CSCI 230 Data Structures and Algorithms
Project 2 - Guitar

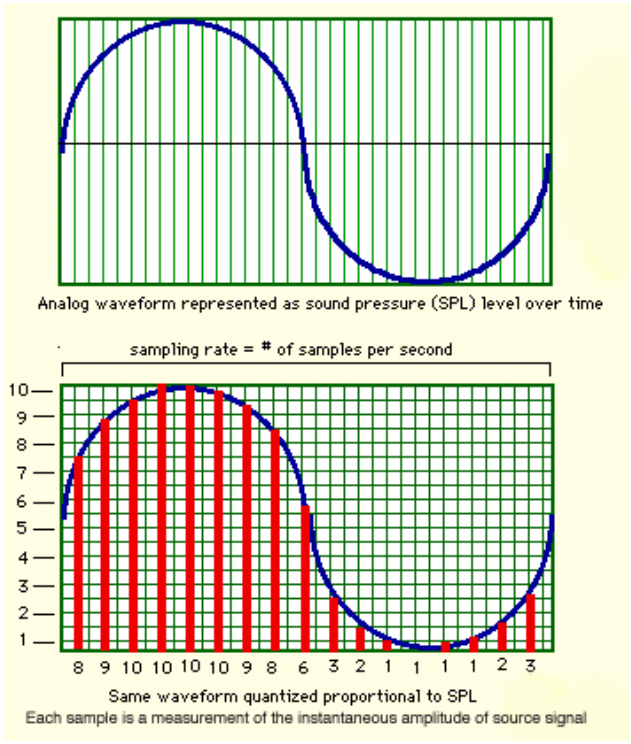
1 Overview

In this project, you will write a program to simulate plucking a guitar string using the Karplus-Strong algorithm. This algorithm played a seminal role in the emergence of physically modeled sound synthesis (where a physical description of a musical instrument is used to synthesize sound electronically). The Karplus-Strong models the vibration of a guitar string with a remarkably simple process: Maintain a buffer of displacements; then, repeatedly delete the first displacement from the front of the buffer and append the average of the first two displacements, scaled by an energy dissipation factor, to the end of the buffer. The size of the buffer determines the frequency of the guitar string. To represent a guitar, you will create many guitar strings (of different frequencies), process keystrokes to identify which strings get plucked, superpose the resulting sound waves, and sonify the results. The key data structure needed to perform the Karplus-String simulation efficiently is a ring buffer (bounded queue).

2 Background

2.1 Digital Audio

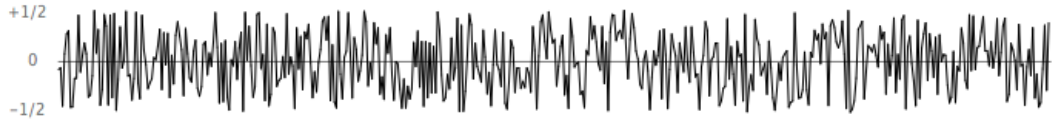
Sound waves travel through the air by the changes of air pressure at different moments in time. The stronger the change in pressure, the louder the sound. As the speed of the air pressure increases, the perceived pitch becomes higher. These changes of amplitude (loudness) and frequency (pitch) are the two principal factors of sound. Digital recording of sound is performed at regular time intervals, when the amplitude of an analog sound wave is recorded, generating a sample. The closer together that samples are recorded, the more accurate the recording becomes. The rate of sample measurement is called the sampling rate (or sampling frequency). The figure on the right shows how an analog sound wave is sampled at regular time intervals.



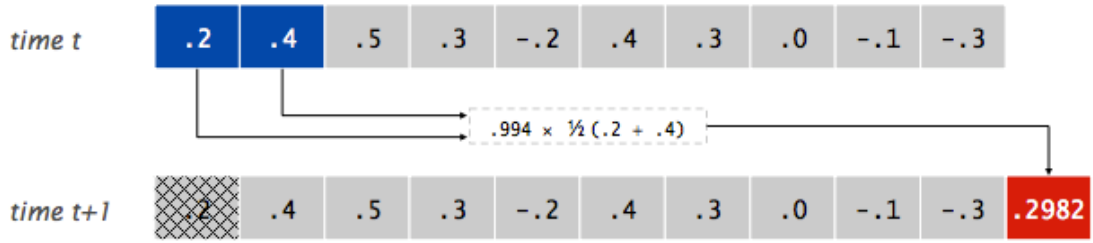
2.2 Simulate the plucking of a guitar string

When a guitar string is plucked, the string vibrates and creates sound. The length of the string determines its fundamental frequency of vibration. We model a guitar string by sampling its displacement ($r \pm 1/2$) at N equally spaced points (in time), where N equals the sampling rate (44,100 samples/second) divided by the fundamental frequency, which is the lowest frequency produced by the oscillation of an object (rounding the quotient up to the nearest integer).

- **Plucking the string**. The excitation of the string can contain energy at any frequency. We simulate the excitation with white noise: set each of the N displacements to a random real number between $\pm 1/2$.



- **The resulting vibrations**. After the string is plucked, the string vibrates. The pluck causes a displacement which spreads wave-like over time. The Karplus-Strong algorithm simulates this vibration by maintaining a ring buffer of the N samples: the algorithm repeatedly deletes the first sample from the buffer and adds to the end of the buffer the average of the first two samples, scaled by an energy decay factor of 0.994.



the Karplus-Strong update

2.3 Why it works?

The two primary components that make the Karplus-Strong algorithm work are the ring buffer feedback mechanism and the averaging operation.

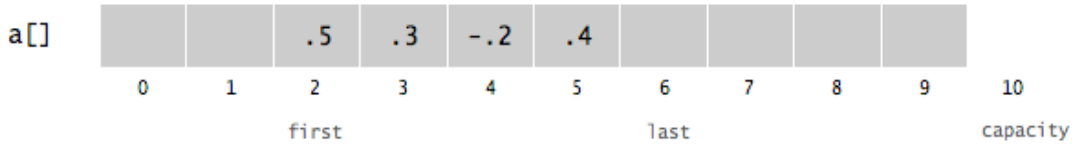
- **The ring buffer feedback mechanism**. The ring buffer models the medium (a string tied down at both ends) in which the energy travels back and forth. The length of the ring buffer determines the fundamental frequency of the resulting sound. Sonically, the feedback mechanism reinforces only the fundamental frequency and its harmonics (frequencies at integer multiples of the fundamental). The energy decay factor (.994 in this case) models the slight dissipation in energy as the wave makes a roundtrip through the string. **The averaging option**.The averaging operation serves as a gentle low-pass filter (which removes higher frequencies while allowing lower frequencies to pass, hence the name). Because it is in the path of the feedback, this has the effect of gradually attenuating the higher harmonics while keeping the lower ones, which corresponds closely with how a plucked guitar string sounds.

2.4 Project Requirements

(1) Ring Buffer

- Write a class named `RingBuffer` that models a ring buffer by implementing the following API:
  - `RingBuffer(int capacity)`: creates an empty ring buffer, with given max capacity.
  - `RingBuffer()`: creates an empty ring buffer with capacity of 100
  - `int size()`: returns the number of items currently in the buffer
  - `boolean isEmpty()`: returns whether the buffer is empty (size equals zero)
  - `boolean isFull()`: returns whether the buffer is full (size equals max capacity)
  - `void enqueue(double x)`: add item `x` to the end or throws a `RingBufferException` exception if the buffer is full (you can define `RingBufferException` to be a subclass of `Exception`, as a public class nested in the `RingBuffer`)

  - `double dequeue()`: delete and return item from the front; throws a `RingBufferException` exception if the buffer is empty
  - `double peek()`: returns (but does not delete) the item from the front of the list.
- Implement `RingBuffer` using a double array. For efficiency, use a cyclic wrap-around: Maintain one integer instance variable `first` that stores the index of the least recently inserted item; maintain a second integer instance variable `last` that stores the index one beyond the most recently inserted item. To insert an item, put it at index `last` and increment `last`. To remove an item, take it from index `first` and increment `first`. When either index reaches the end of the array, make it wrap-around by changing the index to 0. *Note: You must implement the ring buffer in this manner to get full credits.*



a ring buffer of capacity 10, with 4 elements

(2) **Guitar String** Write a class named `GuitarString` that models a vibrating guitar string by implementing the following API:

- `GuitarString(double frequency)`: creates a guitar string of the given frequency, using a sampling rate of 44,000 by creating a `RingBuffer` of the desired capacity N (sampling rate 44,100 divided by frequency, rounded up to the nearest integer), and initializes it to represent a guitar string at rest by enqueueing N zeros.
- `GuitarString(double[] init)`: creates a guitar string whose size and initial values are given by the array by creating a `RingBuffer` of capacity equal to the size of the array, and initializes the contents of the buffer to the values in the array.
- `void pluck()`: sets the buffer to white noise by replacing all items in the ring buffer with random values between -0.5 and +0.5.
- `void tic()`: advances the simulation one time step by applying the Karplus-Strong update: delete the sample at the front of the ring buffer and add to the end of the ring buffer the average of the first two samples, multiplied by the energy decay factor of 0.994.
- `double sample()`: returns the current sample by returning the value of the item at the front of the ring buffer.
- `int time()`: return number of tics by returning the total number of times tic() was called.

(3) Write a program `GuitarHero` that is similar to `GuitarHeroLite` but supports a total of 37 notes on the chromatic scale from 110 Hz to 880 Hz. The program should also plot the sound wave in real-time.

`GuitarHeroLite.java` is a sample `GuitarString` client that plays the guitar in real-time, using the keyboard to input notes. When the user types the lowercase letter `'a'` or `'c'`, the program plucks the corresponding string. Since the combined result of several sound waves is the superposition of the individual soundwaves, we play the sum of all string samples. (See **Code 1: GuitarHeroLite**)

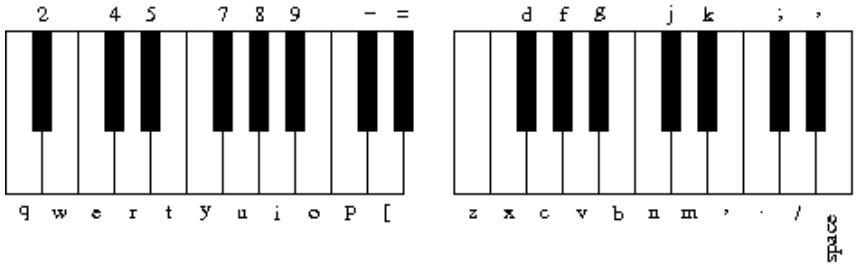
Code 1: GuitarHeroLite

```
public class GuitarHeroLite {
      public static void main(String[] args) {

          // create two guitar strings, for concert A and C
          double CONCERT_A = 440.0;
          double CONCERT_C = CONCERT_A * Math.pow(1.05956, 3.0);
          GuitarString stringA = new GuitarString(CONCERT_A);
          GuitarString stringC = new GuitarString(CONCERT_C);

          while (true) {

              // check if the user has typed a key; if so, process it
              if (StdDraw.hasNextKeyTyped()) {
                  char key = StdDraw.nextKeyTyped();
                  if      (key == 'a') { stringA.pluck(); }
                  else if (key == 'c') { stringC.pluck(); }
              }

              // compute the superposition of samples
              double sample = stringA.sample() + stringC.sample();

              // play the sample on standard audio
              StdAudio.play(sample);

              // advance the simulation of each guitar string by one step
              stringA.tic();
              stringC.tic();
          }
       }
  }
```

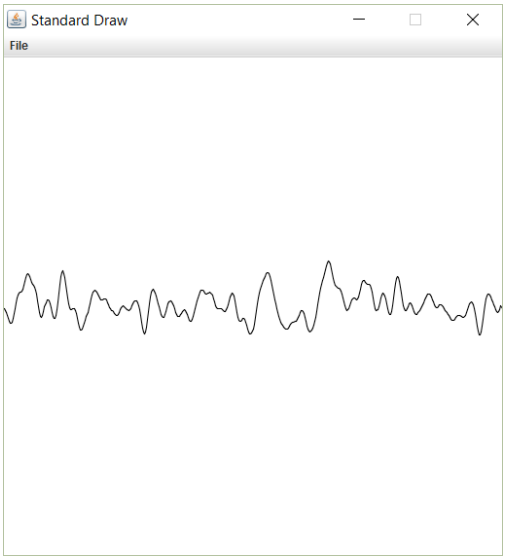
In general, make the i-th character of the string

```
String keyboard = "q2we4r5ty7u8i9op-[=zxdcfvgbnjmk,;/' "
```

play the i-th note.



- The i-th character of the string corresponds to a frequency of $440 \times 1.05956^{(i-24)}$, so that the character `'q'` is approximately 110Hz, `i` is close to 220 Hz, `'v'` is close to 440Hz, and `' '` is close to 880 Hz. *Do not include 37 individual* ***`GuitarString`*** *variables or a 37-way* ***`if`*** *statement!* Instead, create an array of `GuitarString` objects and use the `indexOf` member method to figure out which key was typed. Make sure your program does not crash is a key is played that is not one of your 37 notes.
- Plot the sound waves in real-time, as the user is playing the keyboard guitar. The output should look something like this, but change over time.



Hint: Use a `RingBuffer` object to store the sound samples that have been generated. Drawing methods of the `StdDraw` class are described here. Use the `StdDraw.line` method to draw the sound wave (you can calculate the coordinates using the samples stored in the ring buffer). To avoid significant time delay when the curve is being drawn use the `StdDraw.show` method to display the animation. For example:

Code 2: `DrawExample`

```
StdDraw.show(10);
StdDraw.clear();
// use StdDraw.line to generate the sound waveform
StdDraw.show(10);
```

(4) Object-oriented design principles, including data encapsulation and information hiding, should be used.

(5) Code should be well-documented, including all classes and class members. Follow the style guidelines for the Javadoc tool.

(6) All source code and runnable jar files must be submitted properly on Canvas. (**Note**: If I am not able to read your source files and/or run your executable, your project will not be graded. It is your responsibility to make sure all files are submitted correctly before the deadline.)

2.5 Create a Runnable JAR File

To create a new runnable JAR file in the Eclipse workbench:

- From the menu bar's **File** menu, select **Export**
- Expand the Java node and select **Runnable JAR** file. Click Next.
- In the Runnable JAR File Specification page, select a 'Java Application' launch configuration to use to create a runnable JAR.
- in the **Export destination** field, either type or click **Browse** to select a location for the JAR file.
- Select an appropriate library handling strategy. Choose "Package required libraries into generated JAR".
- Be sure to test the runnable JAR file by opening it.