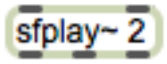
Introducing `sfplay~`

Electronic Music II

Spring 2013

A. `sfplay~`

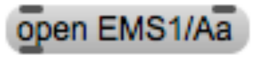
1. Create a blank object box, and type `sfplay~ 2`. This will create the `sfplay~` object, with 2 output channels (pictured below).



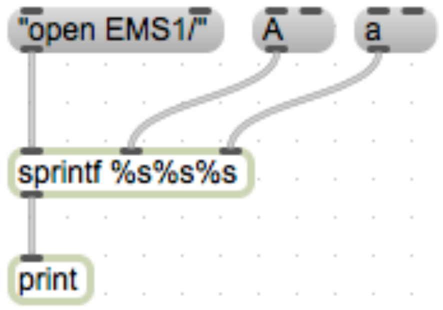
2. The use of `sfplay~` requires two particular pieces of information:
   a. An open message, with the path to the audio file
   b. A trigger to play. We will be using a message containing just the number 1.

B. To create an open message, we will use the `sprintf` object. This object accepts inputs of various types, and outputs a single message containing them all.

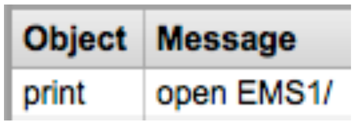
1. The message we need to create will look like this:



2. Looking at its components, we have:
   a. "open EMS1/" : Max considers this to be a **symbol** data type. Technically, it consists of **two** symbols, "open" and "EMS1/".
   b. "A" : another symbol
   c. "a" : another symbol
3. We can create separate message objects containing these three pieces of information, and send them to `sprintf`:

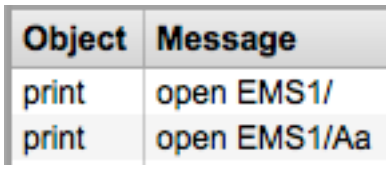


   a. A brief explanation of the arguments I gave `sprintf`: each %[something] creates a 'slot' in `sprintf`'s output for a specific data type. %s creates a slot for a symbol, and our target message contains three symbols. So, three %s's.
   b. For testing purposes, I have also created a `print` object. This will send our output text to the Max window; this is extremely useful for double checking patch components.
   c. Also notice that I enclosed "open EMS1/" in quotes. This is so that Max will see the entire message as one symbol, and not two individual symbols.
4. If we click the message boxes from left to right, the Max window shows us this output:



Object	Message
print	open EMS1/

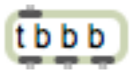
5. This is due to `sprintf`'s method of handling its inlets. The middle and right inlets accept their input, and store them in the respective position in the (eventual) output. The left inlet accepts its input, and immediately constructs an output list from whatever is stored in the object.
6. Since I clicked "open EMS1/" **first**, `sprintf` created a list from what it had: a symbol ("open EMS1/"), a blank symbol slot (added nothing to the output), and another blank symbol slot (also added nothing).
7. If we click the message boxes from left to right **a second time**, we see the following output:



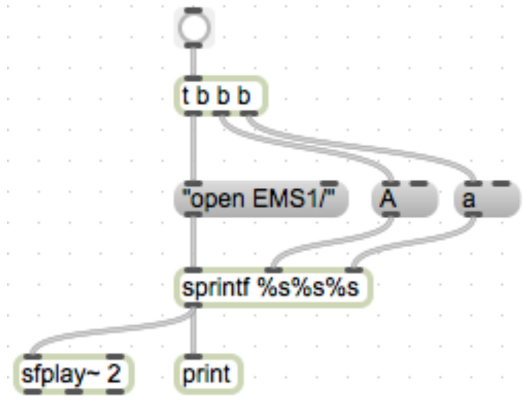
Object	Message
print	open EMS1/
print	open EMS1/Aa

(our first attempt, followed by our second attempt)

8. Now `sprintf` has output an appropriate open message for `sfplay~`.
9. Clicking the message boxes in the wrong order, twice, is **not** an ideal solution for our patch. We can ensure that the messages are triggered in the correct order by using the `trigger` object (abbreviated `t`).

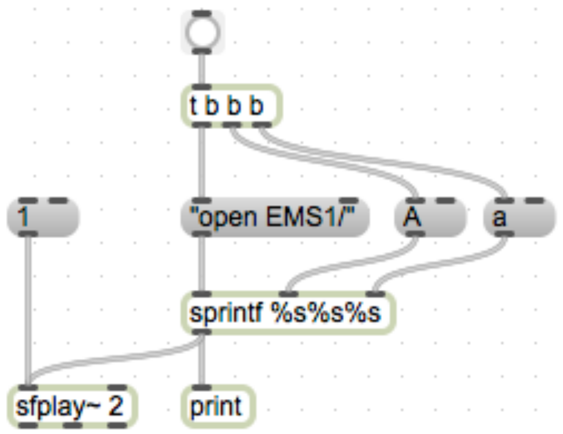


10. `trigger` outputs sequentially from right to left when it receives any input in its inlet. The object shown above will output three bangs, one each coming from the right, middle, then left outlet.
11. Once we connect the `trigger` outputs to our message boxes, and `sprintf` to `sfplay~ 2`, our basic open message-creating setup is complete.

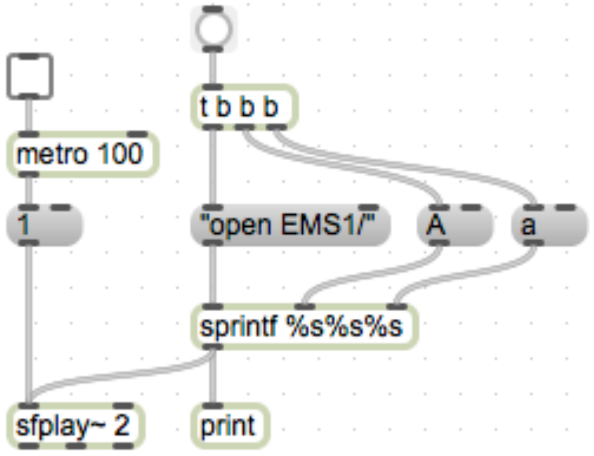


Note that I added a bang button, to initiate our `trigger` object. And for troubleshooting's sake, I've retained the `print` object.

C. The trigger for `sfplay~` to begin playing a file is any non-zero number. So, at its simplest, this would be a sufficient setup:

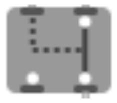


1. However, if we trigger this setup quickly enough, it will stop playing the selected sound file somewhere in the middle, and create a click. (demo)

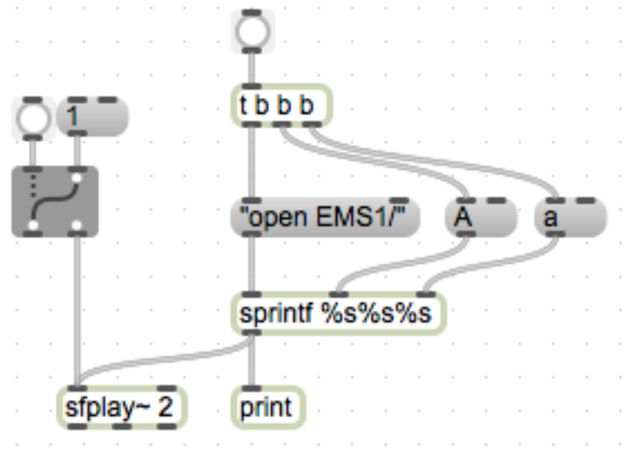


(a clicking setup)

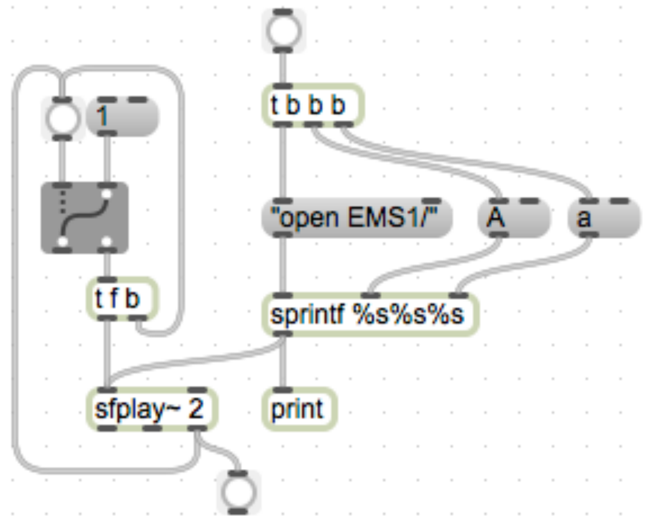
2. One method of preventing this, is to prevent a new trigger being sent to `sfplay~` before the current sound file is finished playing.
3. Luckily, `sfplay~`'s right-most outlet send out a bang when the file is finished. (demo)
4. We can take advantage of this, by introducing the `ggate` object (shown below).



5. `ggate` has two inlets:
   a. left : control. This changes the position of the switch, from one outlet to the other.
   b. right : data. This is the information that will pass out from the outlet corresponding to the switch's current position.
6. Adding `ggate` to our triggering setup, we get this:



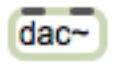
7. Right now the switching has to be done manually. If we want to automatically turn the switch off while a sound file is playing, and back on when it is done, we add the following objects:



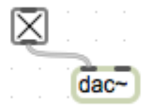
   a. Note that I have connected the right outlet of `sfplay~` directly to `ggate`. The bang button I added is purely for troubleshooting, so that I can be sure when it's sending a bang.
   b. Also note the `trigger` object I added: first it sends a bang back to the `ggate`, to close it, and then it sends the trigger number to `sfplay~`.

8. Now we are protected from cutting a sound file short and getting a click.

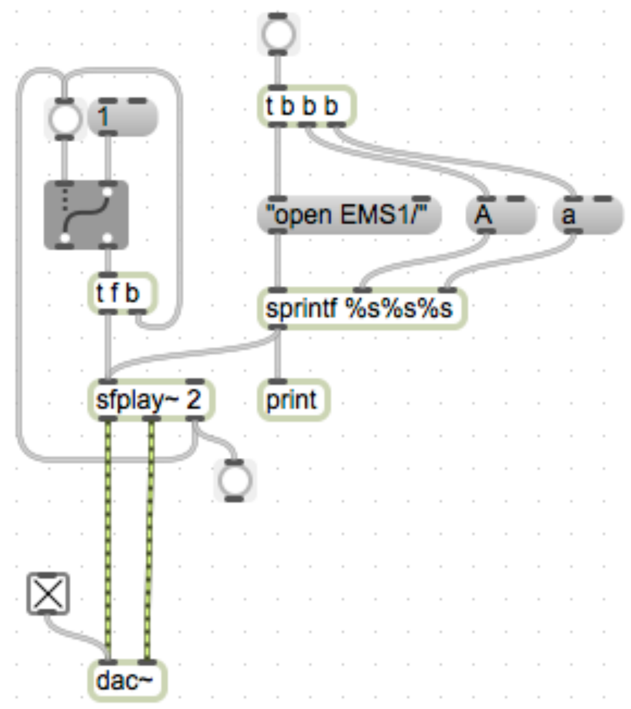
D. Now the last step will be to hear the output of `sfplay~`. We accomplish this using `dac~`.
   1. Create a blank object and type `dac~`. Notice that it defaults to 2 channels (stereo output).



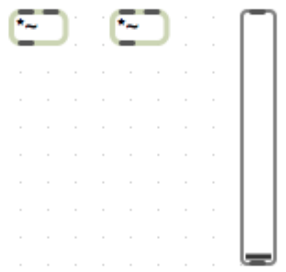
   2. To make `dac~` start outputting audio, we need to tell it to start. An effective way to accomplish this is with a toggle object.



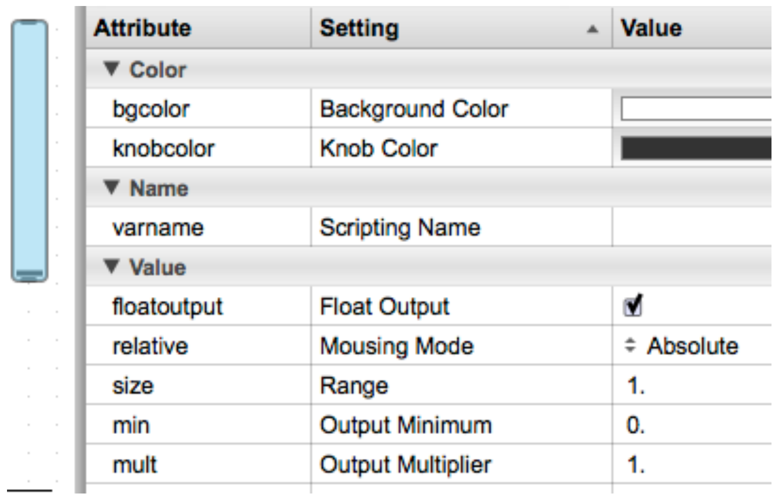
   3. When the toggle is checked, audio is being processed.
   4. Now, connect the left and middle outlets of `sfplay~` to `dac~`.



5. This is a functional basic setup.
6. If we want to have control over the volume of `sfplay~`'s output, we need to add the following objects:
    a. `*~` : this multiplies all sample values passing through it by either a fixed argument included with the object, or by a value passed to its right inlet.
    b. A slider. This will set the value of `*~`.

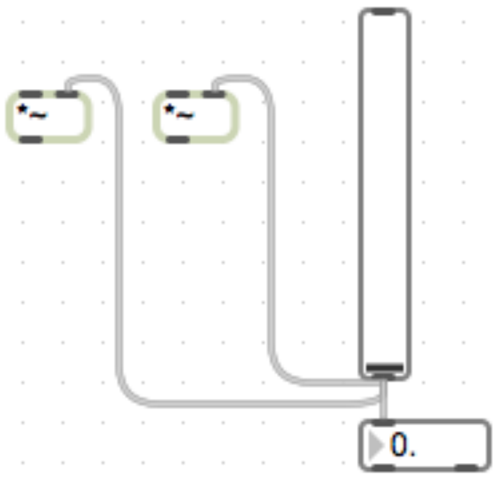


    c. Note that I'm using two `*~`'s, one for each channel of our stereo audio.
7. First, it is wise to keep the value of `*~` from exceeding 1. If the sound file being played back is normalized, this will usually create distortion. We can accomplish this by using the Inspector on our slider object.



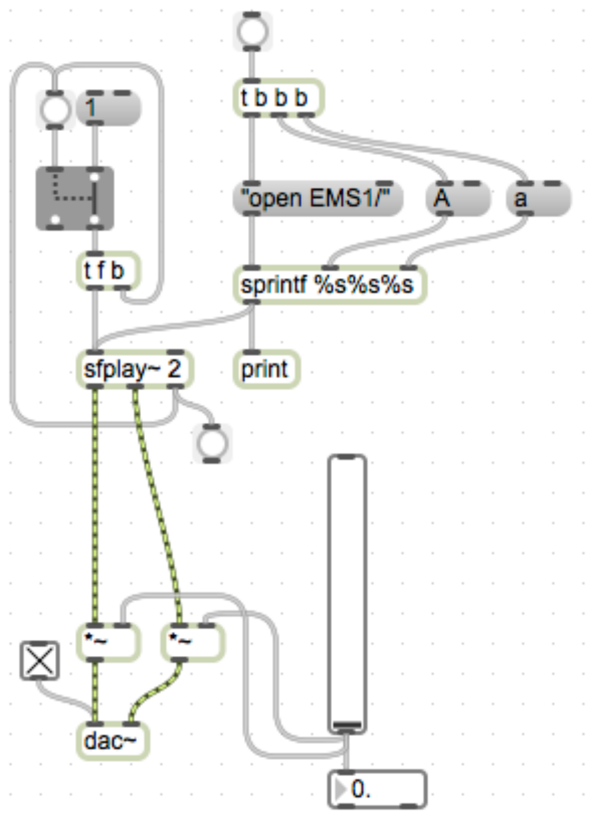
Attribute	Setting	Value
▼ Color		
bgcolor	Background Color	□
knobcolor	Knob Color	■
▼ Name		
varname	Scripting Name	
▼ Value		
floatoutput	Float Output	✓
relative	Mousing Mode	Absolute
size	Range	1.
min	Output Minimum	0.
mult	Output Multiplier	1.

8. I have changed the following settings in the above picture:
    a. floatoutput : with this checked, the slider will output decimal values between its minimum and size (here, values from 0 – 1).
    b. size : sets the maximum value of the slider, I've set it to 1.
9. Now, connect the slider outlet to the right inlets of the two `*~`'s.



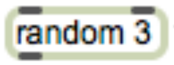
Note that I've also added a float number box, in order to monitor exactly what the slider's value is.

10. Now, connect the left and middle outlets of `sfplay~` to the left inlets of the two `*~`'s, and connect the outlets of the `*~`'s to `dac~`:

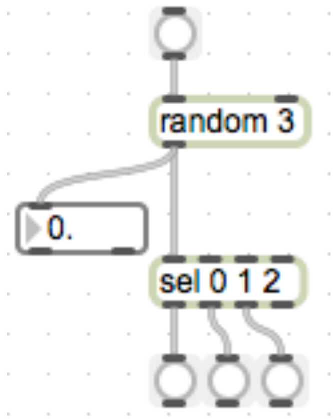


E. Expanding, adding randomness

1. In order to use one instance of this setup to randomly select a sound file from among several classes and instances (ie, Aa, Bc, Zz, etc.), we need to introduce several objects.
2. The first is `random`. This object accepts a bang in its left inlet, and generates a random number between 0 and *n*-1, where *n* is either an argument created with the object, or a value passed to the right inlet.

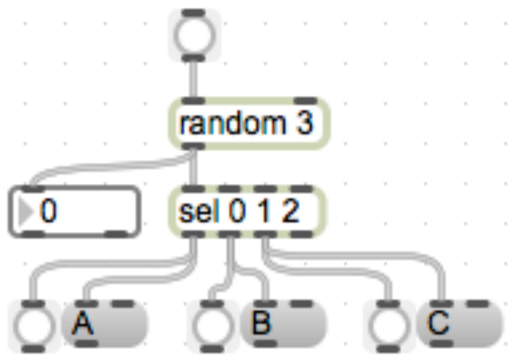
   

   Here, the argument is 3, so the output will range from 0 to 2 (that is, 3-1).
3. Now, to associate each of our `random` object's three possible output values with a sound class, we use the `select` object (abbreviated `sel`).

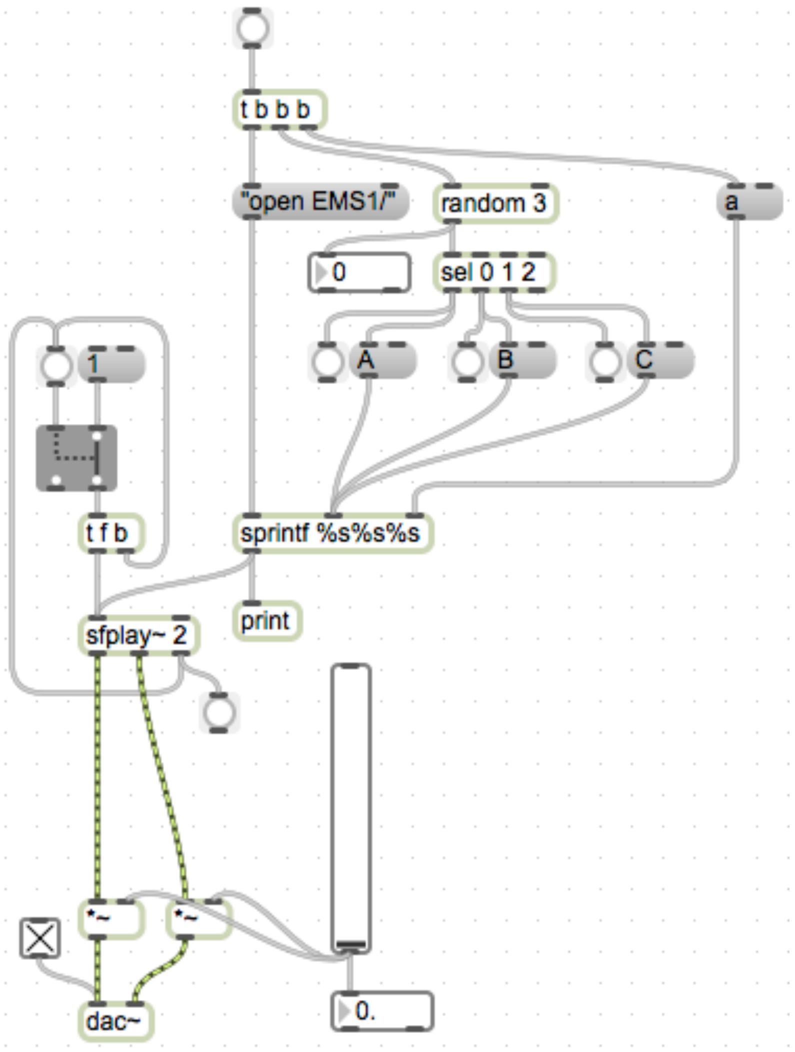
4. `select` accepts various types of input, and outputs a bang when this input matches one of its arguments. In this setup, the output of `random` will trigger one of the three bang buttons.

5. To integrate this with our existing, basic `sfplay~` setup, we need to create message boxes with the sound class labels we'll be using (in this example, A, B, and C). Connect the outlets of `select` to these message boxes.

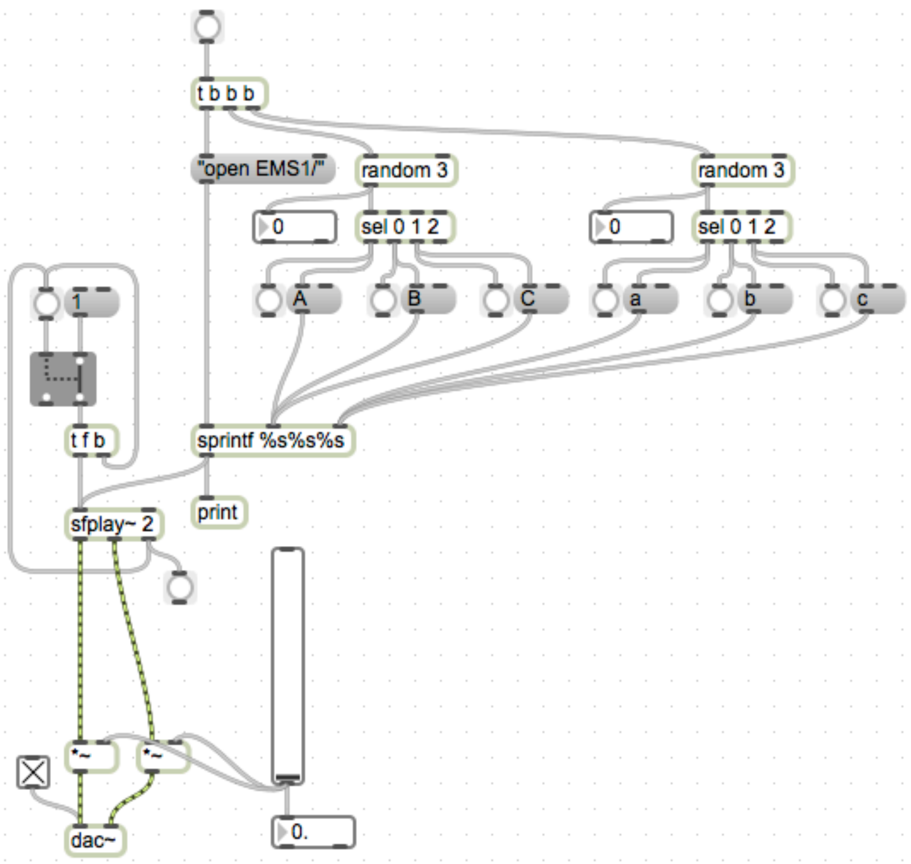


Note that I've retained the bang buttons attached to `select`, to be sure of what's being triggered.

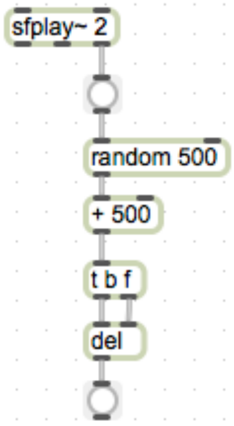
6. Now, connect the bang that in our basic setup (see D.10) would trigger the message box "a" to the inlet of `random 3`. Connect all of the new message box outlets to the middle inlet of `sprintf`. This gives us:

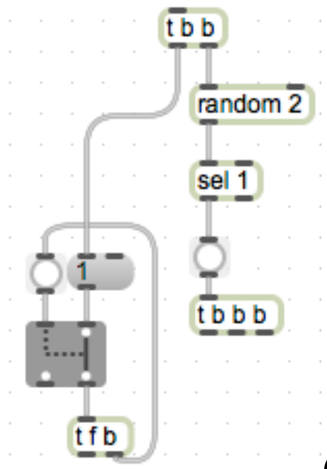


7. To select a random sound file within each sound class, we will use another `random` + `select` setup, this time replacing the message box "a".

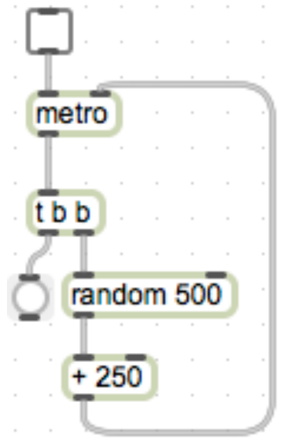


8. Keep in mind that `random` outputs values starting from 0. This means that your sound files will have to be numbered either starting at 0, or the output of `random` will have to be altered to fit your numbering scheme.
9. Some other modifications we can make relate to retriggering `sfplay~`, and selecting new sound files. Consider the following mini-setups (discuss in class):

a.  *(random delay before reopening* `ggate`*)*

b. 

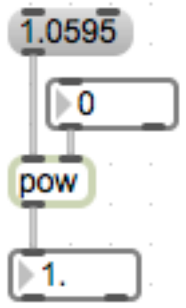
(randomly open a new sound file)

c. 

(automatic triggering, randomized timing)

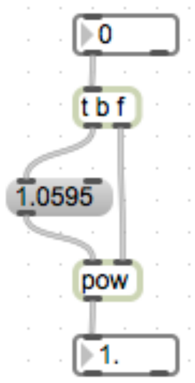
F. Playback Speed and `sfplay~`

1. The right inlet of `sfplay~` is used to set the playback speed. The default setting is 1, which results in an unaltered playback of the sound file selected.
2. To alter the playback speed of `sfplay~` by *x* number of semitones, we need to raise the 12th root of 2 to the *x* power. We accomplish this by using the following:

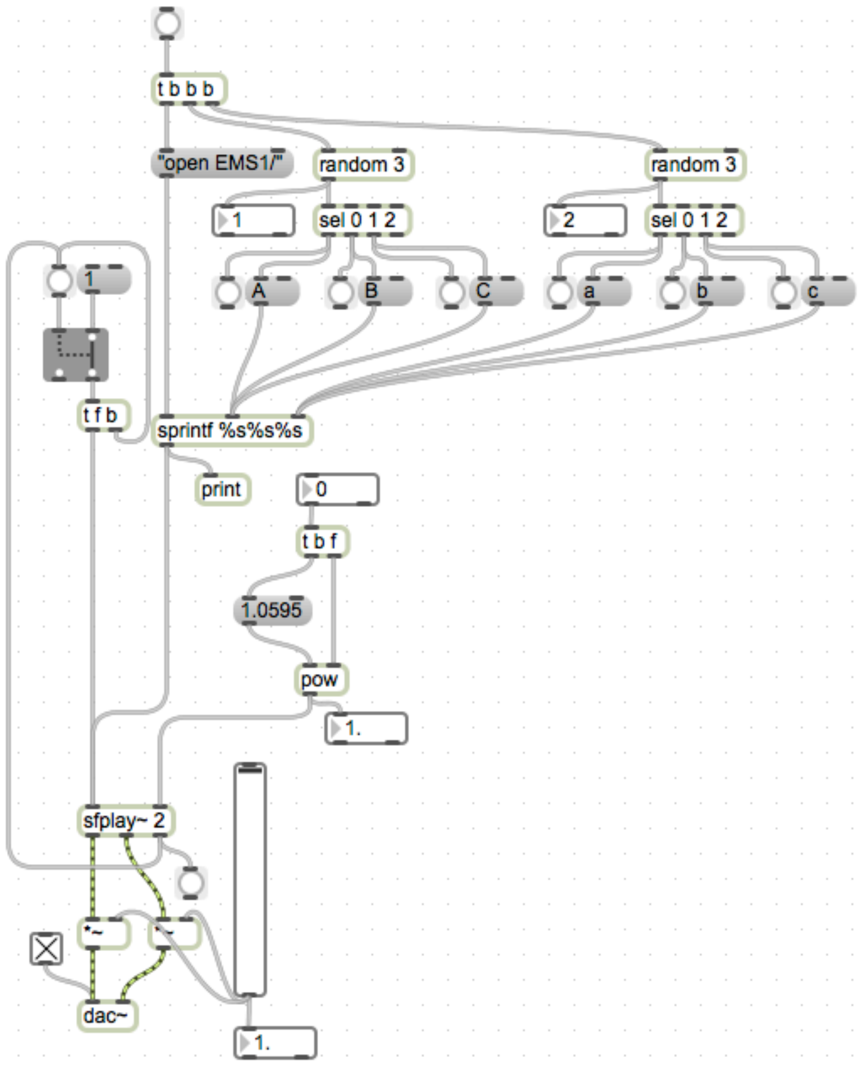


Here, 1.0595 is the 12th root of 2 (rounded slightly), fed into the left inlet. The right inlet is set by the integer box attached; this is the power to which we're raising the 12th root of 2, therefore this is the number of semitones we are shifting our playback speed.

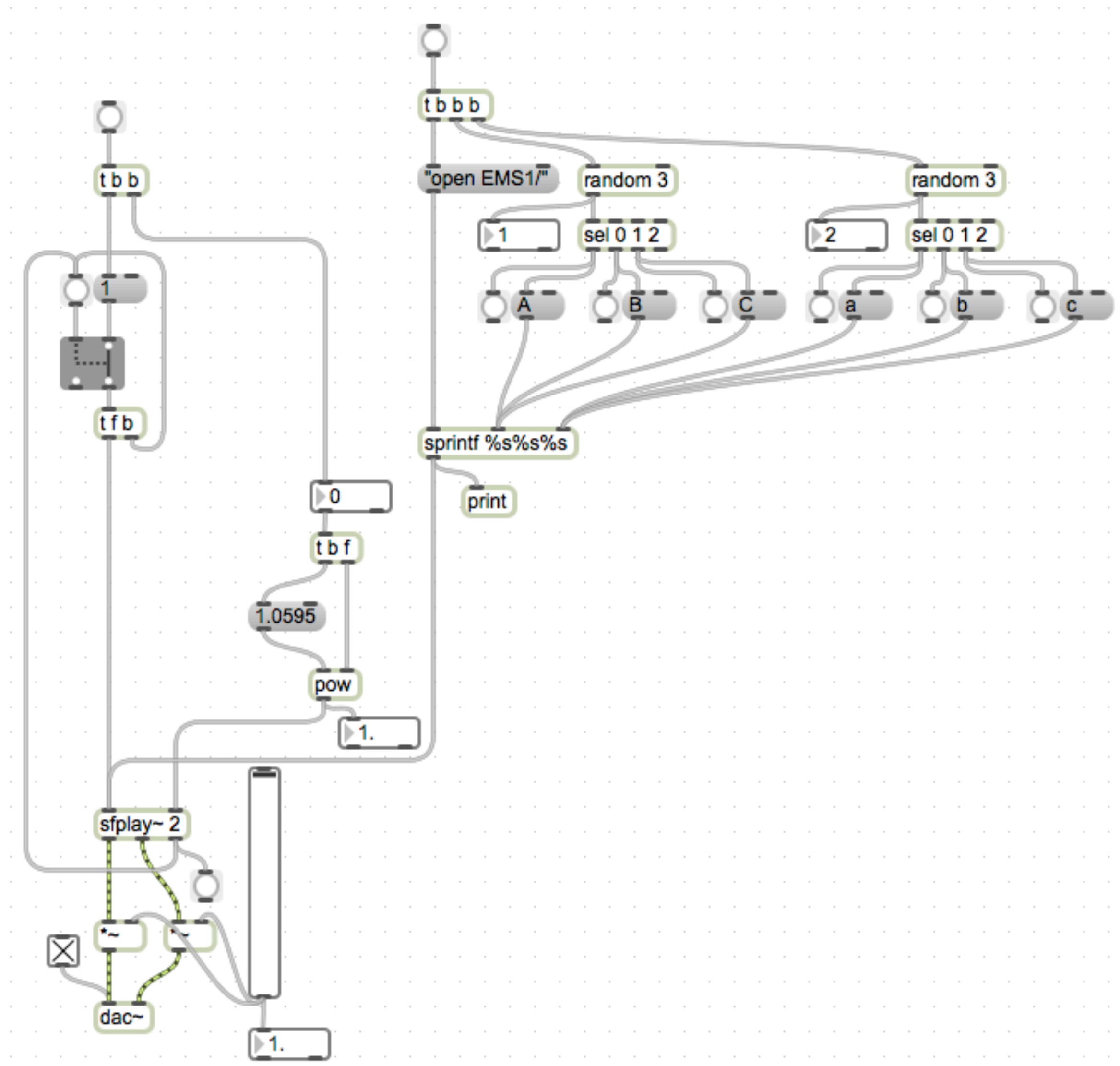
3. We can create a `trigger` object, such that changing the value of the integer box automatically recalculates our playback speed:



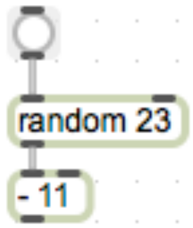
4. Now, connect the outlet of `pow` to the right inlet of `sfplay~`.



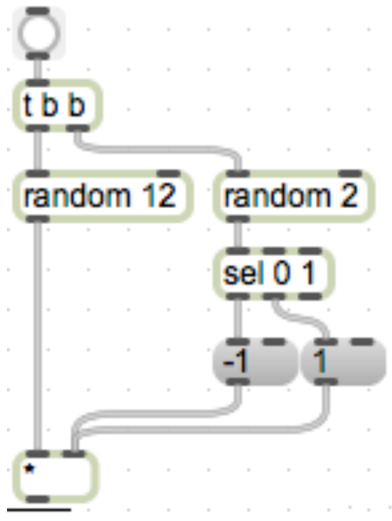
5. The value of the playback speed should be set before triggering playback from `sfplay~`. We can accomplish this with another `trigger` object.



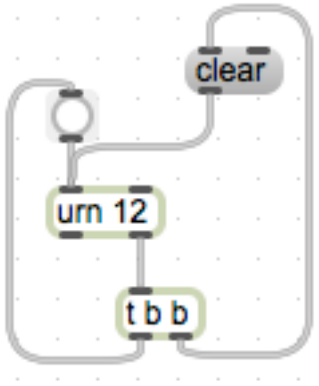
6. From this point, we can experiment with adding randomness to the playback speed value.
7. Consider the following mini-setups (discuss in class):

a. 

(select semitone value between -11 and 11)

b. 

(select semitone value between 0 and 11, and randomly choose up or down)

c. 

(This implements the `urn` *object, explained in class)*