Problem Set 9: Synthesizer

Please send back to me via NYU Brightspace
- A zip archive named as
  `PS09_Last_First.zip`
  Where `Last` and `First` is you last and first name and
  The zip archive contains the C code files that implements all aspects of all problem.

Total points: 100
Points are awarded as follows:
- **40 Points** – Write `paCallback()` in main.cpp
- **40 Points** – Complete `synth_block()` in synth.cpp
- **10 Points** – Submit waveform showing one note from onset to "drop out"
- **10 Points** – Clear code, sensible formatting, good comments

You are given the following in PS09:
- These instructions
- build.sh which should be used to compile your C++ code.

References
The following URL provides a good reference for C language library function usage:
http://www.cplusplus.com/reference/cstdlib/

libsndfile reference
http://www.mega-nerd.com/libsndfile/api.html/

portAudio reference
http://www.portaudio.com/

Problem Overview
First, this project is written in C++, which is also able to link with files written in C. You will write code in the C++ files.

In this assignment you will create a keyboard synthesizer, where the synthesizer keyboard is the computer keyboard. Each key is associated with a note to play, and each note is the fundamental frequency of that note (that is, there are no harmonics). Multiple notes can be played simultaneously.

The tone samples are computed as
```
v = sin(phase);
```
For each new sample of an output channel, the phase is incremented according to the frequency of the tone. The increment for a frequency f0 is
```
phase_inc = 2*PI*f0/samping_rate;
```

If f0 is 1 Hz, then after `sampling_rate` audio samples, the phase will have run from 0 to 2*PI, or one complete cycle. If f0 is 2 Hz, then 2 complete cycles.

In addition, the synthesized tone has an exponential attack so there is no "click" at tone onset and slower exponential decay so the tone "fades away" after the key is pressed.

Each segment of the assignment creates a portion of the synthesizer or adds additional functionality to the synthesizer. You need to add code where you see the comment
```
//Your code here
```

You are given the complete code of the following files:
```
main.h
synth.h
user_io.cpp
freq.h
key_tables.c
key_tables.h
paUtils.c
paUtils.h
build.h
```
You are given portions of the code for the following files, and your assignment is to add to the code in these files, as described below:
```
main.cpp
synth.cpp
```

Code Overview
main.cpp
There are no command line arguments for this program, so main.cpp does not have code to parse the command line or print a Usage statement.

An instance to the Synth Class is defined at the top of main(), and a pointer to the instance:
```
Synth synth, *ps = &synth;
```

Class Synth, and also struct Tone are declared in synth.h. An array of struct Tone is in the Class Synth.

Next in main() is that the table key2index[] is initialized (in file key_tables.c).
```
/* initialize tables */
init_key2index();
```
This provides a mapping from your laptop keys to synthesizer tone frequencies. The actual frequencies associated with note (e.g., A, A#, B, C, etc.) are in
```
freq.h
```

Next is initialization of the values in the Class Synth. The definition of member function `init_synth()` is in synth.cpp.

```
/* initialize Synth */
ps->init_synth(SAMP_RATE);
```

Next is that the pointer to the Class Synth is copied to the buf struct, which is the PortAudio callback structure.

```
/* initialize struct for callback */
buf.ps = ps;
```

Next the code starts PortAudio and then starts Ncurses (or alternative user I/O) and waits for user key input. Ncurses prints:

```
Welcome to my synthesizer!
Keyboard to piano key mapping is:
wer yu opl   -> F3# to A4#
asdfghjkl;\' -> F3 to B4
'.' to stop playing oldest tone
SpaceBar to quit

Key:
```

If the user hits SpaceBar, then Ncurses exits its loop, PortAudio exits and the program exits.

Key_table.c
This file is complete as is. It implements the following functions:

```
bool valid_key(int key);
```
This function returns `false` if the key pressed in the Ncurses loop is not associated with a note.

```
double key2freq(int key);
```
This function maps a valid key to a tone frequency to synthesize.

```
void init_key2index(void);
```
This function initializes the key2freq table, which is used in the `key2index()` function.

Synth.cpp
This defines the Class Synth.

```
void Synth::init_synth(int samp_rate);
```
This function converts initializes the Class Synth member data.

```
void execute_cmd(int cmd);
```
This function converts a command (cmd) to call to the appropriate functions.

```
double *Synth::synth_block(int len);
```
You will complete this function. It is described below.

```
void Synth::add_key(double new_freq)
```

This function adds key information to the Tone array of Class Synth that are the keys that are being voiced by your synthesizer.

```
void Synth::rm_key(void);
```
This function removes key information from the Tone array.

```
void Synth::shift_keys(void);
```
This function shifts the information in the Tone array down by one place.

```
void Synth::init_key(double new_freq);
```
This function writes information for a new key into the highest available position in the Synth array.

Your Assignment
You must add code to two files: main.cpp and synth.cpp. The specifics are described below.

main.cpp
(40 Points) paCallback()
Write the callback function
```
static int paCallback(
    const void *inputBuffer,
    void *outputBuffer,
    unsigned long framesPerBuffer,
    const PaStreamCallbackTimeInfo* timeInfo,
    PaStreamCallbackFlags statusFlags,
    void *userData)
```

Define a
```
    double *po;
```
The callback output block is synthesized by a call to `synth_block()`, which is located in the file synth.cpp, as shown here:
```
    po = ps->synth_block(framesPerBuffer);
```
where `synth_block()` returns a pointer to
```
    double output[FRAMES_PER_BUFFER];
```
which is defined in `synth.h` and filled in by `synth_block()`. This is a buffer of framesPerBuffer mono audio samples, and must be written to the callback output buffer in interleaved form.

Inside a loop of framesPerBuffer audio frames, write the mono values to the Left and Right channel values to the callback output buffer for each audio frame.

Finally, if DB_WAV_OUT has the value (1), write the callback output buffer to the debug file.

synth.cpp

4

(40 Points) synth_block()
25 Points – basic synthesizer
In function
```
double *Synth::synth_block(int framesPerBuffer)
```

Check for cmd
As the first block of code in the `synth_block()` function, get a local copy of cmd (which is an atomic read) and check if it is set, that is, if cmd > 0. If so, then call function
```
execute_cmd(local_cmd);
```
and reset `cmd` (which is an atomic write)
```
ps->cmd = 0;
```

Generate a block of tones
Simple synthesizer loop
- The synthesizer will have a loop that runs over `framesPerBuffer` frames, where a frame is the interleaved samples of all channels.
- For each new output value, Initialize the output value to zero
   ```
   double v = 0;
   ```
- Within this loop is a second loop that computes an output sample for each of the tones that are currently voicing. These tones are described in the first `num_keys` entries in the Tone structure array.
- If the phase_inc for that tone is greater than -1, that is if
   > pt[n].phase_inc > -1

   Then compute the next tone sample value is computed using the expression
   ```
   v += ampl * sin(pt[n].phase);
   ```
   where
   > `v` is the output sample value. Use the "+=" operator since each tone sample sums into the output sample value.
   >
   > `pt[n]` is a pointer to index n of the `Tone` array

   After tone sample value Is computed, its `phase` in incremented by the `phase_inc`. Note that, first, the `phase_inc` for all structures in the `Tone` array were initialized to zero, and the output value `v` was set to zero prior to entering the inner loop. Hence, initially, no tones will be generated by the processing loops.

You now have a complete synthesizer, which should compile and run.

15 Points – Add Exponential Attack and Decay
However, the synthesizer has two negative features. The first is that once a key is pressed, that tone plays forever, or until you press '.' or you exit the program. The second is that the tones start and stop playing as full-scale values. That is not very realistic, and furthermore, causes an audible "pop" when play starts and stops.

To remove the audible pop and make the synthesizer more realistic, add an "attack" and "decay" to the envelope of the synthesized tone. The envelope of a tone will have an

exponential attach and decay, you will stop any tone whose amplitude decays below a specified level DROP_LEVEL, as defined in synth.h

Exponential Decay

An exponential decay is specified by

$$f(t) = e^{-\left(\frac{t}{\tau}\right)}$$

Where τ is the *time constant* and f(t) has the value 1.0 when t has the value 0, and the value 1/e = 1/(2.71828) = 0.368 when t has the value τ . A plot of f(t) for a time constant of 1.0 seconds looks like this:
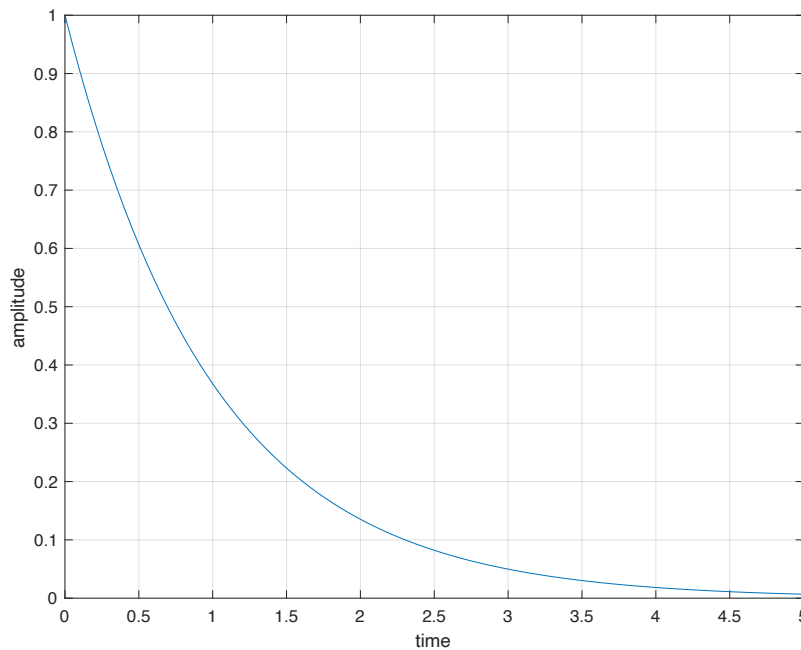
This can be expressed as a function of n, the digital sample index as

$$f(n) = e^{-\left(\frac{n/Fs}{\tau}\right)}$$

Where Fs is the sampling frequency so that n/Fs is the sample-domain equivalent of t. This can be expressed as

$$f(n) = \left(e^{-\left(\frac{1/Fs}{\tau}\right)}\right)^n = (decay\_factor)^n$$

Hence, implementing an exponential decay consists of multiplying the output sample value by

$$(decay\_factor)^n$$

Where n is the sample frame index. This can be computed iteratively. Let

$$decay\_amp(n) = (decay\_factor)^n$$

Then

$$decay\_amp(n) = decay\_amp(n-1) * decay\_factor$$

where

$$decay\_amp(0) = 1.0$$

And the indexes are the sample index.

In our program, there is actually just `decay_amp` and `decay_factor`, where `decay_amp` is initialzed to 1.0 when a new key is pressed and, for each sample frame in the output
- the output value is multiplied by `decay_amp` and
- `decay_amp` updated as `decay_amp *= decay_factor`.
- The resultant `decay_amp` value is saved (in the Tone structure).

The value for `decay_factor` corresponding to a time constant of 1.0 seconds is given in the `synth.h` header file.
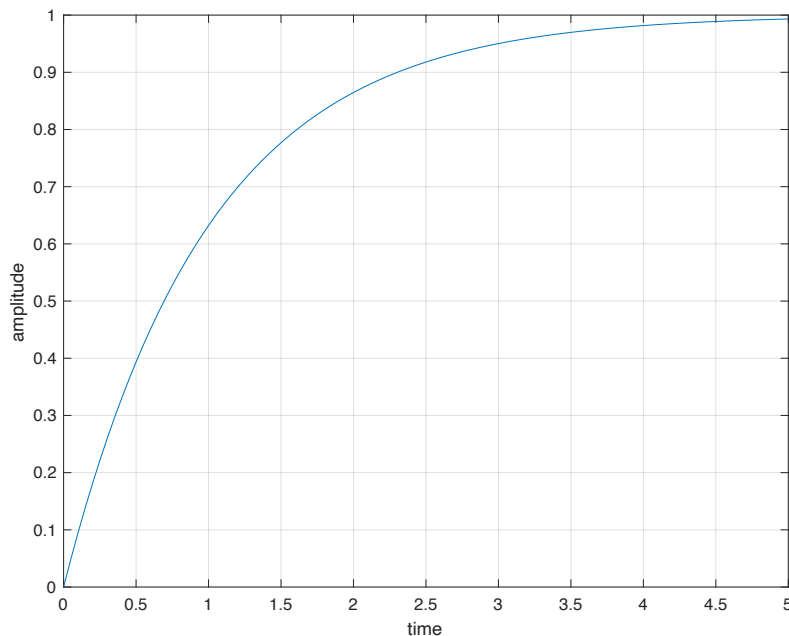
Exponential Attack

In a similar way, we can add an exponential attack to the tone amplitude
- An `attack_amp` value is initialized to 1.0 when a key is added to the key list (see `add_key()`, above)

However, there is one important difference:
- the output value is multiplied by $(1 - \texttt{attack\_amp})$

This is a plot of $(1 - \texttt{attack\_amp})$ for a time constant of 1.0 seconds:

However, a realist value for the attack time constant of e.g. a piano key is perhaps 10 ms, and a realistic decay time constant is 1.0 seconds.

The value for `attack_factor` corresponding to a time constant of 10 ms is given in the `synth.h` header file.

Finally, to apply both attack and decay time constants, simply multiply the output value by (1 – `attack_amp`) and (`decay_amp`). At the key onset, the attack will dominate the decay, but will quickly reach a value near 1.0. As the tone rings out, the decay will dominate. This will produce a tone with an envelope like this:
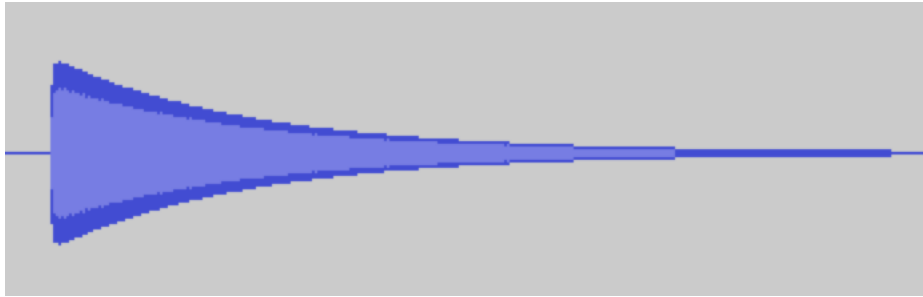

Figure – Note Waveform.

The exponential decay is clearly visible. The horizontal time scale for the plot is approximately 5 seconds. Notice that the tone drops to zero amplitude at one point. This is because the program stops the tone playout when the envelope falls below the `DROP_LEVEL`. Zooming in on the tone onset shows the attack, where the horizontal time scale is approximately 50 ms.
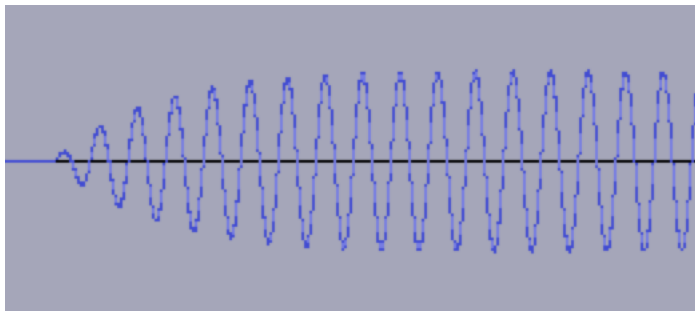
Attack and Decay Summary
So, you must add code that does the following:
- Inside the inner loop in `synth_block()`, for each tone: multiply the tone sample value by (1 – `attack_amp`) and (`decay_amp`), and then update the `attack_amp` and `decay_amp` factors.
- At the end of the inner loop, add code that stops the tone playout if the envelope of the tone is (as measured by decay_amp) below some `drop_level` (set to -60 dB in the `synth.h` header file). Stop the tone as follows:
```
if (pt[n].decay_amp < drop_level) {
    rm_key();
}
```

(10 Points) – Waveform for Concert A

In file main.h set DB_WAV_OUT to 1 (true) as

```
#define DB_WAV_OUT          1
```

and re-compile your program. This will cause all output from paCallback() to be written to file `test_file.wav.` Start the program and immediately hit the ';' key, which is Concert A or 440 Hz. After the note rings out, hit Space Bar to exit the program, which closes the WAV file. Use a DAW to edit the waveform so that it looks like the "Note Waveform" figure, above. Then revert back to

```
#define DB_WAV_OUT          0
```

and submit `test_file.wav` along with your code for `main.cpp` and `synth.cpp` files.