Chapter 3

Fitness in K

REDO! In **K**, just as in nature, selection reduces proportions of less fit genotypes. Fitness in **K** is a function of the mutations carried in the load class $L(i, j)$. 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.

In **K**, fitness is a function of the load class and genotype in combination with any number of user-specified parameters. Typically, the load class and parameters alone determine fitness and the genotype is considered to be fitness-neutral.

In **K**, fitness is a function of the load class-genotype and the configuration `KConfig`, which contains the fitness parameters necessary to determine the fitness of load class-genotypes.

3.1 Specifying the selection model

A fitness function in **K** returns a `KScalar`.

REDO! Kondrashov's original formulation specified a fitness model that enabled threshold selection.

$$w_{i,j} = 1 - \left(\frac{1+dj}{k}\right)^{\alpha} \tag{3.1.1}$$

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 inf correspond to linear, intermediate and threshold selection, respectively ((Kondrashov 1985)).

REDO! For the moment, **K** does not provide Kondrashov's fitness function. Instead, it provides a familiar fitness function parameterized by selection coefficient $s \in (0, 1)$ and dominance $h \in (0, 1)$.

$$w_{i,j} = (1 - hs)^{i}(1 - s)^{j} \tag{3.1.2}$$

3.2 Defining your own fitness function

In the source code of **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, you can easily define your own fitness function and incorporate it 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`, `K->fit_h`) that are being used as parameters. The fitness is computed as a `KScalar` and returned as the function value.

```
KScalar  myfitnessfunction  (KConfig K, KInt i, KInt j, KInt g)
{
    KScalar w;       /* w will hold the computed fitness */
    KScalar t1, t2;  /* for holding 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 your fitness function it must be registered with **K**. This is done prior to the start of the model run, generally within `main()`. In the declarations section you must provide a *function prototype* that describes your fitness function. Note that the function prototype is identical to the first line of the function definition, but does not have a function body and ends with a semicolon '`;`'. Note that because a fitness function has a specific order and type of arguments and returns the same type of value, all fitness function prototypes will be identical except for the name of the function.

```
KScalar  myfitnessfunction  (KConfig K, KInt i, KInt j, KInt g);
```

To use your fitness function, call the function `register_fitness_function()` and pass the name of your fitness function as the sole argument. You then must tell **K** to use your registered fitness function by assigning the value returned by `register_fitness_function()` to `K->fitness_function`. Don't forget to assign values to the members of `K` that your fitness function requires.

```
/* Register the fitness function */
K->fitness_function = register_fitness_function(myfitnessfunction);
/* Set the members of K that are used by myfitnessfunction() */
K->fit_s = 0.95;  /* E.g., highly recessive sublethal mutations*/
K->fit_h = 0.02;
```

3.3 Precomputed fitness

For computational efficiency, **K** automatically precomputes fitnesses for all $L(i, j; g)$ and stores these values in `K->fitness_precomputed[i][j][g]`. The function `fitness(K,i,j,g)` returns the corresponding element of this array.

Precomputation of fitness values occurs as a consequence of calling the function `initialize_model_state()` just prior to starting the model, therefore all relevant members of `K` must have values prior to the calling of this function. Precomputation is valid for the fitness functions supplied with **K**, and perhaps for most fitness functions of interest. The function `fitness_computed(K,i,j,g)` bypasses the precomputed values and determines fitness directly.

If you define a fitness function that may determine different fitness values for identical $L(i, j; g)$ at different times during model execution, then you must decide 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 your fitness function fits case (1), then you should identify times during model execution at which to recompute the precomputed values in `K->fitness_precomputed`. Make sure all relevant members of `K` have been updated, and then call the function `initiate_fitness_precomputed()`.

```
/*
** 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 your fitness function instead fits case (2), then you must disable precomputation of fitness values by telling **K** to directly compute fitness every time a fitness value is required. This is *much* more computationally intensive, so please make sure that case (1) does not apply before doing this. 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 probably stored in `K->fitness_function`) as the sole argument:

```
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 precomputation of fitness values once `must_compute_fitness_function()` has been called.

Chapter 6

Source code conventions for K

Listing 6.1: **K** function template

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

Bibliography

[Charlesworth et al., 1991] Charlesworth, B., Morgan, M. T., and Charlesworth, D. (1991). Multilocus models of inbreeding depression with synergistic selection and partial self-fertilization. *Genetical Research*, 57(2):177–194.

[Haldane, 1949] Haldane, J. B. S. (1949). The association of characters as a result of inbreeding and linkage. *Annals of Eugenics*, 15:15–23.

[Kondrashov, 1985] Kondrashov, A. S. (1985). Deleterious mutations as an evolutionary factor. ii. facultative apomixis and selfing. *Genetics*, 111:635–653.