PS4a: CircularBuffer for StringSound

Due: Thursday, June 9, 11:59pm

For this project, we will use the Karplus-Strong algorithm to simulate the plucking of a guitar string. This algorithm played a foundational role in the emergence of physically modelled sound synthesis (where a physical description of a musical instrument is used to syntesize sound electronically).

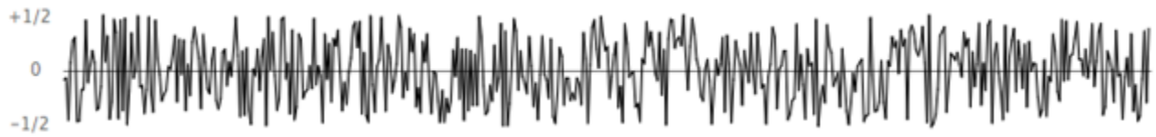
In part A, we will build the structures needed. We will use unit testing to ensure that they work correctly, including throwing exceptions when appropriate.

1 Simulating 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 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 nearest integer). For example, for A_4 (middle A), which has a frequency of 440 Hz, $n = \lceil 44100/440 \rceil = 11$.

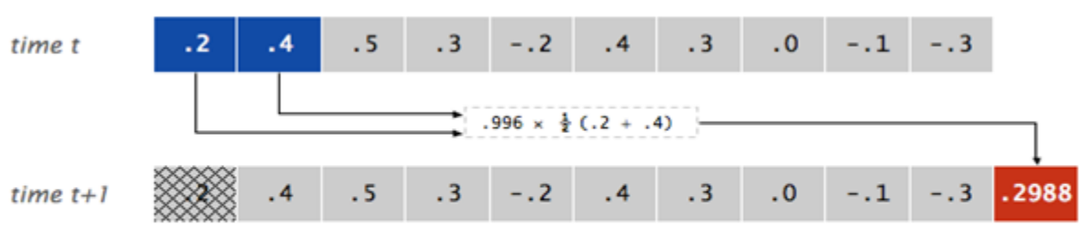
1.1 Plucking the string

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 `int16_t` (between -32768 to 32767).



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 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

1.3 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 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 (0.996 in this case) models the slight dissipation in energy as the wave makes a round trip through the string.

1.3.1 The averaging implementation

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.

2 Implementation

Write a class named `CircularBuffer` that is templated on a type `T` and implements the following API.

```
class CircularBuffer {
 public:
    CircularBuffer(size_t capacity); // Create an empty ring buffer with
   given max capacity

    size_t size() const; // The number of items currently in the buffer
    bool isEmpty() const; // Is the buffer is empty?
    bool isFull() const; // Is the buffer full?
    void enqueue(T item); // Add item to the end
    T dequeue(); // Delete and return item from the front
    T peek() const; // Return (but do not delete) item from the front
}
```

2.0.1 Important notes

1. As a templated class, the implementation will be in your `CircularBuffer.hpp` file. The implementation should appear below the class declaration rather than being inlined.
2. Attempts to instantiate a `CircularBuffer` with a capacity of less than 1 should result in a `std::invalid_argument` exception and an appropriate error message.
3. Attemts to enqueue to a full buffer should result in a `std::runtime_error` exception and an appropriate error message.
4. Attempts to dequeue or peek from an empty buffer should result in a `std::runtime_error` exception and an appropriate error message.
5. You *are* allowed to use any of the classes in the STL, some of which can guarantee $O(1)$ time for the required operations. Remember that some of your requirements are different from the provided behavior.

3 Debugging and testing

You must write a `test.cpp` file that uses the Boost function `BOOST_REQUIRE_THROW` and `BOOST_REQUIRE_NO_THROW` to verify that your code properly throws the specified exceptions when appropriate (and does not throw an exception when it shouldn't). As usual, use

`BOOST_REQUIRE` to excercise all methods of the class. **Your test file must excercise all methods of your `CircularBuffer` and must excercise all exceptions**. You may optionally write a `main.cpp` file that drives around your `CircularBuffer` class and use that for additional testing.

4 Extra Credit

You can earn extra credit by creating a lambda expression and passing it as a parameter to another function (such as those within the `<algorithm>` library). Calling your lambda from within the function where it is created is worth no points. If you do any of the extra credit work, make sure to describe exactly what you did in `Readme-ps4a.md`.

5 What to turn in

Your makefile should build a program named `test`.

Submit a tarball to Blackboard containing:

- Your CircularBuffer (`CircularBuffer.hpp`). Since it is templated, there will not be a `.cpp` file.
- Your unit tests (`test.cpp`)
- The makefile for your project. The makefile should have targets `all`, `test`, `lint` and `clean`. Make sure that all prerequisites are correct.
- Your `Readme-ps4a.md` that includes
    1. Your name
    2. Statement of functionality of your program (e.g. fully works, partial functionality, extra credit)
    3. Key features or algorithms used
    4. Any other notes
- Any other source files that you created.

Make sure that all of your files are in a directory named `ps4a` before archiving it and that there are no `.o` or other compiled files in it.

6 Grading rubric

Feature	Points	Comment
Core Implementation	4	Full & Correct Implementation
	1	Has required functions
	1	Can enqueue elements
	1	Can dequeue elements
	1	All required functionality is correct
Testing	5	Should test
	1	Test both good and bad constructors
	1	Test enqueue and peek
	1	Test dequeue
	1	Tests exceptions
	1	Tests all public functions
Makefile	1	
	1	Has targets `test`, `lint`, `all`, and `clean` and builds
Readme	2	Complete
	1	Describes the data structures used and the time and space complexities of the required operations.
	1	Describes the testing decisions.
Extra Credit	1	
	+1	Uses a lambda expression as a parameter describes in readme.
Penalties		
	-2	Linting problems
	-1	Non-private fields
	-1	Submission includes `.o` files
	-1	Submission not in a directory named `ps4a`
	-10%	Each day late
Total	12	