K

A system for modeling the population dynamics of multilocus mutation load,

based on a model originally developed by Alexei Kondrashov

Douglas G. Scofield[1]
Department of Plant Physiology
Umeå University, Umeå, Sweden

August 22, 2012

[1]Supported by grants from the NSF and the Department of Biology, University of Miami

Abstract

The basic structure of the **K** software package is described. **K** may be used to construct models, based on the description of Kondrashov (1985), that may be used to study the population dynamics of multilocus mutation load. **K** is designed to allow the relatively easy reproduction of newer models based on Kondrashov's original description. **K** also facilitates the construction of new extensions to these models, through the use of novel fitness functions, different sequences of per-generation events, and the specification of additional model parameters. Furthermore, **K** allows the construction of models based on two mutation classes. **K** is covered by the Gnu Public License and will be placed in the `SourceForge.net` open-source project repository.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

K is a software package designed to enable the study of mutation-selection balance and the population dynamics of genetic load. **K** is based on an iterative, deterministic model described by Kondrashov (1985), and possibly derived from Heller & Maynard Smith (1979). The original model has been expanded upon by several authors since (e.g., Charlesworth *et al.* 1990; Lande *et al.* 1994; Muirhead & Lande 1997; Morgan 2001, and others). **K** is designed so that each of these previous models may be easily reproduced.

In the infinite-population model of Kondrashov (1985), genomes are represented by an infinite number of unlinked diploid *load loci*, with mutation carried by each having an identical effect on fitness. Populations are represented by proportional membership within *load classes* $L_{i,j}$, within which individuals carry i heterozygous mutations and j homozygous mutations. Changes in load class membership due to mutation and reproduction are based upon probabilistic expectations from Poisson and binomial distributions, respectively. Changes in load class membership due to selection are based upon differences in relative fitness. The model is run until an equilibrium distribution of load class membership is reached, or until it is apparent that an equilibrium will not be reached within a preset number of model generations using the current model configuration.

Load classes may be subdivided according to *functional genotype*, which is determined by one to several loci unlinked to the load loci and without a direct effect on fitness. The functional genotype is generally defined to control characteristics of the mating system. Several studies have used this model to explore the consequences of associations between fitness, as determined by load classes, and functional genotypes (Kondrashov 1985; Charlesworth *et al.* 1990, and others). "Genotype" in the traditional sense is thus represented by two parts:

- A *load class* portion subject to recurrent mutation that only has a fitness effect; and
- A *functional* portion that determines mating characteristics, etc. and is not subject to mutation.

Here, the load class is indicated by subscripts on symbols $x_{i,j}$ and the functional genotype by Greek letters in parentheses, e.g. $x_{i,j}(\gamma)$.

Mutations are assumed to always be unique. As a result, new mutation only increases the number of heterozygous mutations in a population and does not affect the number of homozygous mutations. Similarly, identity by descent other than that caused by selfing within a generation is not possible. When an outcrossed zygote is formed from the union of two haploid gametes carrying m and n mutations, respectively, the new outcrossed zygote carries $m + n$ heterozygous mutations and 0 homozygous mutations. The distribution of zygotes produced via selfing by individuals carrying i heterozygous mutations and j homozygous mutations is more complex (see Equation [3.1.1]).

The population is described by proportional membership in load class-genotypes $L_{i,j}(\gamma)$ where $i \in (0, M_i)$ and $j \in (0, M_j)$ describe the load portion of the genome and γ is the corresponding genotype of the genotype portion of the genome. Note that in the discussion that follows, the notation $L_{i,j}$ indicates a general statement about the

load class containing i heterozygous mutations and j homozygous mutations and applies to all load class-genotypes belonging to that load class. Similar notational conventions will be used for other model terms.

An analysis that does not consider selfing (that is, that examines outcrossing and/or apomixis) will never encounter homozygous mutations. Kondrashov (1985) provides representative formulae. Here I will assume the need to keep track of both allelic states.

The model cannot be solved analytically (Kondrashov 1985), although simplified recursion equations are available in Schultz & Willis (1995). Instead, the model is specified in a computer program and allowed to iterate until convergence to equilibrium is determined numerically. Equilibrium may be obtained within a few hundred generations, depending upon initial conditions (Charlesworth *et al.* 1990).

The sequence of steps performed, with the notation to be used here for the $L_{i,j}(\gamma)$ proportions resulting from each step, is:

1. reproductive adults, $x_{i,j}(\gamma)$
2. mutation, $x'_{i,j}(\gamma)$
3. reproduction, total progeny $x''_{i,j}(\gamma)$
    (a) production of selfed progeny, $x''_{(S)i,j}(\gamma)$
    (b) production of apomictic progeny, $x''_{(A)i,j}(\gamma)$
    (c) production of outcrossed progeny, $x''_{(O)i}(\gamma)$
4. progeny following selection, $X_{i,j}(\gamma)$
5. next generation of reproductive adults, $x_{i,j}(\gamma)$, and the previous generation of reproductive adults, $x_{(\mathrm{prev})i,j}(\gamma)$.

1.1 Choosing M_i and M_j

Model execution times may be long because of the need for repeated matrix operations to determine membership in each $L_{i,j}(\gamma)$ resulting from each step. Convergence analysis as a shortcut to equilibrium might be possible (M. Morgan, pers. comm.), but has not yet been attempted. Several algorithms in the model have baseline execution time and space requirements of at least $\mathbf{O}({M_i}^2 \times {M_j}^2)$, thus it is critical not to choose M_i and M_j that are unnecessarily large.

The values of M_i and M_j should be chosen based upon classical expectations for frequencies of heterozygous and homozygous deleterious alleles under selection, such that membership in classes where $i > M_i$ or $j > M_j$ is highly unlikely. For example, values of M_i must be greater if mutations have both low dominance and low selective effect, while values of M_j may be very low for highly recessive lethal mutations. Due to selection, there is no strict requirement that $M_j = M_i$ as would be expected for neutral mutations. One implementation known to the author uses $M_i = 200$ and $M_j = 40$. Neither Kondrashov (1985) nor Charlesworth *et al.* (1990) explicitly note the need to determine M_i and M_j.

1.2 Initiating and running K

The start of a **K** session begins with the creation and initiation of the `KConfig` structure (typically named `K`). Below, `initiate_Kconfig()` creates a `KConfig` structure, and the subsequent statements create the arrays describing the load classes, fill in the per-diploid genome per generation mutation rate, fitness function, and fitness parameters desired, as well as reproductive parameters such as selfing rate.

```
KConfig K;
K = initiate_KConfig();
initiate_load_classes(K, MI, MJ);
K->U = 1.0;
```

```
K->fitness_function = FITNESS_MULTIPLICATIVE;
K->fit_s = 1.0;  /* highly recessive lethals */
K->fit_h = 0.02;
K->S = 0.0;
set_repro(K, K->S, 0.0);
```

Once the `K` structure is initialized, a number of subsequent initialization steps are accomplished using a single function.

```
initiate_model_state(K);
```

Once the model state is ready, then execution of the model begins. Because this is an iterative model that is declared finished when a suitable equilibrium is approached, model execution occurs within a loop that checks for equilibrium at each iteration.

```
compute_adults_initial(K);
while (! is_equilibrium(K)) {
    if (K->generation > GENERATION_CUTOFF)
        break;
    if (K->option_truncate)
        truncate_KArray(K, K->x, LOADCLASS_TRUNCATE);
    compute_mutation(K);
    compute_self_progeny(K);
    compute_apomixis_progeny(K);
    compute_gametes(K);
    compute_zygotes(K);
    compute_summed_progeny(K);
    compute_fitness(K);
    compute_adults_nextgen(K);
    normalize_KArray(K, K->x);
}
```

Each of the stages creates results in a separate array that is part of `K`, and these arrays are recycled each generation. Descriptions of the operations occurring within most of these functions are given below. **K** assumes that the order of execution (and thus preferences for use of arrays in `K`) is identical to that used in Charlesworth *et al.* (1990); **K** provides the `compute` functions that implement these assumptions. However, these stages may be performed in any order. For example, if as in Muirhead & Lande (1997)

additional cycles of mutation were desired prior to sexual reproduction, then calls to the function `apply_mutation()` would follow (or replace) the call to `compute_mutation()`.

Following the attainment of equilibrium, statistics are computed from the contents of the arrays.

```
stats_all(K);
stats_print(K);
```

1.3 Duplicating results from published studies

The above model initiation and execution, with appropriate modifications to mutation parameters, may be easily used to duplicate published results. For example, Lande *et al.* (1994) examined the equilibrium consequences of high mutation rate to highly recessive lethal alleles and found patterns important to other areas of my research Scofield & Schultz (e.g. 2006). By using parameters identical to those used by Lande *et al.*, identical results were achieved (Figures 1.1, 1.2).

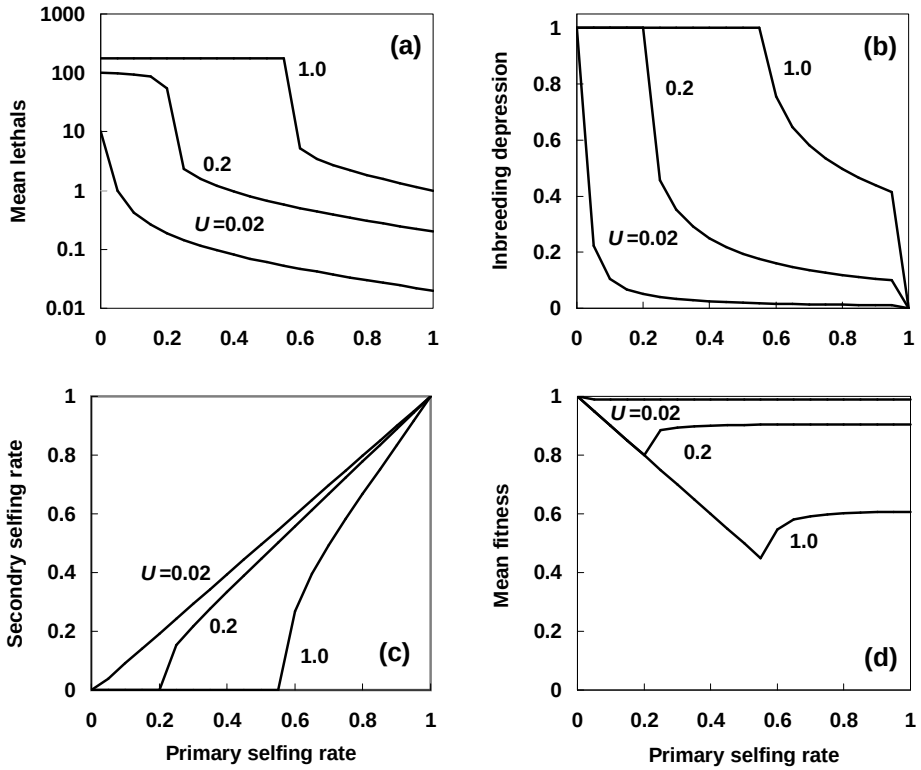

Figure 1.1: Duplicate of the results reported by Lande *et al.* (1994) in their Figure 2, for selection coefficient $s = 1.0$ and dominance $h = 0.0$ at the given mutation rate and range of selfing rates. The sudden drop in inbreeding depression at a primary selfing rate of 1 visible in panel (b) is an artifact of the method used to compute inbreeding depression. As is apparent in panel (d), this does not reflect rapid changes in mean population fitness.

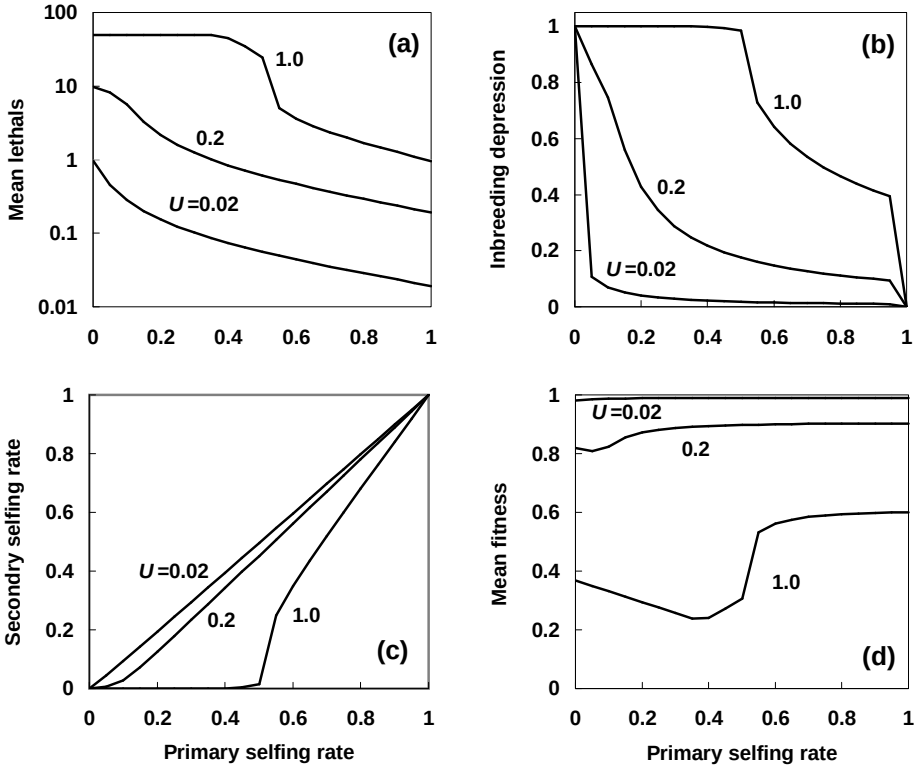

Figure 1.2: Duplicate of the results of Lande *et al.* (1994) in their Figure 3, for selection coefficient $s = 1.0$ and dominance $h = 0.02$ at the given mutation rate and range of selfing rates. The same caution applies for panel (b) as in Figure 1.1.

Chapter 2

Mutation

In **K** each new mutation is assumed to be unique. Therefore, each new mutation increases the number of heterozygous mutations in the population and does not affect the number of homozygous mutations. The mutation rate U is per-diploid genome, per-generation, and is taken to specify the mean of a Poisson distribution Poi(U) specifying the distribution of the number of new mutations (Kondrashov 1985; Charlesworth *et al.* 1990, and many others). The proportion of load class $L_{n,j}$ that is moved to load class $L_{i,j}$ (where $n \leq i$) as a result of mutation is equal to the probability that $i-n$ mutations occur, and is thus the value of the $i-n$-th term of Poi(U). The total proportion of the population in $x'_{i,j}(\gamma)$ after mutation is:

$$x'_{i,j}(\gamma) = \begin{cases} \sum_{n \leq i} e^{-U} \frac{U^{i-n}}{(i-n)!} \cdot x_{n,j}(\gamma) & U > 0 \\ x_{i,j}(\gamma) & U = 0 \end{cases} \tag{2.0.1}$$

There is currently no provision for specifying an alternative mutation function as there is for fitness (section 4.3).

2.1 Precomputed mutation terms

For computational efficiency, **K** automatically calculates values for each term $t \in (0, M_i)$ of Poi(U), where $t = i - n$ as in equation (2.0.1), and stores these values in the array `K->mut_term[t]`. The function `mut_term(K,t)` returns the corresponding element of this array.

Precomputed mutation values are established as a consequence of calling the function `initiate_model_state()` just prior to starting the model. The mutation rate `K->U` must have the desired value prior to this time. The function `mut_term_computed(K,x)` may be used to bypass the precomputed values and calculate the mutation term directly.

Chapter 3

Reproduction

The structure of **K**'s modeling of reproduction reflects the independence of the load class and genotype. *Mating functions* define the result of different types of matings for both load classes and functional genotypes. The membership in load classes following reproduction depends upon the mating functions for load classes and functional genotypes involved, the population rate of the mating system involved, and membership in load classes involved in matings.

The mating functions for selfing and apomixis reflect the fact that just one parental genotype is involved in the production of progeny. The mating functions for outcrossing, on the other hand, require two parental genotypes.

3.1 Load class mating functions

Load class mating functions for selfing and apomixis follow Kondrashov (1985):

$$s_{i,j}(n,v) = \begin{cases} \binom{n}{i}\binom{n-i}{j-v} \cdot \left(\frac{1}{2}\right)^{2n-i} & i \in (0,n),\ j \in (v, v+n),\ i+j \leq n+v \\ 0 & \text{otherwise.} \end{cases} \tag{3.1.1}$$

$$a_{i,j}(n,v) = \begin{cases} 1 & i = n,\ j = v \\ 0 & \text{otherwise.} \end{cases} \tag{3.1.2}$$

For outcrossing, the modification of Charlesworth *et al.* (1990) is used, as it is much more efficient than the function given in Kondrashov (1985). Note, though, that there are typographical errors in Charlesworth *et al.*'s Equation (3). This method first forms pools of haploid male and female gametes, and the proportion of available female gametes is weighted by the outcrossing rate. A function for production of gametes with functional genotypes is required, as is a function for production of functional genotypes for zygotes formed from two genotyped gametes:

$$f_i = \begin{cases} \sum_{v=0}^{i} \sum_{n=i-v}^{M_i} \binom{n}{i-v} \left(\frac{1}{2}\right)^n x'_{n,v} & v \leq M_j \\ 0 & \text{otherwise} \end{cases} \tag{3.1.3}$$

$$f_i^{\female} = O(\gamma) \cdot f_i \tag{3.1.4}$$

$$f_i^{\male} = f_i \tag{3.1.5}$$

Zygotes are formed by matings among male and female gametes:

$$x''_{(O)i}(\gamma) = \frac{1}{2} \sum_{k=0}^{i} f_k^{\female} f_{i-k}^{\male} + f_{i-k}^{\female} f_k^{\male} \tag{3.1.6}$$

Table 3.1: Mating functions based upon perfect Mendelian segregation for functional genotypes controlled by a single biallelic locus as described in the text. Note that for $\mathcal{T}_{(S)\gamma}(\xi)$ and $\mathcal{T}_{(A)\gamma}(\xi)$ each submatrix represents the resulting proportion of the progeny that carry each functional genotype ($\gamma_1\ \gamma_2\ \gamma_3$), and that the proportions in each submatrix sum to 1.

$\mathcal{T}_{(S)\gamma}(\xi)$	
ξ_1	$(1\ 0\ 0)$
ξ_2	$(\frac{1}{4}\ \frac{1}{2}\ \frac{1}{4})$
ξ_3	$(0\ 0\ 1)$

$\mathcal{T}_{(A)\gamma}(\xi)$	
ξ_1	$(1\ 0\ 0)$
ξ_2	$(0\ 1\ 0)$
ξ_3	$(0\ 0\ 1)$

$\mathcal{T}_{(O)\gamma}(\alpha,\beta)$	β_1	β_2
α_1	γ_1	γ_2
α_2	γ_2	γ_3

3.2 Genotype mating functions

The genotype mating functions shown in Table 3.1 likewise follow Kondrashov (1985). For the outcrossed genotype mating function $\mathcal{T}_{(O)\gamma}(\alpha,\beta)$, the resulting genotype γ_i is the Mendelian product of both parents' genotypes.

The tables described here apply to genotypes determined by a single diploid locus. Alternative genotype mating functions may be easily defined that determine these genotypes based on other assumptions (e.g., more than one locus).

3.3 Net effect of mating

Note that each genotype γ_i may have its own selfing, apomixis and outcrossing rate $S(\gamma_i)$, $A(\gamma_i)$, and $O(\gamma_i)$, respectively.

$$x''_{(S)i,j}(\gamma) = \sum_n \sum_v s_{i,j}(n,v) \sum_\xi \mathcal{T}_{(S)\gamma}(\xi) \cdot S(\xi) \cdot x'_{i,j}(\xi) \tag{3.3.1}$$

$$x''_{(A)i,j}(\gamma) = \sum_n \sum_v a_{i,j}(n,v) \sum_\xi \mathcal{T}_{(A)\gamma}(\xi) \cdot A(\xi) \cdot x'_{i,j}(\xi) \tag{3.3.2}$$

$$x''_{(O)i}(\gamma) = \sum_n \sum_v \sum_l \sum_m o_{i,j}(n,v,l,m) \sum_\xi \sum_\zeta \mathcal{T}_{(O)\gamma}(\alpha,\beta) \cdot O(\xi) \cdot x'_{i,j}(\xi) \tag{3.3.3}$$

Total progeny frequencies are simply the sum of the proportions of selfed, apomictic and outcrossed progeny:

$$x''_{i,j}(\gamma) = \begin{cases} x''_{(O)i}(\gamma) + x''_{(S)i,j}(\gamma) + x''_{(A)i,j}(\gamma) & j = 0 \\ x''_{(S)i,j}(\gamma) + x''_{(A)i,j}(\gamma) & \text{otherwise} \end{cases} \tag{3.3.4}$$

Chapter 4

Fitness

In **K**, just as in nature, selection reduces proportions of less fit genotypes. Fitness in **K** is typically a function of the mutations carried in the load class $L_{i,j}$, together with any user-specified parameters carried in the `KConfig` structure `K`. In order to maintain the infinite population at constant sum 1, the application of selection reduces the proportions of load classes below the population mean fitness and increases the proportions of load classes above the population mean fitness. Fitness may also be dependent upon the genotype γ, but this has not typically been considered.

4.1 Specifying the fitness model

Kondrashov (1985) specified a fitness model that enabled threshold selection:

$$\omega_{i,j} = 1 - \left(\frac{1+dj}{k}\right)^{\alpha}. \tag{4.1.1}$$

The numbers of heterozygous and homozygous mutations are specified by i and j, respectively; individuals with $> k$ mutations died; d specifies dominance with $d = 2$ equal

to codominance; and $\alpha = 1$, 2, and ∞ correspond to linear, intermediate and threshold selection, respectively.

At the moment, **K** does not provide Kondrashov's fitness function. Instead, it provides the familiar multiplicative fitness function parameterized by selection coefficient $s \in (0, 1)$ and dominance $h \in (0, 1)$. The fitness of a given genotype is thus:

$$\omega_{i,j} = (1 - hs)^i (1 - s)^j. \tag{4.1.2}$$

Mean population fitness is determined by:

$$\bar{\omega} = \frac{\sum_i \sum_j \omega_{i,j} x''_{i,j}}{\sum_i \sum_j x''_{i,j}} \tag{4.1.3}$$

and is returned by the function `mean_fitness(K,K->xpp)`.

4.2 Precomputed fitness

For computational efficiency, **K** automatically calculates fitnesses for all $L_{i,j}(\gamma)$ and stores these values in the array `K->fitness_precomputed`. The function `fitness(K,i,j,g)` returns the corresponding element of this array.

Precomputed fitness values are established as a consequence of calling the function `initiate_model_state()` just prior to starting the model. All relevant members of `K` must have values prior to this time. The use of precomputed fitness values is appropriate for the fitness functions supplied with **K**. The function `fitness_computed(K,i,j,g)` may be used to bypass the precomputed values and calculate fitness directly.

If a fitness function is defined (section 4.3) that may determine different fitness values

for identical $L_{i,j}(\gamma)$ at different times during model execution, then it must be determined which of the following cases applies:

1. Fitness values may change one to several discrete times; or

2. Fitness values may change many times.

If case (1) fits, then points should be identified during model execution at which precomputed values in `K->fitness_precomputed` should be recomputed. At these times, all relevant members of `K` should be updated, and then the function `initiate_fitness_precomputed()` is called. The fitness function itself may be changed at this time by changing the value of `K->fitness_function`:

```
if ( fitness values must change ) {
    /*
    ** First, change all relevant members of K (e.g., K->fit_s
    ** and K->fit_h) and then recompute the values.
    */
    K->fit_s = 0.80;  /* decrease the selection coefficient */
    initiate_fitness_precomputed(K);
}
```

If the fitness function instead fits case (2), then the use of precomputed fitness values must be disabled by telling **K** to directly compute fitness every time a fitness value is required. This is more computationally intensive. The use of precomputed fitness values is disabled by calling the function `must_compute_fitness_function()` and passing the registered value of the fitness function (which was stored in `K->fitness_function)` as the sole argument:

```
/* Force computation of fitness every time */
must_compute_fitness_function(K->fitness_function);
```

After doing this, the function `fitness(K,i,j,g)` is operationally identical to the function `fitness_computed(K,i,j,g)`. There is currently no mechanism to return to precomputed fitness values once `must_compute_fitness_function()` has been called.

4.3 Defining a new fitness function

In **K**, fitness functions share a common interface. The first argument is always the active `KConfig K`, the second and third arguments are both `KInt` and specify the number of heterozygous and homozygous mutations, respectively, in the load class of interest, and the fourth argument specifies the genotype of interest and is also `KInt`. Once the fitness value is computed from the load class and genotype arguments together with any parameters from `KConfig K`, the fitness value returned as a `KScalar`.

Because fitness functions share a common interface, a new fitness function may be easily defined and incorporated into **K**. Within the body of the fitness function, fitness is calculated using the load class and genotype arguments along with any members of `K` (e.g. `K->fit_s` and `K->fit_h`) that are being used as parameters. The fitness is computed as a `KScalar` and returned as the function value:

```
KScalar  newfitnessfunction  (KConfig K, KInt i, KInt j, KInt g)
{
    KScalar w;        /* w will hold the computed fitness */
    KScalar t1, t2;   /* intermediate values */
    /*
    ** The example fitness function is the multiplicative
    ** fitness function w_{i,j} = (1 - hs)^i (1 - s)^j .
    **
    ** The C function pow(a,b) computes a^b .
    */
    t1 = pow((1 - K->fit_h*K->fit_s), i);
    t2 = pow((1 - K->fit_s), j);
    w = t1 * t2;
    return w;
}
```

In order to use a new fitness function it must be registered. This is done prior to the start of the model run, generally within `main()`. In the declarations section the new fitness function is described with a *function prototype*. A function prototype is identical to the first line of the function definition, but does not have a function body and ends

with a semicolon ';'. All fitness function prototypes will be identical except for the name of the function. Prior to use, a new fitness function is registered by calling the function `register_fitness_function()`. This function requires two arguments: the name of the new fitness function without parentheses, and a short character-string description. **K** uses the new fitness function once the value returned by `register_fitness_function()` is assigned to `K->fitness_function`. Any members of `K` used by the new fitness function, e.g. `K->fit_s`, must have values assigned prior to model execution:

```
/* Function prototype for new fitness function within declarations */
KScalar  newfitnessfunction  (KConfig K, KInt i, KInt j, KInt g);
... intervening code ...
/* Register the fitness function */
K->fitness_function = register_fitness_function(newfitnessfunction,
                                                "new multiplicative");
/* Set the members of K that are used by newfitnessfunction() */
K->fit_s = 0.95;  /* E.g., highly recessive sublethal mutations*/
K->fit_h = 0.02;
... continue with model loop ...
```

Chapter 5

Using nested mutation classes

K provides the ability to simultaneously examine the consequences of the number of mutation classes $N_c > 1$ via *nesting*. No published model has yet made use of this feature (but see Scofield, unpubl.), and the extensions described here are entirely novel to **K**. As a result, this portion of **K** has not been adequately tested.

Space and execution time with nesting scale at approximately $\mathbf{O}([reqs.\ of N_1]^{N_c})$. We will consider the case where $N_c = 2$, which represents a squaring of space and time requirements. We make two further simplifications. We will consider load classes only, not genotypes, and we will also lose some generality in the functions used for computing fitness. Interested users may extend **K**'s implementation of nested mutation classes by following the examples given here and elsewhere in this document.

Each mutation class c has its own mutation rate U_c, fitness function, and fitness coefficients. Load classes are called *nested load classes* and are represented by L_{i_0,j_0,i_1,j_1}, where $i_0 \in (0, M_{i_0})$ and $j_0 \in (0, M_{j_0})$ represent the number of heterozygous and homozygous mutations in the first mutation class and $i_1 \in (0, M_{i_1})$ and $j_1 \in (0, M_{j_1})$ represent the number of heterozygous and homozygous mutations in the second mutation class. The total number of nested load classes is $(M_{i_0} + 1)(M_{j_0} + 1)(M_{i_1} + 1)(M_{j_1} + 1)$. Model

operations are based upon the joint probability distribution of independent events in each mutation class, thus the probability of a mutation event, a reproduction event, etc. affecting L_{i_0,j_0,i_1,j_1} is the probability of the required events affecting each mutation class individually:

$$Pr[\text{event in } L_{i_0,j_0,i_1,j_1}] = Pr[\text{event in } L_{i_0,j_0}] \times Pr[\text{event in } L_{i_1,j_1}]. \tag{5.0.1}$$

5.1 Mutation in nested models

As in an unnested model, mutation increases the number of heterozygous mutations in the population and does not affect the number of homozygous mutations. The mutation process for each mutation class c is equivalent to the single-class process described in Chapter 2 determined by Poi(U_c), thus mutation increases the population proportion in L_{i_0,j_0,i_1,j_1} at the expense of all $L(n_0, j_0, n_1, j_1)$ where $n_0 \leq i_0$ and $n_1 \leq i_1$. The total proportion of the population in x'_{i_0,j_0,i_1,j_1} after mutation is thus:

$$x'_{i_0,j_0,i_1,j_1} = \sum_{n_0=0}^{i_0} \sum_{n_1=0}^{i_1} x_{n_0,j_0,n_1,j_1} \cdot e^{-U_0} \frac{U_0^{[i_0-n_0]}}{(i_0-n_0)!} \cdot e^{-U_1} \frac{U_1^{[i_1-n_1]}}{(i_1-n_1)!} \tag{5.1.1}$$

If $U_c = 0$ then the corresponding term of Poi(U_c) is replaced with 1 if $i_c - n_c = 0$ and 0 otherwise.

5.2 Reproduction in nested models

As for mutation, reproduction in an nested model is a straightforward extension of the unnested model. Because mutation classes are independent, results for each genotype are determined as the product of results for each mutation class. Reproduction is by far the most time-consuming step in a nested model, so judicious implementation of the qualifying conditions on indices appended to each equation may speed execution by an order of magnitude or more:

$$\begin{aligned} s_{i_0,j_0,i_1,j_1}(n_0,v_0,n_1,v_1) = &\binom{n_0}{i_0}\binom{n_0-i_0}{j_0-v_0}\binom{n_1}{i_1}\binom{n_1-i_1}{j_1-v_1}\left(\frac{1}{2}\right)^{2(n_0+n_1)-i_0-i_1}; \\ &i_0 \in (0,n_0),\ j_0 \in (v_0, v_0+n_0),\ i_0+j_0 \le n_0+v_0, \\ &i_1 \in (0,n_1),\ j_1 \in (v_1, v_1+n_1),\ i_1+j_1 \le n_1+v_1 \end{aligned} \tag{5.2.1}$$

$$a_{i_0,j_0,i_1,j_1}(n_0,v_0,n_1,v_1) = \begin{cases} 1 & i_0=n_0,\ j_0=v_0,\ i_1=n_1,\ j_1=v_1 \\ 0 & \text{otherwise.} \end{cases} \tag{5.2.2}$$

Haploid gametes carry one copy of mutations from each mutation class. As for the unnested model, the proportion of available female gametes is weighted by the outcrossing rate:

$$f_{i_0,i_1} = \sum_{v_0=0}^{i_0} \sum_{v_1=0}^{i_1} \sum_{n_0=i_0-v_0}^{M_{i_0}} \sum_{n_1=i_1-v_1}^{M_{i_1}} \binom{n_0}{i_0-v_0}\binom{n_1}{i_1-v_1}\left(\frac{1}{2}\right)^{n_0+n_1} x'_{n_0,v_0,n_1,v_1}; \quad v_0 \le M_{j_0},\ v_1 \le M_{j_1} \tag{5.2.3}$$

$$f^{\female}_{i_0,i_1} = O \cdot f_{i_0,i_1} \tag{5.2.4}$$

$$f^{\male}_{i_0,i_1} = f_{i_0,i_1} \tag{5.2.5}$$

Outcrossed zygotes are formed by matings among male and female gametes:

$$x''_{(O)i_0,i_1} = \frac{1}{4}\sum_{k_0=0}^{i_0}\sum_{k_1=0}^{i_1} f^{\female}_{k_0,k_1} f^{\male}_{i_0-k_0,i_1-k_1} + f^{\female}_{k_0,i_1-k_1} f^{\male}_{i_0-k_0,k_1} + f^{\female}_{i_0-k_0,k_1} f^{\male}_{k_0,i_1-k_1} + f^{\female}_{i_0-k_0,i_1-k_1} f^{\male}_{k_0,k_1}. \tag{5.2.6}$$

Adding up all progeny proportions gives:

$$x''_{i_0,j_0,i_1,j_1} = \begin{cases} x''_{(O)i_0,i_1} + x''_{(S)i_0,j_0,i_1,j_1} + x''_{(A)i_0,j_0,i_1,j_1} & j_0 = 0,\ j_1 = 0 \\ x''_{(S)i_0,j_0,i_1,j_1} + x''_{(A)i_0,j_0,i_1,j_1} & \text{otherwise} \end{cases} \tag{5.2.7}$$

5.3 Fitness and selection in nested models

Fitness is determined for each mutation class as for an unnested model, and the fitness of a nested load class is the product of the fitness functions of the two mutation classes:

$$\omega_{i_0,j_0,i_1,j_1} = \omega_{i_0,j_0} \cdot \omega_{i_1,j_1}. \tag{5.3.1}$$

For example, if a multiplicative fitness model is used for each mutation class then each mutation class has its own selection coefficient and dominance:

$$\omega_{i_0,j_0,i_1,j_1} = (1-h_0 s_0)^{i_0}(1-s_0)^{j_0} \cdot (1-h_1 s_1)^{i_1}(1-s_1)^{j_1}. \tag{5.3.2}$$

The computation of ω_{i_0,j_0,i_1,j_1} is a consequence of setting the fitness functions for the mutation classes; no more need be done by the user. The computation of mean fitness with nested load classes $\bar{\omega}_{01}$ is a straightforward extension of Equation (4.1.3):

$$\bar{\omega}_{01} = \frac{\sum_{i_0}\sum_{j_0}\sum_{i_1}\sum_{j_1} \omega_{i_0,j_0,i_1,j_1} \cdot x''_{i_0,j_0,i_1,j_1}}{\sum_{i_0}\sum_{j_0}\sum_{i_1}\sum_{j_1} x''_{i_0,j_0,i_1,j_1}}. \tag{5.3.3}$$

This value is returned by the function `mean_fitness_n()`.

Fitness values are precomputed for each mutation class c prior to the start of the model (see section 4.2) and kept in the array `KN->fitness_precomputed[c][i][j]`. The function `fitness_n(KN,i0,j0,i1,j1)` uses this array to determine the fitness of the nested load class L_{i_0,j_0,i_1,j_1}. Fitness values may be computed directly via `fitness_computed_n()`. Similar to unnested models, precomputed fitness values can be recomputed with `initiate_fitness_precomputed_n()`; a call to this function will recompute fitness values for both mutation classes. Direct computation of fitness values for each mutation class individually can be specified with `must_compute_fitness_function_n()`.

Selection is applied as for an unnested model:

$$X_{i_0,j_0,i_1,j_1} = \frac{\omega_{i_0,j_0,i_1,j_1} \cdot x''_{i_0,j_0,i_1,j_1}}{\bar{\omega}_{01}} \tag{5.3.4}$$

5.4 Data structures for a nested model

The basic structure of the nested model's data structures is very similar to the unnested model. The load class array names are all identical, with a change in array shape to accommodate the nested load class L_{i_0,j_0,i_1,j_1} and the lack of genotype, e.g., `KN->x[i0][j0][i1][j1]` and `KN->xpp[i0][j0][i1][j1]`. Most functions and datatypes used in a nested model are special versions named by adding the suffix "`_n`".

The datatype `KConfig_n` holds the parameters and data structures used in a nested model, and references to this datatype within **K** use the name `KN`. Changes from the `KConfig` datatype used in unnested models reflect both the greater complexity of nested models and the simplified set of model options. Real-valued parameters that differ between mutation classes are of the `KScalar_n` datatype, which is an array of `KScalar` values, one per mutation class. For example, in a nested model the mutation rate is modified using `KN->U[0] = value0` for the zeroth mutation class and `KN->U[1] = value1` for the first.

5.5 Initiating and running a nested model

The initiation of a **K** session with nested mutation classes looks very much like that for one mutation class. except for the use of the `KConfig_n` structure and the functions used are all special versions named by adding the suffix "`_n`". Also, all values U_c, s_c, etc. that are distinct between load classes are vectors declared as `KScalar_n` or `KInt_n`.

```
KConfig_n KN;
KN = initiate_KConfig_n();
initiate_load_classes_n(KN, MI0, MJ0, MI1, MJ1);
KN->U[0] = 0.1;
KN->U[1] = 0.9;
KN->fitness_function[0] = FITNESS_MULTIPLICATIVE_n;
KN->fitness_function[1] = FITNESS_MULTIPLICATIVE_n;
```

```
KN->fit_s[0] = 1.0;  /* zeroth mut class highly recessive lethals */
KN->fit_h[0] = 0.02;
KN->fit_s[1] = 0.1;  /* first mut class more dominant & mild */
KN->fit_h[1] = 0.3;
KN->S = 0.0;
set_repro_n(KN, KN->S, 0.0);
initiate_model_state_n(KN);
```

As for model initiation, running a **K** session with nested mutation classes looks very much like that for one mutation class, except the `KConfig` structure and the functions used are all special versions named by adding the suffix "`_n`".

```
compute_adults_initial_n(KN);
while (! is_equilibrium_n(KN)) {
    if (KN->generation > GENERATION_CUTOFF_n)
        break;
    if (KN->option_truncate)
        truncate_KArray_n(KN, KN->x, LOADCLASS_TRUNCATE);
    compute_mutation_n(KN);
    compute_self_progeny_n(KN);
    compute_apomixis_progeny_n(KN);
    compute_gametes_n(KN);
    compute_zygotes_n(KN);
    compute_summed_progeny_n(KN);
    compute_fitness_n(KN);
    compute_adults_nextgen_n(KN);
    normalize_KArray_n(KN, KN->x);
}
stats_all_n(KN);
stats_print_n(KN);
```

Chapter 6

Model parameters in K

Model parameters as used in **K** and in Kondrashov (1985). †Kondrashov did not model the coexistence selfing and apomixis, and used the same parameter names for the rate and discount of each.

Table 6.1: Model parameters in **K**

Model Parameter	**K**	Kondrashov	Value modified	**K** source code
Genotype designation	g, ξ, ζ	κ, ξ, ζ	various	`g`, `gg`, `xi`, `zeta`, *etc.*
Selfing rate	$S(\gamma)$	z_κ	model start	`K->S[`*genotype*`]`
Pollen discount to selfing	$\mathcal{D}_S(\gamma)$	y_κ	model start	`K->D_S[`*genotype*`]`
apomixis rate	$A(\gamma)$	$z_\kappa\dagger$	model start	`K->A[`*genotype*`]`
Pollen discount to apomixis	$\mathcal{D}_A(\gamma)$	$y_\kappa\dagger$	model start	`K->D_A[`*genotype*`]`
Outcrossing rate	$O(\gamma)$	not named	model start	`K->O[`*genotype*`]`
Resources for selfed ovules	$\mathcal{R}_{SO}(\gamma)$	not named	model start	`K->rsrc_SO[`*genotype*`]`
Resources for apomictic ovules	$\mathcal{R}_{AO}(\gamma)$	not named	model start	`K->rsrc_AO[`*genotype*`]`
Resources for outcrossed ovules	$\mathcal{R}_{OO}(\gamma)$	not named	model start	`K->rsrc_OO[`*genotype*`]`
Resources for outcrossed pollen	$\mathcal{R}_{OP}(\gamma)$	not named	model start	`K->rsrc_OP[`*genotype*`]`
Total resources used for ovules	F_{fem}	F_{fem}	each generation	`K->F_female`

Table 6.1: Model parameters in **K** (continued)

Model Parameter	**K**	Kondrashov	Value modified	**K** source code
Total resources used for pollen	F_{male}	F_{male}	each generation	`K->F_male`
Proportion of F_{fem} resources used for $L_{i,j}$ selfed ovules	$\gamma_{i,j}$	$\gamma_{i,j}$	each generation	`K->gamma[`i`][`j`]`
Proportion of F_{fem} resources used for $L_{i,j}$ apomictic ovules	$\alpha_{i,j}$	$\alpha_{i,j}$	each generation	`K->alpha[`i`][`j`]`
Proportion of F_{fem} resources used for $L_{i,j}$ outcross ovules	$\beta_{i,j}$	$\beta_{i,j}$	each generation	`K->beta[`i`][`j`]`
Proportion of F_{male} resources used for $L_{i,j}$ outcross pollen	$\rho_{i,j}$	$\rho_{i,j}$	each generation	`K->rho[`i`][`j`]`
Proportion of $\gamma_{i,j}$ used for $L_{i,j}(\gamma)$ selfed ovules	$SO_{i,j}(\gamma)$	$a_{i,j}(\kappa)$†	each generation	`K->SO[`*genotype*`][`i`][`j`]`
Proportion of $\alpha_{i,j}$ used for $L_{i,j}(\gamma)$ apomictic ovules	$AO_{i,j}(\gamma)$	$a_{i,j}(\kappa)$†	each generation	`K->AO[`*genotype*`][`i`][`j`]`

Table 6.1: Model parameters in **K** (continued)

Model Parameter	**K**	Kondrashov	Value modified	**K** source code
Proportion of $\beta_{i,j}$ used for $L_{i,j}(\gamma)$ outcross ovules	$OO_{i,j}(\gamma)$	$g_{i,j}(\kappa)$	each generation	`K->OO[`*genotype*`][`i`][`j`]`
Proportion of $\rho_{i,j}$ used for $L_{i,j}(\gamma)$ outcross pollen	$OP_{i,j}(\gamma)$	$h_{i,j}(\kappa)$	each generation	`K->OP[`*genotype*`][`i`][`j`]`

Note to Doug: consider sorting this table by parameter name, and changing the columns around. Also note the typesetting disaster that is the table caption.

Chapter 7

Debugging Guide for K

REDO! **K** is a complex system, and as such it is only to be expected that modifications or new implementations of model portions will have bugs that will need to be found and corrected. This chapter is a guide to resolving these problems and to the facilities provided within **K** to assist in resolving these problems.

7.1 Reproduction

7.1.1 Diagnostics relevant to problems with reproduction

7.1.2 Progeny proportions

If the integrity of functions described in Chapter 3 for determining $L_{i,j}(\gamma)$ of progeny resulting from matings, then variation in load class genotypes may be unnecessarily complicating the diagnosis of the problem. It would be better to start with a highly simplified adult $L_{i,j}(\gamma)$ to reduce the load classes of progeny that can be produced. The first step is to disable the mutation process. This can be done by either commenting out references to the function `compute_mutation(K)` or by specifying the diag-

nostic switch `DEBUG_DISABLE_MUTATION` (see subsection [whatever] for the specification of diagnostic switches). Then, new adult load classes must be specified. First, use `fill_KArray(K,K->x,0.0)` to set all adult $L_{i,j}(\gamma)$ to zero. Then, specify $L_{i,j}(\gamma)$ proportions that may help to resolve the problem. For example, to examine what appears to be a basic problem with `o_outcross()`, place all adults in load class $L(1,0;0)$ with `K->x[1][0][0]=1.0`. Then execute the model and examine $L_{i,j}(\gamma)$ of progeny produced.

Remember that the sum of adult $L_{i,j}(\gamma)$ must equal `1` prior to reproduction; this can be verified by checking that `(sum_KArray(K,K->x)==1.0)` or by specifying the diagnostic switch `DEBUG_NORMALIZATION`.

7.2 Decreasing execution time

There are two important rules to consider before attempting to decrease the execution time of a computer program:

1. Be sure that the program is actually taking too much time, by your own definition. If so, then ...

2. Only work on those parts of the program that actually use a lot of time.

If you don't follow these rules, you may find that you've either been trying to improve an already fast program, or that you've been making only minor improvements to a slow program. If Rule 1 applies, first try compiling **K** with compiler optimization turned on. In Visual Studio on Windows, select the release build configuration via '`Build > Set Active Configuration ... > Win32 Release`'. On Unix/Linux, type '`make opt`' or add the `-O3` option while compiling.

If Rule 1 still applies after optimized compilation, then you can apply Rule 2 by identifying functions that take too much time. In Visual Studio on Windows, I have no

idea how to do this. On Unix/Linux systems, the `gprof` utility will tell you exactly the information you require. Type '`man gprof`' at a shell prompt for more information.

Once you've identified functions that are too slow, first consider algorithmic improvements. For example, the generation of outcross offspring via Kondrashov's (1985) function requires $\mathbf{O}(M_i^3 \times M_j^2)$ time to execute. By modifying the outcrossing process to first produce gametes $\mathbf{O}(M_i^2 \times M_j)$ and then zygotes $\mathbf{O}(M_i^2)$ as did Charlesworth *et al.* (1990), an increase in speed was obtained.

Chapter 8

Source code conventions for K

Listing 8.1: **K** function template

```
/*////////////////////////////////////////////////////////////////*/
KScalar     my_function         (KConfig K, KArray arg1, KInt arg2);
{
    const char* thisfunction = "my_function";
    /* ... the rest of the function ... */
}
```

Bibliography

Charlesworth, D., Morgan, M. T., & Charlesworth, B. 1990. Inbreeding depression, genetic load, and the evolution of outcrossing rates in a multilocus system with no linkage. *Evolution*, **44**(6), 1469–1489.

Heller, J., & Maynard Smith, J. 1979. Does Muller's ratchet work with selfing? *Genetical Research*, **32**, 289–293.

Kondrashov, Alexey S. 1985. Deleterious mutations as an evolutionary factor. II. Facultative apomixis and selfing. *Genetics*, **111**, 635–653.

Lande, R., Schemske, D. W., & Schultz, S. T. 1994. High inbreeding depression, selective interference among loci, and the threshold selfing rate for purging recessive lethal mutations. *Evolution*, **48**(4), 965–978.

Morgan, M. T. 2001. Consequences of life history for inbreeding depression and mating system evolution in plants. *Proceedings of the Royal Society of London Series B-Biological Sciences*, **268**, 1817–1824.

Muirhead, C. A., & Lande, R. 1997. Inbreeding depression under joint selfing, outcrossing, and asexuality. *Evolution*, **51**(5), 1409–1415.

Schultz, S. T., & Willis, J. H. 1995. Individual variation in inbreeding depression: The roles of inbreeding history and mutation. *Genetics*, **141**, 1209–1223.

Scofield, Douglas G., & Schultz, Stewart T. 2006. Mitosis, stature and evolution of plant mating systems: Low-Φ and high-Φ plants. *Proceedings Of The Royal Society Of London Series B-Biological Sciences*, **273**(1584), 275–282.