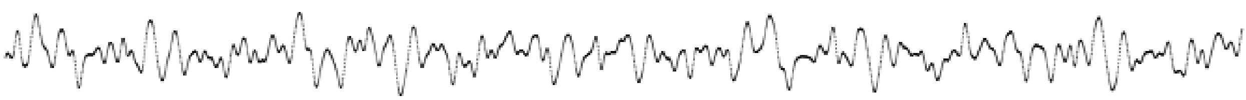
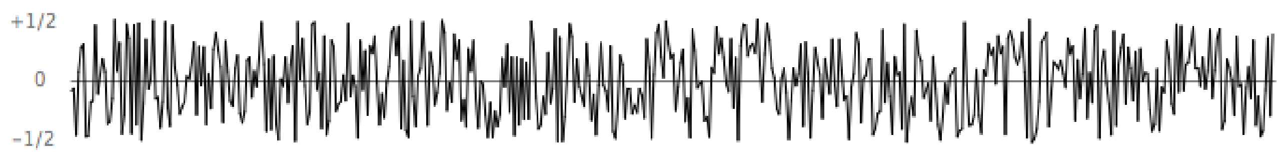
The purpose of this project is to write a program to simulate the plucking of a guitar string using the *Karplus-Strong* algorithm. This algorithm played a seminal role in the emergence of physically modeled sound synthesis, where a physical description of a musical instrument is used to synthesize sound electronically.

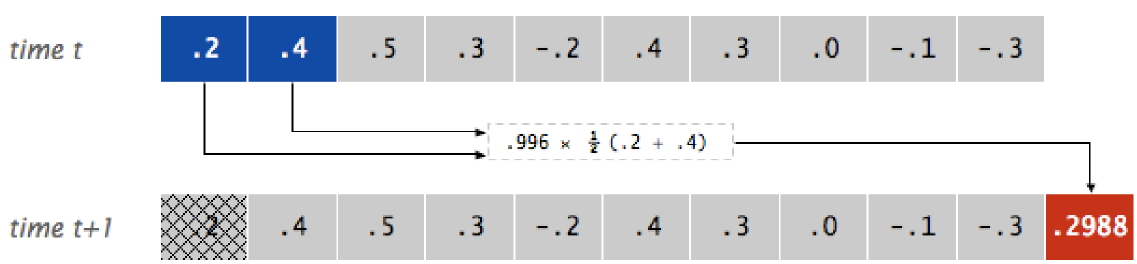
**Simulate the Plucking of a Guitar String** When a guitar string is plucked, the string vibrates and creates sound. The length of the string determines its *fundamental frequency* of vibration. We model a guitar string by sampling its displacement (a real number between $-1/2$ and $+1/2$) at $N$ equally spaced points in time. The integer $N$ equals the *sampling rate* (44,100 Hz) divided by the desired fundamental frequency, rounded **up** to the next integer.



- *Plucking a 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 $-1/2$ and $+1/2$.




- *The Resulting Vibrations* After the string is plucked, the string vibrates. The pluck causes a displacement that 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 deleted sample and the first sample, scaled by an energy decay factor of 0.996. For example:




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

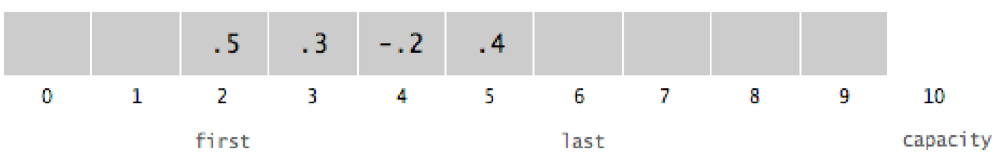
- *The Averaging Operation* The averaging operation serves as a gentle *low-pass filter*, which removes higher frequencies while allowing lower frequencies to pass, hence the name. Because it is in the path of the feedback, this has the effect of gradually attenuating the higher harmonics while keeping 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.

**Problem 1.** (*Ring Buffer*) Your first task is to model the ring buffer. Write a module `ring_buffer.py` that implements the following API:

| function | description |
|---|---|
| `create(capacity)` | create and return a ring buffer, with the given maximum capacity and with all elements initialized to `None` |
| `capacity(rb)` | capacity of the buffer $rb$ |
| `size(rb)` | number of items currently in the buffer $rb$ |
| `is_empty(rb)` | is the buffer $rb$ empty? |
| `is_full(rb)` | is the buffer $rb$? full? |
| `enqueue(rb, x)` | add item $x$ to the end of the buffer $rb$ |
| `dequeue()` | delete and return item from the front of the buffer $rb$ |
| `peek(rb)` | return (but do not delete) item from the front of the buffer $rb$ |

Since the ring buffer has a known maximum capacity, we implement it as a list (`buff`) of floats of that length, with the number of elements in the buffer stored in `size`. For efficiency, we use *cyclic wrap-around*, which ensures that each operation can be done in a constant amount of time. We maintain an index `first` that stores the index of the least recently inserted item, and an index `last` that stores the index one beyond the most recently inserted item. To insert an item into the buffer, we put it at index `last` and increment `last`. To remove an item from the buffer, we take it from index `first` and increment `first`. When either index equals capacity, we make it wrap around by changing the index to 0. The ring buffer can thus be represented as a list of four elements: the buffer (`buff`); number of elements (`size`) currently in `buff`; the index (`first`) of the least recently inserted item; and the index (`last`) one beyond the most recently inserted item. For example, the ring buffer shown in the figure below can be represented as the list `[[•, •, 0.5, 0.3, -0.2, 0.4, •, •, •, •], 4, 2, 6]`.




A ring buffer of capacity 10, with 4 elements

Calling `enqueue()` on a full buffer should terminate the program with the message "Error: cannot enqueue a full buffer". Calling `peek()` or `dequeue()` on an empty buffer should terminate the program with the message "Error: cannot peek an empty buffer" or "Error: cannot dequeue an empty buffer". Use `sys.exit(msg)` to terminate a program with the message `msg`.

```
$ python ring_buffer.py 10
Size after wrap-around is 10
55
$ python ring_buffer.py 100
Size after wrap-around is 100
5050
```

**Problem 2.** (*Guitar String*) Next, create a module `guitar_string.py` to model a vibrating guitar string. The module must implement the following API:

| function | description |
|---|---|
| `create(frequency)` | create and return a guitar string of the given frequency, using a sampling rate given by SPS, a constant in `guitar_string.py` |
| `create_from_samples(init)` | create and return a guitar string whose size and initial values are given by the list *init* |
| `pluck(string)` | pluck the given guitar string by replacing the buffer with white noise |
| `tic(string)` | advance the simulation one time step on the given guitar string by applying the Karplus-Strong update |
| `sample(string)` | current sample from the given guitar string |

Some details about the functions:

- `create(frequency)` creates and returns a ring buffer of 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. A guitar string is represented as a ring buffer of capacity $N$, with all values initialized to 0.0
- `create_from_samples(init)` creates and returns a ring buffer of capacity equal to the size of the given list `init`, and initializes the contents of the buffer to the values in the list. In this assignment, this function's main purpose is for debugging and grading.
- `pluck(string)` replaces the $N$ items in the ring buffer `string` with $N$ random values between -0.5 and 0.5.
- `tic(string)` applies the Karplus-Strong update: deletes the sample at the front of the ring buffer `string` and adds to the end of the ring buffer the average of the first two samples, multiplied by the energy decay factor.
- `sample(string)` returns the value of the item at the front of the ring buffer `string`.

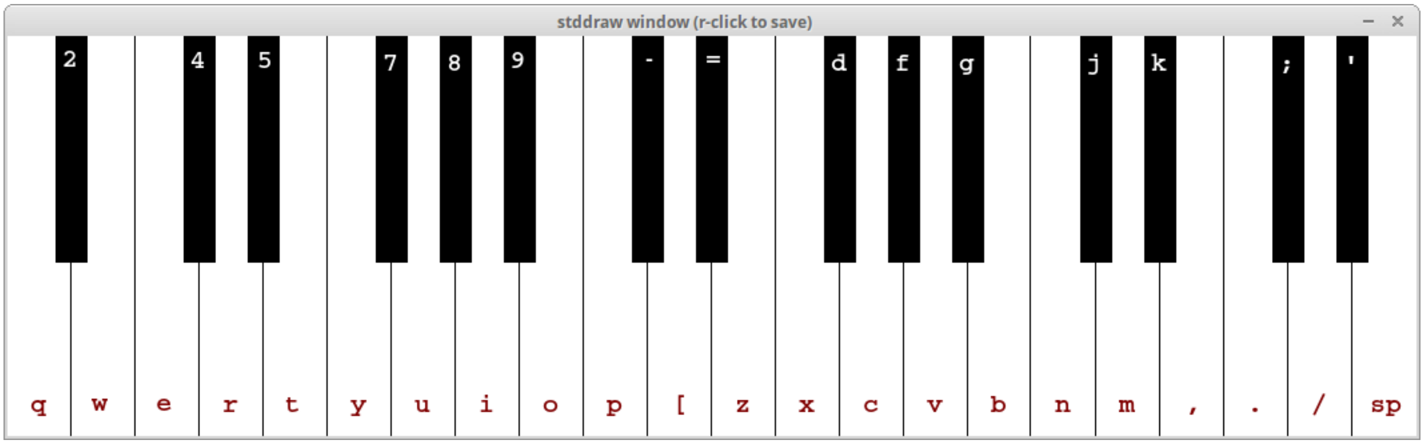
```
$ python guitar_string.py 25
     0   0.2000
     1   0.4000
     2   0.5000
     3   0.3000
     4  -0.2000
     5   0.4000
     6   0.3000
     7   0.0000
     8  -0.1000
     9  -0.3000
    10   0.2988
    11   0.4482
    12   0.3984
    13   0.0498
    14   0.0996
    15   0.3486
    16   0.1494
    17  -0.0498
    18  -0.1992
    19  -0.0006
    20   0.3720
    21   0.4216
    22   0.2232
    23   0.0744
    24   0.2232
```

**Problem 3.** (*Interactive Guitar Player*) The program `guitar_hero.py` is a `guitar_string` client that plays the guitar in real-time, using the keyboard to input notes. When the user types the characters `'q'`, `'2'`, or `'w'`, 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.

Your task is to complete the implementation of `guitar_hero.py` such that it supports a total of 37 notes on the chromatic scale from 110 Hz to 880 Hz. Use the following 37 keys to represent the keyboard, from lowest note to highest note:

```
keyboard = 'q2we4r5ty7u8i9op-[=zxdcfvgbnjmk,.;/\' '
```

The 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 `keyboard` corresponds to a frequency of $440 \times 2^{(i-24)/12}$, so that the character `'q'` is 110 Hz, `'i'` is 220 Hz, `'v'` is 440Hz, and `' '` is 880 Hz. Don't even think of including 37 individual `guitar_string` variables or a 37-way `if` statement! Instead, create a list of 37 `guitar_string` objects and use `guitar_hero.index(keyboard, key)` to figure out which key was typed. Make sure your program does not crash if a key is pressed that does not correspond to one of your 37 notes.

Superposing 37 notes to create a sample to play is computationally expensive, so only superpose the notes within a small window (`2 * guitar_hero.SUPERPOSITION_RANGE`) around the pressed key. For example, if the pressed key is `'p'`, assuming `SUPERPOSITION_RANGE` is 2, you should only superpose the notes corresponding to the keys `'9'`, `'o'`, `'p'`, `'-'`, and `'['`.

### Files to Submit

1. `ring_buffer.py`
2. `guitar_string.py`
3. `guitar_hero.py`
4. `report.txt`

Before you submit:

- Make sure your programs meet the input and output specifications by running the following command on the terminal:

  ```
  $ python run_tests.py [<problems>]
  ```

  where the optional argument `<problems>` lists the numbers of the problems you want to test; all the problems are tested if no argument is given.

- Make sure your programs meet the style requirements by running the following command on the terminal:

  ```
  $ pep8 <program>
  ```

  where `<program>` is the `.py` file whose style you want to check.

- Make sure your report doesn't exceed 400 lines, doesn't contain spelling mistakes, and doesn't contain lines that exceed 80 characters.

**Acknowledgements** This project is an adaptation of the Guitar Hero assignment developed at Princeton University by Andrew Appel, Jeff Bernstein, Maia Ginsburg, Ken Steiglitz, Ge Wang, and Kevin Wayne.