Data Structures

CS284 C/D—Spring 2019

Homework 3 - Rock Band

Instructor: Antonio R. Nicolosi — Due 9 April 2019, 11:59PM

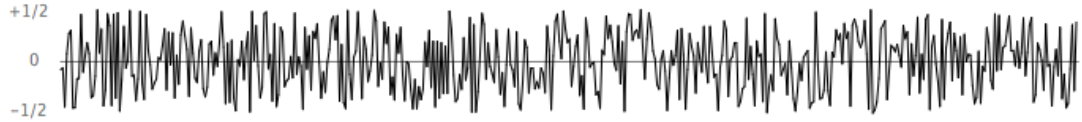
Course Staff: Justin Barish, Zach Hackett, Chris Hittner, Vidya Rajagopalan

1 Overview

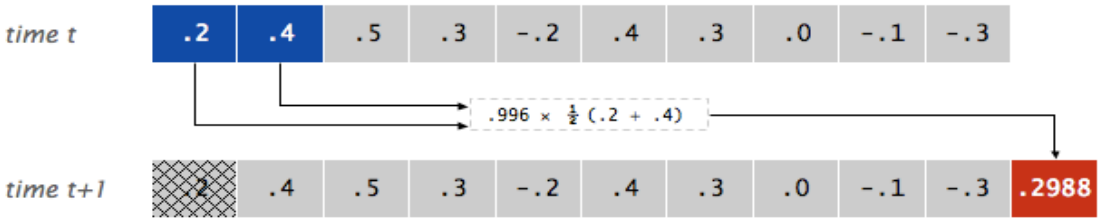
In this assignment, you will write a program to simulate four different musical instruments using the *Karplus-Strong* algorithm. This algorithm played a seminal role in the emergence of physically modeled sound synthesis (where a physical description of a musical instrument is used to synthesize sound electronically). The best way to understand this algorithm is by first looking at how we can use it to model a guitar.

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 regular time intervals, resulting in N samples, where N equals the ratio of the *sampling rate* (44,100) divided by the fundamental frequency of the string (rounded to the nearest integer).

- *Plucking the string*. When a string is plucked, its excitation can contain energy at any frequency. We simulate the excitation by filling the buffer with *white noise*: set each of the N sample 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 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 0.996.



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. 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 roundtrip 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 actually plucked strings sound.

At a high level, the Karplus-Strong algorithm approximately solves the 1D wave equation, which describes the motion of the string as a function of time. The wave shape spreads out over time. The vibration gradually eliminates those frequencies introduced by the pluck that don't match the string's fundamental frequency, with the higher frequencies decaying faster than the lower ones. Eventually, the wave shape will be sinusoidal with frequency equal to the string's fundamental frequency. The Karplus-Strong averaging formula is an extremely simplified method that results in a similar effect, though it remains a bit of a mystery even to experts.

2 How To Use This Document

1. Be sure to reference the FAQ, Suggestions, and sample test code at the end of the document.
2. Pay attention to the Assignment Policies in Section 7.
3. For the programming, program each part in the order presented in the document.
   - Start with parts 1-3, run the test program. Don't forget to compile the code in cos126
   - Then complete steps 4-11
   - If you would like, complete the extra credit.
4. The document guides you through a good way to program this project, (abstract classes, inheritance, etc). This is included not just for an exercise, but because it **greatly reduces the amount of code you have to write!** Make sure you follow the suggested programming means! Each method in steps 1-10 is just a few lines of code.

3 Programming Parts

1. **RingBuffer.**You already completed this part of the assignment in Assignment 2. A completed one has been given to you.

   Next, we need to model our strings.

2. **InstString**. This will be an abstract class. It will have all instance variables needed by any type of instrument string. It will have the following methods:

```
public abstract void pluck();
public abstract void tic();
public double sample();
public int time();
```

   - The sample() method returns the front value of our ring buffer. The default implementation should be in this abstract class.
   - The time() method returns the number of times tic() was called. The default implementation should be in this class.

   See the next section for details about pluck() and tic().

3. **GuitarString.** This will be class that implements the code needed for a Guitar String. It will **extend** the *abstract class InstString*. It will have the following methods:

```
GuitarString(double frequency);
GuitarString(double[] init);
void pluck();
void tic();
```

   - *Constructors.* There are two ways to create a GuitarString object:
     - The first constructor creates a RingBuffer of the desired capacity N (sampling rate divided by frequency, rounded 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 purpose is solely for debugging.
   - *Pluck.* Replace the N elements in the buffer with N random values between -0.5 and +0.5. In order to generate a pseudorandom number, use Math.random() which returns a double value in the interval [0.0,1.0). Since displacements are in [-0.5,0.5], you will have to come up with a function mapping values from the former interval to the latter.
   - *Tic.* Apply the Karplus-Strong update: delete the sample at the front of the buffer and add to the end of the buffer the average of the first two samples, multiplied by the energy decay factor. This is called a *Low-Pass-Filter.*

     Note: GuitarString extends an abstract class. That means it MUST implement all abstract methods of InstString. Since *sample* and *time* are already defined in InstString (not marked abstract!), there is no reason to re-define them here. Also, note that all instance variables needed for GuitarString should probably live in the InstString class (including our RingBuffer instance variable). Since all types of instrument strings are similar, they all need similar things, and doing this properly reduces code duplication.

     Another reason why we use the abstract class is for polymorphism. We can now do:

```
InstString s = new GuitarString(f);
```

     Which will help us group together all of our different type of strings.

****

At this point, enough code is written where running RockBandLite will allow you to test what you have so far. Dont move on until what you have so far works!

****

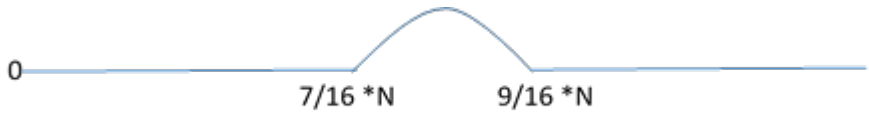
4. **PianoString.** This will be the class that implements the code needed for a Piano String. It will **extend** the abstract class InstString. It will have the following methods:

```
PianoString(double frequency);
PianoString(double[] init);
void pluck();
void tic();
```

   - *Constructors*. There are two ways to create a PianoString object:
     - The first constructor creates a RingBuffer of the desired capacity N (sampling rate divided by frequency, rounded 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 purpose is solely for debugging.
   - *Pluck.* Replace the N elements in the buffer using the following function:

$$f(x) = \begin{cases} 0 & x < \frac{7}{16} * N \text{ or } x > \frac{9}{16} * N \\ .25 * \sin 8\pi(\frac{x}{N} - \frac{7}{16}) & x \geq \frac{7}{16} * N \text{ and } x \leq \frac{9}{16} * N \end{cases}$$

     Where x is the x^{th} element the buffer.
     This simulates the small, localized disturbance caused when a hammer hits the string. Our wave should have the following look:

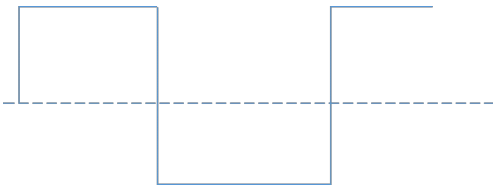


     **NOTE: Ensure you do DOUBLE division. That is, in java, $\frac{j}{N} = 0$ (when $N > j$, and N, j are integers), while $\frac{j}{((double)N)}$ = a decimal value.

5. **DrumString** This will be class that implements the code needed for a Drum String (We will refer to it as a string, even though it is technically a Drum Head). It will **extend** the abstract class InstString. It will have the following methods:

```
DrumString(double frequency);
DrumString(double[] init);
void pluck();
void tic();
```

- *Constructors.* There are two ways to create a DrumString object:
  - The first constructor creates a RingBuffer of the desired capacity N (sampling rate divided by frequency, rounded 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 purpose is solely for debugging.
- *Pluck.* Replace the N elements in the buffer using a square function. To generate our square function, we will use a sine function, using Math.sin(), with a frequency of $1/N$. For each value of x in our sine function, if $sin(x) > 0$ we will change it to a 1. Else, we will change it to a -1. This will give a function looking something like:



- *Tic.* Apply the following filter (with b=0.5, and S, our Stretch Factor, set to 5/3):

$$Y_t = \begin{cases} +Y_{t-p} & \text{probability } b\left(1 - \frac{1}{S}\right) \\ -Y_{t-p} & \text{probability } (1-b)\left(1 - \frac{1}{S}\right) \\ +\frac{1}{2}(Y_{t-p} + Y_{t-p-1}) & \text{probability } b\frac{1}{S} \\ -\frac{1}{2}(Y_{t-p} + Y_{t-p-1}) & \text{probability } (1-b)\frac{1}{S}. \end{cases}$$

Source: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.144.5585&rep=rep1&type=pdf

In this filter, we randomly decide whether or not to use the low-pass-filter described in GuitarString, or to just re-enqueue our dequeued element. We also randomly decide whether to enqueue our number as a positive or negative number. To do this, we dequeue our first element (call it s1, which is Y_{t-p} in the above), and peek at our second element (call it s2, which is Y_{t-p-1} in the above). We then use a random number generator to determine which algorithm to use. For this use **S=5/3.** This means that we have a 60% probability of randomly choosing a low-pass filter [from the above, 1/s, or 1/ (5/3), or .6, or 60%] (averaging s1 and s2), and a 40% probability of simply re-enqueuing s1. We then use a random number generator, with **b=.5** to decide whether or not to negate what we enqueue.

Now, we move on to modeling our instruments.

6. **Instrument**. This is an **abstract** class. It will have all instance variables needed by any type of instrument. It will have the following methods/instance variables:

```
protected InstString[] strings;
public void playNote(int i)
public double ringNotes()
```

*Note that the **class is abstract**, but **the methods are not.** They MUST be defined in Instrument.java. Also, only 1 instance variable is shown, but your implementation can (and should) have more.

   - *playNote(int i).* Given an index, pluck the appropriate string for that instrument.
   - *ringNotes().* Lets the notes keep ringing on the instrument. Calls tic() and adds up the sample for each string. Returns the sum of all string samples.

7. **Guitar.** This is a class that extends our Instrument class, and represents a guitar. By inheritance, it will inherit playNote, and ringNotes. With simple calls to those methods, we can play our instruments!

   All we need to play our guitar are those methods, which should have sufficient implementation for a guitar to play. Thus, all we need in this class is a constructer for Guitar:

   ```
   public Guitar(numNotes)
   ```

   This constructor will initialize our abstract class Instruments *strings* variable to size *numNotes*, and create *numNotes* amount of *GuitarStrings* for that instrument, with each string having the appropriate frequency (see formula below). All of these strings will be stored in that *strings* array. Each string i should be assigned a frequency using the formula: $CONCERT\_A * 2^{(i-24)}/12.0$ with i going from 0 to *numNotes*, where $CONCERT\_A = 440.0$ (Hz)

8. **Drum.** This is a class that **extends** our Instrument class, and represents a drum. By inheritance, it will inherit playNote, and ringNotes.

   All we need to play our drum are those methods, which should have sufficient implementation for a drum to play. Thus, all we need in this class is a constructer for Drum:

   ```
   public Drum(numNotes)
   ```

   Like with Guitar, this constructor will initialize our abstract class Instruments *strings* variable to size *numNotes*, and create *numNotes* strings for that instrument, with each string having the appropriate frequency, using the formula mentioned in Guitar. All of these strings will be stored in that *strings* array.

9. **Bass.** This class will **extend** Instrument, and be identical to Guitar, except use the following formula to generate frequencies for the strings: $CONCERT\_A * 2(i - 48)/12.0$. You can use GuitarString as your string type for the Bass (since a Bass is sort-of a low-pitched guitar). All of these strings will be stored in that *strings* array.

10. **Piano.** This is a class that **extends** our Instrument class, and represents a piano. By inheritance, it will inherit playNote, and ringNotes.

A piano, however, is different that all other instruments, in that it has 3 strings per note. Thus, we need to override our inherited methods, so that Piano plucks all the strings in playNote, and collects samples from and tics all strings per note. Thus, Piano has the following methods/instance variables:

```
protected InstString[][] pStrings;
public Piano(numNotes)
public void playNote(int i)   //Overrides the one in Instrument
public double ringNotes()     //Overrides the one in Instrument
```

The constructor will initialize the 2D array of strings (with numNotes rows and 3 columns (3 strings per note)), and using the frequency formula discussed in guitar, it will, per note, create 1 PianoString of that exact frequency, 1 PianoString of that frequency +0.45, and 1 PianoString of that frequency -0.45. (Since pianos are not tuned exactly, and it is more interesting to program). All of these strings are stored in our local pStrings array (not our abstract classs strings array, since that one is 1D and we need a 2D array).

Each of the other 2 methods will act similarly to the implementation in Instrument, but instead, will pluck each of 3 strings per note, tic each of 3 strings per note, and add the sample for each of 3 strings per note

11. **RockBand.** This is our main class for the program. This program will have a public static void main(String [] args).

    - Initialize an array of Instruments (a Guitar with 37 strings, a Piano with 37 strings, a bass with 37 strings, and a Drum with 19 strings).
    - Map key presses to notes/instruments. In RockBandLite, we use if(key==a), if(key==c) DO NOT DO THIS 95 times! Instead, use the provided:

      ```
      String guitarBassKeyboard = "'1234567890-=qwertyuiop[]\\asdfghjkl;'";
      String pianoKeyboard = "~!@#$%^&*()_+QWERTYUIOP{}|ASDFGHJKL:\"";
      String drumKeyboard = "ZXCVBNM<>?zxcvbnm,.";
      ```

      - Use _____Keyboard.indexOf(key) to get an index that corresponds to a string for that instrument. Fill in the blank for each type of keyboard. That is, if you type 3, it should play a guitar sound, plucking the 3rd string (assuming zero indexed). If you type C, it should play a drum sound, plucking the 2nd string (assuming zero indexed).
      - Use the Enter key to toggle LowMode. When the Enter key (char '\n') is first pressed, it will turn on LowMode, which will instead play bass notes instead of guitar notes, since the guitar and bass share the same Keyboard variable. When Enter is hit again, it will turn off LowMode, disabling the bass and re-enabling the guitar. (I.E. hitting Enter, then 2, will sound the 2nd bass string, hitting Enter again, then 2, will sound the 2nd guitar string)
      - Once you have your index, pass it to playNote for the appropriate instrument.
      - Send the sound to the sound card, by repeatedly calling ringNotes() for each instrument, adding up the results, and calling StdAudio.play(sumOfAllInstrumentSamples);

    At this point, you should be hearing instrument sounds when you hit your keys!

****

Extra Credit:

Implement the **play_from_file** command-line option. When this option is specified, the program will read the notes from a text file given as a command-line argument, play them back, and exit the program (no user real-time input). For example, the program will be invoked with:

```
java assign3.RockBand -play_from_file notes.txt
```

Two different txt files are provided for you, with their format described in Song_Format.txt. You are also to write an additional song txt file of your own!

Make sure to leave a comment on canvas if you complete the bonus.

4 Deliverables:

Submit RingBuffer.java (given to you), InstString.java, GuitarString.java, PianoString.java, DrumString.java, Instrument.java, Piano.java, Guitar.java, Drum.java, Bass.java, and RockBand.java. (11 files in total). Also, if you did the extra credit, submit your song .txt file

- Do not submit the std library files in cos126.
- Submit your assignment as a tar file, named: `<LASTNAME>_<FIRSTNAME>_assign3.tar`
- To create the tar, from above the assign3 directory, type:

  ```
  tar -cf <LASTNAME>_<FIRSTNAME>_assign3.tar ./assign3
  ```

- Ensure that your Instrument.java and InstString.java are abstract classes
- Ensure that all of your classes are named correctly, and that all of the required methods are present.
- Make sure to check that your tar file is correct. You can 'untar' it by typing

  ```
  tar -xf <LASTNAME>_<FIRSTNAME>_assign3.tar
  ```

5 Suggestions:

These are suggestions for how you might make progress. You do not have to follow these steps.

- To implement `pluck()` for GuitarString, use a combination of the RingBuffer methods `size()`, `dequeue()`, and `enqueue()` to replace the buffer with values between -0.5 and 0.5.
- To implement `tic()`, use a combination of `enqueue()`, `dequeue()`, and `peek()`.
- To implement `sample()`, use `peek()`.

6 Frequently Asked Questions:

You specify certain methods/ variables. Can I have more? Yes. As long as you have the required methods/variables that are listed in this document, you can add any methods you want.

Where do I enter keystrokes in RockBandLite and RockBand? Be sure that the standard draw window has focus by clicking in it. Then, type the keystrokes.

How do I round a `double` to the nearest `int`? Use Math.round. For example, '(int) Math.round(val)' will round 'val' and narrow it to integer.

How do I do double division? Ensure at least one of your numbers is a double, or cast one of your numbers to a double (BAD 7/16, GOOD 7/16.0, GOOD 7/(double)16)

What happens if I send a sample whose value is greater than 1 or less than -1 to StdAudio? The value is clipped - it is replaced by the value 1.0 or -1.0, respectively.

I get a `Ring buffer underflow` error in RockBandLite before I type any keystrokes. Why? Did you forget to initialize the ring buffer to contain N zeros in your GuitarString constructor?

When I run `RockBandLite` for the first time, I hear no sound. What am I doing wrong? Make sure you have tested with the `main()` provided for GuitarString. If that works, it is likely something wrong with `pluck()` since the `main()` provided for GuitarString does not test that method. To diagnose the problem, print out the values of `sample()` and check that they become nonzero after you type upper case characters 'A' and 'C'.

When I run `RockBandLite` , I hear static (either just one click, and then silence or continual static). What am I doing wrong? It's likely that `pluck()` is working, but `tic()` is not. The best test is to run the `main()` provided for GuitarString.

How do I use `keyboard.indexOf(key)`? If keyboard is a String and key is a character, then `keyboard.indexOf(key)` return the integer index of the first occurrence of the character key in the string `keyboard` (or -1 if it does not occur).

7 Assignment Policies

Homework will be done individually: each student must hand in their own answers. It's acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students, be taken from online sources, or be taken from previous offerings of this or any other class.

Excerpts of code presented in class can be used.

MOSS will be used when grading the assignment to check for plagiarism.

8 Testing:

Be sure to thoroughly test each piece of code as you write it. We offer a suggestion below.

Guitar string. You can test your `GuitarString` data type, using the `main()` provided:

```
public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      double[] samples = { .2, .4, .5, .3, -.2, .4, .3, .0, -.1, -.3 };
      GuitarString testString = new GuitarString(samples);
      for (int i = 0; i < N; i++) {
          int t = testString.time();
          double sample = testString.sample();
          System.out.printf("%6d %8.4f\n", t, sample);
          testString.tic();
      }
  }
```

```
% java assign3.GuitarString 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
```