Assignment 1: Sine Synthesis

Overview

In lecture, we learned about creating a tone with a basic sine wave. We created a `SineGenerator` class as concrete example of a generator, and connected this generator to the `Audio` class in order to hear the generated sound.

In this assignment, you will expand the functionality to:

- Specify pitches (A, B-flat, C, etc...) instead of frequencies
- Allow multiple simultaneous tones
- Specify a duration for each note as well as an amplitude shape (ie, an envelope)
- Allow for notes of differing timbres

You will be creating a very simple synthesizer, which you can then play by mapping keys on your laptop keyboard to "play note" commands.

Once you get the whole thing working, create something fun. For example, you don't have to just have the chromatic scale mapped to your keyboard. What other mappings can you create that are more interesting?

Part 1a

In class, we learned that a generator's job is to create audio data. The `Audio` class repeatedly asks its generator for a small bit of audio data and plays it out the speaker. `SineGenerator` is a simple example of a generator. The generator's interface is:

```
class Generator: 0
   def generate(self, num_frames, num_channels): 0
      return (array, continue_flag) 0
```

A generator must return a floating-point `numpy` array of length (`num_frames * num_channels`) and a flag (`True` or `False`) indicating whether the generator is done (`False`), or has more audio to generate next time around (`True`). Note that for this assignment, we will only generate mono signals, so you can assume `num_channels == 1`. In subsequent assignments, we will switch to stereo.

Create a `NoteGenerator` class (very similar to `SineGenerator`) that is instantiated with a frequency (Hertz), duration (seconds), and gain (ranging from 0.0 to 1.0). When `Audio` is assigned the generator, it will play the said frequency for the given duration at the given loudness (gain). When the note duration is done, the `NoteGenerator` should return `False` for the `continue_flag`.

In `MainWidget`, have a few key-down messages play notes with specific pitches, durations, and gains of your choice. Each time a key is pressed, create a new `NoteGenerator` instance and tell `Audio` about it.

Part 1b

Make your note generator take a pitch value instead of a frequency. The pitch value is an integer, and a continuous set of integers defines the chromatic scale. We will learn about MIDI soon, so we'll stick with the MIDI convention that frequency A440 = pitch value 69. That also means that middle C is 60 (9 semitones lower than the A).

As discussed in class, we will use equal-tempered tuning. To find the frequency one octave higher than a note with frequency *F*, we multiply by 2. To find the frequency one semitone higher than a note with frequency *F*, we multiply by the 12th root of 2. See: http://en.wikipedia.org/wiki/Pitch_%28music%29 for more details.

Write your Pitch to Frequency conversion in separate function:

```
def pitch_to_frequency(pitch): 0
   ... 0
   return freq
```

Change your key-down test functions to test the pitch (rather than frequency) interface.

Part 2

Create a new generator called `Mixer`. `Mixer` is an object that combines many separate audio sources (ie, other generators) into a single audio stream, which is then fed into the main `Audio` class for playback. Create methods for `Mixer`: `add(), remove(), set_gain(), get_gain()`. Note that Mixer's interface allows you to add or remove generators at any time.

`Mixer`'s `generate()` function should iterate through its array of generators, mixing (ie, adding) their results into a master buffer, then applying the gain, and finally returning the calculated audio buffer. If any generator returns `False`, it should be removed from `Mixer's` array. Note that `Mixer` itself should always return `True` in the `continue_flag`.

Important note: Careful when you implement `generate()`. It is a bad idea to remove an item from a list while iterating through that list.

Now hook it up. `Mixer` should feed into `Audio` and `NoteGenerators` should be added to `Mixer`. With this architecture, you should have polyphony working. You should be able to play and hear multiple notes at the same time.

Part 3

Right now, your note generator plays a single volume pitch for N seconds and abruptly turns off (most likely causing a slight pop/click). Let's make that nicer by creating an amplitude envelope. This envelope lasts the full duration of the note. Look at Section 9.2.1 In *Musimathics, Volume 2*. You will see how to generate an envelope with a specific attack time and decay time. For simplicity, ignore the attack time (so set a=0). This makes the curve simpler to compute because it is single continuous function. Create the decay envelope as described in Section 9.2.1. You can play around with the parameter n2 to get different types of decay curves.

As an optional part of this assignment, implement both the attack and decay portion of the envelope curve. Hint, the `numpy function where()` might be useful.

Part 4

A sine wave is boring. Let's make other kinds of sounds. Expand the capabilities of your note generator to add a series of harmonic frequencies (overtones) to the fundamental frequency. You can compactly represent the harmonic series as an array of amplitudes [a0, a1, a2, a3…], where aN is the amplitude of the N'th overtone. In *Musicmathics, Volume 2* Section 9.25 - 9.2.7, you can see how to construct the "Geometric Waveforms": Square Wave, Triangle Wave, and Sawtooth Wave.

Give your note generator an additional argument - a list of harmonic amplitudes. Since these amplitude values drop off fairly quickly, you only need to provide the first 10 or so values. Beyond that, the effect becomes less audible. Create a few different note timbres (Square Wave, Sawtooth). You can also try other values to see how a note's timbre changes with different strengths of the overtone series.

Part 5

Come up with a creative way to use this system that lets you “perform” something. Create mappings between your keyboard and your synthesizer.
Some ideas:

- Define a sequence of notes (like you saw in the first day of class) so you can trigger a melody.
- Hitting a single key can play more than one note – it can play an entire chord!
- You can trigger staccato (short) notes and legato (long) notes to create different melodic effects, or have different timbres for different parts of your piece.
- You can have a set of “meta keys” that have larger effects and can affect other mappings. For example, if you have 1-9 mapped to a C-major scale, hitting a single other key can change that mapping to a different scale.

Write up a short description of the how to control your system in a README file.

Create a quick / rough / unedited video of your performance. It doesn’t need to be long: 30-60 seconds is fine. You can either submit the video file or (better) upload it to YouTube/Vimeo and provide a link in the README.

Finally...

Please have good comments in your code. When submitting your solution, submit a zip file that has all the necessary files. For example, if you used other files that I provided (like core.py), re-provide those files back to me in your submission.

21M.385 Interactive Music Systems

Fall 2016