasis. (**b**, **c**) AxD and DxA epistasis. In both cases the *M* locus shows additivity; however, the resulting effect on phenotype is switched. (**d**) DxD epistasis. See text for further explanation

and consider different *E*-locus genotypes. Non-additive patterns for *E*-locus genotypes are evident when the *M*-locus genotype is either M_1/M_1 or M_0/M_0. Finally, Fig. 9.6d shows DxD epistasis in which each locus—*M and E*—shows dominance, the pattern of which is dependent upon genotype at the other locus. Figure 9.6 uses equal additive effects (*X*) for all loci. Effect sizes can of course differ between the two loci in a pair, which we will consider when developing FORTUNA to simulate quantitative trait evolution.

Finally, we expand Eq. 9.1 by breaking down genetic variance into its components:

$$V_P = V_A + V_D + V_I + V_E, \tag{9.4}$$

where V_A, V_D, and V_I represent additive, dominance, and epistatic genetic variances, respectively. Environmental variance represents the sum effect of the environment on quantitative phenotype determination. As a simple example, consider two female humans with exactly the same alleles at the QTLs determinative of adult height. If $V_E = 0$, then both women should show exactly the same height as adults. However, environment certainly does play a role in the determination of adult height. In particular, nutrition during development—say, calcium intake—will determine whether or not each individual obtains the adult height to which she is *genetically* predisposed.

9.2 Neutral Quantitative Trait Evolution

9.2.1 Some Preliminaries

We begin with modifications to FORTUNA that will allow us to simulate a quantitative trait whose causative loci show only additive gene action. The "sequence" produced by simulation of quantitative traits will actually be a list of alleles for each *unlinked*, causative locus. Thus, parameter `seqlength` takes on a new meaning when simulating quantitative trait evolution: it represents the number of causative loci. At each locus, the "1" allele has a positive allele effect, while the "0" allele does not add to trait value. Previously, we stored the two haplotypes of an individual in the `sequences` variable of the `Individual` class. This variable holds two `vector<int>`s; each lists the positions of derived alleles in one haplotype of the individual. Variable `sequences` is also repurposed for simulation of quantitative trait evolution.

As an example, consider simulation of a quantitative trait with five causative loci (`seqlength` equals 5) in which an individual's `sequences` variable holds the values $\{\{0, 4\}, \{2\}\}$. In binary, we represent the two "sequences" as `10001` and `00100`, respectively. In essence, the `sequences` variable holds the genotypes of the five causative loci: 1/0, 0/0, 1/0, 0/0, and 1/0. If the "1" alleles at each causative locus contribute allele effect a to phenotype, then, ignoring environmental variance, this individual will show a quantitative phenotype of $b + 3a$, where b is the baseline phenotype associated with an individual lacking *any* alleles of positive effect on phenotype.

A fuller accounting of an individual's phenotype requires us to allow allele effect sizes to differ among the causative loci. Some loci may have a large effect associated with larger values of a and vice-versa. In addition, we need to account for environmental variance V_E. To do so in FORTUNA we draw random variates from a Normal distribution with mean $\mu = 0$ and a standard deviation σ input by the user in the `parameters` file. The random variate is then added to the genetically determined phenotypic value, thereby capturing environmental "noise." The value of an individual's phenotype p_i when only additive and environmental variance are active is then:

$$p_i = b + \sum_{j=1}^{k} u_{i,j} a_j + e_i, \tag{9.5}$$

where k is the number of causative loci *not* involved in epistasis, a_j is the allele effect of the jth locus and coefficient $u_{i,j}$ equals 2, 1, or 0 for the genotypes 1/1, 1/0, and 0/0, respectively, at locus j in individual i, and $e_i \sim \mathcal{N}(0,\sigma^2)$ is the environmental variate for individual i. The standard deviation (σ) rather than the variance (σ^2) of the latter Normal distribution will be specified in the `parameters` file because `normal_distribution<>` objects in C++11 are instantiated with standard deviation.

To model dominance effects, it is necessary to include an additional term relative to Eq. 9.5.

$$p_i = b + \sum_{j=1}^{k} u_{i,j} a_j + \sum_{j=1}^{k} v_{i,j} d_j + e_i, \tag{9.6}$$

where $v_{i,j}$ equals 0, 1, or 0 for the genotypes 1/1, 1/0, and 0/0, respectively at locus j in individual i.

Finally, epistatic effects require one additional term to determine phenotype p_i:

$$p_i = b + \sum_{j=1}^{k} u_{i,j} a_j + \sum_{j=1}^{k} v_{i,j} d_j + \sum_{q=1}^{l} \mathbf{M}_x \left[m_{i,q}, c_{i,q} \right] + e_i, \tag{9.7}$$

where l is the number of epistatically paired loci, $\mathbf{M}_x$ is a matrix representing the effects of epistasis (indexed by x for the type of epistasis), $m_{i,q}$ is the genotype of individual i at the qth main locus, and $c_{i,q}$ is the genotype of individual i at the qth epistatic locus. Again, reference to main and epistatic loci is not standard; it just makes things simpler to code. It may be helpful to look again at Figs. 9.5b,c and 9.6, where the main and epistatic loci are represented by M and E along the axes.

9.2.2 *Modifying FORTUNA to Model Quantitative Trait Evolution*

Considerable additions are required to model quantitative trait evolution using FORTUNA. We begin by detailing several new global parameters specified in the `parameters` file.

Additions to **`parameters`**

```
polygenicTrait 0
baseTraitValue 0.
alleleEffects 0.01
dominanceEffects 0.005
epistatic 0
epistaticTypes 5 ... //number of entries depends on number of causative
    ↪ loci
enviroSD 0.02
```

Declaration of new variables in **`params.h`**

```
\\ GLOBAL PARAMETERS
...
extern bool polygenicTrait;
extern double baseTraitValue;
extern vector<double> alleleEffects;
extern vector<double> dominanceEffects;
extern double enviroSD;
extern vector<int> epistaticTypes;
extern int epistaticLoci;
extern bool epistatic;
extern Matrix<int> epiD;
extern Matrix<int> epiR;
extern Matrix<int> epiAA;
extern Matrix<int> epiAD;
extern Matrix<int> epiDA;
extern Matrix<int> epiDD;
...
```

Modifications to the `params.cc` file allow reading of `parameters` file values:

Modifications to **`params.cc`**

```
...
int epistaticLoci;
double baseTraitValue, enviroSD;
bool polygenicTrait, epistatic;
vector<double> alleleEffects, dominanceEffects;
vector<int> epistaticTypes;

int process_parameters() {
   ...
10      if (iter->first == -1) { // block of global parameters
11         ...
12        polygenicTrait = atoi(parameters["polygenicTrait"].c_str());
13        baseTraitValue = atof(parameters["baseTraitValue"].c_str());
14        alleleEffects = get_multi_double_param("alleleEffects",
             ↪ parameters);
15        if (polygenicTrait && seqlength > 1 && alleleEffects.size() == 1)
             ↪ // then set all effect sizes to this value
16          for (int i=1; i<seqlength; ++i)
17            alleleEffects.push_back(alleleEffects[0]);
18        dominanceEffects = get_multi_double_param("dominanceEffects",
             ↪ parameters);
19        if (seqlength > 1 && dominanceEffects.size() == 1)
20          for (int i=1; i<seqlength; ++i)
21            dominanceEffects.push_back(dominanceEffects[0]);
22        enviroSD = atof(parameters["enviroSD"].c_str());
23        epistatic = atoi(parameters["epistatic"].c_str());
24        epistaticTypes = get_multi_int_param("epistaticTypes",
             ↪ parameters);
25        epistaticLoci = epistaticTypes.size();
26        if (epistatic) {
27          seqlength = seqlength + epistaticLoci;
28          for (int i = 0; i<epistaticLoci; ++i)
29            alleleEffects.push_back(0.);
30        } else {
31          epistaticLoci = 0;
32        }
33      }
34    ...
35  }
36  int ed[] = {0, 0, 0, 1, 0, 0, 2, 0, 0};
37  Matrix<int> epiD(3, 3, ed); // dominance epistasis as in
        ↪ Fig.~\ref{ch9:fig5}b
38  int er[] = {0, 0, 0, 1, 1, 0, 2, 2, 0};
39  Matrix<int> epiR(3, 3, er); // recessive epistasis as in
        ↪ Fig.~\ref{ch9:fig5}a
40  int aa[] = {0, 1, 2, 1, 1, 1, 2, 1, 0};
41  Matrix<int> epiAA(3, 3, aa); // AxA epistasis
42  int ad[] = {2, 0, 2, 1, 1, 1, 0, 2, 0};
43  Matrix<int> epiAD(3, 3, ad); // AxD epistasis
44  int da[] = {0, 2, 0, 1, 1, 1, 2, 0, 2};
45  Matrix<int> epiDA(3, 3, da); // DxA epistasis
46  int dd[] = {0, 2, 0, 2, 0, 2, 0, 2, 0};
47  Matrix<int> epiDD(3, 3, dd); // DxD epistasis
```

Lines 2–6 provide the local declarations of the new parameters, `baseTraitValue` is the lowest trait value possible to which the allele effects may add, while `enviroSD` is the standard deviation of the Normal distribution used to generate random environmental the error/noise term in Eqs. 9.5 and 9.6. The Boolean parameters `polygenicTrait` and `epistatic` specify whether or not to simulate a quantitative trait (as opposed to sequence/haplotypes) and whether to model epistatic effects, respectively. Each locus, the number of which is equal to `seqlength`, must have a real number specifying its effect on trait value (`alleleEffects`) and a real number specifying any `dominanceEffects`. If the dominance effect for a particular locus is set to 0., then the locus will be purely additive with the heterozygote having an intermediate effect to the two homozygotes. Lines 10–32 provide the additional code needed to define the global parameters—i.e., not deme-specific—just discussed. In most cases, this mimics code seen before. However, a couple twists require explication. After defining the `alleleEffects` parameter in line 14, lines 15–16 allow a short cut. If only one number is specified in the `parameters` file for `alleleEffects`, rather than a number for each locus, allele effects of all loci are set to the single number provided. Lines 19–20 do the same for `dominanceEffects`. The parameter `epistaticTypes` requires a list of integers, one for each epistatic locus. The provided integer specifies use of a specific definition of two-locus epistasis defined on lines 36–47: `0` for "dominance epistasis" as in Fig. 9.5b; `1` for "recessive epistasis" as in Fig. 9.5a; `2` for AxA; `3` for AxD; `4` for DxA; `5` for DxD. These values are read in on line 24 and the number of epistatic loci—`epistaticLoci`—is determined on line 25.[2]

If epistasis is being modeled, then it is necessary to increase `seqlength` (line 27), which is the number of main loci stored in the `sequences` of each `Individual` plus the number of epistatic loci. For each epistatic locus, we add a 0 to `alleleEffects` for each epistatic locus (lines 28–29). If not simulating epistasis, the number of epistatic loci is set to zero and `alleleEffects` is not expanded (lines 30–32).

Lines 36–47 define the matrices of the different two-locus epistatic effects; these are the $\mathbf{M}_x$ of Eq. 9.7 above. You could certainly modify the magnitude of the values of these matrices to suit your specific needs. However, because they are only specified in `params.cc` and not the `parameters` file, you will need to recompile the program to incorporate any changes you make to the values of these matrices.

In cases where you would like to explicitly list the `alleleEffects` and/or `dominanceEffects` for each locus in the `parameters` file, it can be cumbersome to do so if the number of loci is large. However, we can use the R function `rdirichlet( )` provided by the `MCMCpack` package to quickly generate random values for `alleleEffects` and/or `dominanceEffects`. In what follows, I focus on generating appropriate values for `alleleEffects`. The same approach can be used for `dominanceEffects`.

The Dirichlet distribution generates n random variates that sum to one. Variance among the n variates is controlled by a vector of n concentration parameters. In the following snippet, I use the `rep` function to create a vector of ten concentration parameters all equal to 1. When the concentration parameter equals 1, no one of the n variates is *expected* to be greater or lesser

[2] The text immediately following this paragraph provides tips and R code for generating long lists of `alleleEffects`, `dominanceEffects`, and `epistaticTypes`. This is helpful when the number of causative loci is large.

than another. As we see shortly below, we can use concentration parameter > 1 for a small $x < n$ of the variates; this produces larger allele effects for x of the n causative loci. The advantage of using this simple R code is that, particularly for large numbers of simulated loci, you can quickly generate allele effects that show random variation, print the results to a file, and then copy and paste from that file into the `parameters` file.

Generating random allele effects in R for `parameters` file

```
> library(MCMCpack)
> v <- rdirichlet(1, rep(1, 10))
# first parameter to rdirichlet(): number of variate sets to draw
# second parameter to rdirichlet(): vector of concentration parameters
> write.table(v, file = "dirichlet", row.names = F, col.names = F, quote
    ↪ = F)
```

The following shows the results from two calls to the `rdirichlet` function that provide some indication of how changing the vector of concentration parameter affects the variation among the random variates.

```
 1  > a <- rdirichlet(1, c(10, 10, rep(1,8)))
 2  > var(a[1,])
 3  [1] 0.01068256
 4  > a
 5           [,1]      [,2]       [,3]       [,4]      [,5]       [,6]       [,7]
 6  [1,] 0.3078745 0.2722179 0.09732201 0.07284715 0.0253912 0.04615144
        ↪ 0.07955839
 7            [,8]       [,9]      [,10]
 8  [1,] 0.04043488 0.03371233 0.02449015
 9  > b <- rdirichlet(1, rep(1,10))
10  > var(b[1,])
11  [1] 0.002942469
12  > b
13           [,1]      [,2]       [,3]       [,4]      [,5]      [,6]      [,7]
14  [1,] 0.1361805 0.1908266 0.04276769 0.07372195 0.1195973 0.0214621
        ↪ 0.1446838
15            [,8]      [,9]      [,10]
16  [1,] 0.06659471 0.1426053 0.06156005
```

Note that the results held in object `a` (lines 5–8) reflect the greater weight placed on the first two variates in the call to `rdirichlet( )` on line 1. Roughly 0.58 of the total 1.0 (to which all 10 variates sum) is accounted for by the first two variates, while the remaining 0.42 of density are rather evenly divided among the remaining eight variates. These results can be copy-and-pasted to the `alleleEffects` parameter in the `parameters` file to simulate ten additive loci in which two loci are of much greater effect. The results held in object `b` (lines 13–16) reflect the even concentration placed on all ten variates. Importantly, we still see a wide distribution of values among the variates. However, there is no a priori reason to believe that any one variate will have a greater value than another when all concentrations are equal to 1. The same procedure can be used to generate a

In the case of `epistaticTypes`, we can also use R to avoid a lot of unnecessary typing. Consider a case where you want to simulate 50 epistatic loci, 25 of which yield AxA epistatic effects and 25 of which yield DxD effects. The following R code will generate the list of 50 integers and write them to a file.

Generating a list of epistatic effects in R for `parameters` file

```
> a <- c(rep(2,25), rep(5,25));
> write.table(a, file="epistatics", row.names = F, col.names = F, quote =
    ↪ F);
```

You can then open file `epistatics` and copy-and-paste the output to the `parameters` file.

Having covered the meaning and coding of new parameters, we return to modifications of other FORTUNA source files necessary for simulation of quantitative trait evolution. First, in order to draw from a Normal distribution for the environmental variance of each individual a `mt19937` is added to the list of `static` variables at the end of `fortuna.cc`.

Modification to **`fortuna.cc`**

```
mt19937 Individual::n;
```

Now consider the numerous modifications to `individual.h`.

Modifications to **`individual.h`**

```
...
private:
  double trait_value;
  void calculate_trait_value() {
    normal_distribution<double> Ve(0, enviroSD); // environmental
        ↪ variance
    trait_value = baseTraitValue;
    vector<int> genotypes(seqlength); // initialize with seqlength
        ↪ entries set to zero
    for (int i=0; i<2; ++i)
      for (auto iter = sequences[i].begin(); iter !=
          ↪ sequences[i].end(); ++iter)
        genotypes[*iter]++;
    for (int i=epistaticLoci; i<seqlength-epistaticLoci; ++i) { //
        ↪ calculate contributions of non-interacting loci
      if (genotypes[i] == 1)
        trait_value += (alleleEffects[i] + dominanceEffects[i]);
      if (genotypes[i] == 2)
        trait_value += (2 * alleleEffects[i]);
    }
    if (epistatic) {
      int mainLocusIndex = 0;
      // calculate contributions of epistatic pairs
      for (int i=seqlength-epistaticLoci; i < seqlength; ++i) { //
          ↪ index of epistatic locus
        double traitMod;
        switch(epistaticTypes[mainLocusIndex]) {
          case 0: // dominance epistasis
              traitMod =
                  ↪ epiD[genotypes[mainLocusIndex]][genotypes[i]] *
                  ↪ alleleEffects[mainLocusIndex];
              trait_value += traitMod;
               break;
          case 1: // recessive epistasis
              traitMod =
                  ↪ epiR[genotypes[mainLocusIndex]][genotypes[i]] *
                  ↪ alleleEffects[mainLocusIndex];
              trait_value += traitMod;
               break;
          case 2: // AxA epistasis
              traitMod=
                  ↪ epiAA[genotypes[mainLocusIndex]][genotypes[i]]
                  ↪ * alleleEffects[mainLocusIndex];
              trait_value += traitMod;
               break;
          case 3: // AxD epistasis
              traitMod =
                  ↪ epiAD[genotypes[mainLocusIndex]][genotypes[i]]
                  ↪ * alleleEffects[mainLocusIndex];
              trait_value += traitMod;
              break;
          case 4: // DxA epistasis
              traitMod =
                  ↪ epiDA[genotypes[mainLocusIndex]][genotypes[i]]
                  ↪ * alleleEffects[mainLocusIndex];
              trait_value += traitMod;
              break;
          case 5: // DxD epistasis
              traitMod =
                  ↪ epiDD[genotypes[mainLocusIndex]][genotypes[i]]
                  ↪ * alleleEffects[mainLocusIndex];
              trait_value += traitMod;
        }
        mainLocusIndex++;
      }
    }
    trait_value += Ve(n); // add environmental effect
    if (trait_value < baseTraitValue)
       trait_value = baseTraitValue;
  }
...
public:
  inline double get_trait_value() {return trait_value;}
  ...
  Individual (vector<vector<int>> seqs): sequences(seqs) { // generation
      ↪ 0 and migration constructor
    if (polygenicTrait)
```

```
60          calculate_trait_value();
61      }
62      Individual (Individual *p1, Individual *p2, vector<vector<int> >
            ↪ mutation_results, vector<int> breakpoints) { //
            ↪ intra-simulation constructor
63        if (polygenicTrait) {
64          sequences = mutation_results;
65          calculate_trait_value();
66        } else {
67           ... // constructor code previously discussed
68          }
69       }
70       ...
71       static mt19937 n;
```

Line 3 declares the variable `trait_value`, which holds the trait value calculated upon the "birth" of each new member of the class `Individual`. The function `calculate_trait_value( )` (lines 4–53), as the name implies, calculates the `trait_value` of an individual. To do so, we first instantiate a `normal_distribution` member with mean 0 and standard deviation equal to the `extern` parameter `enviroSD` (line 5). Trait value is set to its baseline value in line 6, and the rest of the function modifies this value, potentially adding to it. In order to modify the trait value; however, we first need to determine the `vector<int>` genotypes of each locus, both additive and epistatic (if the latter is applicable; line 7). This is accomplished by the code of lines 8–10, in which each locus represented by a position in `vector<vector<int> > sequences` is interrogated. The `for` loop spanning lines 11–16 determines the additive and dominance effects of all loci *not* subject to epistasis. Note that `i` of this loop is first set to the value of `epistaticLoci`. If there is no epistasis, this variable is equal to zero. Otherwise, the loop begins at the first locus that is *not* subject to epistasis. Furthermore, the loop runs to only the last of the purely additive loci (`seqlength - epistaticLoci`) (see Fig. 9.7). The loop tabulates additions to `baseTraitValue` by determining if the genotype adds one or two `alleleEffects` (lines 12–15), and modifies the addition by any non-zero `doiminanceEffects` specified for heterozygotes of a given locus in the `parameters` file (line 13). The loop spanning lines 17–37, further `trait_value` when `epistatic` is set to true (i.e., 1). Variable `mainLocusIndex` is first set to zero (line 18), followed by a `for` loop that runs from the first of the epistatic loci (`seqlength-epistaticLoci`) through the end of `sequences` (line 20; see Fig. 9.7). For each pair of main and epistatic loci, the quantity by which `trait_value` is modified is kept track of by the variable `trait_mod` (line 21). Then a `switch( )` function is used to apply the proper type of epistatic effect for the locus (lines 22–46). Note that each `case` corresponds to the `epistaticTypes` of the locus, and the matrices defined in `params.cc` are used to make the proper modification to `trait_value`. Finally, environmental effect is added to `trait_value` on line 38. If the calculated `trait_value` is less than `baseTraitValue`, `trait_value` is set to `baseTraitValue` (lines

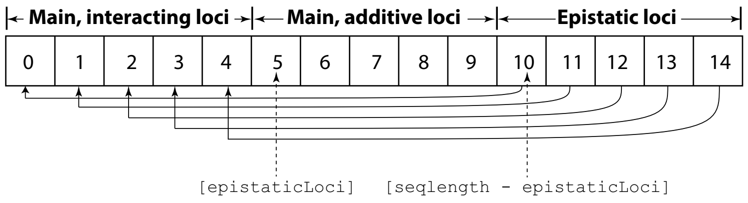


Fig. 9.7 Schematic of the `vector<int> genotypes` Here, the genotypes of 15 loci are stored as 0, 1, or 2—which is the number of phenotype-modifying alleles at the locus. 10 main loci are simulated. Because `epistaticLoci` is set to 5, the first 5 main loci will interact with the 5 epistatic loci. The remaining 5 main loci are purely additive. Solid arrows from epistatic to main loci represent the interaction between pairs of loci. Two important indices of `genotypes` used in the new `Individual` class function `calculate_trait_value()` are indicated by vertical dotted arrows

51–52). This last conditional implies it is impossible to have a `trait_value` less than `baseTraitValue`.

Changes to the `public` members of the `Individual` class include a function to return the calculated `trait_value` (line 56) and modified constructors (lines 58–69). The first constructor (lines 58–61), as before, is used in the first generation or in instances of migration between populations. The only modification to this constructor is the instruction to calculate `trait_value` upon instantiation of the `Individual` object. The second constructor (lines 62–69) is used in the case of reproduction within a population. Values for the `sequences` variable are the result of reproduction and mutation (line 64) and, again, `trait_value` is calculated upon instantiation of the new `Individual` object (line 65). Finally, the `static mt19937` variable is on line 71. Recall that this random number generator is used to draw a random environmental effect (lines 5 and 50).

Before turning to simulation results in Sects. 9.4–9.5, we next discuss haplotype files whose output was covered in Sect. 6.3.3. In particular, we will consider the code needed to start a simulation from a saved haplotype file rather than from MS output.

9.3 Multiple Runs of Sequence or Quantitative Trait Evolution

Steven Jay Gould famously postulated that if the "tape" of evolution on Earth was rewound and run again, the biological world would be very different from the one run of evolution to which we are privy today. In some ways this idea inspires ideas of science fiction. Consider, for example, a run of evolution in which a terrestrial cephalopod becomes the dominant species on Earth, composing epic poems, practicing a variety of spiritual disciplines, and engaging in military conflict. Importantly, the fictional aspect of such a vision is that on "our" Earth things have turned out quite differently. Yet, the idea that identical beginnings can produce distinct evolutionary outcomes is not fiction. Although natural selection is a force that drives phenotype toward an ephemeral optimal value or maintains that optimal value, changes due to natural selection are still stochastic. The probabilistic nature of evolution ensures that things need not have turned out the way they have.

Recall from Sect. 6.3.3 that every `printhapfreq` generations, a haplotype file is output for every extant deme in that generation. When simulating sequences, this file consists of the following:

- First line: integer positions of each polymorphic locus
- Second line: number of the deme in which the mutation leading to the polymorphism arose
- Third line: number of the generation in which the mutation leading to the polymorphism arose
- Each subsequent line: the state of each polymorphic locus (0 or 1) for a chromosome

When simulating quantitative traits, the "haplotype" file consists of the following:

- First line: a name for each main and epistatic locus—e.g., Locus1, Locus2, etc...
- Each subsequent line: the allele at each unlinked locus (0 or 1)

Note that the genotypes for each locus are obtained by combining pairs of lines. For example, if there are four additive loci without epistasis and lines 2 and 3 of the "haplotype" file are `0 1 0 0` and `1 0 0 1`, respectively, then the genotypes for loci 1-4 in one diploid individual are 1/0, 1/0, 0/0, and 1/0.

To this point, we have either started the simulation with no polymorphism or (almost exclusively) used MS to generate the starting variation. However, the generation of haplotype files allows us to begin a simulation from the variation captured in the haplotype file. There are at least two good motivations for choosing to do so. First, as alluded to above, the stochastic nature of evolution can be made apparent by running multiple simulations with a common starting point. Second, we may have an empirical sample with estimates of allele frequencies. In this case, we can create a haplotype file that captures the estimated allele frequencies and use simulation to forecast the future evolution of the sampled population(s).

Next, we cover the additional code necessary to use a haplotype file to instantiate a `Population` object. First, we add two deme-specific parameters to the `parameters` file: `useHapStartingData` and `hapFile`. The former is a `vector<bool>` variable, each entry of which specifies use of a haplotype file to populate a `Population` object when set to 1 rather than 0. The latter is a `vector<string>` that holds the name(s) of the haplotype file(s) to be used for a deme—e.g., `deme0_3000`, which contains the haplotypes or quantitative alleles of deme 0 from generation 3000 of a previous FORTUNA run. To read in the values of these parameters the following modifications are made to the files handling parameters:

Modifications to **`parameters`**; **`params.h`**; and **`params.cc`**

```
\\ parameters
...
DEME /// 0
useMS 0 // set to 0 if using haplotype data
useHapStartingData 1
hapFile deme0_3000
...
DEME /// 1
useMS 0
useHapStartingData 1
hapFile deme1_3000
...

\\ params.h
extern vector<bool> useHapStartingData;
extern vector<string> hapFile;

\\params.cc
vector<bool> useHapStartingData;
vector<string> hapFile;
...
int process_parameters() {
...
     if (iter->first == -1) { // block of global parameters
        ...
     } else {  // block of deme parameters
        ...
       useHapStartingData.push_back(
            ↪ atoi(parameters["useHapStartingData"].c_str()) );
       hapFile.push_back( parameters["hapFile"] );
        ...
     }
...
}
```

The other requisite code is added to the `Population` constructor:

Modifications to **`population.h`**

```
Population (int popnum, int eextant, string rreppy):popn(popnum),
    ↪ extant(eextant), reppy(rreppy) {
   ...
   if (polygenicTrait) {
     if (birthgen[popn] == 0) {
       ...
       map<int, vector<vector<int> > > unlinked_loci;
       for (int i=0; i<pop_schedule[popn][0]; ++i) {
        unlinked_loci[i].push_back({});
        unlinked_loci[i].push_back({});
     }
     if (useHapStartingData[popn]) {
        hapFile[popn].erase(hapFile[popn].find_last_not_of("
            ↪ \n\r\t")+1);
        ifstream hapdata(hapFile[popn]);
        string hap_line;
        getline(hapdata, hap_line); // skip first line
        for (int i=0; i<pop_schedule[popn][0]; ++i) {
           for (int j=0; j<2; ++j) { // two lines per diploid individual
              getline(hapdata, hap_line);
              for (int k=0; k<hap_line.size()-1; k+=2) {
                 if (hap_line[k] == '1')
                    unlinked_loci[i][j].push_back(k/2);
              }
           }
        }
     } else { // use MS to obtain starting data
         ...
     }
     for (int i=0; i<pop_schedule[popn][0]; ++i)
        individuals.push_back( new Individual(unlinked_loci[i]) );
    }
 }
 ...
if (!polygenicTrait) {
    vector<int> allele_positions;
   ...
  if (useMS[popn]) {
      ...
  } else if (useHapStartingData[popn]) {
     hapFile[popn].erase(hapFile[popn].find_last_not_of(" \n\r\t")+1);
     ifstream hapdata(hapFile[popn]);
     string hap_line;
     getline(hapdata, hap_line); // 1st line:polymorphic site positions
     istringstream iss(hap_line);
     string s;
     while (iss >> s) { // read allele positions; create new Allele
          ↪ object for each
        int position = atoi(s.c_str());
        allele_positions.push_back( position );
        alleles.insert( { position , new Allele(position, -1, popn) } );
     }
     for (int i=0; i<2; ++i)
        getline(hapdata, hap_line); // ignore next two lines
     for (int i=0; i<pop_schedule[popn][0]; ++i) {
        vector<vector<int> > ses;
        ses.push_back({});
        ses.push_back({});
        for (int j=0; j<2; ++j) { // two lines per diploid individual
           getline(hapdata, hap_line);
           istringstream iss2(hap_line);
           int site = 0;
60                while (iss2 >> s) {
61                   if (s[0] == '1') ses[j].push_back(allele_positions[site]);
62                   ++site;
63                }
64             }
65             individuals.push_back( new Individual(ses) );
66          }
67       } else { // start with NO variation
68          ...
69       }
70    }
71    ...
72 };
```

When both polygenicTrait and useHapStartingData[popn] are set to 1 (i.e., true), lines 6–24 are used to instantiate the population. The variable unlinked_loci is declared and defined on lines 6–10; this variable holds the alleles for each of the polygenic loci determinative of the quantitative phenotype. When hapFile is read from the parameters file, the string holds white space. It is necessary to trim this white space (line 12) so the file name is recognized when instantiating the ifstream hapdata on line 13. Lines 14–15 define the string hap_line, which takes input from the getline() function and reads in the first three lines, which are ignored because they contain metadata for each locus.

The for loop on lines 16–24 populates unlinked_loci, where the int key of this map object specifies the individual. The inner for loop (lines 17–23) reads the next two lines in the "haplotype" file, and the innermost for loop (lines 19–22) parses each line by advancing the index variable k by two each iteration to skip the white space between the 1s and 0s. Finally, whether the starting data are obtained from a "haplotype" file or MS, the data in unlinked_loci are used to instantiate each Individual of the deme (lines 29–30).

If useHapStartingData[popn] is set to 1 (line 38), but a sequence rather than a polygenic trait is simulated (line 33), we begin the same way as with a polygenic trait by declaring the ifstream hapdata using hapFile[popn] trimmed of trailing white space (lines 39–40) and reading the first line of the file (lines 41–42). An istringstream named iss is used to parse the first line of the input file, obtain the polymorphic positions, store each position in allele_positions, and instantiate an Allele object for each position (lines 43–49). Lines 51–52 read and skip the information in the next two lines of the haplotype file. Each Individual of the deme is instantiated via the for loop on lines 52–66. This involves creating a vector<vector<int> > ses (line 53) that initially holds two empty vectors (lines 54–55). For each diploid individual, the next two lines of the haplotype file are read one at a time using the for loop on lines 56–64. Again, an istringstream object is used to parse the input (line 58) and, if a given site in the haplotype is set to the derived "1" allele, its position is added to the appropriate vector<int> of `ses` (lines 60–63). After both `vector<int>`s of `ses` are established, they are used to instantiate the `Individual` object (line 65).

As a quick example, I now consider simulating a quantitative trait evolving neutrally for 3001 generations. This first simulation starts with MS-generated variation at 10 purely additive QTLs, each with an `alleleEffect` of 0.05 and a mutation rate of 10^{-5} per locus. The black line in Fig. 9.8a shows the neutral evolution of quantitative phenotype. I then ran five separate simulations using the `deme0_3000` file to start each simulation. The evolving values of the quantitative trait for all five, independent simulations are shown in gray (Fig. 9.8a). In the absence of selection, phenotype wanders freely, with two of the simulations ending with a phenotype greater than the starting value after 3000 further generations of evolution and three of the simulations ending with a phenotype less than the starting value.

To demonstrate the use of a haplotype file as the starting point for a *sequence* simulation, I first simulated positive selection on a new variant at position 50,000 within a 100,000bp sequence. Mutation and recombination rates were set to 10^{-8} per nucleotide, $N_e = 10{,}000$, $s = 0.05$, and $h = 0.5$. The simulation was stopped after 100 generations, at which point the frequency of the favored allele at position 50,000 was equal to approximately 0.018. Using the `deme0_100` haplotype file for starting variation, I then ran two independent simulations for an additional 900+ generations. Figure 9.8b shows the results, with the heavy black line showing the value of Tajima's D for the window centered on the selected site for the first 100 generations. One major difference between the two replicate simulations (gray lines in Fig. 9.8b) was the fixation time of the favored allele: generation 298 for the dark gray line and generation 513 for the darker gray line (indicated by vertical dashed lines in Fig. 9.8b). Further comparison of the dynamics of Tajima's D between the two independent simulations reveals some interesting similarities and differences. In the simulation where fixation occurs earlier (dark gray line), Tajima's D declines below −2.0 well before final fixation and remains at its nadir until after fixation. In the simulation where fixation occurs later (lighter gray line), Tajima's D also descends rapidly below −2.0. Unlike the other simulation, however, its nadir is maintained for a longer period but recovers to values greater than −2.0 long before fixation takes place. This simple example shows how multiple runs starting from identical genetic variation help build heuristic understanding of the evolutionary process and the summary statistics we use to understand it.

9.4 Random Genotypes for Initiating Quantitative Trait Simulation

Two disadvantages to using MS-generated variants to start a simulation of quantitative trait evolution are: (1) we cannot predict what the distribution

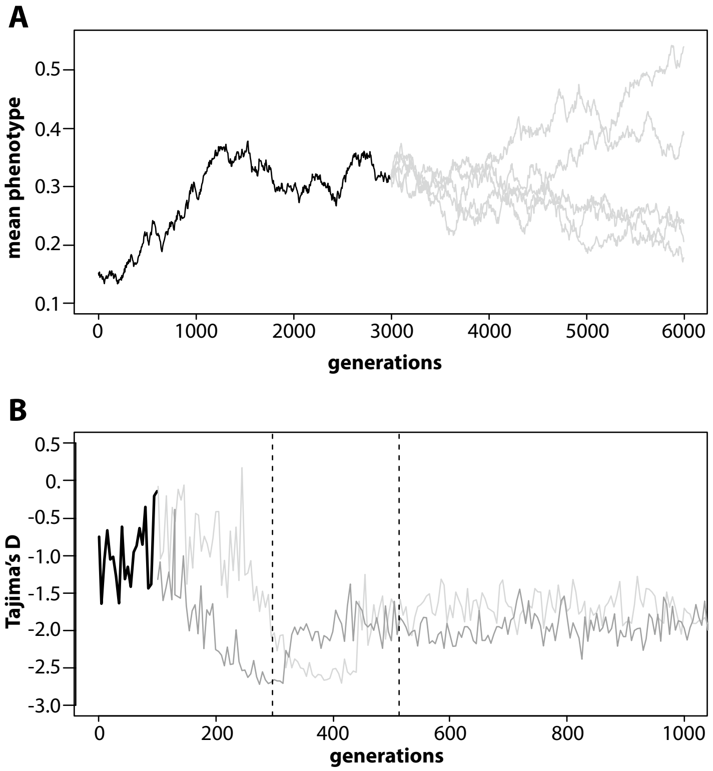


Fig. 9.8 Running multiple simulations of quantitative trait and sequence evolution from a common starting point. (**a**) Neutral quantitative trait evolution. MS-generated variation was used to start the simulation of a quantitative trait evolving neutrally for 3000 generations (black line). The genetic variation data in the "haplotype" file generated at generation 3000 was then used as the starting point for five independent simulations that ran for another 3000 generations (five gray lines). (**b**) Sequence evolution. Positive selection on a new variant was simulated for 100 generations and the value of Tajima's *D* at the 10,000bp window centered on the selected site was plotted (black line). After 100 generations the favored allele was at a frequency of approximately 0.018. The resulting haplotype file was then used as a starting point for two independent simulations that were run for more than 900 additional generations (light and dark gray lines). The dashed vertical lines indicate the generation when the favored allele became fixed—earlier for the simulation represented by the dark gray line. See text for more detail regarding the parameter values of the simulations

of phenotypes will be, and (2) many of the loci that determine quantitative phenotype become monomorphic in short order because the derived "1" allele begins at very low frequency as predicted by the neutral allele frequency spectrum. The second disadvantage may actually be desirable in specific cases where we want to investigate the role of mutation on continued increases or decreases in phenotype over long evolutionary periods; in these cases, new mutations at previously monomorphic loci serve as new conduits for phenotypic change.

Given these potential disadvantages, it is desirable to produce a data set of random genotypes in which we specify the initial frequency of the derived "1" allele at each locus. The following R script allows for rapid generation of a random set of genotypes with which we can initialize a quantitative simulation by setting the `useHapStartingData` parameter detailed in Sect. 9.3 and setting the `hapFile` parameter to the output file specified in the following R script.

generatePolygenicData.r

```
makeStartingDataset <- function(n, freqvec, ofname)
{
  m <- matrix(0, ncol = length(freqvec), nrow = n);
  names <- vector();
  for (i in 1:length(freqvec)) {
    random_vecky <- sample( c( rep(1,freqvec[i]), rep(0, n-freqvec[i]) )
        ↪ );
    m[,i] <- random_vecky;
    names <- c(names, paste("Locus", i, sep = ""));
  }
  d <- as.data.frame(m);
  colnames(d) <- names;
  write.table(d, file = ofname, row.names = F, col.names = T, quote= F,
      ↪ sep = " ");
}
```

The arguments to the function include:

- `n`: $2N_e$
- `freqvec`: a vector holding the number of derived alleles in the data set for each locus
- `ofname`: the name of the file two which the data set is written

For example, the function call `makeStartingData(20000, c(400, 5000, 10000, 5000), "starter")` will write a file named `starter` with 20,000 rows of alleles for four loci at starting frequencies of 0.02, 0.25, 0.50, and 0.25 as well as a header line of the form `Locus1 Locus2 Locus3 Locus4`. Thus, we ensure that the minor allele frequency at each locus is sufficiently large to prevent rapid loss due to drift and assert control over the expected phenotype of individuals represented by the starting data set. Regarding the latter point, the expected phenotype determined by additive variance alone is $s\sum_{j=1}^{k} q_j a_j$, where k is the number of additive loci, q_j is the specified frequency of the jth locus, and a_j is the specified allele effect of the jth allele. For example, if we model a quantitative trait determined by $j = 20$ additive loci, each with an allele effect of $a = 0.05$ and a starting frequency of $q = 0.25$, the expected phenotype is 0.5. Of course, this does not account for noise due to environmental variance. Moreover, it is impossible to predict the phenotypes of data sets generated with this function when we model a locus that includes dominant gene action and/or epistasis. For the same reason, biological phenotype becomes less predictable when dominance effects and/or epistasis are active.

9.5 Quantitative Traits Under Selection and Evolutionary Constraints

Next, we examine quantitative trait evolution under selection. We begin by introducing the code necessary, followed by examples of directional selection, evolutionary constraints on phenotype, and artificial selection in the form of truncation selection.

The variable `polysel`, which will control the type and strength of selection, is first declared and defined. Note that this is a deme-specific parameter.

Final modifications to **`parameters`**; **`params.h`**; and **`params.cc`**

```
\\ parameters
...
DEME  /// 0
...
polysel 0 2 10 // added to each deme
...

\\ params.h
extern map<int, vector<double> > polysel;

\\ params.cc
...
map<int, vector<double> > polysel
    ...
     } else {  // block of deme parameters
         ...
        polysel[iter->first] = get_multi_double_param("polysel",
             ↪ parameters);
          ...
     }
```

Next, we review additional modifications to `population.h` necessary to incorporate selection on a quantitative trait.

Modifications to **population.h**

```
private:
...
bool polyselection{};
...
public:
...
void reproduce(int gen, vector<int> actives) {
    ...
   if (polyselection) {
     if (polysel[popn][0] == 3) { // then truncation selection
        multimap<double, int> ordered_by_trait;
        for (int i=0; i<individuals.size(); ++i)
          ordered_by_trait.insert(pair<double,int> (
               ↪ (*individuals[i]).get_trait_value(), i) );
        if (polysel[popn][1] == 1) { // selection for lesser phenotypes
          for (auto iter=ordered_by_trait.begin(); iter !=
               ↪ ordered_by_trait.end(); ++iter) {
             recode_parents.push_back(iter->second);
             if (recode_parents.size() == polysel[popn][2])
                break;
          }
        } else { // selection for greater phenotypes
          for (auto riter=ordered_by_trait.rbegin(); riter !=
               ↪ ordered_by_trait.rend(); ++riter) {
             recode_parents.push_back(riter->second);
             if (recode_parents.size() == polysel[popn][2])
                break;
          }
        }
     } else {
        for (int i=0; i<individuals.size(); ++i)
          if (randomnum(e) <=
               ↪ (*individuals[i]).get_polygenic_fitness(popn) )
               ↪ recode_parents.push_back(i);
     }
   }
   if (activeselection || negselection || polyselection) {
     randomind.param(uniform_int_distribution<int>::param_type(0,
          ↪ recode_parents.size() - 1) );
   for (int i=0; i<N; ++i) {
     vector<int> parents;
     if (activeselection || negselection || polyselection ) {
         ...
       } else {
          ...
       }
       ...
    }
    ...
}

46  Population (int popnum, int eextant, string rreppy):popn(popnum),
        ↪ extant(eextant), reppy(rreppy) { // constructor
47     ...
48    if (possel[popn][0] != 0 || nfdsel[popn][0] !=0 || negsel[popn][0] != 0
          ↪ || polysel[popn][0] != 0) { // added last conditional
49    ...
50      } else if (polysel[popn][0] != 0) {
51        polyselection = true;
52      }
53      ...
54    }
55     ...
56  }
```

On line 3, the `bool polyselection` is declared. This will be set to true in the constructor when the first entry of the `polysel` parameter is set to something other than zero (lines 46–56). Selection on a quantitative trait is implemented in the `reproduce()` function (lines 7–45). We will implement three distinct types of selection on polygenic traits: (1) natural selection in which fitness is determined based on the distance of an individual's phenotype from a target phenotype specified by the second entry (index 1) of the `polysel` parameter; (2) evolutionary constraints, where two phenotypic values are listed in `polysel` as lower and upper constraints on phenotype; and (3) truncation selection, as a form of artificial selection, in which individuals in the top or bottom *n*% of the current phenotypic distribution are selected as the parents of the next generation, where *n* is specified by the third entry (index 2) of the `polysel` parameter.[3]

Lines 10–27 implement truncation selection, which holds when the first entry of `polysel` is set to 3. We use a `multimap<double, int>` object named `ordered_by_trait` to enable ordering of individuals by phenotype. `multimaps` are a peculiar form of `map`-like container in which the key is a potentially non-unique value. We use this container because, like `maps`, `multimaps` are indexed in ascending order by key. This will make it easy for us to identify the *n* individuals with the least or greatest phenotypes as parents of the next generation. Lines 12–13 populate `ordered_by_trait`. Under truncation selection, the second entry (index 1) of the `polysel` parameter specifies whether selection is for lesser phenotype (when equal to 1; line 14) or greater phenotype (line 20). In the former case, we iterate through the `multimap` in the forward direction (lines 15–19), adding the `multimap` *value*, which is the integer linked to the `Individual` object with the phenotype recorded by the `multimap` *key*. Iteration continues until the number of individuals stored as parents in `recode_parents` reaches the required number *n*, as specified by `polysel[popn][2]` (lines 17–18). The same procedure is used when the selected parents are the *n* individuals with the *greatest* phenotype, with the exception that we iterate in the backwards direction through the

[3] See Sects. 9.5.1–9.5.3 for the meanings of the `polysel` entries in each of these cases.

`multimap` (lines 21–25). Lines 32 and 36 are previously coded lines to which the `polyselection` condition has been added. Addition of individuals with the least or greatest trait values to `recode_parents` puts these individuals in a position to be parents of the next generation.

Lines 27–30 implement natural selection and evolutionary constraints. In these lines, each individual's fitness is assessed through a call to the function `get_polygenic_fitness()` of class `Individual` (discussed next). If `randomnum` returns a value less than or equal to the fitness of the individual, that individual's identifying integer is pushed to `recode_parents`, making it a potential parent of the next generation's progeny.

Calculation of polygenic fitness is the only addition necessary to `Individual.h`:

Modifications to **`Individual.h`**

```
 1  public:
 2  ...
 3    double get_polygenic_fitness(int popn) {
 4      double fitness;
 5      double theta = polysel[popn][1];
 6      double s = polysel[popn][2];
 7      if (polysel[popn][0] == 1) { // Gaussian fitness function
 8        fitness = exp(-1 * s * (pow (trait_value - theta, 2)));
 9      } else if (polysel[popn][0] == 2) { // Evolutionary constraints
10        if (trait_value < polysel[popn][1])
11          fitness = 1 - ((polysel[popn][1] - trait_value) *
                ↪ polysel[popn][3]); // [3] sel. strength
12        else if (trait_value > polysel[popn][2])
13          fitness = 1 - ((trait_value - polysel[popn][2]) *
                ↪ polysel[popn][3]);
14        else
15          fitness = 1;
16      }
17      if (fitness < 0) fitness = 0;
18      if (fitness > 1) fitness = 1;
19      return fitness;
20    }
```

The `fitness` returned on line 19 is declared on line 4. Fitness is then either calculated using a Gaussian fitness function (if the first entry of `polysel` is set to 1) or based on evolutionary constraints (if the first entry of `polysel` is set to 2). Before returning to the code, however, we take a look at the form of the Gaussian fitness function, which specifies the relative fitness of individual i (w_i) as:

$$w_i = e^{-s(p_i-\theta)^2}, \tag{9.8}$$

where s is a selection coefficient that quantifies strength of selection, p_i is the quantitative phenotype of individual i, and θ is the optimal phenotype.

Figure 9.9 plots w_i versus $p_i - \theta$. Note the ways in which the selection coefficient s differs from that used in population genetics for selection on a

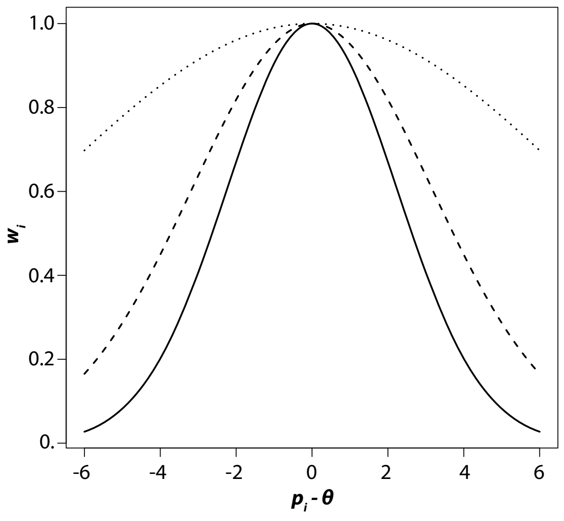


Fig. 9.9 The Gaussian fitness function. Fitness of individual *i* is plotted versus the difference between the phenotype of individual *i* and θ, the optimal phenotype. $s = 0.10$ (solid line); $s = 0.05$ (dashed line), and; $s = 0.01$ (dotted line)

single locus. In the latter case, relative fitness cannot fall below $1 - s$. In the context of the Gaussian fitness function, however, s controls the rate at which w_i falls as the absolute difference between p_i and θ increases. Moreover, we must account for the scale of the phenotype when choosing the value of s to simulate. For example, if the phenotype in question ranges between 1 and 2, the largest value of $p_i - \theta$ possible is 1—e.g., if the $p_i = 1$ and $\theta = 2$. Thus, a relatively larger value of s is required to simulate strong selection at this scale than for a selected phenotype that ranges between 100 and 200, in which case differences from θ may be two orders of magnitude larger.

Returning to the previous listing, lines 5–6 read in the values of θ and s, respectively, which are only used in the case of selection for an optimal trait value. Lines 7–9 define fitness according to the Gaussian fitness function. If evolutionary constraints are simulated, lines 9–16 calculate fitness. Fitness equals 1 whenever the phenotype of the considered individual is between the lower and upper phenotypic constraints—indices [1] and [2] of the `polysel` parameter, respectively. However, if the considered phenotype is below the lower bound (line 10) or above the upper bound (line 12), the difference between the phenotype and the appropriate bound is multiplied by another conception of selective strength—specified by a fourth entry to `polysel` parameter; index [3]. By this definition, fitness declines linearly as the distance from the lower or upper bound increases.

9.5.1 *Natural Selection Using a Gaussian Fitness Function*

To simulate natural selection on a quantitative trait using the fitness function of Eq. 9.8, it is necessary to list three values for the `polysel` parameter in the `parameters` file in this indexed order:

- [0] $\leftarrow$ 1
- [1] $\leftarrow$ value of θ, the targeted phenotype
- [2] $\leftarrow$ value of s, the 'strength' of selection

We now consider an example in which four demes are simulated (each with $N_e = 2500$), only *one* of which experiences natural selection. In this latter deme, optimal phenotype (θ) equals 0.5 and $s = 0.05$. In addition, `enviroSD` was set to 0.2 for all demes. Ten purely additive loci—each with an allele effect of 0.05 were simulated and all four demes began with identical genetic variation at these loci as generated by one run of MS. Finally, mutation rate was set to 1×10^{-6}; this assumes 100 sites at each causative locus that can switch in a time-reversible manner from state 1 to 0 or vice-versa. Clearly, this is a parameter value that requires careful consideration, but I decided on these assumptions for convenience. Figure 9.10a shows the results when there is no gene flow between the demes. As seen here the deme subject to natural selection increases toward the optimum, while the three demes in which the quantitative trait is neutral show three rather independent trajectories of mean phenotype. Figure 9.10b allows for symmetric migration between all demes at a rate of $m = 0.0004$; thus, $4N_em = 4 \times 2500 \times 0.004 = 4$. Although only one of the four demes is subject to natural selection, the high level of migration draws mean phenotype of all four demes upward toward the target phenotype of 0.5.

9.5.2 *Evolutionary Constraints on Quantitative Phenotype*

It is often assumed that a quantitative phenotype cannot fall below or rise above certain values due to hard biological or physical constraints that prevent manifestation of phenotypes outside these bounds. It is also possible, as we now model, that quantitative phenotypes outside these bounds are simply less fit. To simulate evolutionary constraints on a quantitative trait in a given deme, it is necessary to list *four* values for the `polysel` parameter in the `parameters` file in this indexed order:

- [0] $\leftarrow$ 2
- [1] $\leftarrow$ lower bound
- [2] $\leftarrow$ upper bound
- [3] $\leftarrow$ selective strength of constraint when outside these bounds

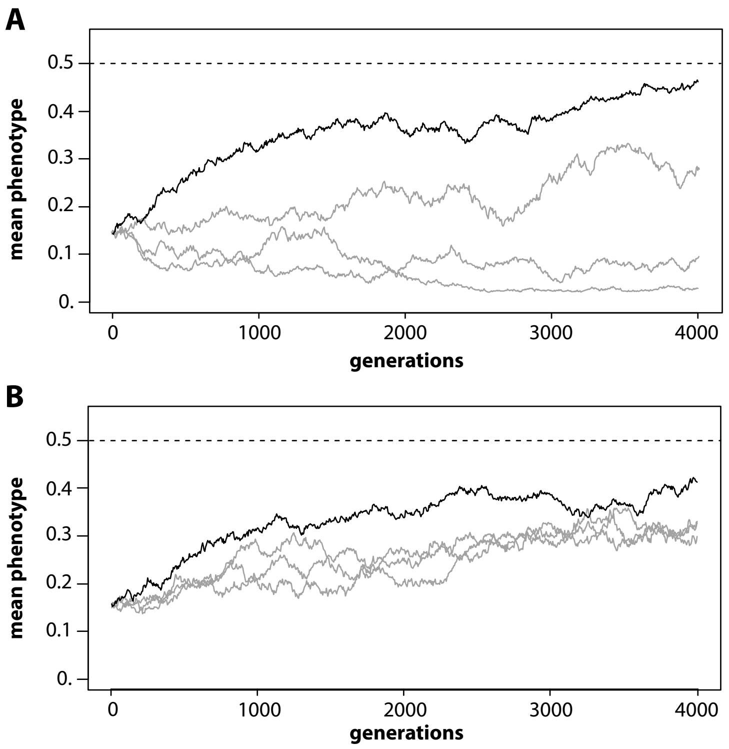


Fig. 9.10 Directional selection on a quantitative trait determined by variation at ten purely additive loci. (**a**) Four demes of $N_e = 2500$ each split from a single deme whose genetic variation was MS-generated. One of the four demes (black line) is subject to directional selection toward an optimal phenotype of 0.5. Per-locus mutation rate was set to 1×10^{-6}. **No migration** was simulated between the four demes. (**b**) The same as in (**a**), except that symmetric migration occurs between all four demes with great frequency ($4N_em = 4$). In the presence of gene flow, mean phenotype increases in all four demes toward the optimal value (dashed line), despite selective pressure present in only one deme (black line)

As described earlier, the relative fitness of an individual is set to its maximum of 1 for any individual whose phenotype is between the specified bounds.

In Fig. 9.11, all simulated demes begin with the same genetic variation at twenty purely additive loci generated randomly as described in Sect. 9.4. Allele effects were set to 0.05, mutation rate to 1×10^{-6}, and `enviroSD` to 0.2. In panel A, four demes with no selection were simulated. The random walk of mean quantitative phenotype is evident. In Fig. 9.11b,c, two pairs of demes were simulated: the first pair had lower and upper bounds of 0.4 and 0.45, respectively, while the second pair had lower and upper bounds of 0.45 and 0.5, respectively. The results shown in Fig. 9.11b are from simulations in which the selective strength of constraint was set to 0.025. The selective

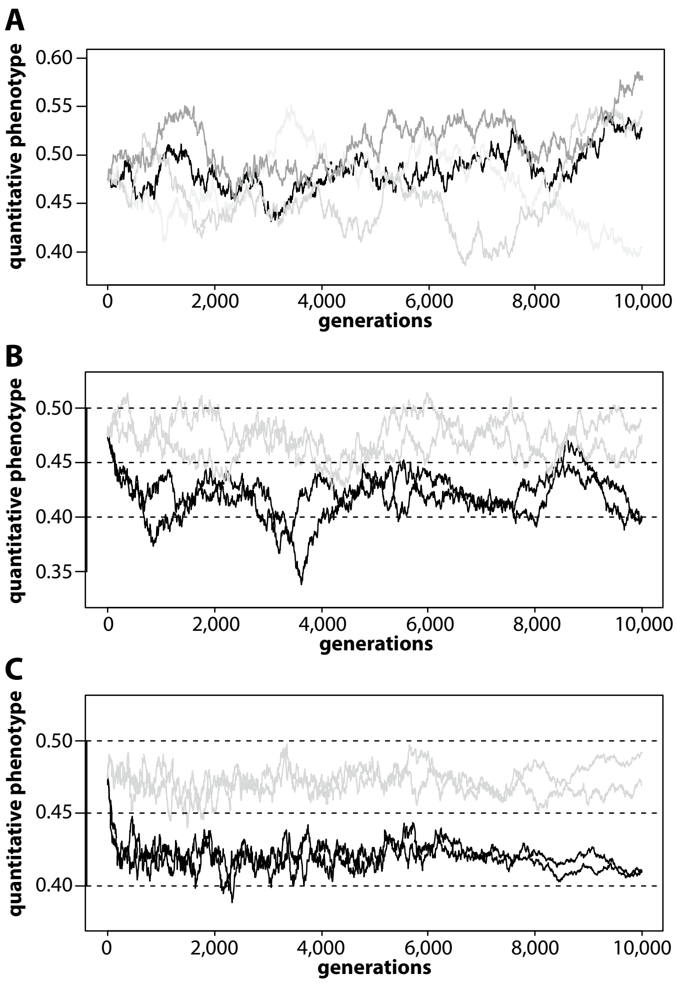


Fig. 9.11 Evolutionary constraints. In all cases `enviroSD` was set to 0.2, twenty purely additive loci with equal allele effects of 0.05 were simulated, and the *mean* phenotype is shown for four independent demes—i.e., no migration. All simulations began with the same starting "haplotype" file and, therefore, identical starting genetic variation. (**a**) Four independent simulations with no selection. (**b**) Weak constraint with the gray-line simulations having an upper constraint of 0.5 and lower constraint of 0.45 and black-line simulations having an upper constraint of 0.45 and lower constraint of 0.40. Selective strength of constraint set to 0.025. (**c**) Strong constraint. Same constraints as in (**b**), but with selective strength of constraint set to 0.10

strength of constraint for the simulation results shown in Fig. 9.11c was set higher to 0.1.

When examining Figs. 9.10 and 9.11 it is important to remember that the lines track the mean phenotype of each deme. Although mean phenotype rarely strays beyond the specified evolutionary bounds in any of the deme results pictured in Fig. 9.11c, a large number of individuals of any given deme in any given generation would be outside these bounds. If your intention is to model "hard" evolutionary constraints that represent, for example, physically impossible values, it is necessary to specify the strength of constraint as 1; this will assign a fitness of 0 to all individuals with phenotypes beyond these bounds.

An interesting pattern observed in Fig. 9.11c is that after generation 6000, the oscillations of mean phenotype become noticeably smaller in magnitude. This suggests that given sufficient time, evolutionary constraints lead to reduced genetic variation as more and more members of the deme obtain constrained phenotypes through very similar genotypes at the causative genes. As a corollary, this implies that fixation and loss of alleles at causative loci occurs randomly over time. You might try modeling dominance and epistatic effects, rather than purely additive effects, to see how these more unpredictable genetic determinants affect this pattern.

9.5.3 *Artificial Selection*

Truncation selection is an intuitive approach to artificial selection in which only those members of the current population with the most desirable phenotypes are selected to be parents of the next generation. In particular, the goal is one of directional selection, either attempting to increase or decrease the mean phenotype of the next generation. Thus, "truncation": we only recruit members in one tail or the other of the population's current phenotypic distribution to produce the next generation. To simulate truncation selection on a quantitative trait in a given deme, it is necessary to list three values for the `polysel` parameter in the `parameters` file in this indexed order:

- [0] ← 3
- [1] ← 1 to select for decreased phenotype, 2 to select for increased phenotype
- [2] ← the number of individuals in the low or high tail (depending on [1]) to retain as parents; the smaller this value the more selective—i.e., the more severe the truncation

The results of truncation selection are rapid, so it is only necessary to simulate a small number of generations to observe the effects. I ran two simulations in which the "herd" contained 100 individuals, `enviroSD` was set to 0.2, and the causative genetic loci were fifty purely additive loci with equal allele effects of 0.01. Each simulation was run for four generations. The only difference between the two simulations was whether smaller or larger phenotypes were selected; in either case the 10 smallest or largest individuals were retained as parents. In practice, it may be preferable to set `enviroSD` to

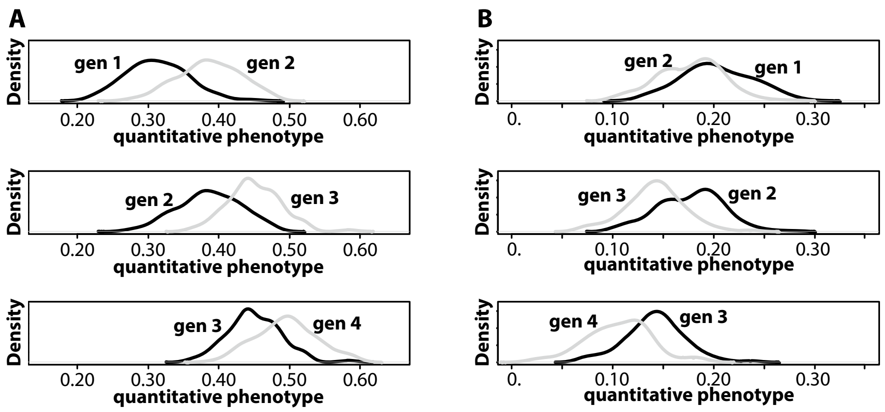


Fig. 9.12 Truncation selection shifts the phenotypic distribution rapidly in the case of purely additive loci. In all cases `enviroSD` was set to 0.2, fifty purely additive loci with equal allele effects of 0.01 were simulated. Each generation the "herd" consists of 100 individuals. (**a**) Artificial selection to *increase* quantitative phenotype. Black lines show the pre-selection kernel density estimate of the "herd" phenotypes estimated using the `density()` function in R. Gray lines show the post-selection kernel density estimate of the "herd" phenotypes. Each generation the 10 individuals with the greatest phenotype were selected to produce the next generation. (**b**) Artificial selection to decrease quantitative phenotype. Each generation the 10 individuals with the lowest phenotype were selected to produce the next generation

zero in simulations of truncation selection, particularly if you are modeling a situation in which all members of the "herd" share a common environment. Indeed, this would allow you to estimate narrow-sense heritability of the simulated trait using the well-known breeder's equation $R = h^2 S$, where R is the response to selection (the difference between mean phenotype of this and next generation) and S is the selection differential (the difference between the current generation's mean phenotype and that of the selected parents).

Design a series of rapid simulations to assess the impact of number of causative loci, variation in allele effects, dominance effects, and epistatic effects on the narrow-sense heritability of a trait.

Here, however, I have kept things simple to demonstrate the efficacy of (1) the code and (2) truncation selection on a purely additive trait. Figure 9.12a shows the steadily increasing distribution of phenotypes when the 10 parents with the greatest phenotypes parent the next generation. Three generations of truncation selection increase mean phenotype from around 0.3 to roughly 0.5. Selection for smaller phenotype is also effective. Figure 9.12b shows mean phenotype declining from roughly 0.2 to roughly 0.1 in three generations of truncation selection.

The examples of evolutionary constraints as well as artificial and natural selection presented here only provide a glimpse of what may be done with the FORTUNA code to simulate quantitative traits and their selection. As an obvious example, it will often be of interest to simulate dominance and epistatic effects as well as variation in allele effect size across causative loci. These tasks can all be accomplished by modifying the values in the `parameters` file. It may also be of interest to test fitness functions besides the Gaussian. This will require you to add to the `get_polygenic_fitness()` function of `Individual.h`, but hopefully the example of the Gaussian will make this a very simple task.

Appendix A
FORTUNA Parameter Documentation

Proper simulation of a scenario using FORTUNA requires the user to know which parameters need to be specified and the appropriate values to specify. Parameters labeled [[sequence simulation]] are introduced in Chaps. 3–6 and 8, while parameters labeled [[quantitative trait simulation]] are introduced in Chap. 9. Note, however, that most parameters associated with [[sequence simulation]] are still used in simulations of quantitative trait evolution. The distinction is made because quantitative trait simulation in FORTUNA uses some program data structures (particularly the variable `sequences` of class `Individual`) in a qualitatively different manner (see Chap. 9). Thus, [[quantitative trait simulation]] parameters only require consideration in simulations of quantitative trait evolution. The global parameters listed in Sect. A.1 are listed at the top of the `parameters` file and their values are only specified once. On the other hand, deme-specific parameters (Sect. A.2) must be listed for each Deme block in the `parameters` file. For some parameters, the distinct meaning and usage of the parameter in the context of sequence and quantitative trait simulation are detailed. Example `parameters` files specific to a variety of evolutionary scenarios can be found at github.com/deltafortuna and driftlessevolution.org.

A.1 Global Parameters

`alleleEffects (double, double, double, ...)`
9.2.2
quantitative trait simulation
The allele effect of each main locus. If only one number is provided, allele effects at all main loci are set to this number. Otherwise, the allele effect of *each* main locus must be specified. See Chap. 9 for suggestions on how to randomly generate allele effects for large numbers of loci using a Dirichlet random variable.

`baseTraitValue (double)`
9.2.2
quantitative trait simulation
Baseline quantitative phenotype to which allele, dominance, epistatic, and environmental effects are added to generate phenotype of each individual.

`diploid_sample (bool)`
6.3.3
sequence simulation
When set to true (1), samples and summary statistics are based on a random selection of diploid individuals. Otherwise, samples and summary statistics are based on a random selection of sequences from separate individuals.

`dominanceEffects (double, double, double, ...)`
9.2.2
quantitative trait simulation
The intra-locus dominance effect of each main locus, which applies when an individual is heterozygous at a main locus. If one number is provided dominance effects at all loci are equal to this number. Otherwise, the dominance effect of *each* main locus must be specified. See Chap. 9 for suggestions on how to randomly generate dominance effects using a Dirichlet random variable.

`enviroSD (double)`
9.2.2
quantitative trait simulation
Standard deviation of the Normal distribution from which environmental effects on quantitative phenotype are drawn. The square of this value is the environmental variance. Standard deviation is specified in FORTUNA because the second argument to the constructor of a C++ `normal_distribution` object is the standard deviation rather than the variance of the distribution.

`epistatic (bool)`
9.2.2
quantitative trait simulation
When set to `0` (`false`), epistatic effects are not simulated. When set to `1` (`true`), epistatic effects are simulated.

`epistaticTypes (int, int, int, ...)`
9.2.2
quantitative trait simulation
The type of epistasis for each pair of main and epistatic loci. While `seqlength` specifies the number of main loci, the number of `int`s listed here specifies the number of main loci whose additive allele effects are modified by epistatic effects of the types listed here (see Figure 9.7). The following values are used to specify the different types of epistasis (see Figures 9.5–9.6 and accompanying text):

- 0: dominance (text specific example)
- 1: recessive (text specific example)
- 2: additive by additive (AxA)
- 3: additive by dominant (AxD)
- 4: dominant by additive (DxA)
- 5: dominant by dominant (DxD)

Any combination of epistatic types can be listed here, as long as the number does not exceed `seqlength`.

`getWindowStats (double);`
5.3.1
sequence simulation
Set to 1 to enable printing of by-window summary statistics to file every `sampfreq` generations. Set to 0 to disable printing.

`hotrecrate (double);`
5.3.1
sequence simulation
The per-site recombination rate within a defined recombination hot spot. It is easiest to specify the value using exponent scientific notation – e.g., `1e-06` indicates one cross-over at a given *site* every $1/10^{-6}$ generations.

`hotrecStart (int);`
5.3.1
sequence simulation
The first site within the simulated sequence of length `seqlength` that is the upstream border of a simulated recombination hot spot.

`hotrecStop (int)`
5.3.1
sequence simulation
The last site within the simulated sequence of length `seqlength` that is the downstream border of a simulated recombination hot spot.

`migration_rates (double, double, double, ...)`
6.2.1.1
sequence simulation
A sequence of real numbers that specify the entries of a `pop_num` $\times$ `pop_num` migration rate matrix. Values entered here are for the migration rate $m_{i,j}$, the fraction of individuals in population j that are immigrants from population i. The first `pop_num` entries of this parameter are interpreted as the first row of the migration rate matrix and so on. As an example,

```
migration_rates 0 0.001 0.0001 0.0005 0 0.0002 0.0015 0.00005 0
```

specifies a migration matrix of:

$$\begin{bmatrix} 0. & 0.001 & 0.0001 \\ 0.0005 & 0. & 0.0002 \\ 0.0015 & 0.00005 & 0. \end{bmatrix}$$

for `pop_num 3`.

`modelMigration (bool))`
6.2.1.1
Set to `1` to simulate migration. Set to `0` to disable simulation of migration between demes.

`mutrate (double)`
3.4.1
sequence simulation
The per-site mutation rate. This can be represented as a decimal or in scientific (exponent) notation – i.e., 0.00000005 or 5e-08.
quantitative trait simulation
The probability of a mutation at each main and epistatic locus simulated. For example, if set to 1e-06, each locus has a one-in-one-million chance of mutating (from 0 to 1 or 1 to 0, depending on its current state). The reason for using a higher mutation rate when simulating quantitative trait simulation is that we envision each QTL as a protein-coding gene in which there are multiple sites that can mutate to change the QTL's effect on phenotype. Thus, at a per-site mutation rate of 1×10^{-8} where 100 sites are capable of bringing about a shift in effect when mutated, the mutation rate is $100 \times 10^{-8} = 10^{-6}$. In short, I am assuming a per-locus rather than a per-site mutation rate when simulating quantitative trait simulation. Set `mutrate` to a lower value if you do not want to make this assumption.

`polygenicTrait (bool)`
9.2.2
quantitative trait simulation
Set to 1 to simulate quantitative trait evolution rather than sequence evolution. Set to 0 to simulate sequence evolution.

`pop_num (int)`
6.2.1.1
sequence simulation
The number of demes/populations to simulate. If `pop_num` is less than the number of deme-specific parameter blocks, only the parameters in the first `pop_num` parameter blocks are used.

`printhapfreq (int)`
6.2.1.1
Haplotype (sequence) samples are printed to file every `printhapfreq` generations.

`recrate (double)`
5.3.1
sequence simulation
The "baseline," per-site recombination rate *outside of* a defined recombination hot spot, should one be specified. It is easiest to specify this value using exponent scientific notation – e.g., `1e-08` indicates one cross-over at a given *site* every $1/10^{-8}$ generations.

`runlength (int)`
6.2.1.1
sequence simulation
The number of generations for which the simulation should run. Note that this is the run length of the *total* simulation. Individual demes have their own time periods of existence that are equal to or less than `runlength`.

`sampfreq (int)`
3.4.1
Samples of all extant populations are taken every `sampfreq` generations.
sequence simulation

`sampsize (int)`
5.3.1
sequence simulation
The number of individuals randomly sampled from each population in sampling generations.

`seqlength (int)`
5.3.1
sequence simulation
The physical length (in nucleotides) of the sequence to be simulated.
quantitative trait simulation
The number of main loci *plus* the number of epistatic loci.

`trackAlleleBirths (bool)`
6.2.1.1
When set to 1, the generation in which an allele emerges by new mutation and the deme in which it first emerged are printed to file. Although this can be highly useful data for certain research questions, keeping track of these data is computationally expensive. This allele history file is not generated if set to 0, which is recommended if the history of alleles is not deemed important output.

`useHotRec (bool)`
5.3.1
sequence simulation

When set to 1, a recombination hot spot is simulated. When set to 0, the recombination hot spot specified by `hotrecrate`, `hotrecStart`, and `hotrecStop` is ignored and not simulated.

`useRec (bool)`
5.3.1
sequence simulation
When set to 1, recombination is simulated. When set to 0, recombination is not simulated.

`windowSize (double)`
5.3.1
sequence simulation
The length (in nucleotides) of each window for which statistics are calculated when `getWindowStats` is set to 1 (`true`).

`windowStep (double)`
5.3.1
sequence simulation
The number of nucleotides the window should advance when `getWindowStats` is set to 1 (`true`). For example if `seqlength` is 10,000, windowSize is `2000`, and `windowStep` is `1000`, then statistics for the following windows will be calculated; [0, 2000); [1000, 3000); [2000, 4000); [3000, 5000); [4000, 6000); [5000, 7000); [6000, 8000); [7000, 9000); [8000, 10000).

A.2 Deme-Specific Parameters

`birthgen (int)`
6.2.1.1
sequence simulation
Not to be confused with the variable `Allele::birthgen`, this is the first generation that simulation of the deme in question begins.

`carrying_cap (int, ...)`
4.2.1
Contains as many elements as `demography`. Elements of this vector are relevant when the entry of `demography` equals 4, which symbolizes logistic growth. For example, if we set the following parameters

```
demography 0 1 4
```
and
```
carrying_cap 15000 10000 50000,
```

`50000` will be used as carrying capacity upon entering the third demographic phase. The values of the first two numbers of `carrying_cap` are irrelevant in this case. `carrying_cap 0 100 50000` would accomplish the same thing.

`dem_end_gen (int, ...)`
4.2.1
sequence simulation
Contains as many elements as `demography`. Each entry specifies the last generation of a phase in the demographic change of a deme. In combination with `dem_start_gen`, defines the time period of each phase of demographic change.

`dem_parameter (int)`
4.2.1
sequence simulation
Contains as many elements as `demography`. Each entry specifies a parameter used with the type of demographic change specified by the element of the same index in `demography`.

`dem_start_gen (int, ...)`
4.2.1
sequence simulation
Contains as many elements as `demography`. Each entry specifies the first generation of a phase in the demographic change of a deme. In combination with `dem_end_gen`, defines the time period of each phase of demographic change.

`demography (int, ...)`
4.2.1
For a deme, the series of *types* of demographic change it experiences. The following integers specify the corresponding type of change, if any.

- `0` no change in size
- `1` instantaneous change
- `2` linear change
- `3` exponential change
- `4` logistic change

For example, `demography 0 3 0` indicates the deme in question remains at a constant size initially, followed by exponential change in size, after which constant population size is renewed. The parameters `dem_start_gen`, `dem_end_gen`, and `dem_parameter` specify the quantitative details of these changes (or lack thereof). See section 4.2 for more information and other examples.

`extinctgen (int)`
6.2.1.1
sequence simulation

One generation *after* the deme in question stops being simulated, either because this number is less than the global parameter `runlength`, the deme merges with another, or it is lost to a split into two demes.

`hapFile (string)`
9.3
sequence simulation
The full name of the file that contains haplotype data from a previous simulation from which another simulation is initiated when `useHapStartingData` rather than `useMS` is true.

`mergegenesis (int, int, int)`
6.3.1
sequence simulation
Controls creation of a deme through merging of two extant demes. To specify a merger origin, the first element is set > 0. The second element is the first source deme, while the third element of the parameter is the second source deme. For example,

```
mergegenesis 10 1 2
```

indicates that the deme originates by the merger of demes 1 and 2.

`mscommand (string)`
3.7
sequence simulation
The full MS command-line call. Do not place in quotations. If necessary, include the pathway to the program. For example,

```
./ms 100 1 -t 4
```

`negsel (double, double, double, double, double)`
8.1, 8.3
sequence simulation
Used to simulate purifying natural selection in a specified subsequence. The first two elements of the parameter are selection coefficients s and h, respectively. The next two elements specify the first and last base pair of the subsequence subject to purifying selection. The last element is the number of sites within the subsequence whose positions will be randomly chosen where the derived ″1″ allele is the deleterious allele. For example,

```
negsel 0.5 0.1 40000 60000 150
```

specifies a 20,000bp subsequence from base pair 40,000 through base pair 60,000. At 150 randomly selected positions/sites within the subsequence, relative fitnesses are $w_{0/0} = 1$, $w_{0/1} = 1 - hs = 0.95$, and $w_{1/1} = 1 - s = 0.5$. Purifying selection is simulated whenever the first elements value is greater than 0. Make sure that other selection parameters (`ndfsel`, `possel`) are set to 0 at their first elements; this avoids conflicting signals of selective targets.

`nfdsel (double, double)`
8.1, 8.2.5
sequence simulation
Used to simulate negative frequency-dependent selection in which a variant is favored when at low frequency and vice-versa. The first element is the coefficient s, while the second element is the coefficient h. See section 8.2.5 for a discussion of the relative fitness functions for negative frequency-dependent selection.

`polysel (int, int, int, ...)`
9.5
quantitative trait simulation
Used to perform one of three types of selection on a quantitative trait. The first element indicates which of the three types to simulate: `1` for directional selection toward an optimal phenotype according to a Gaussian fitness function; `2` for evolutionary constraints beyond which a fitness penalty is incurred; `3` for the truncation (artificial) selection.
When simulating directional selection, the second element is the optimal/-targeted trait value, and the third element is the strength of selection, s. When simulating evolutionary constraints, the second and third elements are the upper and lower bounds on trait value, respectively. In addition, a fourth element should be specified, which is equal to selective strength, s. Finally, when simulating truncation selection, the second element of parameter `polysel` is set to 1 if selecting for a decreasing trait value and set to 2 if selecting for an increasing trait value. The third element of the parameter in this case is the *number* of individuals to choose as parents of the next generation; these will either be individuals of the current generation that have the greatest or smallest trait values.

`popsize (integer)`
3.4.1, 6.2.1.1
The effective population size of diploid individuals. Initially, `popsize` is introduced as a global parameter. In the complete version of FORTUNA, each deme has its own `popsize`.

`possel (double, double, double)`
8.1
sequence simulation
Used to simulate positive selection on a specified site within the simulated sequence. Both selection on a new variant or a standing variant of specified frequency can be simulated using this parameter. In addition, overdominance selection can be simulated with this parameter. Parameter values indicating the position of the selected site are supplied to parameter `sellocus`. In the context of positive natural selection, the derived ”1” allele represents

the adaptive allele. The three elements of `possel` are the values of s, t, and h, respectively. The parameter value t is only used when simulating overdominance selection.

Examples of `possel` usage:

```
possel 0.1 0. 0.5,
```

specifies the selective regime, $w_{1/1} = 1$, $w_{0/1} = 1 - hs = 0.95$, $w_{0/0} = 1 - s = 0.9$

```
possel 0.1 0.15 0.,
```

specifies the selective regime, $w_{1/1} = 1 - s = 0.9$, $w_{0/1} = 1$, $w_{0/0} = 1 - t = 0.85$ (overdominance)

`sellocus (double, double, double, double, double)`
8.1
sequence simulation
Used when simulating positive natural selection (including negative-frequency-dependent selection and overdominance) to specify the characteristics of the targeted site. The first element specifies the (preferred) position of the selected site. The second element of the parameter is the *count* of the adaptive, derived ′1′ allele at the beginning of the simulation. If set to 1, the starting frequency of the adaptive allele is $p = 1/(2\times$`popsize`$)$; this simulates selection on a new variant generated by mutation in the previous generation. If the second element is set to a value/count greater than 1, then simulation on standing variation is requested. Because the standing variation at the beginning of the simulation is drawn from an MS simulation or a haplotype file, there is great chance that the preferred position of the selected site is not polymorphic. The solution used in FORTUNA is to scan the polymorphic sites of MS output or haplotype file for the one that most closely matches the position specified in the first element of the `possel` parameter; the closest match is the polymorphic site that minimizes (1) distance from the preferred position of the selected site and (2) difference between its derived allele frequency and the preferred frequency of the derived allele implied by its starting count. The fifth element of the `sellocus` parameter should be set to 1 or 0. Setting it to 1 causes the simulation to start over if the adaptive allele is lost. When set to 0, the simulation will continue despite loss of the adaptive allele.

Examples of `sellocus` usage:

```
sellocus 500000 1 0. 0. 1
```

specifies positive selection on the derived allele at base pair 500,000, which begins as a single copy in the deme. In addition, the simulation will start over (overwriting output files from failed simulations) any time the adaptive allele is lost.

```
sellocus 500000 250 0.05 0.0125 1
```

specifies positive selection on a standing variant with a starting frequency of $p = 250/(2\times$`popsize`). The third element indicates that a derived allele from an MS output or haplotype file within 5% of the preferred position at base pair 500,000 is acceptable as a selective target – i.e., any position on the domain [450000, 550000]. The fourth element indicates that a derived allele from an MS output or haplotype file with a frequency within 0.0125 of the preferred starting allele frequency of $p = 250/(2\times$`popsize`) is acceptable – e.g., if `popsize` equals 10,000 diploid individuals, the desired starting allele frequency is $p = 250/20000 = 0.0125$ and any polymorphic site with a derived allele frequency on the domain [0., 0.025] is acceptable.

`splitgenesis (int, int, int)`
6.3.1
sequence simulation
Controls creation of a deme through the splitting of an extant deme. To specify a split origin, the first element is set > 0. The second element is the source deme that is splitting, while the third element of the parameter is the percentage of individuals in the source deme that are randomly assigned to the new deme. Note that another new deme should be specified as well to receive the remaining percentage of individuals from the source deme. For example,

```
splitgenesis 1 5 45
```

indicates that the deme originates through receipt of 45% of the individuals in deme 5.

`useHapStartingData (bool)`
9.3
sequence simulation
When set to 1, set `useMS` to 0, and specify the name of the input haplotype file using parameter `hapFile`. Setting this parameter true directs the program to read starting variation data from a haplotype file produced by a previous run of FORTUNA. When set to 0, starting variation is drawn from MS simulation when `useMS<-1` or the deme(s) begin(s) with no variation when `useMS<-0`.

`useMS (bool)`
3.7
sequence simulation
When set to 1, set `useHapStartingData` to 0, and specify the MS command to run as the entry to the `mscommand` parameter. Setting this parameter true directs the program to run `mscommand` and use the output file `ms_output` as the source of starting genetic variation.