SLiM

Workshop Series

#2: The Chromosome Hierarchy

Benjamin C. Haller

Messer Lab, Cornell University

The initialize() callback

```
// set up a simple neutral simulation
initialize()
{
   // set the overall mutation rate
   initializeMutationRate(1e-7);

   // m1 mutation type: neutral
   initializeMutationType("m1", 0.5, "f", 0.0);

   // g1 genomic element type: uses m1 for all mutations
   initializeGenomicElementType("g1", m1, 1.0);

   // uniform chromosome of length 100 kb
   initializeGenomicElement(g1, 0, 99999);

   // uniform recombination along the chromosome
   initializeRecombinationRate(1e-8);
}
```

Callbacks

```
initialize()
{
    ...
}
```

- **Callbacks** are script blocks called by SLiM

Callbacks

```
initialize()
{
   ...
}
```

- **Callbacks** are script blocks called by SLiM
- Various types:
  - `initialize()` callbacks are called at the **start**
  - `early()` events are called **early** each generation
  - `late()` events are called **late** each generation
  - `fitness()`, `mateChoice()`, `modifyChild()`, `recombination()`, `reproduction()`, `...`

The `initialize()` callback

```
// set up a simple neutral simulation
initialize()
{
    // set the overall mutation rate
    initializeMutationRate(1e-7);

    // m1 mutation type: neutral
    initializeMutationType("m1", 0.5, "f", 0.0);

    // g1 genomic element type: uses m1 for all mutations
    initializeGenomicElementType("g1", m1, 1.0);

    // uniform chromosome of length 100 kb
    initializeGenomicElement(g1, 0, 99999);

    // uniform recombination along the chromosome
    initializeRecombinationRate(1e-8);
}
```

Function calls

```
// set the overall mutation rate
initializeMutationRate(1e-7);
```

- Function calls *do something* (perform an action)
- Functions are passed *parameters*
- Here, the mutation rate is initialized to `1e-7`

Function calls

```
// set the overall mutation rate
initializeMutationRate(1e-7);
```

- Function calls *do something* (perform an action)
- Functions are passed *parameters*
- Here, the mutation rate is initialized to `1e-7`
- Several types of functions:
  - Eidos provides more than 100 built-in functions
  - SLiM provides a few built-in functions also
  - User-defined functions can be written in script

Mutation types

```
// m1 mutation type: neutral
initializeMutationType("m1", 0.5, "f", 0.0);
```

- Mutation types
  - a **category of mutation**: neutral, beneficial, etc.
  - defined by a *distribution of fitness effects* (DFE)
  - the DFE provides selection coefficients (*s*)
  - also have a dominance coefficient (*h*)
  - here, a fixed DFE (`"f"`) with s=0.0, h=0.5
  - fitness effects are $1+s$ (AA) and $1+hs$ (Aa)
  - named; this mutation type will be `m1`

Genomic element types

```
// g1 genomic element type: uses m1 for all mutations
initializeGenomicElementType("g1", m1, 1.0);
```

- Genomic element types
  - represent a **category of genomic region**
  - exon, intron, non-coding region, UTR, etc.
  - draw mutations from a set of mutation types
  - here, drawing just from `m1` (weight of `1.0`)
  - named; this genomic element type will be `g1`

Genomic elements

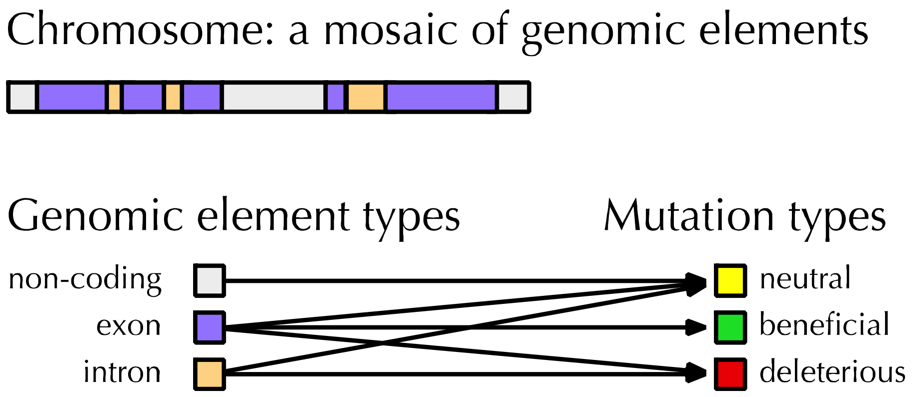
```
// uniform chromosome of length 100 kb
initializeGenomicElement(g1, 0, 99999);
```

- Genomic elements
  - represent a **specific region of the chromosome**
  - span some interval [start, end], in base positions
  - belong to one genomic element type
  - this genomic element is of type `g1` and spans [`0`, `99999`]
  - draw mutations from that genomic element type

The chromosome

Genomic elements, genomic element types, and mutation types

- The chromosome defines the genetic structure
- It is a sequence of elements (`GenomicElement`)
- Each element has a type (`GenomicElementType`)
- Each type draws new mutations from DFEs
- DFEs are defined by a `MutationType`



The `initialize()` callback

```
// set up a simple neutral simulation
initialize()
{
   // set the overall mutation rate
   initializeMutationRate(1e-7);

   // m1 mutation type: neutral
   initializeMutationType("m1", 0.5, "f", 0.0);

   // g1 genomic element type: uses m1 for all mutations
   initializeGenomicElementType("g1", m1, 1.0);

   // uniform chromosome of length 100 kb
   initializeGenomicElement(g1, 0, 99999);

   // uniform recombination along the chromosome
   initializeRecombinationRate(1e-8);
}
```

Callbacks

- **Callbacks** are script blocks called by SLiM

Callbacks

- **Callbacks** are script blocks called by SLiM
- Various types:
  - `initialize()` callbacks are called at the **start**
  - `early()` events are called **early** each generation
  - `late()` events are called **late** each generation
  - `...`

Callbacks

- **Callbacks** are script blocks called by SLiM
- Various types:
  - `initialize()` callbacks are called at the **start**
  - `early()` events are called **early** each generation
  - `late()` events are called **late** each generation
  - `...`
- We defined the *model structure* in `initialize()`
- Now we'll define some *ecology* with `early()`

The `early()` event

```
// create a population of 500 individuals
1 early()
{
    sim.addSubpop("p1", 500);
}
```

- Declares an `early()` event (callback)
- Declared to run in generation `1`
- The "`early()`" part is optional (default)

The `early()` event

```
// create a population of 500 individuals
1 early()
{
   sim.addSubpop("p1", 500);
}
```

- Declares an `early()` event (callback)
- Declared to run in generation `1`
- The "`early()`" part is optional (default)
- Makes a *method call*: `target.method()`
  - like a function call, but sent to a *target object*
  - this target object, `sim`, represents the simulation
  - defines a new subpopulation, `p1`, of size `500`

The `late()` event

```
// run to generation 10000
10000 late()
{
   sim.outputFixedMutations();
   sim.simulationFinished();
}
```

- Declares a `late()` event (callback)
- Declared to run in generation `10000`

The `late()` event

```
// run to generation 10000
10000 late()
{
    sim.outputFixedMutations();
    sim.simulationFinished();
}
```

- Declares a `late()` event (callback)
- Declared to run in generation `10000`
- Makes a method call to output substitutions
  - (no parameters here, but you use `()` anyway)

The `late()` event

```
// run to generation 10000
10000 late()
{
   sim.outputFixedMutations();
   sim.simulationFinished();
}
```

- Declares a `late()` event (callback)
- Declared to run in generation `10000`
- Makes a method call to output substitutions
  - (no parameters here, but you use `()` anyway)
- Makes a method call to stop the simulation
  - (the simulation would stop anyway)

The complete model

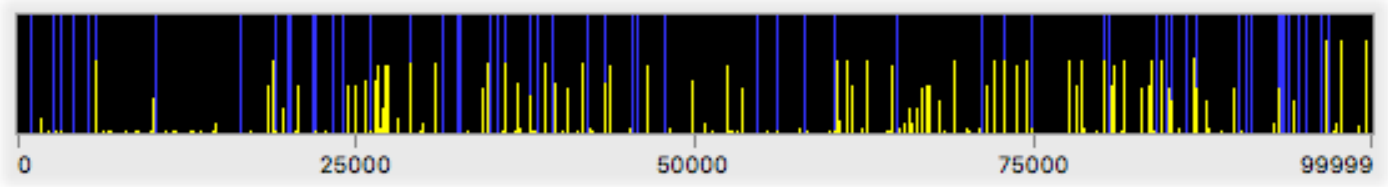
```
// set up a simple neutral simulation
initialize()
{
	// set the overall mutation rate
	initializeMutationRate(1e-7);

	// m1 mutation type: neutral
	initializeMutationType("m1", 0.5, "f", 0.0);

	// g1 genomic element type: uses m1 for all mutations
	initializeGenomicElementType("g1", m1, 1.0);

	// uniform chromosome of length 100 kb
	initializeGenomicElement(g1, 0, 99999);

	// uniform recombination along the chromosome
	initializeRecombinationRate(1e-8);
}

// create a population of 500 individuals
1 early()
{
	sim.addSubpop("p1", 500);
}

// run to generation 10000
10000 late()
{
	sim.outputFixedMutations();
	sim.simulationFinished();
}
```

Model output

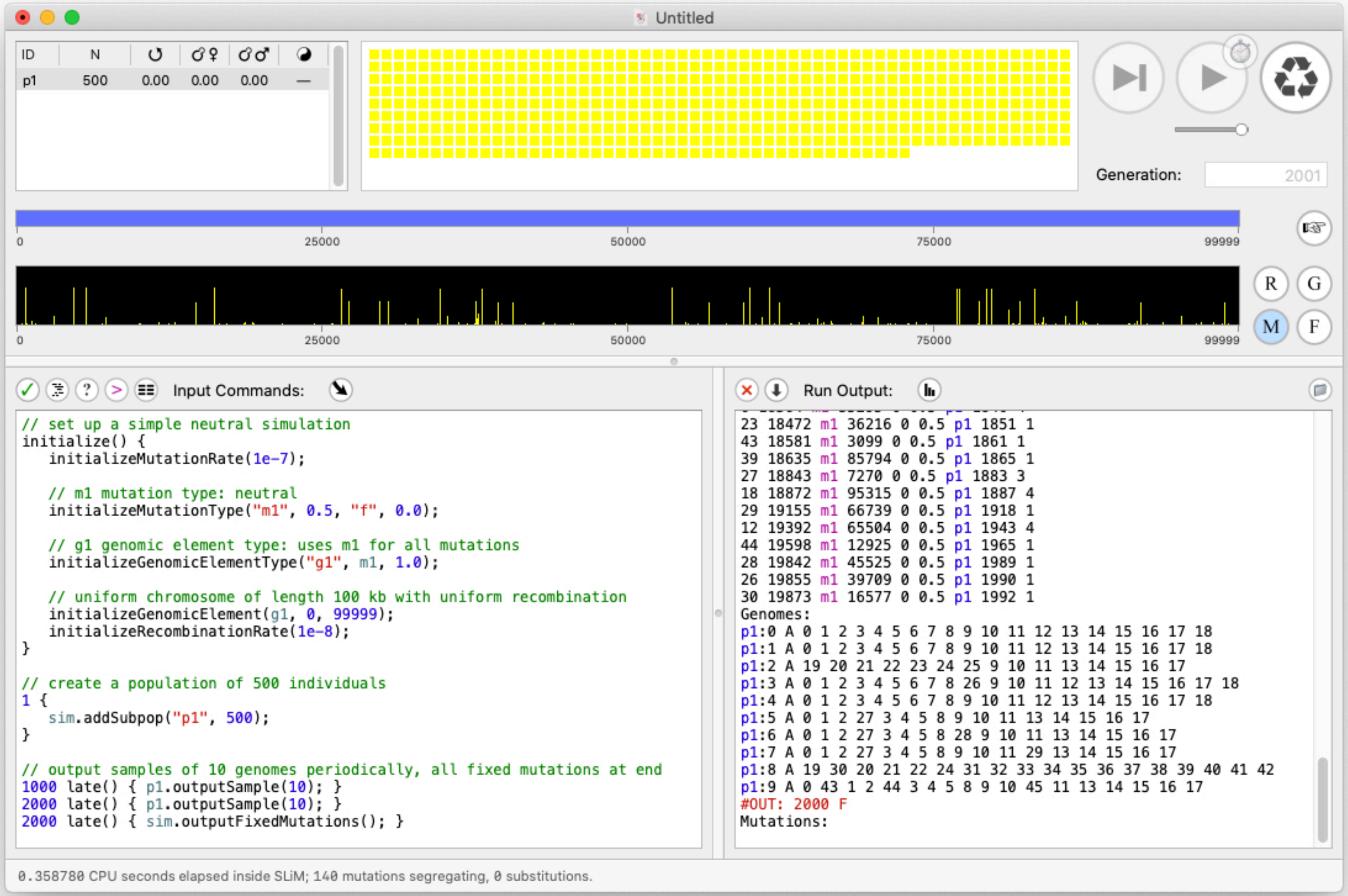
- From `outputFixedMutations()`:

```
#OUT: 10000 F
Mutations:
0 119 m1 80553 0 0.5 p1 14 1219
1 2404 m1 39498 0 0.5 p1 246 1450
...
56 70547 m1 91029 0 0.5 p1 7026 9424
57 69695 m1 95168 0 0.5 p1 6945 9424
```

- In SLiMgui:



Modeling in SLiMgui

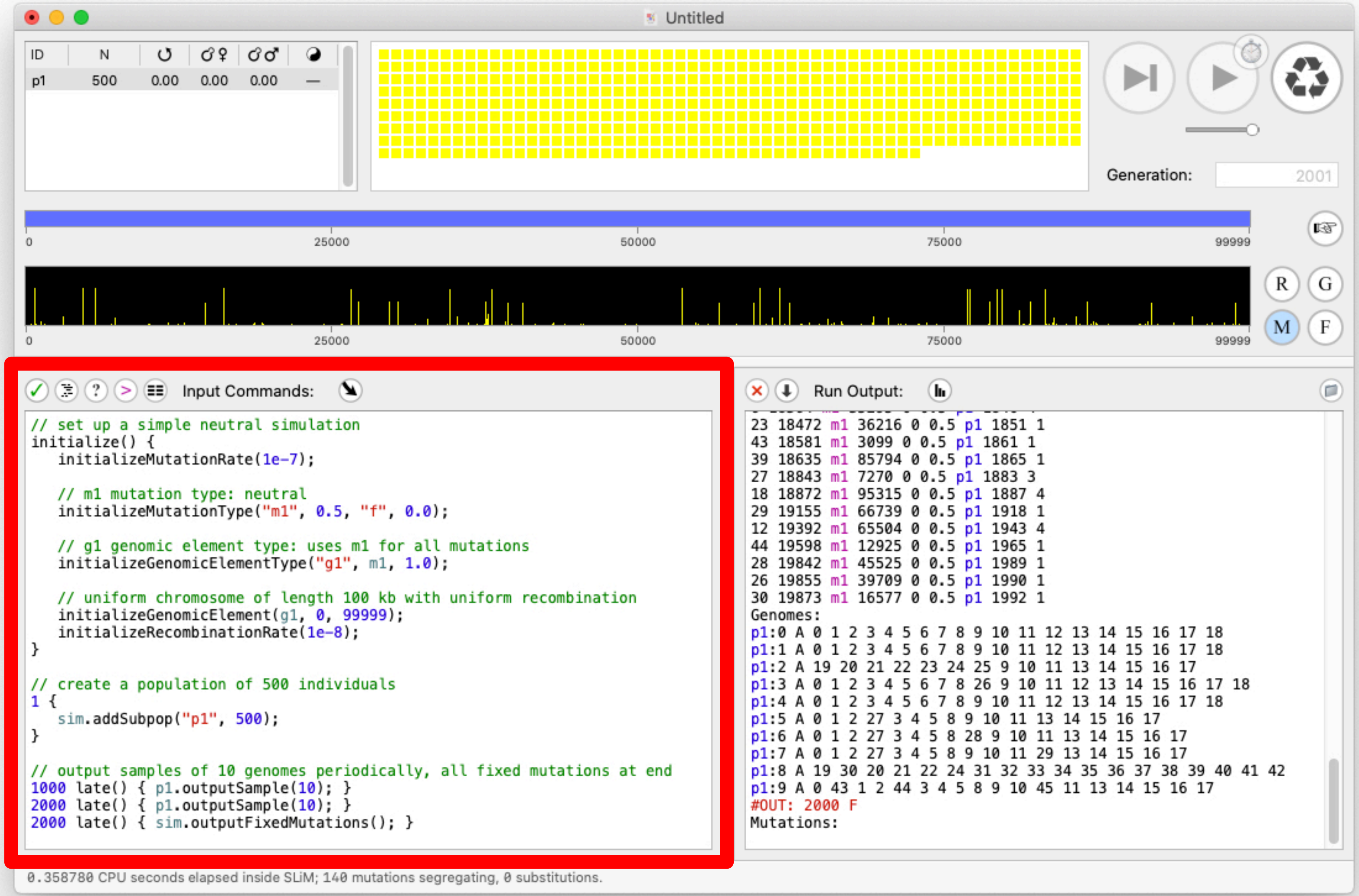


```
// set up a simple neutral simulation
initialize() {
   initializeMutationRate(1e-7);

   // m1 mutation type: neutral
   initializeMutationType("m1", 0.5, "f", 0.0);

   // g1 genomic element type: uses m1 for all mutations
   initializeGenomicElementType("g1", m1, 1.0);

   // uniform chromosome of length 100 kb with uniform recombination
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(1e-8);
}

// create a population of 500 individuals
1 {
   sim.addSubpop("p1", 500);
}

// output samples of 10 genomes periodically, all fixed mutations at end
1000 late() { p1.outputSample(10); }
2000 late() { p1.outputSample(10); }
2000 late() { sim.outputFixedMutations(); }
```

```
23 18472 m1 36216 0 0.5 p1 1851 1
43 18581 m1 3099 0 0.5 p1 1861 1
39 18635 m1 85794 0 0.5 p1 1865 1
27 18843 m1 7270 0 0.5 p1 1883 3
18 18872 m1 95315 0 0.5 p1 1887 4
29 19155 m1 66739 0 0.5 p1 1918 1
12 19392 m1 65504 0 0.5 p1 1943 4
44 19598 m1 12925 0 0.5 p1 1965 1
28 19842 m1 45525 0 0.5 p1 1989 1
26 19855 m1 39709 0 0.5 p1 1990 1
30 19873 m1 16577 0 0.5 p1 1992 1
Genomes:
p1:0 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:1 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:2 A 19 20 21 22 23 24 25 9 10 11 13 14 15 16 17
p1:3 A 0 1 2 3 4 5 6 7 8 26 9 10 11 12 13 14 15 16 17 18
p1:4 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:5 A 0 1 2 27 3 4 5 8 9 10 11 13 14 15 16 17
p1:6 A 0 1 2 27 3 4 5 8 28 9 10 11 13 14 15 16 17
p1:7 A 0 1 2 27 3 4 5 8 9 10 11 29 13 14 15 16 17
p1:8 A 19 30 20 21 22 24 31 32 33 34 35 36 37 38 39 40 41 42
p1:9 A 0 43 1 2 44 3 4 5 8 9 10 45 11 13 14 15 16 17
#OUT: 2000 F
Mutations:
```

0.358780 CPU seconds elapsed inside SLiM; 140 mutations segregating, 0 substitutions.

Modeling in SLiMgui



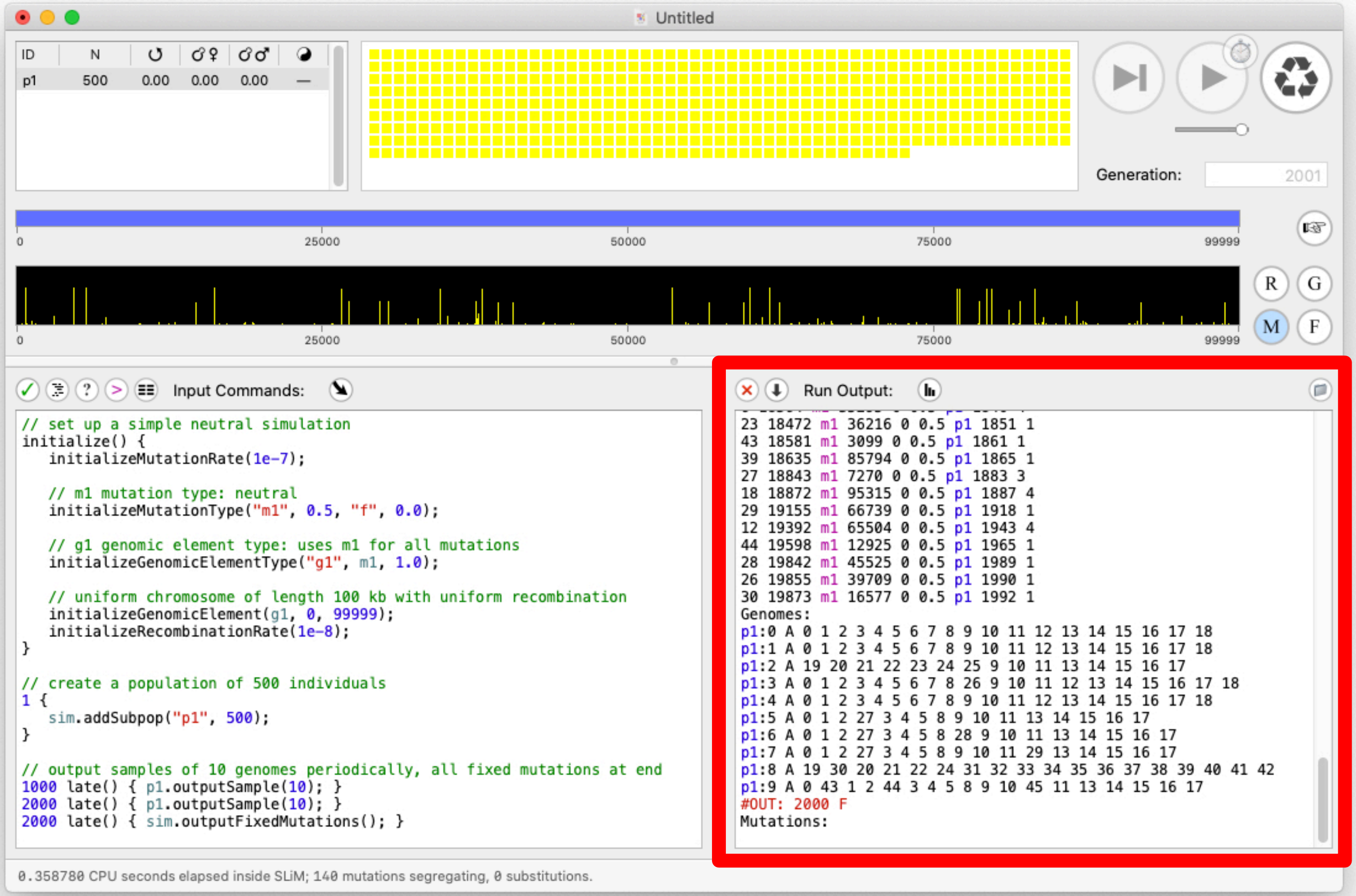
ID	N	↺	♂♀	♂♂	◕
p1	500	0.00	0.00	0.00	—

```
// set up a simple neutral simulation
initialize() {
   initializeMutationRate(1e-7);

   // m1 mutation type: neutral
   initializeMutationType("m1", 0.5, "f", 0.0);

   // g1 genomic element type: uses m1 for all mutations
   initializeGenomicElementType("g1", m1, 1.0);

   // uniform chromosome of length 100 kb with uniform recombination
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(1e-8);
}

// create a population of 500 individuals
1 {
   sim.addSubpop("p1", 500);
}

// output samples of 10 genomes periodically, all fixed mutations at end
1000 late() { p1.outputSample(10); }
2000 late() { p1.outputSample(10); }
2000 late() { sim.outputFixedMutations(); }
```

```
23 18472 m1 36216 0 0.5 p1 1851 1
43 18581 m1 3099 0 0.5 p1 1861 1
39 18635 m1 85794 0 0.5 p1 1865 1
27 18843 m1 7270 0 0.5 p1 1883 3
18 18872 m1 95315 0 0.5 p1 1887 4
29 19155 m1 66739 0 0.5 p1 1918 1
12 19392 m1 65504 0 0.5 p1 1943 4
44 19598 m1 12925 0 0.5 p1 1965 1
28 19842 m1 45525 0 0.5 p1 1989 1
26 19855 m1 39709 0 0.5 p1 1990 1
30 19873 m1 16577 0 0.5 p1 1992 1
Genomes:
p1:0 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:1 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:2 A 19 20 21 22 23 24 25 9 10 11 13 14 15 16 17
p1:3 A 0 1 2 3 4 5 6 7 8 26 9 10 11 12 13 14 15 16 17 18
p1:4 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:5 A 0 1 2 27 3 4 5 8 9 10 11 13 14 15 16 17
p1:6 A 0 1 2 27 3 4 5 8 28 9 10 11 13 14 15 16 17
p1:7 A 0 1 2 27 3 4 5 8 9 10 11 29 13 14 15 16 17
p1:8 A 19 30 20 21 22 24 31 32 33 34 35 36 37 38 39 40 41 42
p1:9 A 0 43 1 2 44 3 4 5 8 9 10 45 11 13 14 15 16 17
#OUT: 2000 F
Mutations:
```

0.358780 CPU seconds elapsed inside SLiM; 140 mutations segregating, 0 substitutions.

Modeling in SLiMgui

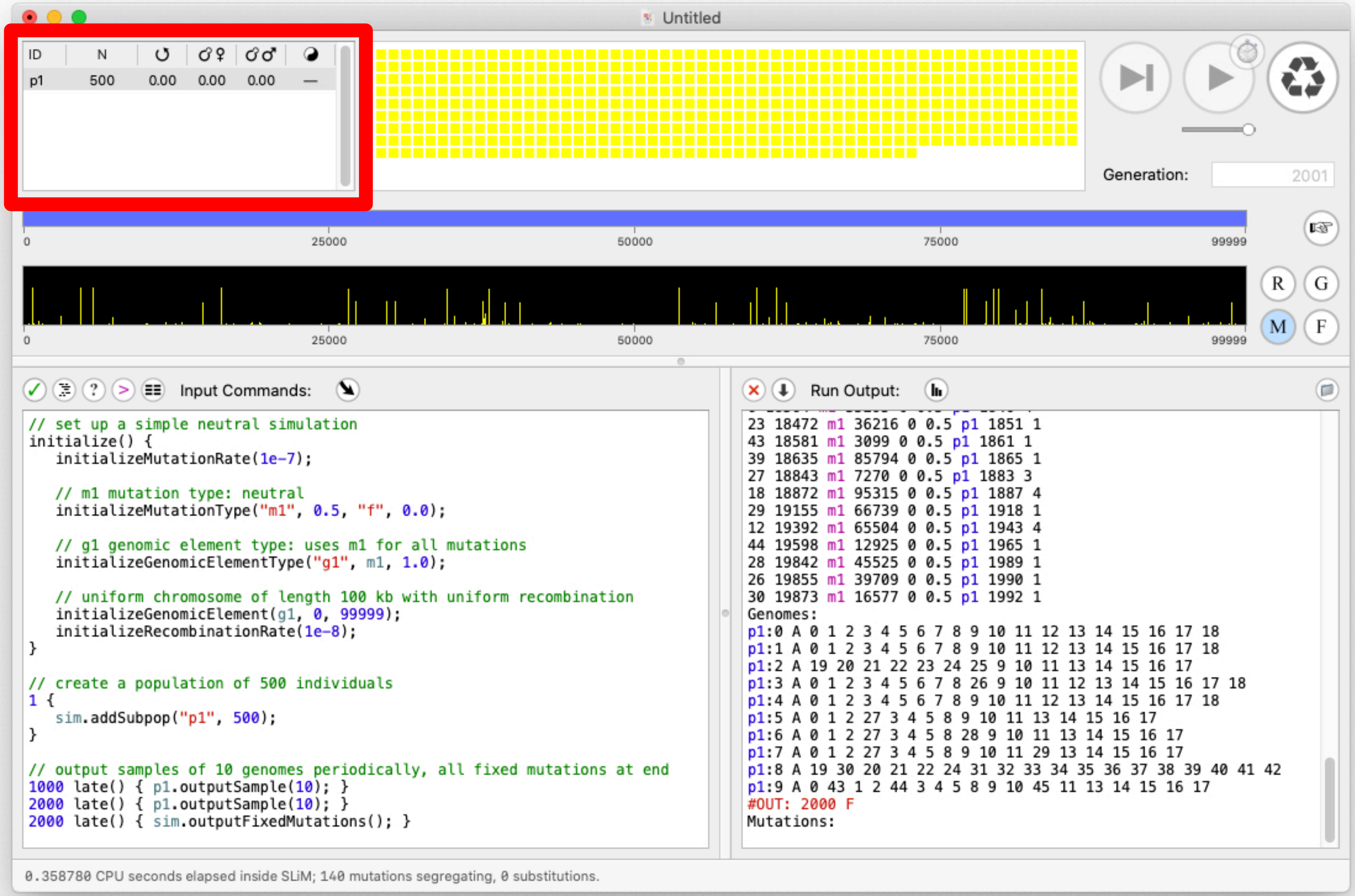


```
// set up a simple neutral simulation
initialize() {
   initializeMutationRate(1e-7);

   // m1 mutation type: neutral
   initializeMutationType("m1", 0.5, "f", 0.0);

   // g1 genomic element type: uses m1 for all mutations
   initializeGenomicElementType("g1", m1, 1.0);

   // uniform chromosome of length 100 kb with uniform recombination
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(1e-8);
}

// create a population of 500 individuals
1 {
   sim.addSubpop("p1", 500);
}

// output samples of 10 genomes periodically, all fixed mutations at end
1000 late() { p1.outputSample(10); }
2000 late() { p1.outputSample(10); }
2000 late() { sim.outputFixedMutations(); }
```

```
23 18472 m1 36216 0 0.5 p1 1851 1
43 18581 m1 3099 0 0.5 p1 1861 1
39 18635 m1 85794 0 0.5 p1 1865 1
27 18843 m1 7270 0 0.5 p1 1883 3
18 18872 m1 95315 0 0.5 p1 1887 4
29 19155 m1 66739 0 0.5 p1 1918 1
12 19392 m1 65504 0 0.5 p1 1943 4
44 19598 m1 12925 0 0.5 p1 1965 1
28 19842 m1 45525 0 0.5 p1 1989 1
26 19855 m1 39709 0 0.5 p1 1990 1
30 19873 m1 16577 0 0.5 p1 1992 1
Genomes:
p1:0 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:1 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:2 A 19 20 21 22 23 24 25 9 10 11 13 14 15 16 17
p1:3 A 0 1 2 3 4 5 6 7 8 26 9 10 11 12 13 14 15 16 17 18
p1:4 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:5 A 0 1 2 27 3 4 5 8 9 10 11 13 14 15 16 17
p1:6 A 0 1 2 27 3 4 5 8 28 9 10 11 13 14 15 16 17
p1:7 A 0 1 2 27 3 4 5 8 9 10 11 29 13 14 15 16 17
p1:8 A 19 30 20 21 22 24 31 32 33 34 35 36 37 38 39 40 41 42
p1:9 A 0 43 1 2 44 3 4 5 8 9 10 45 11 13 14 15 16 17
#OUT: 2000 F
Mutations:
```

0.358780 CPU seconds elapsed inside SLiM; 140 mutations segregating, 0 substitutions.

Modeling in SLiMgui



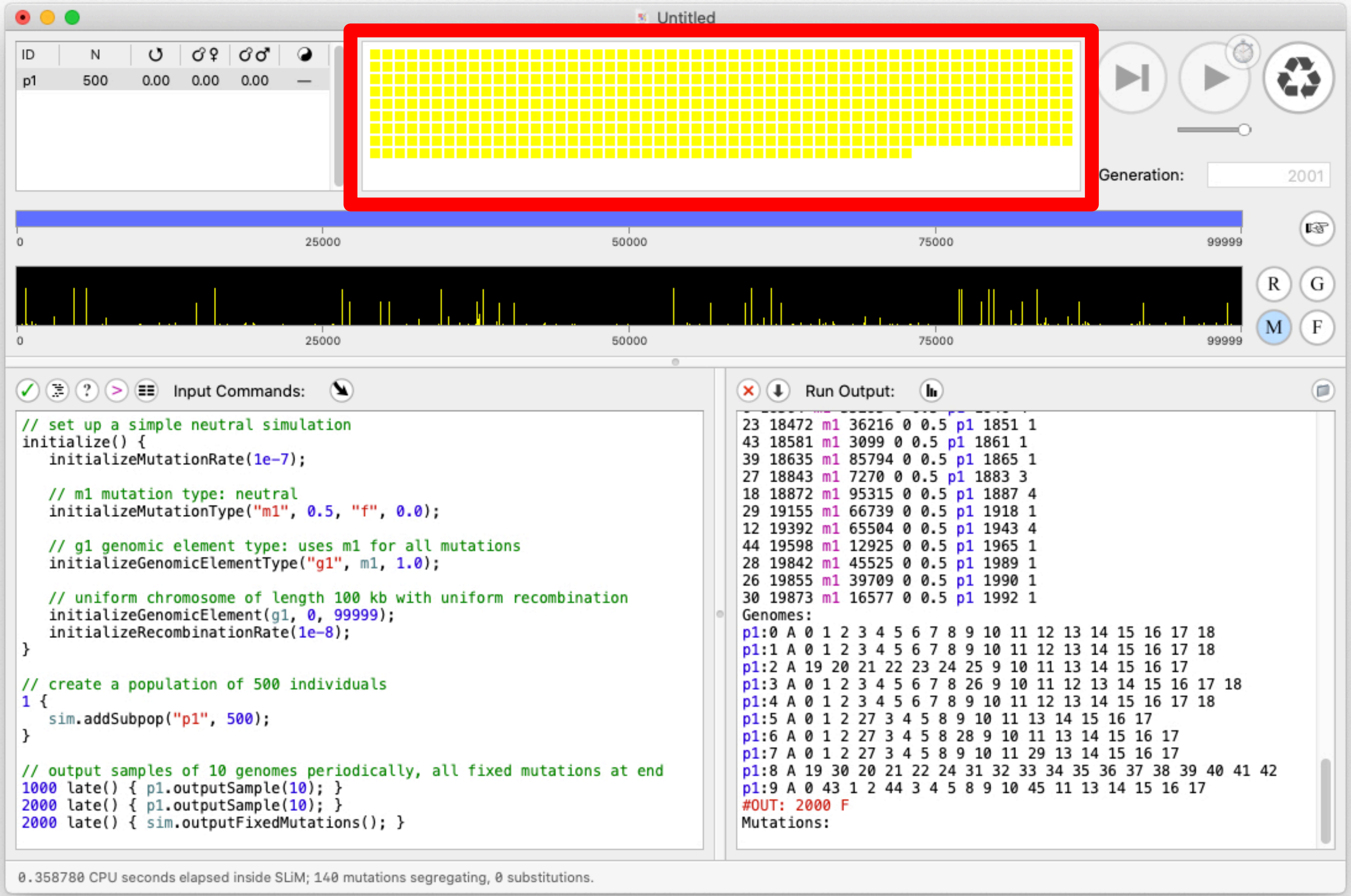
ID	N	↺	♂♀	♂♂	◑
p1	500	0.00	0.00	0.00	—

```
// set up a simple neutral simulation
initialize() {
   initializeMutationRate(1e-7);

   // m1 mutation type: neutral
   initializeMutationType("m1", 0.5, "f", 0.0);

   // g1 genomic element type: uses m1 for all mutations
   initializeGenomicElementType("g1", m1, 1.0);

   // uniform chromosome of length 100 kb with uniform recombination
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(1e-8);
}

// create a population of 500 individuals
1 {
   sim.addSubpop("p1", 500);
}

// output samples of 10 genomes periodically, all fixed mutations at end
1000 late() { p1.outputSample(10); }
2000 late() { p1.outputSample(10); }
2000 late() { sim.outputFixedMutations(); }
```

```
23 18472 m1 36216 0 0.5 p1 1851 1
43 18581 m1 3099 0 0.5 p1 1861 1
39 18635 m1 85794 0 0.5 p1 1865 1
27 18843 m1 7270 0 0.5 p1 1883 3
18 18872 m1 95315 0 0.5 p1 1887 4
29 19155 m1 66739 0 0.5 p1 1918 1
12 19392 m1 65504 0 0.5 p1 1943 4
44 19598 m1 12925 0 0.5 p1 1965 1
28 19842 m1 45525 0 0.5 p1 1989 1
26 19855 m1 39709 0 0.5 p1 1990 1
30 19873 m1 16577 0 0.5 p1 1992 1
Genomes:
p1:0 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:1 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:2 A 19 20 21 22 23 24 25 9 10 11 13 14 15 16 17
p1:3 A 0 1 2 3 4 5 6 7 8 26 9 10 11 12 13 14 15 16 17 18
p1:4 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:5 A 0 1 2 27 3 4 5 8 9 10 11 13 14 15 16 17
p1:6 A 0 1 2 27 3 4 5 8 28 9 10 11 13 14 15 16 17
p1:7 A 0 1 2 27 3 4 5 8 9 10 11 29 13 14 15 16 17
p1:8 A 19 30 20 21 22 24 31 32 33 34 35 36 37 38 39 40 41 42
p1:9 A 0 43 1 2 44 3 4 5 8 9 10 45 11 13 14 15 16 17
#OUT: 2000 F
Mutations:
```

0.358780 CPU seconds elapsed inside SLiM; 140 mutations segregating, 0 substitutions.

Modeling in SLiMgui



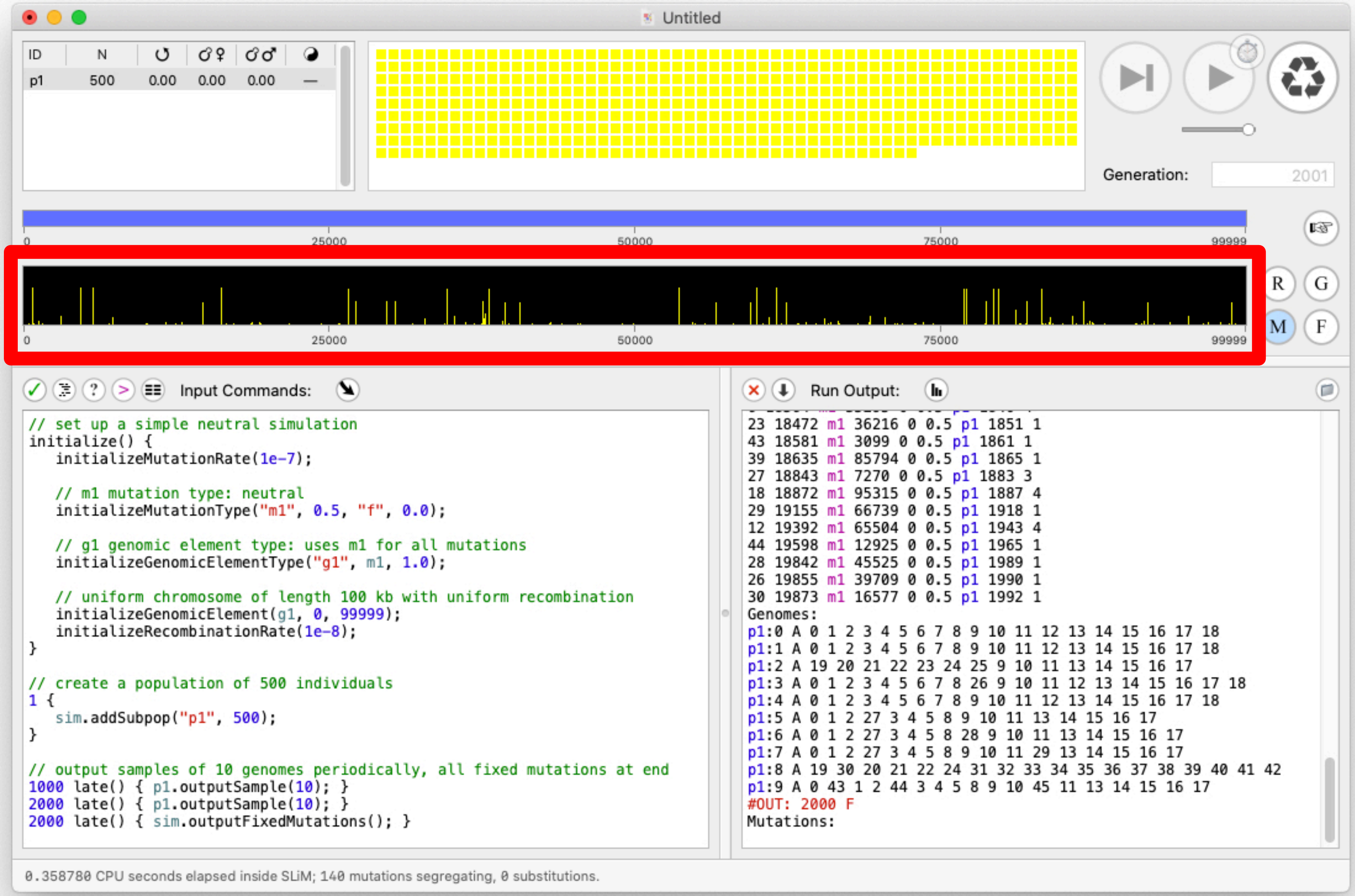
ID	N	↺	♂♀	♂♂	◕
p1	500	0.00	0.00	0.00	—

```
// set up a simple neutral simulation
initialize() {
   initializeMutationRate(1e-7);

   // m1 mutation type: neutral
   initializeMutationType("m1", 0.5, "f", 0.0);

   // g1 genomic element type: uses m1 for all mutations
   initializeGenomicElementType("g1", m1, 1.0);

   // uniform chromosome of length 100 kb with uniform recombination
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(1e-8);
}

// create a population of 500 individuals
1 {
   sim.addSubpop("p1", 500);
}

// output samples of 10 genomes periodically, all fixed mutations at end
1000 late() { p1.outputSample(10); }
2000 late() { p1.outputSample(10); }
2000 late() { sim.outputFixedMutations(); }
```

```
23 18472 m1 36216 0 0.5 p1 1851 1
43 18581 m1 3099 0 0.5 p1 1861 1
39 18635 m1 85794 0 0.5 p1 1865 1
27 18843 m1 7270 0 0.5 p1 1883 3
18 18872 m1 95315 0 0.5 p1 1887 4
29 19155 m1 66739 0 0.5 p1 1918 1
12 19392 m1 65504 0 0.5 p1 1943 4
44 19598 m1 12925 0 0.5 p1 1965 1
28 19842 m1 45525 0 0.5 p1 1989 1
26 19855 m1 39709 0 0.5 p1 1990 1
30 19873 m1 16577 0 0.5 p1 1992 1
Genomes:
p1:0 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:1 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:2 A 19 20 21 22 23 24 25 9 10 11 13 14 15 16 17
p1:3 A 0 1 2 3 4 5 6 7 8 26 9 10 11 12 13 14 15 16 17 18
p1:4 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:5 A 0 1 2 27 3 4 5 8 9 10 11 13 14 15 16 17
p1:6 A 0 1 2 27 3 4 5 8 28 9 10 11 13 14 15 16 17
p1:7 A 0 1 2 27 3 4 5 8 9 10 11 29 13 14 15 16 17
p1:8 A 19 30 20 21 22 24 31 32 33 34 35 36 37 38 39 40 41 42
p1:9 A 0 43 1 2 44 3 4 5 8 9 10 45 11 13 14 15 16 17
#OUT: 2000 F
Mutations:
```

0.358780 CPU seconds elapsed inside SLiM; 140 mutations segregating, 0 substitutions.

Modeling in SLiMgui

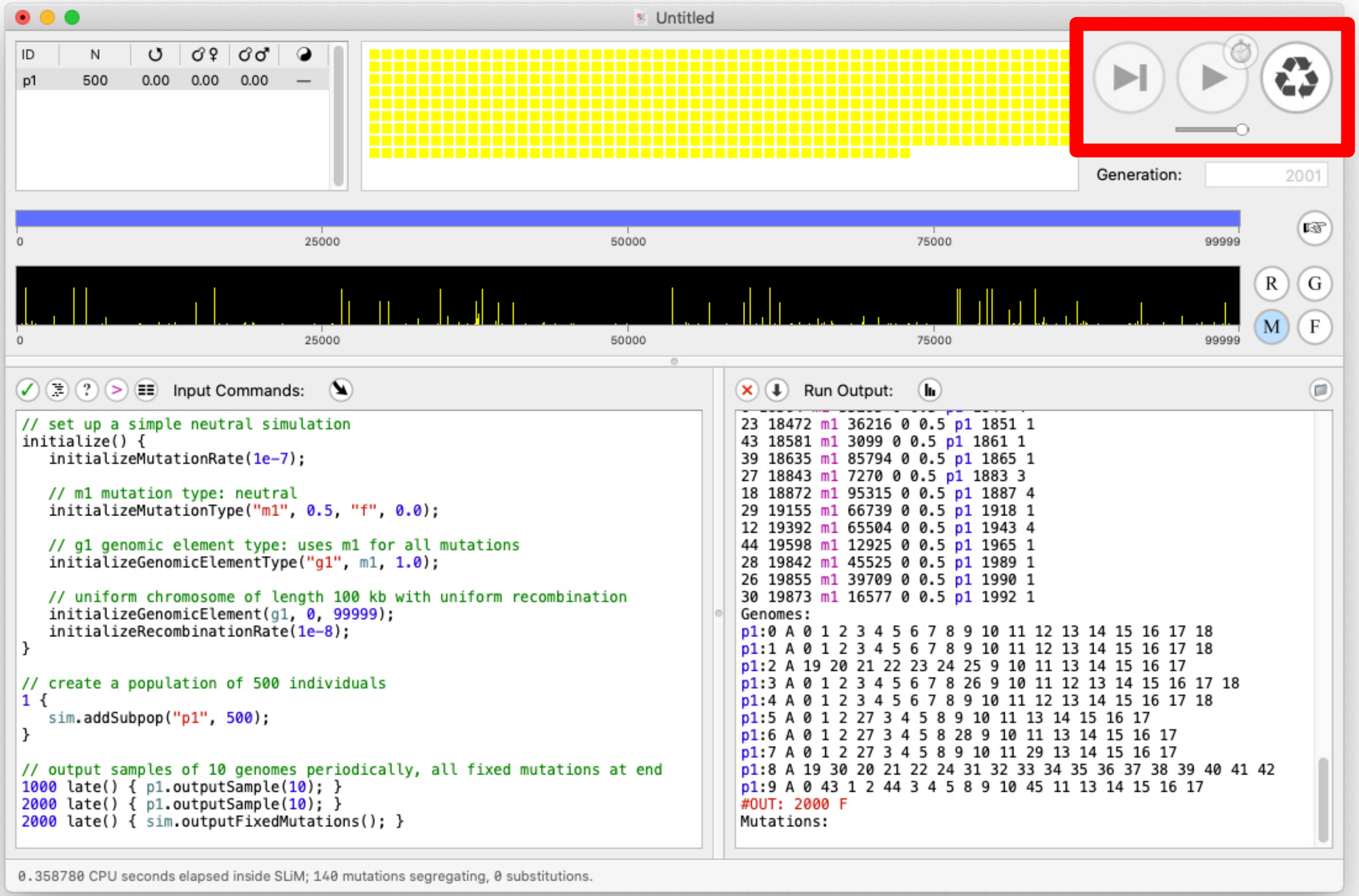


```
// set up a simple neutral simulation
initialize() {
   initializeMutationRate(1e-7);

   // m1 mutation type: neutral
   initializeMutationType("m1", 0.5, "f", 0.0);

   // g1 genomic element type: uses m1 for all mutations
   initializeGenomicElementType("g1", m1, 1.0);

   // uniform chromosome of length 100 kb with uniform recombination
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(1e-8);
}

// create a population of 500 individuals
1 {
   sim.addSubpop("p1", 500);
}

// output samples of 10 genomes periodically, all fixed mutations at end
1000 late() { p1.outputSample(10); }
2000 late() { p1.outputSample(10); }
2000 late() { sim.outputFixedMutations(); }
```

```
23 18472 m1 36216 0 0.5 p1 1851 1
43 18581 m1 3099 0 0.5 p1 1861 1
39 18635 m1 85794 0 0.5 p1 1865 1
27 18843 m1 7270 0 0.5 p1 1883 3
18 18872 m1 95315 0 0.5 p1 1887 4
29 19155 m1 66739 0 0.5 p1 1918 1
12 19392 m1 65504 0 0.5 p1 1943 4
44 19598 m1 12925 0 0.5 p1 1965 1
28 19842 m1 45525 0 0.5 p1 1989 1
26 19855 m1 39709 0 0.5 p1 1990 1
30 19873 m1 16577 0 0.5 p1 1992 1
Genomes:
p1:0 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:1 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:2 A 19 20 21 22 23 24 25 9 10 11 13 14 15 16 17
p1:3 A 0 1 2 3 4 5 6 7 8 26 9 10 11 12 13 14 15 16 17 18
p1:4 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:5 A 0 1 2 27 3 4 5 8 9 10 11 13 14 15 16 17
p1:6 A 0 1 2 27 3 4 5 8 28 9 10 11 13 14 15 16 17
p1:7 A 0 1 2 27 3 4 5 8 9 10 11 29 13 14 15 16 17
p1:8 A 19 30 20 21 22 24 31 32 33 34 35 36 37 38 39 40 41 42
p1:9 A 0 43 1 2 44 3 4 5 8 9 10 45 11 13 14 15 16 17
#OUT: 2000 F
Mutations:
```

0.358780 CPU seconds elapsed inside SLiM; 140 mutations segregating, 0 substitutions.

Modeling in SLiMgui

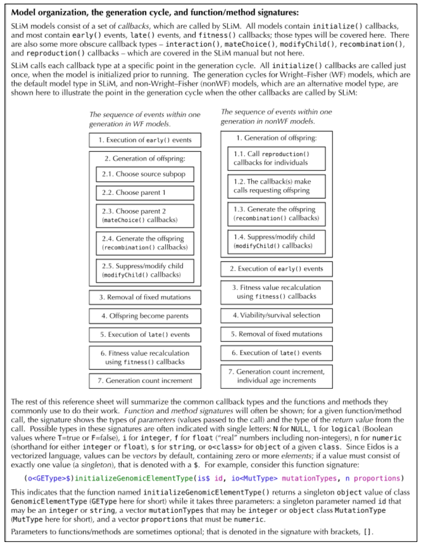


```
// set up a simple neutral simulation
initialize() {
   initializeMutationRate(1e-7);

   // m1 mutation type: neutral
   initializeMutationType("m1", 0.5, "f", 0.0);

   // g1 genomic element type: uses m1 for all mutations
   initializeGenomicElementType("g1", m1, 1.0);

   // uniform chromosome of length 100 kb with uniform recombination
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(1e-8);
}

// create a population of 500 individuals
1 {
   sim.addSubpop("p1", 500);
}

// output samples of 10 genomes periodically, all fixed mutations at end
1000 late() { p1.outputSample(10); }
2000 late() { p1.outputSample(10); }
2000 late() { sim.outputFixedMutations(); }
```

```
23 18472 m1 36216 0 0.5 p1 1851 1
43 18581 m1 3099 0 0.5 p1 1861 1
39 18635 m1 85794 0 0.5 p1 1865 1
27 18843 m1 7270 0 0.5 p1 1883 3
18 18872 m1 95315 0 0.5 p1 1887 4
29 19155 m1 66739 0 0.5 p1 1918 1
12 19392 m1 65504 0 0.5 p1 1943 4
44 19598 m1 12925 0 0.5 p1 1965 1
28 19842 m1 45525 0 0.5 p1 1989 1
26 19855 m1 39709 0 0.5 p1 1990 1
30 19873 m1 16577 0 0.5 p1 1992 1
Genomes:
p1:0 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:1 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:2 A 19 20 21 22 23 24 25 9 10 11 13 14 15 16 17
p1:3 A 0 1 2 3 4 5 6 7 8 26 9 10 11 12 13 14 15 16 17 18
p1:4 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
p1:5 A 0 1 2 27 3 4 5 8 9 10 11 13 14 15 16 17
p1:6 A 0 1 2 27 3 4 5 8 28 9 10 11 13 14 15 16 17
p1:7 A 0 1 2 27 3 4 5 8 9 10 11 29 13 14 15 16 17
p1:8 A 19 30 20 21 22 24 31 32 33 34 35 36 37 38 39 40 41 42
p1:9 A 0 43 1 2 44 3 4 5 8 9 10 45 11 13 14 15 16 17
#OUT: 2000 F
Mutations:
```

0.358780 CPU seconds elapsed inside SLiM; 140 mutations segregating, 0 substitutions.

Beginner reference sheet



Model organization, the generation cycle, and function/method signatures:

SLiM models consist of a set of *callbacks*, which are called by SLiM. All models contain `initialize()` callbacks, and most contain `early()` events, `late()` events, and `fitness()` callbacks; those types will be covered here. There are also some more obscure callback types – `interaction()`, `mateChoice()`, `modifyChild()`, `recombination()`, and `reproduction()` callbacks – which are covered in the SLiM manual but not here.

SLiM calls each callback type at a specific point in the generation cycle. All `initialize()` callbacks are called just once, when the model is initialized prior to running. The generation cycles for Wright–Fisher (WF) models, which are the default model type in SLiM, and non-Wright–Fisher (nonWF) models, which are an alternative model type, are shown here to illustrate the point in the generation cycle when the other callbacks are called by SLiM:

The sequence of events within one generation in WF models.

1. Execution of `early()` events
2. Generation of offspring:
   - 2.1. Choose source subpop
   - 2.2. Choose parent 1
   - 2.3. Choose parent 2 (`mateChoice()` callbacks)
   - 2.4. Generate the offspring (`recombination()` callbacks)
   - 2.5. Suppress/modify child (`modifyChild()` callbacks)
3. Removal of fixed mutations
4. Offspring become parents
5. Execution of `late()` events
6. Fitness value recalculation using `fitness()` callbacks
7. Generation count increment

The sequence of events within one generation in nonWF models.

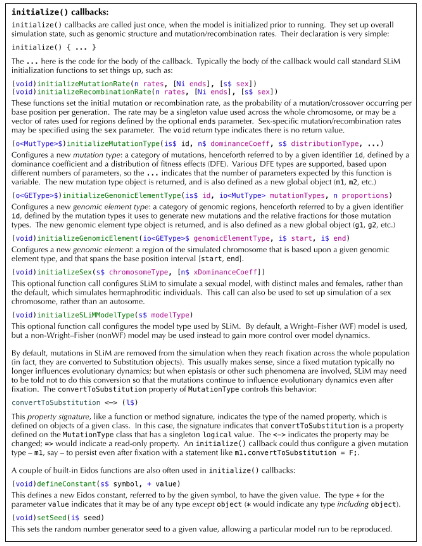
1. Generation of offspring:
   - 1.1. Call `reproduction()` callbacks for individuals
   - 1.2. The callback(s) make calls requesting offspring
   - 1.3. Generate the offspring (`recombination()` callbacks)
   - 1.4. Suppress/modify child (`modifyChild()` callbacks)
2. Execution of `early()` events
3. Fitness value recalculation using `fitness()` callbacks
4. Viability/survival selection
5. Removal of fixed mutations
6. Execution of `late()` events
7. Generation count increment, individual age increments

The rest of this reference sheet will summarize the common callback types and the functions and methods they commonly use to do their work. *Function* and *method signatures* will often be shown; for a given function/method call, the signature shows the types of *parameters* (values passed to the call) and the type of the *return value* from the call. Possible types in these signatures are often indicated with single letters: `N` for `NULL`, `l` for `logical` (Boolean values where `T`=true or `F`=false), `i` for `integer`, `f` for `float` ("real" numbers including non-integers), `n` for `numeric` (shorthand for either `integer` or `float`), `s` for `string`, or `o<class>` for `object` of a given `class`. Since Eidos is a vectorized language, values can be *vectors* by default, containing zero or more *elements*; if a value must consist of exactly one value (a *singleton*), that is denoted with a `$`. For example, consider this function signature:

```
(o<GEType>$)initializeGenomicElementType(is$ id, io<MutType> mutationTypes, n proportions)
```

This indicates that the function named `initializeGenomicElementType()` returns a singleton `object` value of class `GenomicElementType` (`GEType` here for short) while it takes three parameters: a singleton parameter named `id` that may be an `integer` or `string`, a vector `mutationTypes` that may be `integer` or `object` class `MutationType` (`MutType` here for short), and a vector `proportions` that must be `numeric`.

Parameters to functions/methods are sometimes optional; that is denoted in the signature with brackets, `[]`.



`initialize()` callbacks:

`initialize()` callbacks are called just once, when the model is initialized prior to running. They set up overall simulation state, such as genomic structure and mutation/recombination rates. Their declaration is very simple:

```
initialize() { ... }
```

The `...` here is the code for the body of the callback. Typically the body of the callback would call standard SLiM initialization functions to set things up, such as:

```
(void)initializeMutationRate(n rates, [Ni ends], [s$ sex])
(void)initializeRecombinationRate(n rates, [Ni ends], [s$ sex])
```

These functions set the initial mutation or recombination rate, as the probability of a mutation/crossover occurring per base position per generation. The rate may be a singleton value used across the whole chromosome, or may be a vector of rates used for regions defined by the optional `ends` parameter. Sex-specific mutation/recombination rates may be specified using the `sex` parameter. The `void` return type indicates there is no return value.

```
(o<MutType>$)initializeMutationType(is$ id, n$ dominanceCoeff, s$ distributionType, ...)
```

Configures a new *mutation type*: a category of mutations, henceforth referred to by a given identifier `id`, defined by a dominance coefficient and a distribution of fitness effects (DFE). Various DFE types are supported, based upon different numbers of parameters, so the `...` indicates that the number of parameters expected by this function is variable. The new mutation type object is returned, and is also defined as a new global object (`m1`, `m2`, etc.)

```
(o<GEType>$)initializeGenomicElementType(is$ id, io<MutType> mutationTypes, n proportions)
```

Configures a new *genomic element type*: a category of genomic regions, henceforth referred to by a given identifier `id`, defined by the mutation types it uses to generate new mutations and the relative fractions for those mutation types. The new genomic element type object is returned, and is also defined as a new global object (`g1`, `g2`, etc.)

```
(void)initializeGenomicElement(io<GEType>$ genomicElementType, i$ start, i$ end)
```

Configures a new *genomic element*: a region of the simulated chromosome that is based upon a given genomic element type, and that spans the base position interval [`start`, `end`].

```
(void)initializeSex(s$ chromosomeType, [n$ xDominanceCoeff])
```

This optional function call configures SLiM to simulate a sexual model, with distinct males and females, rather than the default, which simulates hermaphroditic individuals. This call can also be used to set up simulation of a sex chromosome, rather than an autosome.

```
(void)initializeSLiMModelType(s$ modelType)
```

This optional function call configures the model type used by SLiM. By default, a Wright–Fisher (WF) model is used, but a non-Wright–Fisher (nonWF) model may be used instead to gain more control over model dynamics.

By default, mutations in SLiM are removed from the simulation when they reach fixation across the whole population (in fact, they are converted to Substitution objects). This usually makes sense, since a fixed mutation typically no longer influences evolutionary dynamics; but when epistasis or other such phenomena are involved, SLiM may need to be told not to do this conversion so that the mutations continue to influence evolutionary dynamics even after fixation. The `convertToSubstitution` property of `MutationType` controls this behavior:

```
convertToSubstitution <–> (l$)
```

This *property signature*, like a function or method signature, indicates the type of the named property, which is defined on objects of a given class. In this case, the signature indicates that `convertToSubstitution` is a property defined on the `MutationType` class that has a singleton `logical` value. The `<–>` indicates the property may be changed; `=>` would indicate a read-only property. An `initialize()` callback could thus configure a given mutation type – `m1`, say – to persist even after fixation with a statement like `m1.convertToSubstitution = F;`.

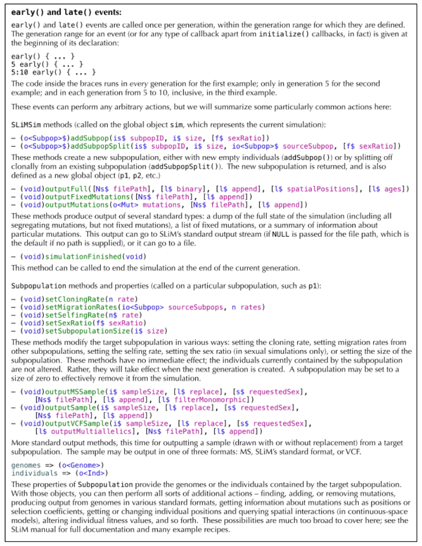
A couple of built-in Eidos functions are also often used in `initialize()` callbacks:

```
(void)defineConstant(s$ symbol, + value)
```

This defines a new Eidos constant, referred to by the given symbol, to have the given value. The type `+` for the parameter `value` indicates that it may be of any type *except* `object` (`*` would indicate any type *including* `object`).

```
(void)setSeed(i$ seed)
```

This sets the random number generator seed to a given value, allowing a particular model run to be reproduced.



`early()` and `late()` events:

`early()` and `late()` events are called once per generation, within the generation range for which they are defined. The generation range for an event (or for any type of callback apart from `initialize()` callbacks, in fact) is given at the beginning of its declaration:

```
early() { ... }
5 early() { ... }
5:10 early() { ... }
```

The code inside the braces runs in *every* generation for the first example; only in generation 5 for the second example; and in each generation from 5 to 10, inclusive, in the third example.

These events can perform any arbitrary actions, but we will summarize some particularly common actions here:

`SLiMSim` methods (called on the global object `sim`, which represents the current simulation):

```
– (o<Subpop>$)addSubpop(is$ subpopID, i$ size, [f$ sexRatio])
– (o<Subpop>$)addSubpopSplit(is$ subpopID, i$ size, io<Subpop>$ sourceSubpop, [f$ sexRatio])
```

These methods create a new subpopulation, either with new empty individuals (`addSubpop()`) or by splitting off clonally from an existing subpopulation (`addSubpopSplit()`). The new subpopulation is returned, and is also defined as a new global object (`p1`, `p2`, etc.)

```
– (void)outputFull([Ns$ filePath], [l$ binary], [l$ append], [l$ spatialPositions], [l$ ages])
– (void)outputFixedMutations([Ns$ filePath], [l$ append])
– (void)outputMutations(o<Mut> mutations, [Ns$ filePath], [l$ append])
```

These methods produce output of several standard types: a dump of the full state of the simulation (including all segregating mutations, but not fixed mutations), a list of fixed mutations, or a summary of information about particular mutations. This output can go to SLiM's standard output stream (if `NULL` is passed for the file path, which is the default if no path is supplied), or it can go to a file.

```
– (void)simulationFinished(void)
```

This method can be called to end the simulation at the end of the current generation.

`Subpopulation` methods and properties (called on a particular subpopulation, such as `p1`):

```
– (void)setCloningRate(n rate)
– (void)setMigrationRates(io<Subpop> sourceSubpops, n rates)
– (void)setSelfingRate(n$ rate)
– (void)setSexRatio(f$ sexRatio)
– (void)setSubpopulationSize(i$ size)
```

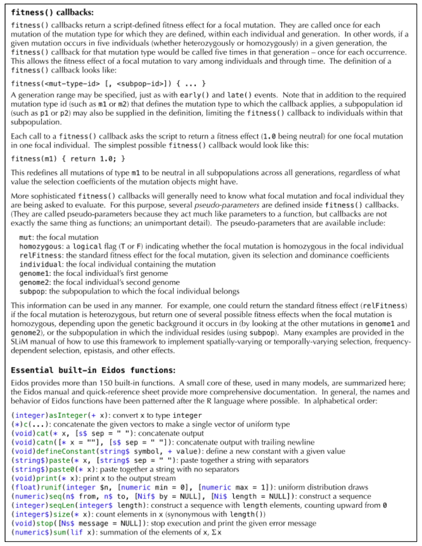
These methods modify the target subpopulation in various ways: setting the cloning rate, setting migration rates from other subpopulations, setting the selfing rate, setting the sex ratio (in sexual simulations only), or setting the size of the subpopulation. These methods have no immediate effect; the individuals currently contained by the subpopulation are not altered. Rather, they will take effect when the next generation is created. A subpopulation may be set to a size of zero to effectively remove it from the simulation.

```
– (void)outputMSSample(i$ sampleSize, [l$ replace], [s$ requestedSex],
     [Ns$ filePath], [l$ append], [l$ filterMonomorphic])
– (void)outputSample(i$ sampleSize, [l$ replace], [s$ requestedSex],
     [Ns$ filePath], [l$ append])
– (void)outputVCFSample(i$ sampleSize, [l$ replace], [s$ requestedSex],
     [l$ outputMultiallelics], [Ns$ filePath], [l$ append])
```

More standard output methods, this time for outputting a sample (drawn with or without replacement) from a target subpopulation. The sample may be output in one of three formats: MS, SLiM's standard format, or VCF.

```
genomes => (o<Genome>)
individuals => (o<Ind>)
```

These properties of `Subpopulation` provide the genomes or the individuals contained by the target subpopulation. With those objects, you can then perform all sorts of additional actions – finding, adding, or removing mutations, producing output from genomes in various standard formats, getting information about mutations such as positions or selection coefficients, getting or changing individual positions and querying spatial interactions (in continuous-space models), altering individual fitness values, and so forth. These possibilities are much too broad to cover here; see the SLiM manual for full documentation and many example recipes.



`fitness()` callbacks:

`fitness()` callbacks return a script-defined fitness effect for a focal mutation. They are called once for each mutation of the mutation type for which they are defined, within each individual and generation. In other words, if a given mutation occurs in five individuals (whether heterozygously or homozygously) in a given generation, the `fitness()` callback for that mutation type would be called five times in that generation – once for each occurrence. This allows the fitness effect of a focal mutation to vary among individuals and through time. The definition of a `fitness()` callback looks like:

```
fitness(<mut-type-id> [, <subpop-id>]) { ... }
```

A generation range may be specified, just as with `early()` and `late()` events. Note that in addition to the required mutation type id (such as `m1` or `m2`) that defines the mutation type to which the callback applies, a subpopulation id (such as `p1` or `p2`) may also be supplied in the definition, limiting the `fitness()` callback to individuals within that subpopulation.

Each call to a `fitness()` callback asks the script to return a fitness effect (`1.0` being neutral) for one focal mutation in one focal individual. The simplest possible `fitness()` callback would look like this:

```
fitness(m1) { return 1.0; }
```

This redefines all mutations of type `m1` to be neutral in all subpopulations across all generations, regardless of what value the selection coefficients of the mutation objects might have.

More sophisticated `fitness()` callbacks will generally need to know what focal mutation and focal individual they are being asked to evaluate. For this purpose, several *pseudo-parameters* are defined inside `fitness()` callbacks. (They are called pseudo-parameters because they act much like parameters to a function, but callbacks are not exactly the same thing as functions; an unimportant detail). The pseudo-parameters that are available include:

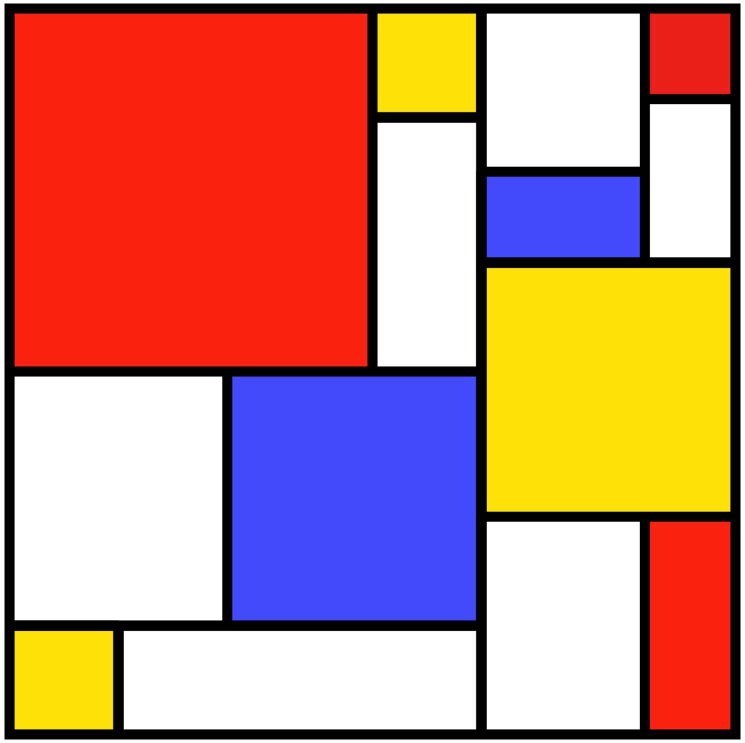
- `mut`: the focal mutation
- `homozygous`: a `logical` flag (`T` or `F`) indicating whether the focal mutation is homozygous in the focal individual
- `relFitness`: the standard fitness effect for the focal mutation, given its selection and dominance coefficients
- `individual`: the focal individual containing the mutation
- `genome1`: the focal individual's first genome
- `genome2`: the focal individual's second genome
- `subpop`: the subpopulation to which the focal individual belongs

This information can be used in any manner. For example, one could return the standard fitness effect (`relFitness`) if the focal mutation is heterozygous, but return one of several possible fitness effects when the focal mutation is homozygous, depending upon the genetic background it occurs in (by looking at the other mutations in `genome1` and `genome2`), or the subpopulation in which the individual resides (using `subpop`). Many examples are provided in the SLiM manual of how to use this framework to implement spatially-varying or temporally-varying selection, frequency-dependent selection, epistasis, and other effects.

`Essential built-in Eidos functions:`

Eidos provides more than 150 built-in functions. A small core of these, used in many models, are summarized here; the Eidos manual and quick-reference sheet provide more comprehensive documentation. In general, the names and behavior of Eidos functions have been patterned after the R language where possible. In alphabetical order:

- `(integer)asInteger(+ x)`: convert x to type `integer`
- `(*)c(...)`: concatenate the given vectors to make a single vector of uniform type
- `(void)cat(* x, [s$ sep = " "])`: concatenate output
- `(void)catn([* x = ""], [s$ sep = " "])`: concatenate output with trailing newline
- `(void)defineConstant(string$ symbol, + value)`: define a new constant with a given value
- `(string$)paste(* x, [string$ sep = " "])`: paste together a string with separators
- `(string$)paste0(* x)`: paste together a string with no separators
- `(void)print(* x)`: print x to the output stream
- `(float)runif(integer $n, [numeric min = 0], [numeric max = 1])`: uniform distribution draws
- `(numeric)seq(n$ from, n$ to, [Nif$ by = NULL], [Ni$ length = NULL])`: construct a sequence
- `(integer)seqLen(integer$ length)`: construct a sequence with `length` elements, counting upward from `0`
- `(integer$)size(* x)`: count elements in x (synonymous with `length()`)
- `(void)stop([Ns$ message = NULL])`: stop execution and print the given error message
- `(numeric$)sum(lif x)`: summation of the elements of x, Σx



SLiM Workshop Exercise #2