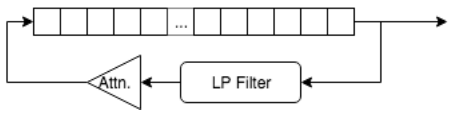
Polyphonic Synthesizer with the Karplus-Strong Algorithm

-by Qingyang (Tom) Xi

Description
The project aims to create a polyphonic synthesizer of guitar tones using the Karplus-Strong algorithm.

The Karplus-Strong algorithm is one of the earliest and most efficient physical modeling algorithms that simulates the tone of a plucked string.

Like many great inventions, the Karplus-Strong algorithm was an accidental discovery. The basic design of the algorithm entails a recirculating delay line that is filtered and attenuated, and it can be seen in the following figure:



The delay line is initially filled with noise, and by shifting the values in this recirculating shift register, the output samples stream to compose the sound of a plucked string.

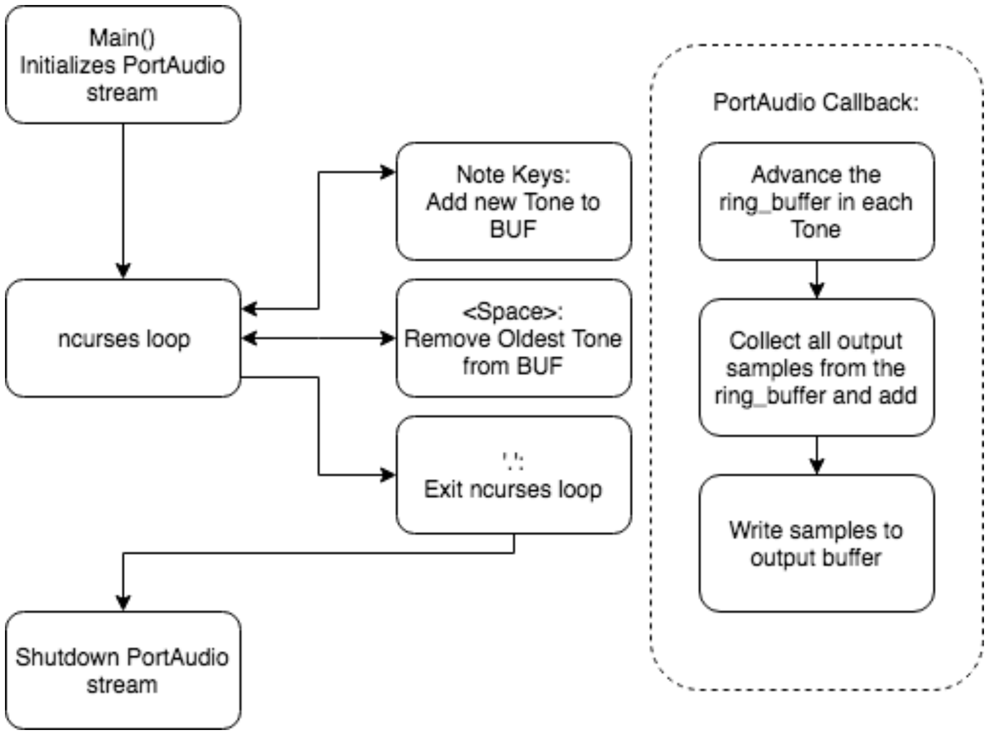
The Karplus-Strong algorithm is seen as the first example of physical modeling for music synthesis what we now call Digital Wave Guides. The central idea of this class of physical models is to use values in a delay lines to simulate the longitudinal displacement of a vibrating string. Since according to the d'Alembert's solution of wave equation, the traveling wave is readily simulated with a shift register. The low pass filter and gain unit simulates the frequency dependent energy loss of real-world vibrating strings over time.

Another way to understand why such a simple structure could produce such complex and harmonic sound is to view the delay line as a resonating body that emphasizes the signal having fundamental period corresponding to the length in time of the delay line.

The program will evoke an instance of the above described structure for every tone that it is asked to synthesis in real time, with a maximum polyphony of 8 voices.

Usage Context
This program can be simply used as a standard polyphonic synthesizer with minimal storage requirement over a very wide range, for it does not require wavetables or other pre-recorded samples. All audio samples are generated during program runtime.

Signal flow and program design


Main()
Initializes PortAudio stream
ncurses loop
Note Keys: Add new Tone to BUF
<Space>: Remove Oldest Tone from BUF
'':
Exit ncurses loop
Shutdown PortAudio stream
PortAudio Callback:
Advance the ring_buffer in each Tone
Collect all output samples from the ring_buffer and add
Write samples to output buffer

The code for this project is adapted from HW11, with modification to the Tone structure:

```
typedef struct {
    int key; /* index into freq array */

    int buffer_len; /* the length of the ring_buffer, bigger than string_len*/
    double string_len; /* the length of the string being simulated.
                        * Due to freqeuncy spacing, this can be fractional.*/
    double* ring_buffer; /* The delay line as a double array.*/
    int write_index; /* where the next sample should be written */
    double read_index; /* the position of the end of the current delay line*/

    double attack_factor; /* to avoid pop at the onset */
    double decay_factor; /* proportional to tone wavelength */
    double attack_amp; /* save attack amplitude */
    double z; // state of the filter
} Tone;
```

and addition of the following three functions, all implemented in "support.c":

```
void init_tone(Tone *pt, int this_key, double string_len);
double tic(Tone *pt);
double fractional_buffer_read(Tone *pt);
```

init_tone initializes the tone structure by providing the desired len of the string to be simulated. The actual buffer created for the tone is always a little bit bigger than the length of the string. This is to enable fractional string_len.

Tic advances the buffer in a Tone by one timestep, and returns a single sample. Tic also updates the write_index and read_index of the Tone structure, as well as the buffer itself. Filtering and attenuation is also included in this function call.

fractional_buffer_read is a helper function for tic, and enables fractional delay lines. Current implementation is a linear interpolation between two adjacent samples.

Computational complexity
By using a ring buffer structure where the content of the array stays put while the read/write index of the array moves, each call of the function tic is constant time.

Every time a note is added, memory allocation and noise initialization for the buffer occurs. This is O(n) where n is the size of the buffer.

Every time a note is ended, either through spacebar or exceeding max polyphony, the memory of the buffer associated with the tone being removed is freed. This is also constant time.

Initializing notes are the most computationally heavy elements in this program right now. However, this only occurs with each valid keystroke.
PA callback however, albeit being constant time, is called once per audio frame.

Platform and compilation

The project was developed on MacOS, using CLion.

The following build.sh is included in the program to build from source code.

```
gcc -Wall -o synth synth.c support.c -I/usr/local/include \
        -L/usr/local/lib -lsndfile -lportaudio -lncurses
```