Assignment 3 patching

Electronic Music II

Spring 2014

1. This demonstration will present the concepts at play in Assignment 3, and how to patch them in Max.
   a. These concepts will be presented in this order:
      i. Creating [sfplay~]
      ii. Onset delay
      iii. Pitch shift
      iv. Gain and panning
      v. Filename automation
      vi. Tying it all together
   b. I have attempted to list these in order from least to most complicated.
   c. NB: When setting out on a patching project, it is **extremely helpful** to "zoom out", look at the desired results or goals are, and isolate components that contribute to those results or goals.
2. Creating [sfplay~]
   a. For this assignment we will be creating a stereo [sfplay~] object.
   b. The basic implementation of this object has been laid out in an earlier handout; since, by the end of this assignment we will have replaced several of the basic components in that handout, for now we only need the object, and a "1" message box.

3. Onset delay
   a. This component will delay the beginning of playback from our [sfplay~] object.
   b. We can accomplish this with two new objects: [random] and [delay].
   c. The function of [trigger] has already been demonstrated. To explain briefly, [random *n*] will produce a random integer between 0 and $n - 1$.
   d. [delay] is typically abbreviated into [del]. It has two inlets:
      i. The left inlet accepts bangs.
      ii. The right inlet accepts float or integer numbers, to set the delay time.
   e. The function of [delay] is to *delay* a bang by a set number of milliseconds.
   f. So, our onset delay component will resemble the following:

g. To explain briefly, the [trigger] first causes [random 100] to output a random integer from 0-99, into the right inlet (delay-setting inlet) of [del]. Then, [trigger] sends a bang into the [del] object, which will emerge from the outlet after the time set by the [random] object.

h. Since this bang emerging from [del] will need to drive several components, we will leave this component 'floating' for the time being.

4. Pitch shift

a. Recall that the frequency ratio between a pitch and an equal-tempered semitone above that pitch is (approximately) 1:1.0595.

b. To create intervals wider than one semitone, either above or below the original pitch, we need to raise 1.0595 to specific exponents. So, the ratio of *two* semitones up would be 1:(1.0595 ^ 2).

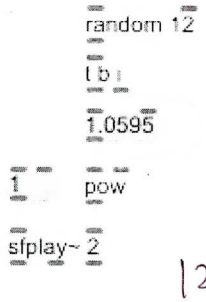
c. We will accomplish this using a [random] object, and the [pow] object.

d. Recall that the [pow] object raises whatever is passed to its left inlet, to the exponent given either by its creation argument or by its right inlet.

e. Our component to create the necessary ratios for equal-tempered semitone transposition, therefore, will resemble the following:

f. Briefly, [random] is generating a value between 0-11, effectively our interval of transposition. [trigger] then sends this integer to the right inlet of [pow], and then bangs the message containing "1.0595" to send it through the left inlet of [pow]. The result will be 1.0595 ^ *n*, where *n* is the output from [random], AKA our transposition interval.

g. Since this output has to be sent to the right inlet of [sfplay~ 2], we can connect that presently:



pitch transposition

random 24 - 12

for ascending + descending } -11 to 11 semitones

$\sqrt[12]{2}$ = semitones

$\sqrt[1200]{2}$ = cents

5. Gain and panning

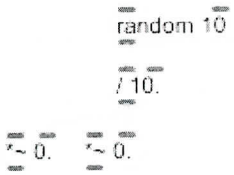
   a. Since these effects both alter the amplitude of our output audio signal, we will be using two pairs of [*~] objects. Recall the function of these objects, laid out in earlier presentations.

   b. We will be using [random] objects to set values for both layers. Beginning with gain:

      i. Create one [random] object, and set the creation argument as 10.

      ii. Connect the output of this object to a [/ 10.] object.

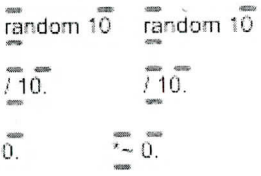
      iii. Connect the outlet of the [/ 10.] object to the right inlets of one pair of [*~]'s. The completed result is shown below:



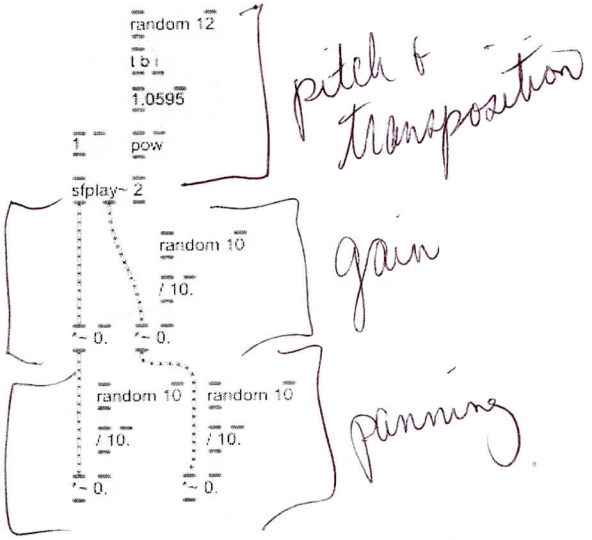
      iv. So, [random 10] creates a value between 0-9; [/ 10.] divides it by 10 and outputs the result as a float number. This float number is then used to change the amplitude of any signal passed through the [*~] objects.

      v. [*~ 0.] objects are used in this demonstration, just as a precaution/paranoia against Max truncating decimal places as they are sent into the [*~] objects.

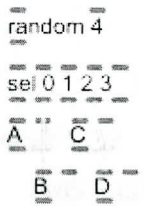
   c. Panning will be created in a similar fashion, but this time using one [random 10] > [/ 10.] setup **per [*~]**. See below.



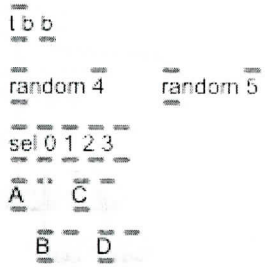
   d. Since these two components will be modifying the output of [sfplay~ 2], we can connect them as shown:



6. Filename automation
   a. This component will be covered in two sections: the [random]/[sel] section, and the [pack]/[sprintf] section.
   b. The new object used in the first section, [select], is typically abbreviated to [sel]. It takes input into its left inlet, attempts to match that input to one of its creation argument, and if there **is** a match it will output a bang through the outlet matched to that creation argument. If there **is not** a match the input will be passed **as is** to the rightmost outlet.
   c. We will assume that the files we are opening with [sfplay~] are named "A-1.aiff", "B-2.aiff", "Z-26.aiff", etc.
   d. To randomly choose from between several letters, we will be using [random] to create random integers, and [select] to trigger messages **that correspond to those random integers**. See below.



   e. To create the second component of the filename, we will need simply a [random] object, with an argument sufficient to include the full scope of our naming system (ie, if we have 5 files of each class, we need [random 5]).
   f. To coordinate these two objects, we will use a [trigger] object:



4 files in each class, we need 4.

   g. Notice that the integer-generating [random] object is being triggered first. This will be explained more fully in the description of the second section of this component.
7. Assembling messages from lists, part 1: [pack]
   a. To create a list of predetermined data types, use the [pack] object.
   b. The [pack] object by default has two inlets and one outlet. The object is shown below:

   c. To speak basically, the inlets store input, and the outlet outputs a list of the inputs. However there are several refinements to this statement that must be made if [pack] is to be used effectively.
   d. The inlets of [pack] represent elements of a list. Each inlet only effects its own element of the list; so, the right inlet can't be used to modify the first element of the list, only the second element.

e. Much like the arithmetic objects demonstrated earlier, the left and right inlets of [pack] have different effects on the output.

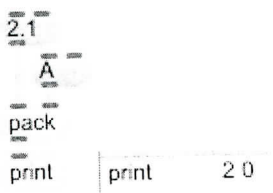
i. The right inlet will only store its input in the list, replacing any previous input sent to this inlet. It will not output the list.

ii. The left inlet will store its input in the list just like the right inlet. **However**, it will also **output the list** using its most recent input and the right inlet's most recent input.

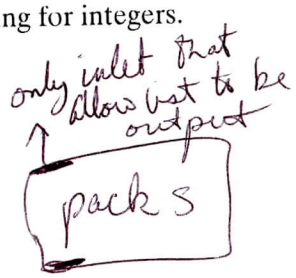
iii. A bang sent to the left inlet will output the list using each inlet's most recent input (bang cannot be packed in a list).

iv. So, the first important thing to remember when using [pack] is to ensure that data is passed to the inlets **in the correct sequence.**

f. The inlets of [pack] are 'looking for' a particular data type. By default, they are looking for integers. Consider the following patch and its resulting output:



Pack = ignores letters



i. Since the left inlet received a symbol, it wasn't stored in the list and the second element remained unchanged (so, 0).

ii. The right inlet received a **float** number, and so the decimal places were truncated to make it an **integer**.

iii. The data types can be specified using creation arguments.

iv. Each creation argument will create one inlet. Note that, with no arguments [pack] has two inlets (both for integers); with one argument [pack] has one inlet (looking for whatever data type is specified):

v. [pack] will always have only one outlet.

vi. Even as more inlets are added, it will always be the **left** inlet that causes the list to be output. Every inlet **to the right of the leftmost inlet** behaves as the right inlet in the two-inlet example.

g. To apply this to our hypothetical naming convention, we will need a symbol inlet and an integer inlet:

h. The output of this object, predictably, will not entirely meet our naming convention:

i. To place the elements of this list within the unchanging name elements ("-" and ".aiff"), we will need a second object.

8. Assembling messages from lists, part 2: [sprintf]

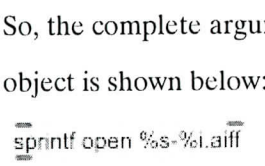
a. The [sprintf] object takes input much like [pack] and assembles it into a text string.

b. The object is extremely flexible; for this demo we are concerned with the following information:

   i. The creation argument given to [sprintf] is **what the output string will look like.**

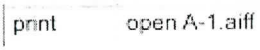
   ii. The input taken from a list will be inserted, element by element, where a "%" argument is present. These arguments represent data types much like the arguments in [pack], but **must be preceded by %.**

c. So, the complete argument we give to [sprintf] for this demo will be "open %s-%i.aiff". The complete object is shown below:

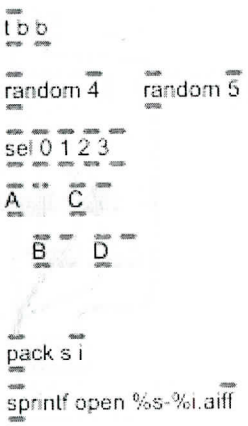


d. Connect the outlet of the [pack] object created in item 3.g to the left inlet of [sprintf].

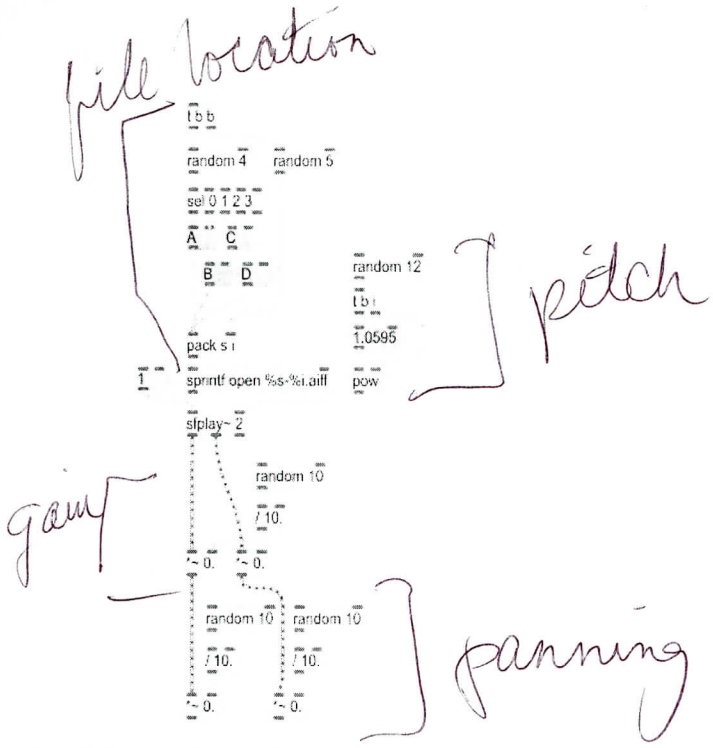
e. This results in the following output (using A and 1 as the list elements for [pack]):



f. This patch segment, then, can be connected first to our [random]/[sel] section:

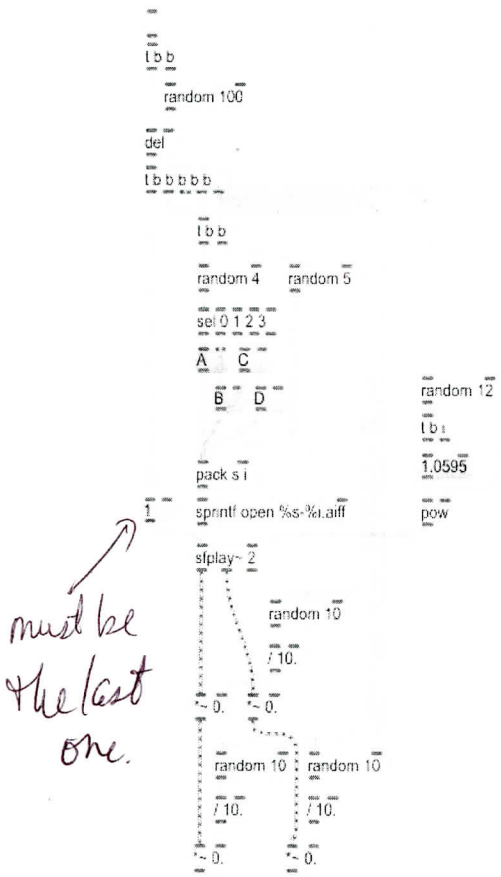


g. And then to our [sfplay~ 2] object:



9. Tying it all together

   a. To activate all these components in sequence, with a single bang, we will use a [trigger] object, as shown below:



   b. Notice that I, by preference/practice, connect one component to one bang from the [trigger].