`survival()` callbacks are provided with information on the survival decision made by SLiM for every individual, including each individual's final calculated fitness value, they can also be used purely observationally – for example, to log out the fitness value of each individual along with the outcome of selection for that individual, or similar information (see section 16.12).

16.23 Tracking separate sexes in script, nonWF style

Section 12.5 showed how to track the sex of individuals yourself in script in a WF model, using the `tag` property to represent the sex of each individual. In this section we will look at the same idea implemented in a nonWF model.

This approach can be useful for implementing unusual mating systems that don't fit well into SLiM's conception of sex. In sexual models in SLiM, for example, SLiM does not allow selfing; it is assumed that females produce eggs and male individuals produce sperm, and that one egg and one sperm are required for fertilization. There are some sexual species, however, that can reproduce by methods such as *automixis* – the fusion of nuclei or gametes produced by the same individual – that violate this assumption. In this section, by way of illustration, we will implement a sexual species that usually reproduces by biparental mating between a female and a male, to produce an offspring of either sex, but in which a female can also sometimes reproduce by automixis, without a male mate, to produce a daughter.

There are various forms of automixis; for simplicity, we will here implement a form of automixis in which any two randomly chosen gametes from the focal female fuse, but we will discuss other types of automixis at the end. The focus here is not really on automixis; that is just used as a pedagogical example to justify the explicit tracking of sex in script, rather than using `initializeSex("A")` and letting SLiM do it for us.

Here's the whole model except the `reproduction()` callback:

```
initialize() {
    initializeSLiMModelType("nonWF");
    defineConstant("K", 500);      // carrying capacity
    initializeMutationType("m1", 0.5, "f", 0.0);
    m1.convertToSubstitution = T;
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeMutationRate(1e-7);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", K);

    // assign random sexes (0 = male, 1 = female)
    p1.individuals.tag = rbinom(p1.individualCount, 1, 0.5);
}
early() {
    p1.fitnessScaling = K / p1.individualCount;
}
1:2000 late() {
    ratio = sum(p1.individuals.tag == 0) / p1.individualCount;
    catn(sim.generation + ": " + ratio);
}
```

The initialization is boilerplate; note that we do not call `initializeSex()` here, since we do not want SLiM to track sex itself. In the `1 early()` event we draw a random sex for each individual from a binomial distribution, and assign those sex values into the `tag` property of the individuals;

completely arbitrarily, we designate `0` as male and `1` as female. In the `1:2000 late()` event we output the sex ratio in each generation (calculated as M:(M+F), following the definition used by SLiM), since the sex ratio will provide a test that the model is working as intended.

The `reproduction()` callback is also quite simple:

```
reproduction() {
    // we focus on the reproduction of the females here
    if (individual.tag == 1)
    {
       if (runif(1) < 0.7)
       {
          // choose a male mate and produce a son or daughter
          mate = subpop.sampleIndividuals(1, tag=0);
          offspring = subpop.addCrossed(individual, mate);
          offspring.tag = rbinom(1, 1, 0.5);
       }
       else
       {
          // reproduce through automixis to produce a daughter
          offspring = subpop.addSelfed(individual);
          offspring.tag = 1;
       }
    }
}
```

We want to focus on the reproduction of females here; usually we could do that by declaring the callback as `reproduction(NULL, "F")`, but since we're tracking sex ourselves in `tag`, we need to test that the focal individual is female ourselves. If the focal individual is female, we then roll the dice; with a 70% chance, we choose a mate with `sampleIndividuals()`, narrowing the pool of eligible individuals to males using `tag=0`, and then call `addCrossed()` to produce one offspring from that biparental mating. That offspring is assigned a sex randomly, with equal probability. The remaining 30% of the time, we have the female reproduce by automixis, by calling `addSelfed()`. This is the step that SLiM would not allow in a normal sexual model; `addSelfed()` can only be called in non-sexual models. (This limitation was put in place to clarify how to implement selfing when modeling sex chromosomes.) Since this is, in SLiM's view, a non-sexual model, we are allowed to call it. The offspring resulting from this automixis is always female (following the biology of some species with automixis).

When the model is run, it can be seen that the calculated sex ratio starts around 0.5 but quickly falls to an average of about 0.35. This is expected, since males are produced only by biparental mating, which happens 70% of the time. Any given offspring will therefore be male with a probability of $0.7 \times 0.5 = 0.35$. Our tracking of sex appears to be working properly.

This model can easily be extended in various directions. One could track more than just two sexes or mating types; the `tag` property can be used however, you wish, and you can control how it affects reproduction with your own custom script. You could model fish that change sex over the course of their lives (as some do), alternative sexual types like "sneaker males", sex determination according to environmental cues like temperature, species with ZW sex chromosomes instead of XY (if there are biological details that make that distinction important for your model), or anything else of the sort.

As mentioned at the outset, automixis is a complex topic, and it is not our real focus here. We have implemented automixis by simply calling `addSelfed()`, which generates two haploid gametes from the focal female using SLiM's standard recombination machinery and fuses them to form the diploid offspring. The biological reality is that depending upon whether a species uses "central fusion automixis", "terminal fusion automixis", or some other type, the gametes that fuse to produce the offspring may have certain properties because they are derived from certain specific points in the process of meiosis (Wikipedia has a helpful diagram in its article on automixis). In this case, `addSelfed()` does not provide the desired behavior, and the `addRecombinant()` method is the more powerful alternative. With `addRecombinant()`, you can control the exact set of recombination breakpoints used to generate each of the two gametes used by SLiM in the offspring, making them exhibit the necessary properties. We will not show an example of that approach here, but other recipes in this manual do use `addRecombinant()` for various other purposes; section 16.17 might be particularly relevant as an example of the general approach. Generating your own breakpoints is reasonably straightforward, either by using the `drawBreakpoints()` method of `Chromosome` (if SLiM's default breakpoint generation is acceptable, perhaps with some modification), or by generating them yourself using functions like `rbinom()` to determine the number of breakpoints and `rdunif()` to draw each breakpoint's position. Section 1.5.6 discusses SLiM's default recombination breakpoint generation algorithm in detail; you might wish to pattern your own breakpoint generation algorithm after it.

16.24 Modeling haplodiploidy with `addRecombinant()`

We have seen ways of handling a variety of mating systems in SLiM, including selfing (section 6.3.1), cloning (section 6.3.2), haploidy (sections 14.9, 16.13, and 16.14), and alternation of generations (section 16.16), as well as how to track separate sexes yourself in script in order to implement more unusual mating systems such as automixis (sections 12.5 and 16.23). One fairly common mating system we have not covered yet is haplodiploidy. In haplodiploidy (sometimes called arrhentoky), males develop from unfertilized eggs and are haploid, while females develop from fertilized eggs and are diploid. It is particularly well-known as the sex-determination system in the Hymenoptera (bees, ants, and wasps), and has interesting effects on the relatedness of individuals that may promote the emergence of eusociality.

The haplodiploidy model presented here was developed in collaboration with Rodrigo Pracana, Richard Burns, Robert L. Hammond, and Yannick Wurm, and a related preprint has been posted on bioRxiv at https://doi.org/10.1101/2021.10.25.465450. The model presented in that publication is fairly different from the one shown here; it implements a Wright–Fisher model design (but written as a nonWF SLiM model), with non-overlapping generations, a fixed population size, and fitness that acts through mating success. In that model, all reproduction occurs in a single "big bang", generating all of the individuals needed to fill out the next generation. It therefore follows the recipe of section 16.15, which shows how to implement a Wright–Fisher model as a nonWF model. (In fact it goes even further than that recipe does, in making fitness influence mating success rather than reproduction!) That paper will likely be of interest to those modeling haplodiploidy in SLiM, as it explores some of the consequences of haplodiploidy for deleterious and beneficial mutations and compares simulations in SLiM to some prior analytical work. If the SLiM code from *either* that paper or this section is adapted for use in a publication, a citation to Pracana, Burns, Hammond, Haller, & Wurm (2021) would be much appreciated. Thanks to my collaborators for welcoming the publication of this recipe here.

In this section, on the other hand, we will look at a very minimal nonWF haplodiploidy model; to keep things simple it will follow typical nonWF model conventions such as overlapping generations, density-dependent population regulation, and fitness that acts through survival. Each female will reproduce independently, via a separate callout to the `reproduction()` callback. In all these respects, this recipe follows the very simple nonWF template of the recipe in section 16.1.

OK, with that background information out of the way, let's look at the model! Note that this model requires SLiM 3.7 to run. Here's the `initialize()` callback:

```
initialize() {
    defineConstant("K", 2000);
    defineConstant("P_OFFSPRING_MALE", 0.8);
    initializeSLiMModelType("nonWF");
    initializeMutationRate(1e-8);
    initializeMutationType("m1", 0.0, "f", 0.0);
    m1.convertToSubstitution = T;
    m1.haploidDominanceCoeff = 1.0;
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 999999);
    initializeRecombinationRate(1e-6);
    initializeSex("A");
}
```

We define a carrying capacity `K` of `2000`. We also define a parameter, `P_OFFSPRING_MALE`, that is the probability that a given offspring generated by a female will be a (haploid) male, as opposed to a (diploid) female. Here that is set to `0.8`, quite arbitrarily; offspring will be mostly males.

We set up a neutral mutation type, `m1`; this model can also accommodate non-neutral mutations, but we won't do that here for simplicity. We set `convertToSubstitution` to `T` so these neutral mutations get substituted by SLiM when they fix (see section 1.5.2); note that SLiM will correctly detect fixation even though the model contains a mixture of haploids and diploids. We also set the `haploidDominanceCoeff` property to `1.0`. This property is used as the "dominance" coefficient in haploids, where mutations can only be present in a single copy. Recall that in diploids, a homozygous mutation has a fitness effect of $1+s$,where s is the selection coefficient of the mutation, and a heterozygous mutation has a fitness effect of $1+hs$, where h is the dominance coefficient kept in the `dominanceCoeff` property of `MutationType`. You might want a mutation to have a different fitness effect when found in a haploid – not $1+s$, and not $1+hs$, but rather $1+zs$, where z is the value of the `haploidDominanceCoeff` property (note that I just made up the use of the symbol z here, that is not a standard convention). Here we set this to `1.0`, but it doesn't matter anyway since this is a neutral model. Note that if you wanted even finer control over the fitness effects of mutations in haploids you could achieve that with a `fitness()` callback.

The rest of the initialization is quite standard. We use `initializeSex("A")` to ask SLiM to track separate sexes for us; we don't do anything here that requires us to use the techniques of section 16.23 to track sex ourselves.

Here's the rest of the framework of the model, apart from the `reproduction()` callback:

```
1 early() {
    // make an initial population with the right genetics
    mCount = asInteger(K * P_OFFSPRING_MALE);
    fCount = K - mCount;
    sim.addSubpop("p1", mCount, sexRatio=1.0, haploid=T);     // males
    sim.addSubpop("p2", fCount, sexRatio=0.0, haploid=F);     // females
    p1.takeMigrants(p2.individuals);
    p2.removeSubpopulation();
}
early() {
    p1.fitnessScaling = K / p1.individualCount;
}
10000 late() {
    sim.simulationFinished();
}
```

The `early()` event implements density-dependent fitness, and the `10000 late()` event defines the end of the simulation. The interesting bit is the `1 early()` event that creates the initial population. First, it uses P_OFFSPRING_MALE to calculate the initial number of males and females in the subpopulation. Then it creates all the males in p1, passing sexRatio=1.0 to make them all male, and haploid=T to make them all haploid. Similarly, it creates all the females in p2, passing sexRatio=0.0 to make them all female, and haploid=F to make them all diploid. It moves the females into p1 with takeMigrants(), and then removes p2 from the model. This sleight of hand lets us use addSubpop() to easily create the males and females we want. We could avoid the use of p2 by creating p1 with an initial size of 0, and then filling it up with new empty individuals with the appropriate genetics using addEmpty() in a 1 reproduction() callback, but this design is much simpler, and the ephemeral existence of p2 is fairly harmless. (If you use tree-sequence recording with this model, the initial creation of the females in p2 might make for problems with recapitation and so forth; in that case, the addEmpty() technique might be better. You could also simply call sim.addSubpop("p1", K, sexRatio=P_OFFSPRING_MALE) to create the initial subpopulation; this will make the males in the first generation be diploid, but that ought to be harmless since their second genomes will never be used for anything.)

Finally, here is the reproduction() callback, which runs only for females since it is declared with "F" for its focal sex:

```
reproduction(NULL, "F") {
    // choose an initial copy strand based on a coin flip
    strand = rbinom(1, 1, 0.5);
    gen1 = strand ? individual.genome1 else individual.genome2;
    gen2 = strand ? individual.genome2 else individual.genome1;
    breaks = sim.chromosome.drawBreakpoints(individual);

    // decide whether we're generating a haploid male or a diploid female
    if (rbinom(1, 1, P_OFFSPRING_MALE))
    {
       // didn't find a mate; make a haploid male from an unfertilized egg:
       //   - one genome comes from recombination of the female's genomes
       //   - the other genome is a null genome (a placeholder)
       subpop.addRecombinant(gen1, gen2, breaks, NULL, NULL, NULL, "M");
    }
    else
    {
       // found a mate; make a diploid female from a fertilized egg:
       //   - one genome comes from recombination of the female's genomes
       //   - the other genome comes from the mate (a haploid male)
       mate = subpop.sampleIndividuals(1, sex="M");
       subpop.addRecombinant(gen1, gen2, breaks, mate.genome1, NULL, NULL, "F");
    }
}
```

This handles all the key details of haplodiploidy. We will generate the offspring using the method addRecombinant(); it is a very low-level method, so we will need to tell it exactly what to do (see section 24.14.12). For this reason, we need to choose which of the female's genomes will be the initial copy strand when generating the egg. We do that up front, using rbinom() as a coin flip, and assign gen1 to be the initial copy strand and gen2 to be the other strand. Next, since we will need crossover breakpoints to pass to addRecombinant() to generate the egg, we get them from sim.chromosome, which draws them following SLiM's standard procedure. Finally, we need to decide whether we're going to generate a (haploid) male or a (diploid) female offspring. We again use rbinom(), with P_OFFSPRING_MALE as the probability for male.

If the offspring is male, there is no mate; the egg is unfertilized. In this case, we call addRecombinant() with gen1, gen2, breaks for the parental genomes and breakpoints for the egg, and `NULL, NULL, NULL` for the sperm to indicate that there is no sperm. This makes the second genome of the offspring a null genome, indicating that the offspring is haploid. Finally, we pass `"M"` to `addRecombinant()` to tell it the offspring is male.

If the offspring is female, we first have to choose a mate. We use `sampleIndividuals()` to do this, requiring only that the mate be male; of course, further restrictions could be placed on that selection. Then we call `addRecombinant()`, passing `gen1, gen2, breaks` as before for the egg. Here, however, we pass `mate.genome1, NULL, NULL` for the sperm, indicating that the sperm carries a copy of the male's haploid genome, without a second strand and without recombination. Finally, we pass `"F"` to tell `addRecombinant()` the offspring is male.

That's all there is to it; it's really quite straightforward. Implementing these sorts of mating system is mainly a matter of thinking through the biological logic: who mates with whom, who is what ploidy, and where each offspring chromosome comes from. Translate that logic into the appropriate calls to `addCrossed()`, `addCloned()`, `addSelfed()`, and `addRecombinant()` in your `reproduction()` callback, and things will often go quite smoothly. If you need to violate rules that SLiM normally enforces – having a non-hermaphroditic individual self, for example, or having more than two sexes – you may want to track the sex of individuals yourself, as shown in section 16.23.

Breaking with our usual procedure, we will not show results from this recipe here; demonstrating the correctness of this model is really beyond the scope of this manual. Instead, the reader is encouraged to consult Pracana, Burns, Hammond, Haller, & Wurm (2021), which analyzes results from a fairly similar model to show that they match the predictions for haplodiploidy made by analytical models.