Mission 10: Musical Diversions

Start date: 21 September 2017
Due: 29 September 2017, 23:59

Readings:

- Textbook Section 2.5

Fundamentals of Music

Note and Frequency

The basic unit of music is a *note*. A note in music is like a word in written language, or a syllable in spoken language. Each note is a pitched sound, and the pitch is based primarily on the *frequency* of the sound. Since there are infinitely many frequencies in nature, we usually pick a particular frequency as the reference pitch, and use musical temperament to determine the width of a semitone, which is the distance between two adjacent notes. In western music, we set 440.00Hz as reference pitch, and apply twelve-tone equal temperament to define the notes. In twelve-tone equal temperament, the width of a semitone is the twelfth root of two:

$$\sqrt[12]{2} \approx 1.0595$$

Therefore, if we have a note with a frequency of 440Hz, then the note just higher than it will have the frequency:

$$\sqrt[12]{2} \times 440.00\text{Hz} \approx 466.16\text{Hz}$$

The next note differs from the result by the same factor and thus has the frequency:

$$\sqrt[12]{2}^2 \times 440.00\text{Hz} \approx 493.88\text{Hz}$$

MIDI Note and Letter Name

MIDI stands for Musical Instrument Digital Interface. It is a technical standard that describes a protocol, digital interface and connectors and allows a wide variety of electronic musical instruments, computers and other related devices to connect and communicate with one another. In order to digitize a note, we have the following MIDI note representation.

The note with reference frequency 440.00Hz is denoted as 69 in MIDI note representation, and the MIDI note 70 has a frequency of 466.16Hz and so on. In Source, we have a function `midi_note_to_frequency` to convert a MIDI note to the corresponding frequency. The function `midi_note_to_frequency` has the type

`midi_note_to_frequency` : $Number \rightarrow Number$

Other than a MIDI note name, a note also has a letter name. In letter name representation, we use the letters from A to G to represent notes. The interval between B and C, E and F are one semitone, and between A and B, C and D, F and G, G and higher A are two semitones. Hence, the frequency ratio of F to E is $\sqrt[12]{2}$, and the frequency ratio of G to F is $\sqrt[12]{2}^2$.

Since the note two semitones higher than G is A again, we need a way to differentiate those two As, so instead of saying the note with frequency 440.00Hz is A, we call it $A4$. Then next A, which is 12 semitones higher than $A4$, is called $A5$, and thus has frequency 880.00Hz.

Notice that the distances between two adjacent letter names are not the same. There is no other note between E and F but there is one note between A and B. We need to add accidentals to represent that note. In this case, it is a semitone higher than A, so it can be called $A\sharp$ (pronounced as `A sharp`, and typed as `A#`; besides, it is also a semitone lower than B, thus it can also be called $B\flat$ (pronounced as `B flat`, and typed as `Bb`).

In Source, you can use `letter_name_to_midi_note` to convert a letter name to corresponding midi note. The function has the type:

`letter_name_to_midi_note` : $String \rightarrow Number$

The argument for `letter_name_to_midi_note` should be a proper letter name as a Source String; for example: `"A5"`, `"B3"`, `"D#4"`.

Here is a mapping between letter names, midi note names, and frequencies:

midinotes

Scale

A scale is any set of notes ordered by fundamental frequency or pitch. The most common scale might be the heptatonic scale. In a heptatonic scale, notes are separated in octaves; a note with twice the frequency of the other note is called an octave higher. Within an octave, there are seven pitches arranged in different ways based on their frequency. The most famous heptatonic scales are the major and minor scales. Each scale starts with a base frequency, which we call *tonic*.

In a major scale, the interval between the first note (tonic) and the second note (supertonic) is a tone (two semitones), and the interval between the second and third note (mediant) is a tone. The following intervals are semitone, tone, tone, tone, and finally semitone, to reach the tonic one octave higher.
If we write it in a sequence of steps, it will be:

$$(2, 2, 1, 2, 2, 2, 1)$$

If the major scale starts with the MIDI note 60, which is $C4$, it will be:

$$60, 62, 64, 65, 67, 69, 71, 72$$

or:

$$C4, D4, E4, F4, G4, A4, B4, C5$$

This scale will be named based on its tonic, which is a C note; so we call it C major scale.

Chord, Arpeggio and Arpeggiator

In music, a chord is a set of notes that sound simultaneously; the notes chosen should be in harmony. The minimum number of notes to form a chord is three, and chords formed by three notes are called triadic chords, or (triads)
. The interval between any two of the notes determines the type of the chord. Based on the differences of the intervals, there are four basic triads:

- Major triad: the list of intervals is: `list(4, 3)`
- Minor triad: the list of intervals is: `list(3, 4)`
- Augmented triad: the list of intervals is: `list(4, 4)`
- Diminished triad: the list of intervals is: `list(3, 3)`

Hence, if the base note is C, the C major triad is C, E, G; the C minor triad is C, $E\flat$, G; the C augmented triad is C, E, $G\sharp$; and the C diminished triad is C, $E\flat$, $G\flat$.

An arpeggio is a musical technique where notes in a chord are played or sung in sequence, one after the other, rather than ringing out simultaneously. A simple arpeggio can be made using the first, third and fifth notes in a scale. For example, the first, third, and fifth notes in a G major scale is G, B and D, so if we play the notes one by one, then it is considered a G major arpeggio.

If we play these notes in some pattern and cross different octaves, it may create a good effect which is suitable for accompaniment. This feature is called arpeggiator. For more information and demonstration, you can read this article: http://www.dawsons.co.uk/blog/what-is-an-arpeggiator

This mission has **five** tasks.

Task 1:

In the introduction of scale above, you are told that you can use a sequence of steps to define a scale. Now, other than the boring major scale, Pixel would like to generate different scales based on different sequences of steps; also, instead of just using 'C4' as the tonic, he want his scale to be played in different keys. Write a function `generate_list_of_note` that takes a base note and a list of intervals as arguments, and returns a list of *MIDI notes*. The function should have the type:

$$\texttt{generate\_list\_of\_note} : (String, List(Number)) \rightarrow List(Number)$$

As a test, try

```
var c_major_scale = generate_list_of_note("C4",
    list(2, 2, 1, 2, 2, 2, 1, -1, -2, -2, -2, -1, -2, -2));
```

You should obtain a list of midi notes:

```
list(60, 62, 64, 65, 67, 69, 71, 72, 71, 69, 67, 65, 64, 62, 60)
```

After you have generated a list of MIDI notes, you cannot just show the list of numbers to Pixel as the numbers make no sense. Hence, you may need to write a function `list_to_sound` using `sine_sound` to convert the list of MIDI notes to a `sound` which can be played using the `play` function. The `list_to_sound` function should take two arguments, one is the list of MIDI notes, and the other is the duration for each note, and returns a `sound`.

$$\texttt{list\_to\_sound} : (List(Number), Number) \rightarrow Sound$$

Finally, use the given `harmonic_minor_scale_interval` and `melodic_minor_scale_interval` to generate the C harmonic minor scale and C melodic minor scale. Can you hear the differences amongst these three C scales?

Task 2:

Now you can play three types of scales and have a function `generate_list_of_note` which can produce and infinite combination of notes. Pixel wants you to do something even crazier using the tool, that is Arpeggiator!

To play the arpeggio, you firstly need to know what are the notes that build the arpeggio. From the introduction we know a simple arpeggio contains three notes, the tonic, the mediant and the dominant. For C major, they would be C, E, and G. As usual, you will be given the list of intervals and you can use it to generate a list of MIDI notes using `generate_list_of_note` you just defined. Fourtunatly, this task is too simple, so Pixel has done it for you, so that you can use the function `generate_arpeggio` to create a list of notes that form the arpeggio.

Using the provided list of intervals, the list of notes generated should be tonic, mediant, dominant, tonic (an octave higher), mediant (an octave higher), dominant (an octave higher), and tonic (two octaves higher). Thus, the result of

```
generate_arpeggio("C4", major_arpeggio_interval);
```

should be a list of MIDI notes representing

```
C4, E4, G4, C5, E5, G5, C6
```

Now, instead of playing through the notes, you find it more interesting to play it in a 'sawtooth way', which is

```
C4, E4, G4, C5, E4, G4, C5, E5, G4, C5, E5, G5, C5, E5, G5, C6
```

If we draw it in a musical staff, it would look like:

arpeggio

Write a function `arpeggiator_up` which takes a list of MIDI notes as argument and the duration for each note, and returns a sound of the arpeggio played in the 'sawtooth way', which is playing the first four notes, then follwed by the 2^{nd}, 3^{rd}, 4^{th} and 5^{th} notes, then 3^{rd}, 4^{th}, 5^{th}, and 6^{th} as so on, until you hit the highest note. If the length of the list of MIDI notes is less than 4, return it back without doing anything.

ADSR

Now you can play a scale and even an arpeggiator, but the sound is not good enough. Pixel wants you to change the individual notes also, and he shouts again, "Attack, Delay, Sustain, Release - ADSR - ADSR, that's how you move boys and girls"; suddenly, these words remind you of your music lesson in primary school, it's about ADSR.

When an acoustic musical instrument produces sound, the loudness and spectral content of the sound change over time in ways that vary from instrument to instrument. In order to control a sound's amplitude at any point in its duration, we can apply an ADSR envelope on it. Basically, there are four phases between the time a sound is produced and the time it disappears, namely, Attack, Decay, Sustain, and Release. To customize the ADSR envelope, we need these four parameters:

- Attack time: the time taken for a sound to run up from silence to peak (loudest point);
- Decay time: the time taken for the subsequent run down from the peak to the designated sustain level;
- Sustain level: the level of amplitude during the main sequence of the sound's duration, until the sound enters it released stage;
- Release time: the time taken for the level to decay from the sustain level to zero.

A typical ADSR envelope looks like this:

midinotes

However, in nature, most of the sounds will decay exponentially. Therefore, the decay and release stage should be modified and the envelope should look like this:

midinotes

Task 3:

If we have all the information needed for an ADSR envelope, we can define the envelope in this way:

- during the attack phase, increase the amplitude of the sound from 0 to 1 linearly;
- during the decay phase, reduce the amplitude of the sound from 1 to `sustain_level` exponentially;
- during the sustain phase, keep the amplitude of the sound at the `sustain_level`;
- during the release phase, which is the last `release_time` seconds in the duration time of the sound, reduce the amplitude of the sound from `sustain_level` to 0 exponentially.

In this task, you need to figure out exponential decay, before you can tackle a whole envelope in the next task. If a quantity decreases at a rate proportional to its current value, the decay is exponential. Let N denote the quantity and λ denote the decay constant, we have:

$$\frac{dN}{dt} = -\lambda N$$

Solving this equation, we have:

$$N(t) = N_0 e^{-\lambda t}$$

where $N(t)$ is the quantity at time t, and $N_0 = N(0)$ is the initial quantity. For exponential decay, there is another commonly-used characteristic, its half-life $t_{\frac{1}{2}}$. The half-life of an exponential decay is the time required for the decaying quantity to fall to one half of its initial value; the value of half-life can be obtained by:

$$t_{\frac{1}{2}} = \frac{\ln(2)}{\lambda}$$

To simplify the task, the initial value N_0 will be fixed at 1. Write a function `exponential_decay` which takes one argument, `decay_time`, which is the time taken for the quantity to be fully decayed, and returns a exponential decay function (You can assume that the parameter that the exponetional decay function takes in is always positive). Notice that in this task, when a quantity decays till $\frac{\mathbf{1}}{\mathbf{2^4}}$ of its orginal value, we consider it 'fully decayed' (the value will turn to zero after 'fully decayed'); that is to say, the `decay_time` is four times the half-life:

$$\texttt{decay\_time} = 4 \times t_{\frac{1}{2}}$$

The function `exponential_decay` should have the type

$$\texttt{exponential\_decay} : (Number) \to (Number \to Number)$$

Task 4:

Complete the function `adsr` using the given template; `adsr` has four arguments, `attack_time`, `decay_time`, `sustain_level`, and `release_time`, and returns a function that applies the corresponding envelope to its argument sound. The function should have the type:

$$\text{adsr} : (Number, Number, Number, Number) \to (Sound \to Sound)$$

We provide you with several sample envelopes. After implementing the `adsr` function, you can test them and try to determine what's the instruments we are trying to simulate.

Note that in addition to sine waves, we also used sawtooth, square and triangle waves. You can try them out using the same ADSR envelope and the same pitch, and find the differences. If you are interested in waveforms, please refer to Wikipedia: http://en.wikipedia.org/wiki/Waveform.

Task 5:

Finally, you have your generator of ADSR envelopes, but the greedy Pixel again wants more. This time he wants to stack multiple ADSR envelopes on different harmonics and play them together. If you play a base note with a frequency f, the harmonics of this base note will have a frequency that is an integer multiple of the frequency of the base note. The first harmonic will have the frequency $2f$, the second the
frequency $3f$ and so on.

Write a function `stacking_adsr`, which has four arguments:

- `wave_form`: the function used to generate different waveform; it should be `sine_sound`, `sawtooth_sound`, `square_sound` or `triangle_sound`; [The three new waveform functions are similar to the `sine_sound`, and are available in the library.]
- `base_frequency`: which is the frequency of the base note;
- `duration`: the duration of the sound;
- `list_of_envelope`: should be a list of ADSR envelopes.

The function `stacking_adsr` should return a sound in which multiple sounds are played simultaneously. The number of sounds should be equal to the length of `list_of_envelope`. The first element in `list_of_envelope` is applied to the base note, the second element in `list_of_envelope` is applied to the first harmonic of the base note, and so on.

We have a sample bell sound generated by stacking four ADSR envelopes together. After implementing the `stacking_adsr` function, you can use the sample to test your answer and check whether you hear a bell sound. Pixel also randomly writes down some combinations and asserts that they are the simulated sound of trombone, piano, violin and cello, respectively.

You can help him to test them and tell whether they are similar to the timbre of those instruments.
If you are ambitious enough, you can even generate your own stacking-ADSR envelopes and imitate your favorite instrument. (Not graded)

It takes some time to generate the sound, please be patient. :)

Submission

To submit your work to the Source Academy, place your program in the "Source" tab of the online editor within the mission page, save the program by clicking the "Save" button, and click the "Submit" button. Please ensure the required function from each Task is included in your submission. Note that submission is final and that any mistakes in submission requires extra effort from a tutor or the lecturer himself to fix.

IMPORTANT: Make sure you've saved the latest version of your work by clicking the "Save" button before finalizing your submission!