SLiM Workshop Series #7: SLiMgui Shortcuts

1. Start a new model in SLiMgui

Run SLiMgui, and close any existing model windows. Press command-N to make a new window.

We've already seen a few nice features and shortcuts in SLiMgui that accelerate your workflow. Here we're going to gather them all into one place, and introduce a few new ones. Some of this will be review, for reinforcement, but hopefully some of it will be new and useful.

2. Code completion

Click after the opening brace of `initialize()` and press return to make a new line. Then type "iS" and press the escape key. This initiates auto-completion: SLiM finds likely symbols starting with the prefix you supplied. In this case, we want to complete to `initializeSex()`, but it is not the first choice provided, so press the down-arrow key until it is selected and then press return to choose it. If we had completed on "iSe" instead – a more suggestive prefix – `initializeSex()` would have been the first choice.

Note that code completion is case-sensitive, and that is important. By completing on "iS", we told SLiMgui that we wanted something like a two-word symbol where the second word started with "S". (The matching is somewhat fuzzy, so it also accepts `initializeSLiMOptions()`, with a third word that we didn't specify, and `initializeTreeSequence()`, with an intervening word.) If you complete on "is" instead – all lowercase – you will get matches for things like `isInteger()` and `isFinite()` that "iS" does not match. (Indeed, toward the bottom of the list you will even be offered things like `exists()` and `rpois()`, both of which contain "is".)

3. Function/method signatures in the status bar

You should now have this in your script:

```
initialize() {
initializeSex()
   initializeMutationRate(1e-7);
   ...
```

The next question is what `initializeSex()` expects to be passed. Click in between the two parentheses for that function call, and the status bar at the bottom of the window will display the function signature as a reminder:

```
(void)initializeSex(string$ chromosomeType,
[numeric$ xDominanceCoeff = 1])
```

This is where knowing how to interpret Eidos function/method signatures comes in handy. The first parameter, `chromosomeType`, must be a singleton `string` (the singleton requirement being indicated by the `$`). The second, `xDominanceCoeff`, must be either `integer` or `float` (`numeric` is a shorthand for that), and must also be a singleton; that parameter is optional, however, as indicated by the brackets around it, with a default value of `1`.

4. Option-click for help

Perhaps we're still not sure what to pass to `initializeSex()`; what sort of singleton string value does it expect? Option-click on "initializeSex" to open the help window with a search on that term. That clarifies things; unless we want to simulate a sex chromosome, chromosomeType should be `"A"` for "autosome". Enter `"A"` now as the parameter to `initializeSex()`.

5. Check script

Your script should now have this:

```
initialize() {
initializeSex("A")
   initializeMutationRate(1e-7);
   ...
```

Are we good to go? We could try recycling the model to find out; but that would reset the current simulation state, clear the output pane, and so forth. Perhaps we'd like to just find out whether our script is error-free without recycling. To do this, press the "check script" button:

In response, SLiMgui will display an error panel saying:

> ERROR (EidosScript::Match): unexpected token '@initializeMutationRate' in expression statement; expected ';'.

The same error message is also displayed in the status bar at the bottom of the window, so you can see it even after you dismiss the error panel. SLiMgui also highlights `initializeMutationRate` since that token occurs in the script where Eidos expected a semicolon to be (ending the previous line's statement). Fair enough; add a semicolon after `initializeSex()`, and then click the "check script" button again to confirm that the script is now free of syntax errors.

6. Prettyprinting

Now your script should have something like this:

```
initialize() {
initializeSex("A");
   initializeMutationRate(1e-7);
   ...
```

This is fine, except that the indentation of the i line is poor. We could go in and add a tab at the beginning of the line to clean it up; but there's an easier way, especially if the indentation of a bunch of lines is wrong. Click the "prettyprint script" button above the script input pane:

The indentation is automatically fixed to follow standard Eidos coding conventions:

```
initialize() {
   initializeSex("A");
   initializeMutationRate(1e-7);
   ...
```

You can try messing up the indentation of a bunch of the lines in the script; pressing "prettyprint script" should clean it all up instantly.

By default, prettyprinting only reformats your script by re-indenting lines. If you want it to do more, try holding down the option (alt) key and clicking the button. This does a more complete prettyprinting, and will change the whitespace within lines, the whitespace between script blocks, etc., to provide a reformatted script in a completely standard format.

Prettyprinting is undoable, so feel free to experiment with it a bit to see what it does.

7. Indent/outdent

The "prettyprint script" button cleans things up automatically, but sometimes you want to change the indentation of a block of code in a non-standard way. There's a shortcut for that in SLiMgui. Drag to select a range of text in the script from `initializeMutationRate()` through to `initializeGenomicElement()`. Then press command-] (right bracket). The selected text will be indented by one tab stop; i.e., an extra tab will be inserted at the start of each line.

Hmm, that doesn't look great. Press command-[(left bracket) to outdent the text by one tab stop. Problem solved! (Or if not, for some reason, then press "prettyprint script" to clean it up.)

8. Selecting whole lines

This is a macOS pro tip, not a SLiMgui tip specifically. It's often convenient to select entire lines at a time, including tabs at the start and the newline at the end. You can then cut the selected lines and paste them somewhere else, without indentation and newlines getting screwed up.

To select one entire line, you triple-click: click the mouse three times in rapid succession. Try triple-clicking on `initializeMutationRate` to select that whole line. Then cut the line by pressing command-X, click before the ending brace `}` of the `initialize()` callback, and paste by pressing command-V. The whole line has been cleanly moved.

Even more useful is dragging out a selection composed of entire lines. To do this, you triple-click, but on the third click you keep the mouse button pressed instead of releasing it. You are now drag-selecting whole lines until you release the mouse button. Try triple-click-dragging from `initializeRecombinationRate` through to `initializeMutationRate` to select both lines in their entirety; then cut and paste to move them to the beginning of the `initialize()` callback, before `initializeSex()`.

9. Comment/uncomment

Suppose we want to disable all of the output events in this model, but we don't want to remove the code for them entirely; this is just a temporary change. You could add a comment token, `//`, at the beginning of each line to comment them all out, but there's an easier way. First, triple-click-drag to select all three lines. Then press command-/ (forward slash) to comment out the selected lines.

Once you're done, and you want the code to be active again, the same steps will uncomment them: triple-click-drag to select the relevant lines, then press command-/ to uncomment them again. SLiMgui sees that the lines are already commented out, and uncomments them for you.

10. Find/replace shortcuts

This is another macOS shortcut; it works in all apps that follow Apple's standard user interface guidelines (and should work in SLiMgui even on Linux). Suppose you want to find every occurrence of the word "output" in this script. You could press command-F to bring up the Find user interface (find: "F"), and then type "output", but there's an easier way. Select the word "output" anywhere it occurs, and press command-E. The selection is now entered (enter: "E") as the find string. You could press command-F now to use the Find user interface, and you would see "output" there. But again there's an easier way: press command-G to go (go: "G") to the next occurrence of "output", without ever even showing the Find user interface. To go to the previous occurrence, press shift-command-G.

11. Find Recipe

While we're on the topic of finding things, suppose you want to find a recipe that uses initializeSex. You could search through the manual PDF, but how tedious is that? Instead, go to the **File** menu, to the **Open Recipe** submenu, and choose **Find Recipe…** to open a panel that will help you. In

the first Keywords field, type "initializeSex" (without the quotes). Notice that as you type, the list of recipes in the box below filters down to just the recipes that contain the given keyword.

Now you can click on the entries in that list to see a preview of their code, with their usage of the keyword `initializeSex` highlighted. This should make it easy to find the particular recipe you want.

12. Execute to a given generation

We've made a few changes to out model. Now we'd like to execute up to generation 1000, and then single-step across that generation to see it generate its output sample. Unless you've got extraordinarily quick reflexes, it's hard to click the Play button in such a way as to stop at exactly generation 1000. Instead, recycle the simulation, then click in the Generation textfield. Type "1000" (without the quotes) and press return. The simulation will run to the start of generation 1000 and stop. Many people don't realize that the Generation textfield is editable until this is pointed out!

13. Changing the working directory

When running under SLiMgui, the default "working directory" where files will be read and written (unless a full Unix-style absolute path is given) is the user's Desktop folder. You may wish to change this, so that you can base your runs in a particular working directory without having to hard-code the path to that directory in your script. To do this, just click the folder-icon button:

SLiMgui will ask you to choose your preferred working directory.

14. The splitview dividers

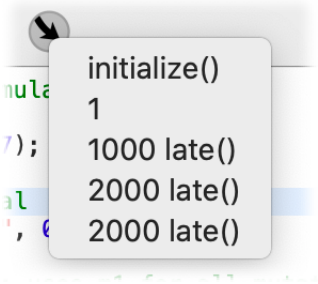
The SLiMgui window has various sub-panes: the input script pane, the output pane, and the pane at the top that contains components like the chromosome view, the individuals view, and the population table. Depending on what you're doing, you might want to focus more on one or another of these panes. It's easy to miss them, but there are draggable splitters between the panes of the window, marked with little circular indentations:

You can click and drag those to resize the panes inside the window. They will snap at particular spots to constrain the panes to reasonable sizes; but if you continue to pull them, all the way to the edge of the window, you can completely hide one or another of the window's panes if you wish.

15. The Jump button

This model is quite short, but when working on a large SLiM model it can start to be difficult to navigate around the code and find the place you want to be. For this problem, SLiMgui provides the Jump button:

If you click on it, you'll see a pop-up menu with all of the events and callbacks in your model:



If you select one of these items, you will see that SLiMgui will select that line in your script. Complex SLiM models often have a dozen or more events and callbacks, spanning pages of code; the Jump menu then proves invaluable.

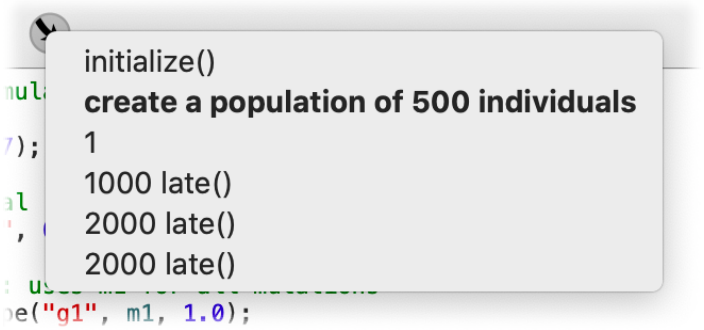
But it does more than this. There's a comment in the default script that reads:

```
// create a population of 500 individuals
```

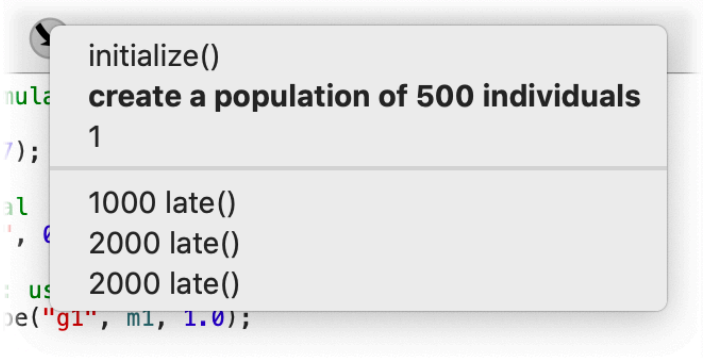
Add a third slash to the start of this line, so that it starts with `///`. This is a special comment format that the Jump menu recognizes; a block comment with an extra asterisk, like:

```
/** try this! */
```

also works. With this change, click on the Jump button again. You will see that your special comment has resulted in the addition of a new line, in bold, in the Jump menu:



You can even divide the Jump menu into subsections for organizational purposes. Try adding a comment that is simply `///`, with no text following, above the three output events. Now the Jump menu will have a divider in it:

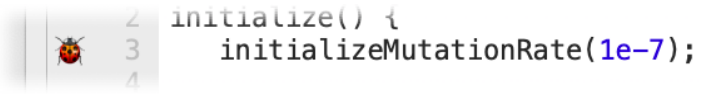


The Jump button may not seem very exciting right now, but keep it in the back of your mind for when you start writing larger SLiM models.

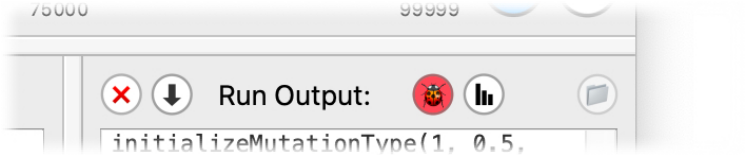
16. Debug points

Debugging is an art, and skill at it takes lots of practice. It often requires a lot of detective work; in essence, you need to walk through all of your assumptions about your model, and test them one by one to confirm that they are true – that your code is doing what you expect it to do. If you do this methodically, at some point you will generally find that a particular spot in your code is behaving differently than you expected it to, and that will be the source of the bug that you observe.

To help with these investigations, SLiMgui provides "debug points". These are points that you have marked in your code, in SLiMgui's interface; when they are executed, SLiMgui provides debugging output that describes what happened during their execution. Try clicking in the left-hand margin of the script pane to add a debug point, like this:



Click where the ladybug icon is shown in the screenshot above, and you should be able to set your own debug point. Now recycle and run the model. You may notice a button flashing red:



This is the Debugging Output button; it flashes when new debugging out is generated by debug points (or other things). Click it, and a new window will open with some output:

```
#DEBUG CALL (line 3, gen 0): call to function initializeMutationRate() arguments:
    rates == float [0:0] 1e-07
    ends == NULL
    sex == string [0:0] "*"
#DEBUG CALL (line 3, gen 0): function initializeMutationRate() return: void
```

Here you can see details of the function call, its parameters, and its return value. This works for many kinds of statements; assignments with operator = will log out the value assigned, for example.

17. Other semi-hidden features

It's time to wrap up, but there are two more useful features in SLiMgui that are pretty hidden. One is the variable browser, which displays the names and values of all of the Eidos symbols defined in the simulation. Open it by clicking its button, above the script input pane:

The other is the tables drawer, opened with the pointing hand button:

These are both fairly self-explanatory; try recycling your model and then playing it to some middle generation, and have a look at the information displayed in the variable browser and the tables drawer. These facilities can prove quite useful for debugging!

16. BEEP! No recommended reading; just relax and bask in your knowledge of SLiMgui.