PS4

Synthesizing a Plucked String Sound (Part A)

CircularBuffer implementation with unit tests and exceptions

We will

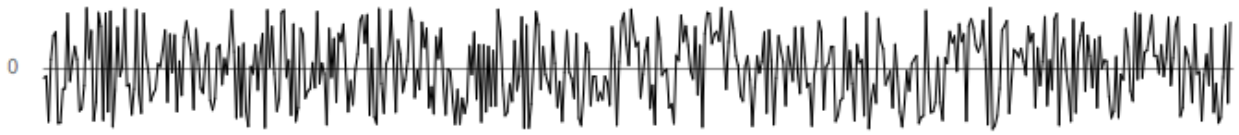
- Write a program to simulate plucking a guitar string using the Karplus-Strong algorithm. This algorithm played a seminal role in the emergence of physically modeled sound synthesis (where a physical description of a musical instrument is used to synthesize sound electronically).
- Learn about cpplint; best practices in C++ coding
- Use Unit testing and exceptions in the implementation.

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 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.

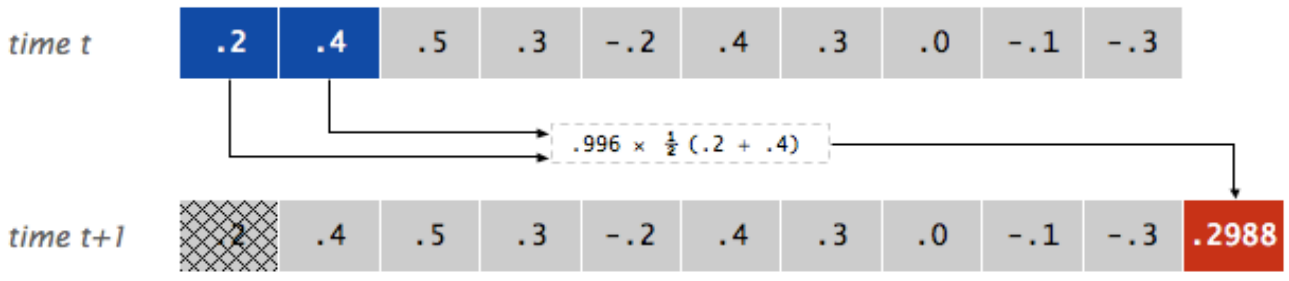
Plucking the string

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 between -32768 to 32767.



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.

Implementation

Write a class named `CircularBuffer` that implements the following API:

```
class CircularBuffer
---------------------------------------------------------------------------
        CircularBuffer(int capacity)  // create an empty ring buffer, with given
max
                                      // capacity
int     size()                        // return number of items currently in the buffer
bool    isEmpty()                     // is the buffer empty (size equals zero)?
bool    isFull()                      // is the buffer full  (size equals capacity)?
void    enqueue(int16_t x)            // add item x to the end
int16_t dequeue()                     // delete and return item from the front
int16_t peek()                        // return (but do not delete) item from the front
---------------------------------------------------------------------------
```

Your code must `#include <stdint.h>` header that defines the standard 16-bit integer type `int16_t`.

Important notes:

1. The code should be in a pair of files named `CircularBuffer.cpp` and `CircularBuffer.h`.
2. Attempts to instantiate with a capacity less than 1 should result in a `std::invalid_argument` exception, and the error message `CircularBuffer constructor: capacity must be greater than zero`.
3. Attempts to enqueue to a full buffer should result in a `std::runtime_error` exception, and the error message `enqueue: can't enqueue to a full ring`.
4. Attempts to dequeue or peek from an empty buffer should result in a `std::runtime_error` exception, and an appropriate error message.

Debugging and testing

- You should write a `test.cpp` file that uses the Boost functions `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 exercise all methods of the class.
- **Your test file must exercise all methods of your ring buffer, and must exercise all exceptions.**
- You can write a `main.cpp` file that drives around your `CircularBuffer` class, and use that for additional testing.

cpplint

Google's style guide is here: https://google.github.io/styleguide/cppguide.html

The `cpplint.py` file can be retrieved from https://github.com/cpplint/cpplint

or https://raw.githubusercontent.com/google/styleguide/gh-pages/cpplint/cpplint.py

Save the `cpplint.py` file onyour machine, and then:

```
chmod +x cpplint.py
sudo mv cpplint.py /usr/local/bin
```

Now, you can style-check a file using `cpplint.py` as an executable:

```
cpplint.py ''filename''
```

Alternately, you could run it using Python:

```
python cpplint.py ''filename''
```

You may use "`//NOLINT`" for not including full path for the .h files guards.

Additional Files

Produce and turn in a `Makefile` for building your class.

The executable that the `Makefile` builds must be called `ps4a`. This should be the result of linking your `test.o` and your `CircularBuffer.o`.

Produce and turn in a plain-text `ps4a-readme.txt` file that explains what you have done.

In particular, describe:

- how you implemented the ring buffer
- exactly what works or doesn't work

Submit your work

You should be submitting at least five files:

1. `CircularBuffer.cpp`
2. `CircularBuffer.h`
3. `test.cpp`
4. `Makefile`
5. `ps4a-readme.txt`

If you create a `main.cpp` with `printf`-style tests, you may submit that as well.

Place the files in subdirectory called `ps4a`, and archive with:

```
tar czvf ''<archive-file-name>''.tar.gz ps4a
```

The name of your submit file should include your name and the project name "PS4a" – e.g., PS4a_JoeSmith.tar.gz. Submit the archive via Blackboard.

Grading rubric

Feature	Value	Comment
core implementation	4	full & correct implementation = 4 pts; nearly complete = 3pts; part way=2 pts; started=1 pt
Makefile	1	Makefile included
your `test.cpp`	4	should test that you: generate `std::invalid_argument` exception on bad constructor; don't generate exception on good constructor; `enqueue`, `dequeue`, and `peek` work; generate `std::runtime_error` when calling `enqueue` on full buffer; generate `std::runtime_error` when calling `dequeue` or `peek` on empty buffer.
`cpplint`	2	Your source files pass the style checks implemented in `cpplint`
`readme`	2	Readme should say something meaningful about what you accomplished 0.5 point for explaining how you tested your implementation 0.5 point for explaining the exceptions you implemented 1 point for correctly explaining the time and space performance of your CircularBuffer implementation
naming	1	Your files (including tar.zip should be named according requirements)
Total	**14**	
extra credit	2	Use of the `lambda` expression