Guitar Hero

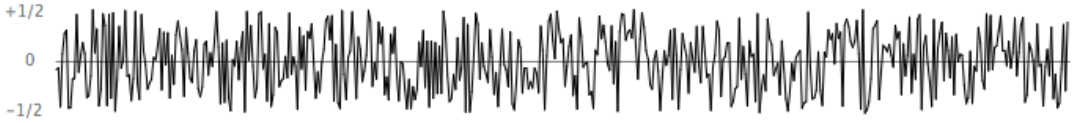
Write a program to simulate plucking a guitar string using the *Karplus-Strong* algorithm. This algorithm played a seminal role in the emergence of physically modeled sound synthesis (where a physical description of a musical instrument is used to synthesize sound electronically).

Digital audio. Before reading this assignment, review this background material:

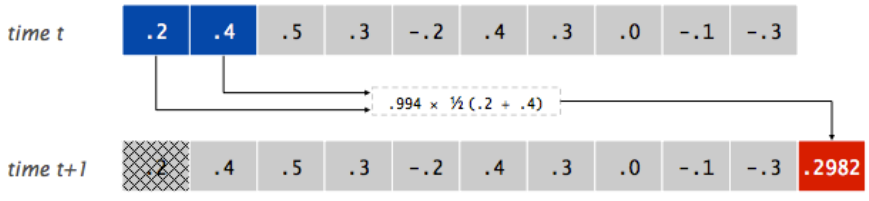
- Basic sound concepts (in lab folder)
- http://en.wikipedia.org/wiki/Karplus%E2%80%93Strong_string_synthesis
    - This info is interesting, but you don't need to be able to understand it to complete the lab

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* (a real number between -1/2 and +1/2) at *N* equally spaced points (in time), where *N* equals the *sampling rate* (44,100) divided by the fundamental frequency (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 number from -0.5 and 0.5.



- *Resulting vibrations.* When 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 *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

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*** (0.994 in this case) models the slight dissipation in energy as the wave makes a round-trip through the string.

- *The averaging operation.* 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.

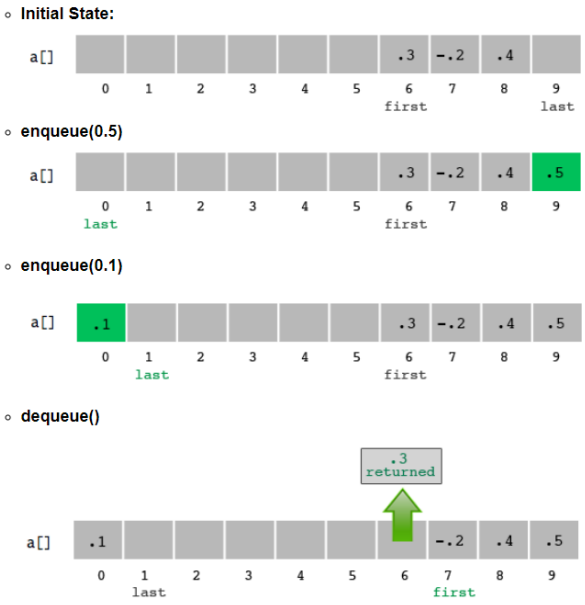
From a mathematical physics viewpoint, the Karplus-Strong algorithm approximately solves the 1D wave equation, which describes the transverse motion of the string as a function of time.

Ring buffer. Your first task is to create a data type to model the ring buffer. Complete the class (starter code provided) named RingBuffer that implements the following API:

```
class RingBuffer
---------------------------------------------------------------------------------------------------------------
        RingBuffer(int capacity)  // create an empty ring buffer, with given max capacity
    int size()                    // return number of items currently in the buffer
    int getCapacity()             // return the total number of items the buffer can contain
   void enqueue(double x)         // add item x to the end
 double dequeue()                 // delete and return item from the front
 double peek()                    // return (but do not delete) item from the front
```

Since the ring buffer has a known maximum capacity, implement it using a `double` array of that length. For efficiency, use *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 equals `capacity`, make it wrap-around by changing the index to `0`.

Have RingBuffer throw a RuntimeException if the client attempts to `dequeue` from an empty buffer or `enqueue` into a full buffer. Example of a ring buffer in action:



For convenience, a `main` method has been provided with some test code. Feel free to add any tests you see fit. Ensure your RingBuffer class works before proceeding to the next step.

Guitar string. Next, create a data type to model a vibrating guitar string. Complete the class (starter code provided) named GuitarString that implements the following API:

```
class GuitarString
-----------------------------------------------------------------------------------------------------------
       GuitarString(double frequency)  // string of the given frequency, using a sampling rate of 44,100
       GuitarString(double[] init)     // string whose size and initial values are given by the array
  void pluck()                         // set the buffer to white noise
  void tic()                           // advance the simulation one time step
double sample()                        // return the current sample
   int time()                          // return number of tics
```

- *Constructors.* There are two ways to create a GuitarString object.
    - The first constructor creates 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.
    - The second constructor creates a RingBuffer of capacity equal to the size of the array, and initializes the contents of the buffer to the values in the array. On this assignment, its main purpose is for debugging and grading.
- *Pluck.* Replace the *N* items in the ring buffer with *N* random values between -0.5 and +0.5.
- *Tic.* Apply 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.
- *Sample.* Return the value of the item at the front of the ring buffer.
- *Time.* Return the total number of times `tic` was called.

For convenience, a `main` method has been provided with some test code. Feel free to add any tests you see fit. Ensure your GuitarString class works before proceeding to the next step.

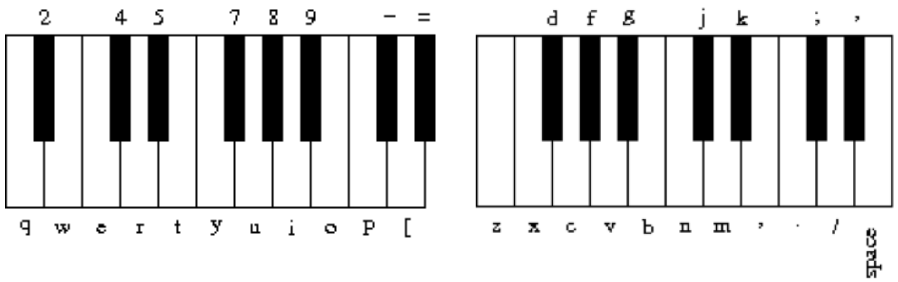
Interactive guitar player. *GuitarHeroLite.java* is a sample GuitarString client that plays the guitar in real-time, using the keyboard to input notes. When the user types the lowercase letter 'a' or 'c', the program plucks the corresponding string. Since the combined result of several sound waves is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

```
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();
        }
    }
}
```

Write a program GuitarHero that is similar to GuitarHeroLite, but supports a total of 37 notes on the chromatic scale from 110Hz to 880Hz. In general, make the i^{th} character of the string below play the `i` note.
`String keyboard = "q2we4r5ty7u8i9op-[=zxdcfvgbnjmk,.;/' ";`

This keyboard arrangement imitates a piano keyboard: The "white keys" are on the `qwerty` and `zxcv` rows and the "black keys" on the `12345` and `asdf` rows of the keyboard.



The i^{th} character of the string corresponds to a frequency of $440 \times 2^{(i-24)/12}$, so that the character `'q'` is approximately 110Hz, `'i'` is close to 220Hz, `'v'` is close to 440Hz, and `' '` is close to 880Hz. Don't even think about including 37 individual GuitarString variables or a 37-way `if` statement! Instead, create an array of 37 GuitarString objects and use `keyboard.indexOf(key)` to figure out which key was typed. Make sure your program does not crash if a key is played that is not one of your 37 notes. **Check the FAQ if you have problems.**

Run your program and play the following sequence, you should (hopefully!) hear a familiar tune:

```
i p z v b z p b n z p n d [ i d z p i p z p i u i i
```

(Advanced) Guitar Hero Visualizer

Write a program `GuitarHeroVisualizer.java` (by modifying `GuitarHero.java`) that plots the sound wave in real-time, as the user is playing the keyboard guitar. The output should look something like the below, but change over time. (Check out a video here.)

You should not re-draw the wave on every sample. Instead, draw the wave of the last `n` samples every `n` time steps for an appropriate value of `n`. Experiment with different values of `n` to find one that you think looks good and draws smoothly. The StdDraw class uses double buffering – there is an onscreen image, and an off-screen buffer that stores all drawing changes until it gets drawn to the screen.

There is more than one way to handle the drawing — there is not a "right" way to do this. You may also do a different visualization, as long as it is tied to the audio samples.

(Advanced) Clickable Keyboard

Create a class GuitarHeroKeyboard that also displays a keyboard and allows the user the click the corresponding keys to play the notes (alongside the ability to press the keys).

You can use the "keyboard.png" image, a horizontal black- and white-key keyboard contributed by Liberty student Tejas Bogguram in 2018. This image makes creating a working keyboard easier than using the standard locations of the black keys, which force you to check for mouse clicks in an X- and Y-range (rather than just the X-range).

(Advanced) Synthesize Other Instruments

Modify the Karplus-Strong algorithm to synthesize a different instrument. Consider changing the excitation of the string (from white-noise to something more structured) or changing the averaging formula (from the average of the first two samples to a more complicated rule) or anything else you might imagine.

For example, here is a college project that uses the GuitarHero framework to model a piano / player piano.

*Adapted from the **Guitar Hero** project*
http://nifty.stanford.edu/2012/wayne-guitar-heroine/