DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
IMPERIAL COLLEGE LONDON

EE3-19
REAL-TIME DIGITAL SIGNAL PROCESSING
LAB 2 REPORT

YASMIN BABA - 01196634
ZHI WEN SI - 01243716

Table of Contents

1. QUESTIONS

1.1 Provide a trace table of Sinegen for several loops of the code. How many samples does it have to generate to complete a whole cycle?

Using an IIR Filter method, the given ***sinegen*** function generates a sinewave by iteratively solving the following difference equation:

$$y[n] = \sqrt{2}y[n+1] - y[n+2] + \frac{1}{\sqrt{2}}x\,[n]$$

The input signal $x[n]$, which denotes the initial impulse needed to start the filter, is defined as,

$$x = \begin{cases} 1, & n = 0 \\ 0, & otherwise \end{cases}$$

The initial conditions for this filter are,

(1) $$y[0] = y[1] = y[2] = 0$$

Note, that after each iteration the following values are moved through the array:

$$y[2] = y[1] \text{ and } y[1] = y[0]$$

After calculating a few iterations, the following sample values were found and placed in the trace table below.

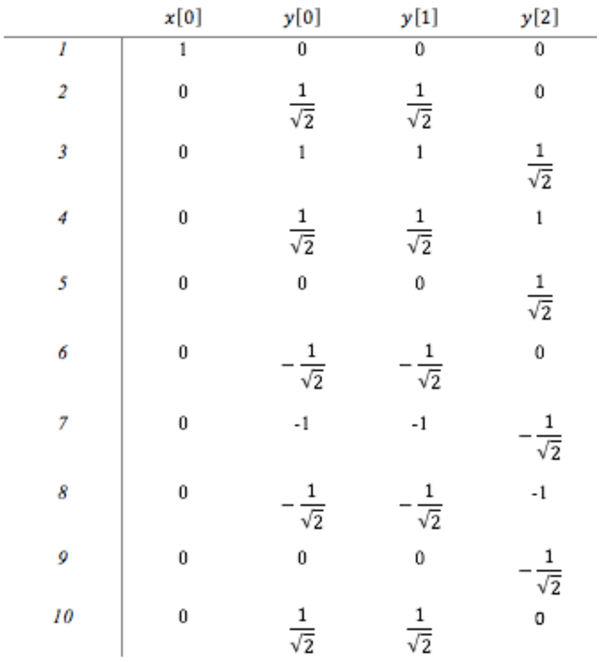
(2) $y[0] = \sqrt{2}(0) - 0 + \frac{1}{\sqrt{2}}(1) = \frac{1}{\sqrt{2}}$

(3) $y[0] = \sqrt{2}(\frac{1}{\sqrt{2}}) - 0 + \frac{1}{\sqrt{2}}(0) = 1$

(4) $y[0] = \sqrt{2}(1) - \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}(0) = \frac{1}{\sqrt{2}}$

(5) $y[0] = \sqrt{2}(\frac{1}{\sqrt{2}}) - 1 + \frac{1}{\sqrt{2}}(0) = 0$

etc.



	$x[0]$	$y[0]$	$y[1]$	$y[2]$
1	1	0	0	0
2	0	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$	0
3	0	1	1	$\frac{1}{\sqrt{2}}$
4	0	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$	1
5	0	0	0	$\frac{1}{\sqrt{2}}$
6	0	$-\frac{1}{\sqrt{2}}$	$-\frac{1}{\sqrt{2}}$	0
7	0	-1	-1	$-\frac{1}{\sqrt{2}}$
8	0	$-\frac{1}{\sqrt{2}}$	$-\frac{1}{\sqrt{2}}$	-1
9	0	0	0	$-\frac{1}{\sqrt{2}}$
10	0	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$	0

Table 1: Trace Table

The element $y[0]$ stores the sample value returned to the main. To produce a whole cycle, 8 samples are generated following the pattern:

$$0,\ \frac{1}{\sqrt{2}}, 1, \frac{1}{\sqrt{2}}, 0, -\frac{1}{\sqrt{2}}, -1, -\frac{1}{\sqrt{2}}, 0, \ldots$$

1.2 Can you see why the output of the sinewave is currently fixed at 1 kHz? Why does the program not output samples as fast as it can? What hardware throttles it to 1 kHz?

The sampling frequency is defined by the variable ***sampling_freq*** and has been set to 8kHz. As 8 samples per period are produced by the function ***sinegen***, the outputted sinewave will have a frequency of:

$$Signal\ frequency = \frac{8{,}000\ (samples\ per\ second)}{8\ (samples\ per\ cycle)} = 1{,}000Hz$$

Generating a sine wave using the IIR method limits the signal frequency to one-eighth of the sampling frequency. For example, when changing the sampling frequency to 16kHz, the signal frequency doubles to 2kHz, as expected.

The hardware that throttles the output is the DSK's DAC. When the sampling frequency is set to 8kHz, the DAC will only be able to output data every $125\mu s$. Furthermore, it is useful to note that the codec on this DSK is configurable only on sampling frequencies of 8kHz, 16kHz, 24kHz, 32kHz, 44.1kHz, 48kHz and 96kHz.

1.3 By reading through the code can you work out the number of bits used to encode each sample that is sent to the audio port?

By looking at the configuration settings of the audio port in *Listing 1*, the audio interface (7) is initialised such that 32 bits are used to encode each sample sent.

```
/* Audio port configuration settings: these values set registers in the AIC23 audio
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \
             /*********************************************************************/
             /*   REGISTER            FUNCTION                   SETTINGS         */
             /*********************************************************************/\
    0x0017,  /* 0 LEFTINVOL  Left line input channel volume  0dB                  */\
    0x0017,  /* 1 RIGHTINVOL Right line input channel volume 0dB                  */\
    0x01f9,  /* 2 LEFTHPVOL  Left channel headphone volume   0dB                  */\
    0x01f9,  /* 3 RIGHTHPVOL Right channel headphone volume  0dB                  */\
    0x0011,  /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
    0x0000,  /* 5 DIGPATH    Digital audio path control      All Filters off      */\
    0x0000,  /* 6 DPOWERDOWN Power down control              All Hardware on      */\
    0x004f,  /* 7 DIGIF      Digital audio interface format  32 bit               */\
    0x008d,  /* 8 SAMPLERATE Sample rate control             8 KHZ                */\
    0x0001   /* 9 DIGACT     Digital interface activation    On                   */\
             /*********************************************************************/
};
```

Listing 1: Hardware Initialisation Settings

Additionally, *Listing 2* shows code found in the main which is used to send samples to the audio port when it is ready to transmit. It indicates each sample is an *Int32*, i.e. 32 bits.

```
//  send to LEFT channel (poll until ready)
while (!DSK6713_AIC23_write(H_Codec, ((Int32)(sample * L_Gain))))
{};
// send same sample to RIGHT channel (poll until ready)
while (!DSK6713_AIC23_write(H_Codec, ((Int32)(sample * R_Gain))))
{};
```

Listing 2: Sending Samples to Audio Port

2. Explanation of Code

In order to generate a sinewave whose frequency does not vary with sampling frequency, the IIR method is disregarded and replaced with a sine look-up table. The look-up table stores evenly spaced data points, or samples, of one cycle of a sine wave which will be sent to the output. In this case, the look-up table is of size 256.

The following function **sine_init** initialises the look-up table by using a for loop, assigning each cell of the table with a different data point.

```
/******************************* sine_init() ************************************/
void sine_init(void)
{
    int i;
    for (i = 0; i < SINE_TABLE_SIZE; i++){
        table[i] = sin(2*PI*i/SINE_TABLE_SIZE);
    }
}
```

*Listing 3: **sine_init** Function*

To check the function is successful, a graphical representation of the data points was produced. Note how a very small number appears on the y-axis instead of an exact 0. This is believed to be caused by the symbol **PI**, which is used to calculate the samples stored in the look-up table, not being exact.

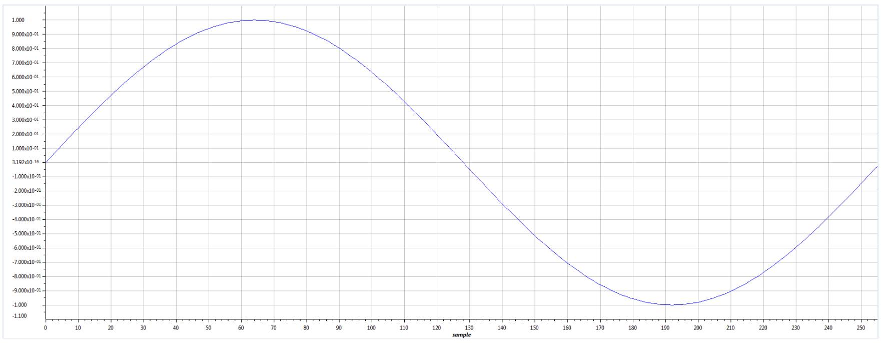


Figure 1: Graphical Representation of Look-Up Table

In order to retrieve these samples and send them to the output, the function **sinegen** was modified accordingly.

```
/********************************** sinegen() ***************************************/
float sinegen(void)
{
    float wave;
    float interval;

    interval = sine_freq/sampling_freq * SINE_TABLE_SIZE;
    wave = table[(int)round(n)];

    if(n < SINE_TABLE_SIZE){
    n = n + interval;
    }

    if(n >= SINE_TABLE_SIZE){
    n = n - SINE_TABLE_SIZE;
    }

    return (wave);
}
```

*Listing 4: **sinegen** Function*

The float variable ***interval*** is used as the index difference between the current sample and the next. The float ***n***, a global variable initially set to 0, represents the current index value itself. Finally, the float ***wave*** stores the sample value taken from the look-up table at the current index and is the variable sent back in to main to be outputted.

The variable ***n*** must be declared as a float in order to avoid calculation errors when later added with ***interval***. However, when passed as an index into ***table***, it is rounded to the nearest integer as the table is indexed by whole numbers.

Since a continuous sine wave is to be generated, and the fact that the look-up table consists of a finite 256 samples, a wrap-around effect is implemented via the if statements. If the index is below 256, the next index ***n*** is incremented by the value ***interval***. On the other hand, if the index is equal to or greater than 256, it will be subtracted by 256. It is not set back to zero in case the first index does not start at zero – using this method, a phase difference may be implemented if desired.

Figures 2,3 and *4* illustrate a fixed 1kHz signal frequency when sampling frequency was varied.

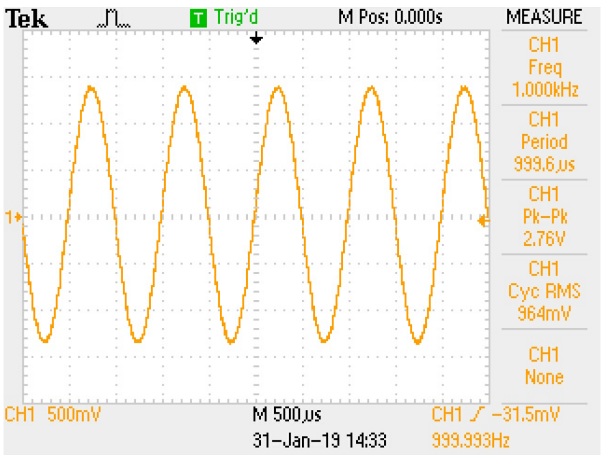


Figure 2: 8kHz Sampling Frequency

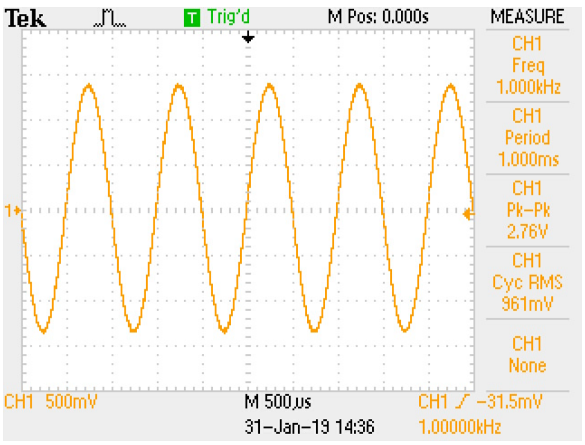


Figure 3: 32kHz Sampling Frequency

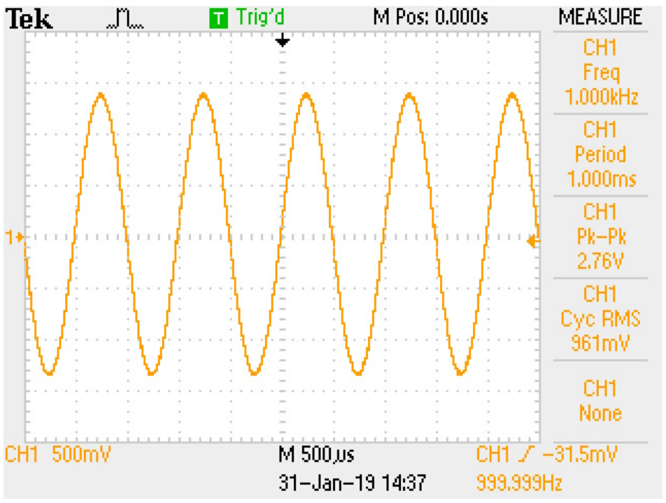


Figure 4: 96kHz Sampling Frequency

3. Increasing The Resolution

Digitised analogue signals, such as this generated sine wave, are approximated with a staircase structure. The smaller the ‘steps’, the higher the resolution. An obvious way to increase the resolution is to increase the size of the look-up table. This allows for more samples in one cycle and thus shrinks these ‘steps’. However, there are many ways in which one can increase the resolution of the output without increasing the table size, the simplest being increasing the sampling frequency. By increasing the sampling frequency, there are now more samples taken from the look-up table, and in other words, more ‘steps’. However, since the DSK is configurable only to sampling frequencies mentioned in 1.2, there is a restriction on varying the sampling frequency in this case.

Instead, one could take advantage of the symmetry of a sine wave. Instead of storing one cycle in 256 samples, one could store a quarter cycle and manipulate the samples accordingly per quarter cycle. For instance, after the first quarter cycle, the wrap-around effect occurs but this time, the samples outputted for the next quarter cycle will be retrieved starting at the bottom of the look-up table up to the top. This method increases the resolution as we now have more samples per one cycle (1024 samples to be exact), again shrinking the ‘steps’.

4. Frequency Limitations

The following observations have been observed using a sampling frequency of 8kHz. As the signal frequency is reduced to 10Hz, the voltage amplitude is observed to decrease as shown in *Figure 5*.

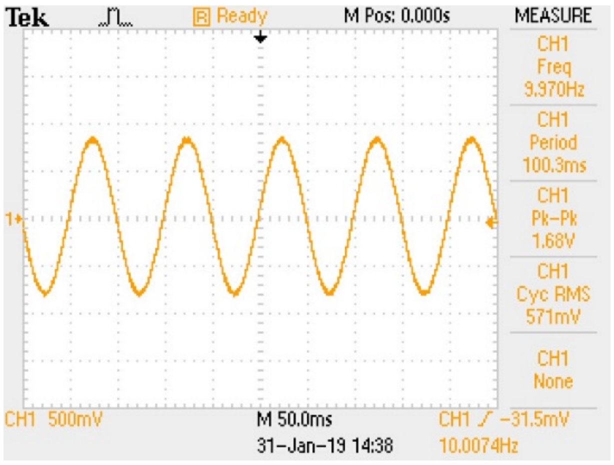


Figure 5: 10Hz Signal Frequency

This happens as there is a high pass filter at the output of the audio chip, as shown in *Figure 6,* which will attenuate low frequency signals. The corner frequency was calculated as,

$$f_c = \frac{1}{2\pi RC} = \frac{1}{2\pi(100 + 47K)(270n)} = 7.19Hz$$

Ideally, this high pass filter would cut off all frequencies below 7.19Hz. However, such a filter cannot be implemented in reality, hence attenuation beings to occur before the corner frequency.

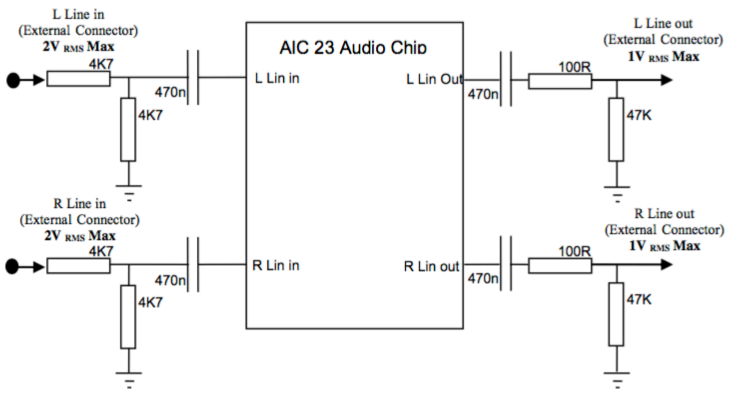


Figure 6: AIC 23 Audio Chip

The maximum signal frequency one can obtain is restricted to the Nyquist frequency ($0.5f_{sampling}$). As the Nyquist frequency was approached, the signal started to become more and more distorted. At Nyquist frequency, only two samples per cycle are produced. This reduces the resolution significantly, producing large 'steps' as shown in *Figure 7*.

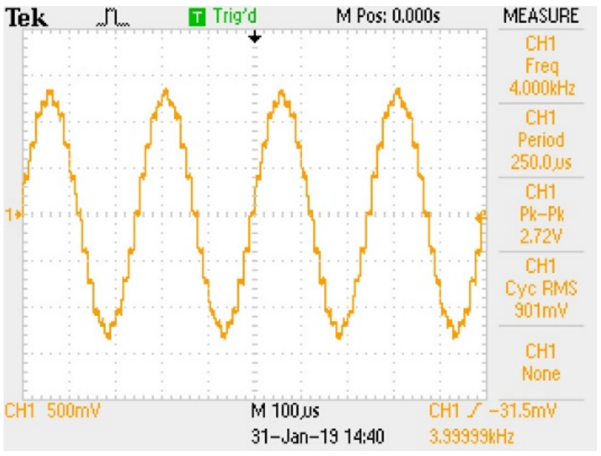


Figure 7: 4kHz Signal Frequency (Nyquist Frequency)

If the Nyquist frequency is exceeded, aliasing will occur, resulting in the wrong output signal frequency. For example, consider a 3kHz signal and a 5kHz signal both sampled at 8kHz. Their frequency spectra are identical hence the output of the DAC will be considered a 3kHz sine wave.

5. APPENDIX

```
/*************************************************************************************
               DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
                              IMPERIAL COLLEGE LONDON 

                      EE 3.19: Real Time Digital Signal Processing
                          Dr Paul Mitcheson and Daniel Harvey

                        LAB 2: Learning C and Sinewave Generation 

                             ********* S I N E . C **********

               Demonstrates outputting data from the DSK's audio port.
               Used for extending knowledge of C and using look up tables.

 ************************************************************************************
               Updated for use on 6713 DSK by Danny Harvey: May-Aug 06/Dec 07/Oct 09
               CCS V4 updates Sept 10
 ************************************************************************************/
/*
 *  Initialy this example uses the AIC23 codec module of the 6713 DSK Board Support
 *  Library to generate a 1KHz sine wave using a simple digital filter. 
 *     You should modify the code to generate a sine of variable frequency.
 */
/***************************** Pre-processor statements ******************************/

//  Included so program can make use of DSP/BIOS configuration tool.  
#include "dsp_bios_cfg.h"

/* The file dsk6713.h must be included in every program that uses the BSL.  This 
   example also includes dsk6713_aic23.h because it uses the 
   AIC23 codec module (audio interface). */
#include "dsk6713.h"
#include "dsk6713_aic23.h"

// math library (trig functions)
#include <math.h>

// Some functions to help with configuring hardware
#include "helper_functions_polling.h"


// PI defined here for use in your code 
#define PI 3.141592653589793

// size of look-up table for values of a sine wave
#define SINE_TABLE_SIZE 256



/******************************* Global declarations ********************************/

/* Audio port configuration settings: these values set registers in the AIC23 audio 
   interface to configure it. See TI doc SLWS106D 3-3 to 3-10 for more info. */
DSK6713_AIC23_Config Config = { \

               /**********************************************************************/
               /*   REGISTER            FUNCTION                      SETTINGS         */ 

               /**********************************************************************/\
    0x0017,    /* 0 LEFTINVOL  Left line input channel volume  0dB                   */\
    0x0017,    /* 1 RIGHTINVOL Right line input channel volume 0dB                   */\
    0x01f9,    /* 2 LEFTHPVOL  Left channel headphone volume   0dB                   */\
    0x01f9,    /* 3 RIGHTHPVOL Right channel headphone volume  0dB                   */\
    0x0011,    /* 4 ANAPATH    Analog audio path control       DAC on, Mic boost 20dB*/\
    0x0000,    /* 5 DIGPATH    Digital audio path control      All Filters off       */\
    0x0000,    /* 6 DPOWERDOWN Power down control              All Hardware on       */\
    0x004f,    /* 7 DIGIF      Digital audio interface format  32 bit                */\
    0x008d,    /* 8 SAMPLERATE Sample rate control             8 KHZ                 */\
    0x0001     /* 9 DIGACT     Digital interface activation    On                    */\

               /**********************************************************************/
```

```c
};


// Codec handle:- a variable used to identify audio interface
DSK6713_AIC23_CodecHandle H_Codec;

/* Sampling frequency in HZ. Must only be set to 8000, 16000, 24000
32000, 44100 (CD standard), 48000 or 96000  */
int sampling_freq = 8000;

// look-up table index
float n = 0;

// Array of data used by sinegen to generate sine. These are the initial values.
float y[3] = {0,0,0};
float x[1] = {1}; // impulse to start filter

float a0 = 1.4142; // coefficients for difference equation
float b0 = 0.707;

// Holds the value of the current sample
float sample;

/* Left and right audio channel gain values, calculated to be less than signed 32 bit
 maximum value. */
Int32 L_Gain = 2100000000;
Int32 R_Gain = 2100000000;


/* Use this variable in your code to set the frequency of your sine wave
   be careful that you do not set it above the current nyquist frequency! */
float sine_freq = 1000.0;

// create an array to store samples of sine wave
float table [SINE_TABLE_SIZE];

/******************************* Function prototypes ********************************/
void init_hardware(void);
float sinegen(void);
void sine_init(void);
/********************************** Main routine ************************************/
void main()
{

        // initialize board and the audio port
        init_hardware();

        // initialize look up table
        sine_init();

    // Loop endlessley generating a sine wave
   while(1)
    {
                // Calculate next sample
                sample = sinegen();

        /* Send a sample to the audio port if it is ready to transmit.
           Note: DSK6713_AIC23_write() returns false if the port if is not ready */

        //  send to LEFT channel (poll until ready)
        while (!DSK6713_AIC23_write(H_Codec, ((Int32)(sample * L_Gain))))
        {};
                // send same sample to RIGHT channel (poll until ready)
        while (!DSK6713_AIC23_write(H_Codec, ((Int32)(sample * R_Gain))))
        {};

                // Set the sampling frequency. This function updates the frequency only if it
                // has changed. Frequency set must be one of the supported sampling freq.
                set_samp_freq(&sampling_freq, Config, &H_Codec);

        }

}

/******************************* init_hardware() ************************************/
void init_hardware()
{
    // Initialize the board support library, must be called first
    DSK6713_init();

    // Start the codec using the settings defined above in config
    H_Codec = DSK6713_AIC23_openCodec(0, &Config);

        /* Defines number of bits in word used by MSBSP for communications with AIC23
         NOTE: this must match the bit resolution set in in the AIC23 */
        MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);

        /* Set the sampling frequency of the audio port. Must only be set to a supported
           frequency (8000/16000/24000/32000/44100/48000/96000) */

        DSK6713_AIC23_setFreq(H_Codec, get_sampling_handle(&sampling_freq));

}

/********************************** sinegen() ***************************************/
float sinegen(void)
{
    float wave;
    float interval;

        // index difference between each sample
        interval = sine_freq/sampling_freq * SINE_TABLE_SIZE;

        // index rounded to nearest integer
        wave = table[(int)round(n)];

        if(n < SINE_TABLE_SIZE){
                n = n + interval;
        }

        // wrap-around effect enabling index to be returned to the top of the table
        if(n >= SINE_TABLE_SIZE){
               n = n - SINE_TABLE_SIZE;
        }

    return (wave);
}

/****************************** sine_init() ************************************/
void sine_init(void)
{
        int i;

        // sine wave evenly split into SINE_TABLE_SIZE number of samples and stored into look-
up table
        for (i = 0; i < SINE_TABLE_SIZE; i++){
               table[i] = sin(2*PI*i/SINE_TABLE_SIZE);
        }
}
```

Listing 5: Full Code Listing