Problem Set 7: Synthesizer

Please send back to me via NYU Brightspace
- A zip archive named as
  `PS07_Last_First.zip`
  Where `First` and `Last` are your first and last name.
  The zip archive contains the C code files that implements all aspects of all problems.

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

References
The following URL provides a good reference for C language library function usage:
https://cplusplus.com/reference/clibrary/

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

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

Problem Overview
In this assignment you will create a simple 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);
```
After computing 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.c
synth.h
user_io.c
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:

```
synth.c
```

Code Overview

main.c

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

An instance to the Synth struct is defined at the top of main(), and a pointer to the instance:

Synth synth, *ps = &synth;

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

Next in main() is initialization of the values in the Synth struct. The definition of `init_synth()` is in synth.c.

/* initialize Synth */
init_synth(ps, NUM_CHAN, SAMP_RATE);

Next is that the pointer to the Synth struct 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 and waits for user key input. Ncurses prints:

```
Welcome to my synthesizer!
Keyboard to piano key mapping is:
  w e r   y u   o p      F3# to G4# (black keys)
  a s d f g h j k l ;     F3  to A4  (white keys)
  '.' to stop playing oldest tone
SpaceBar to quit

Key:
```

This simulates the following keys from the piano:

49	69	a′	A_4 A440	**440.0000**
48	68	g#′/ab′	$G\#_4/A\flat_4$	415.3047
47	67	g′	G_4	391.9954
46	66	f#′/gb′	$F\#_4/G\flat_4$	369.9944
45	65	f′	F_4	349.2282
44	64	e′	E_4	329.6276
43	63	d#′/eb′	$D\#_4/E\flat_4$	311.1270
42	62	d′	D_4	293.6648
41	61	c#′/db′	$C\#_4/D\flat_4$	277.1826
40	60	c′ 1-line octave	C_4 Middle C	261.6256
39	59	b	B_3	246.9417
38	58	a#/bb	$A\#_3/B\flat_3$	233.0819
37	57	a	A_3	**220.0000**
36	56	g#/ab	$G\#_3/A\flat_3$	207.6523
35	55	g	G_3	195.9977
34	54	f#/gb	$F\#_3/G\flat_3$	184.9972
33	53	f	F_3	174.6141

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

Synth.c
```
    void init_synth(Synth *ps, int samp_rate);
```
This function initializes the Synth and Tone structures.

```
    float key2freq(char c)
```
This function converts the key letter to the key frequency.

```
    void execute_cmd(Synth *ps, int cmd);
```
This function converts a command (cmd) to calls to the appropriate functions.

```
    void synth_block(Synth *ps, int len, float *output);
```
You will complete this function. It is described below.

```
void add_key(Synth *ps, int new_key, double new_freq);
```
This function adds key information to the array of Synth struct that are the keys that are being voiced by your synthesizer.

```
void rm_key(Synth *ps);
```
This function removes key information from the array of Synth struct that are the keys that are being voiced by your synthesizer.

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

```
void init_key(Synth *ps, int new_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 in files: synth.c. The specifics are described below.

synth.c
(20 Points) init_synth()
In function
```
void init_synth(Synth *ps, int in_num_chan, int
in_samp_rate)
```

Initialize the values in the local global array
        double key_freq[];
Set key_freq[0] = freq_f3; //local global variable
Set all higher indicies of key_freq[] to be SEMITONE_RATIO higher than the previous value. In other words, the ratio of key_freq[i-1]/key_freq[i] is the 12th root of 2.

Use the function input args to Initialize the num_chan, samp_rate values in the Synth structure and initialize num_keys to zero to indicate that no notes are playing.

Initialize the key value of each element in the `Tone` structure array to Space character (' '). This corresponds to "no note selected."
        ps->tone[i].key = ' '; //space character

(50 Points) synth_block()
30 Points – basic synthesizer
In function
```
Void synth_block(Synth *ps, int framesPerBuffer, float
*output)
```

Add code to synthesize the tones whose information is in the valid array elements of Synth. This is the main part of the assignment.

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

Simple synthesizer loop
- The synthesizer will have a loop that runs over `framesPerBuffer` frames, where a frame is the interleaved samples of all channels.
- 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.
- Compute the next tone sample value using the expression
       `v += FS_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, the its `phase` in incremented by `phase_inc`.

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

20 Points – Add Exponential Attack and Decay
However, the synthesizer has two negative features. The first is that once a key is pressed, the tone plays forever, or until a '.' key ("drop tone") is pressed 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 can 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:
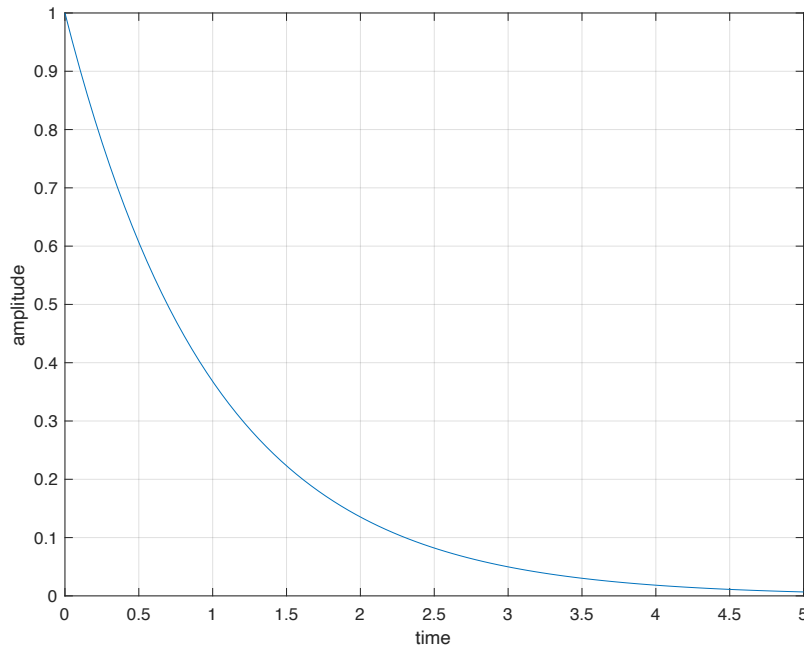
This can be expressed as a function of n, the digital sample index as

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

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

6

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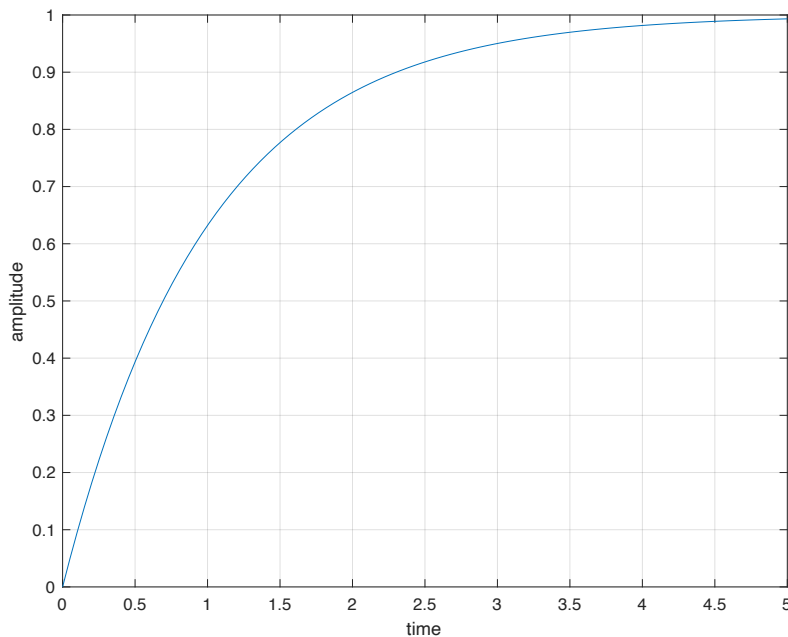
Exponential Attack
In a similar way, we can add an exponential attack to the tone amplitude
- An `attack_amp` value is initialized to 1.0 when a key is added to the key list (see `add_key()`, above)

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

This is a plot of $(1 - \text{attack\_amp})$ 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 $(1 - \text{attack\_amp})$ 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:
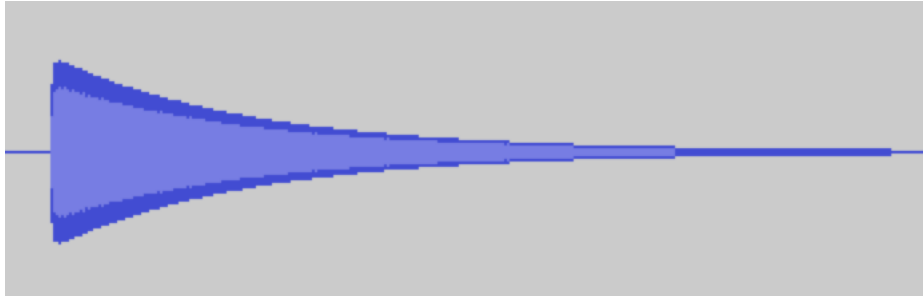
Figure – Note Waveform.

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.
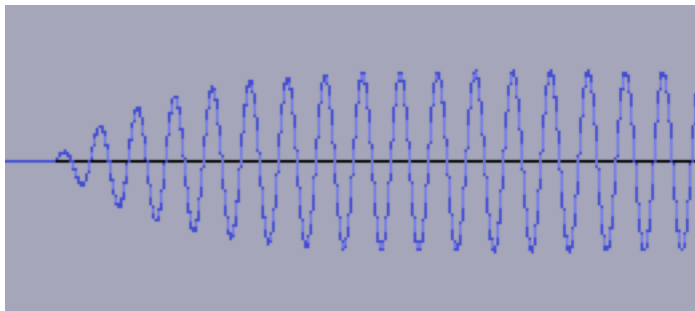
Attack and Decay Summary
So, you must add code that does the following:
- Inside the inner loop in `synth_block()`, for each tone: multiply the tone sample value by `(1 — attack_amp)` and `(decay_amp)`, and then update the `attack_amp` and `decay_amp` factors.
- At the end of the inner loop, add code 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(ps);
}
```

(10 Points) – Waveform for Concert A
Set DB_WAV_OUT to 1 (true) as

```
#define DB_WAV_OUT          1
```

in **main.c** 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 semicolon (;) 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 `synth.c`.