RTDSP Lab 2

Contents

1. Questions

Question 1

- Figure 1 provides a selection of the impulse response of the IIR filter found in sine.c. As we can see the outputs have a periodicity of 8 samples.

#	x	y
-2	0	0.000
-1	0	0.000
0	1	0.707
1	0	1.000
2	0	0.707
3	0	0.000
4	0	-0.707
5	0	-1.000
6	0	-0.707
7	0	0.000
8	0	0.707
9	0	1.000

Figure 1

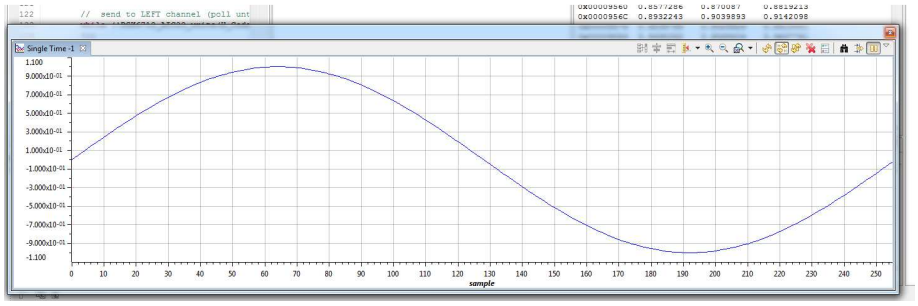
Question 2

- The sampling frequency as defined in sine.c (by sampling_freq) limits the frequency of the DAC, ie the output we see, to (in its original configuration) 8000 Hz. Since our sine wave has a periodicity of 8 samples we get the period of the generated sinewave to be $\frac{8000\ Hz}{8} = 1000\ Hz$. Changing the value of sampling_freq to 16000, as expected, doubles the period of the sinewave to 2kHz.
- The hardware throttling the output is the DSK's DAC

Question 3

```
//  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))))
{};
```

- From the lines above we see that the code encodes each sample to 32bits before sending it to the audio port.



2. Code Explanation

The first part of the code relied on writing a table to store the sine values. This was done in a for loop assigning a different value to each of the 256 different cells with the following formula: $cell[i] = \sin\left(2\pi \times \frac{i}{256}\right)$.

The written code included a simple table lookup increasing in appropriate steps, with the code resembling the following snippet:

```
wave = table[(int)index]; //this is the output value to be used
index += (SINE_TABLE_SIZE)/(sampling_freq/sine_freq); //increment
//index %= SINE_TABLE_SIZE while avoiding quantisation loss;
if (index > SINE_TABLE_SIZE)
{
	index -= SINE_TABLE_SIZE;
}
```

The code works by having a global float variable called index. By having the variable index be a float rather than an integer we avoid phase distortion by skipping relevant values. To illustrate this concept consider this table:

sampling frequency	sine frequency	increment (float)	increment (int)
8000	800	25.6	26

Iteration	Float		Integer		
	Index	Cell Accessed	Index	Cell Accessed	Error
0	0	0	0	0	0
1	25.6	26	26	26	0
2	51.2	51	52	52	1
3	76.8	77	78	78	1
4	102.4	102	104	104	2
5	128.0	128	130	130	2
6	153.6	154	156	156	2
7	179.2	179	182	182	3
8	204.8	205	208	208	3
9	230.4	230	234	234	4
10	0.0	0	4	4	4
11	25.6	26	30	30	4
12	51.2	51	56	56	5
13	76.8	77	82	82	5
14	102.4	102	108	108	6
15	128.0	128	134	134	6
16	153.6	154	160	160	6
17	179.2	179	186	186	7
18	204.8	205	212	212	7
19	230.4	230	238	238	8
20	0.0	0	8	8	8
21	25.6	26	34	34	8
22	51.2	51	60	60	9
23	76.8	77	86	86	9
24	102.4	102	112	112	10
25	128.0	128	138	138	10
26	153.6	154	164	164	10
27	179.2	179	190	190	11
28	204.8	205	216	216	11
29	230.4	230	242	242	12
30	0.0	0	12	12	12

Table 1. This table looks at how each cell of a 256 sample sine table is accessed using 2 different methods when generating an 800Hz sinewave.

Finally to avoid smaller quantisation error from arising each time the index loops back (as it is constrained to the sine table size, in this case 256) the index is not reset to 0 but rather has 256 subtracted from it, meaning any remaining fractional part will be kept, avoid errors in long runtimes.

Thus this solution of indexing using floats results in a short and efficient code, meaning that it should work in real-time even on a slower DSK clock rate.

3. Increasing Resolution

If we define increasing resolution as the ways in which we can reduce the "stepiness" of the output (i.e. increase the number of possible discrete levels) or make it appear smoother (i.e. we are not increasing the number of discrete levels but rather ensuring that the step size is consistent) we can use 3 methods to increase the resolution of the output signal.

The first method would be to increase the sampling frequency. Doing so would mean more samples per second which increase the resolution by creating additional, smaller steps in the output signal. This is only true for output frequencies larger than $\frac{sampling\ frequency}{\#\ of\ samples}[Hz]$. For example with a sampling frequency of 8000 Hz and 256 samples for 1 sine wave we would get a frequency of $\frac{8000\ [Hz]}{256\ samples} = 31.25[Hz]$. Thus any output frequency below this would have duplicate outputs, for example the 2nd element of the table array may be accessed twice.

We notice that increasing the number of samples would decrease the "minimum good resolution" frequency. However this is not an option but we can instead choose what to store inside our 256 samples. For example, instead of storing a full sine wave we can simply store a quarter of a sine wave in the 256 available samples. With appropriate interpolation and inverting such a system would act as if it had 1020 samples. ($1020 = (256 - 1) \times 4$, we are discarding one sample for each wave as we would calculate the values for quarter sine wave from 0 to 1 and either 0 or 1 will be discarded on each iteration.) At higher frequencies this quarter sinewave method would not strictly increase resolution. It will instead make the outputted values possibly more "correct" but it will not help increase the number of steps, which is mainly limited by the sampling frequency. Indeed while there are more samples to choose from there is still the same number of samples outputted per second. Thus implementing a quarter sinewave method would bring little advantage with respect to resolution increase (when generating above $\frac{sampling\ frequency}{\#\ of\ samples}[Hz]$) while bringing a larger processing overheard (though in this case this is not very important due to the amount of processing power available to us).

Table 2 illustrates why a larger sample table would not help very much by looking at how samples would be generated as past 256 samples for a generated frequency of 1300 Hz we noticed very little difference in the final output samples.

Output 1020 samples	Output 512 samples	Output 256 samples	Output 64 samples	Output 16 samples	Output 8 samples
0.000	0.000	0.000	0.000	0.000	0.000
0.850	0.851	0.845	0.831	0.707	0.707
0.892	0.893	0.893	0.924	0.924	1.000
0.080	0.086	0.098	0.098	0.383	0.707
-0.809	-0.803	-0.803	-0.773	-0.707	-0.707
-0.926	-0.924	-0.924	-0.924	-0.924	-1.000
-0.159	-0.159	-0.171	-0.195	-0.383	-0.707
0.759	0.757	0.757	0.707	0.707	0.707
0.951	0.953	0.957	0.957	1.000	1.000
0.238	0.243	0.243	0.290	0.383	0.707
-0.705	-0.707	-0.707	-0.707	-0.383	-0.707
-0.973	-0.973	-0.976	-0.981	-1.000	-1.000
-0.309	-0.314	-0.314	-0.383	-0.383	-0.707
0.646	0.644	0.634	0.634	0.383	0.000
0.988	0.989	0.989	0.995	1.000	1.000
0.384	0.383	0.383	0.383	0.707	0.707
-0.588	-0.586	-0.576	-0.556	-0.383	0.000
-0.997	-0.997	-0.997	-1.000	-1.000	-1.000

Table 3. This table shows the output of a 1300Hz signal with sampling frequency of 8000Hz with a variety of sample table sizes.

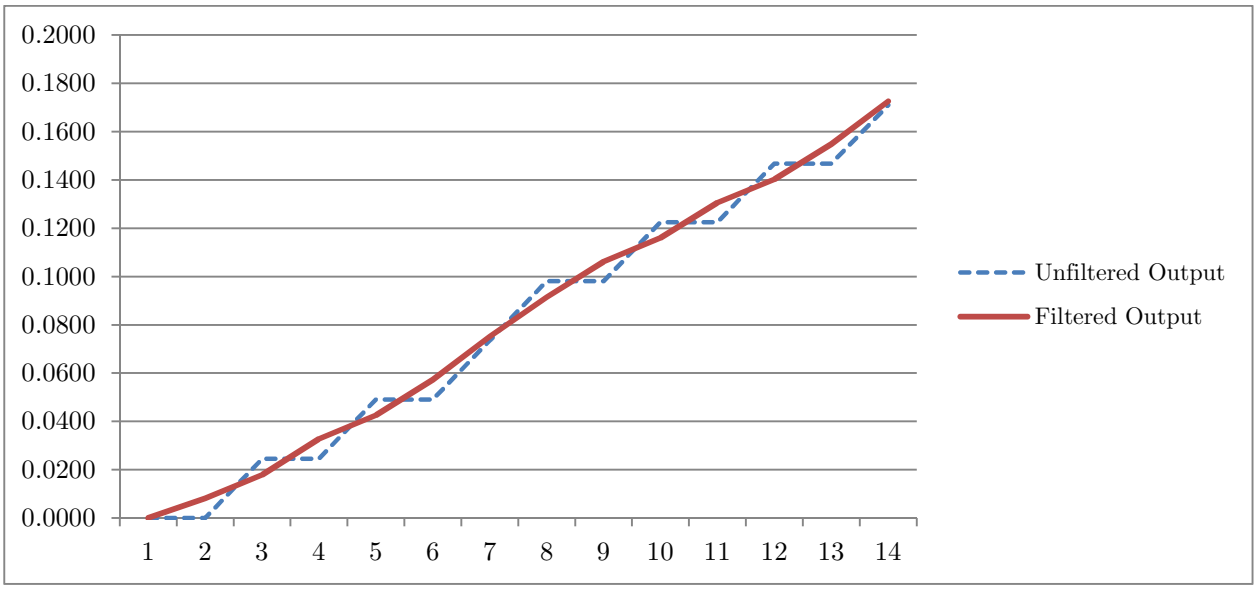
Finally we notice that when we are skipping samples (which happens when the output frequency is above $\frac{sampling\ frequency}{\#\ of\ samples}[Hz]$) the step size may not be consistent, i.e. the increase from one sample to the other Is not the same. When we are repeating samples (which happens when the output frequency is below $\frac{sampling\ frequency}{\#\ of\ samples}[Hz]$) we are essentially outputting less discrete levels than the sampling frequency allows. To solve this we could pass the output through a moving average filter.

For example let us look at the following scenario.

Sampling Frequency	Sine Frequency	Increment
8000	18	0.576

Index	Cell Accessed	Output	Filtered Output
0	0	0.0000	0.0000
0.576	0	0.0000	0.0082
1.152	1	0.0245	0.0164
1.728	1	0.0245	0.0327
2.304	2	0.0491	0.0409
2.88	2	0.0491	0.0572
3.456	3	0.0736	0.0735
4.032	4	0.0980	0.0899
4.608	4	0.0980	0.1061
5.184	5	0.1224	0.1143
5.76	5	0.1224	0.1305
6.336	6	0.1467	0.1386
6.912	6	0.1467	0.1548
7.488	7	0.1710	0.1709

Table 3. This table looks at the scenario where the output frequency is under $\frac{sampling\ frequency}{\#\ of\ samples}[Hz]$ (18Hz in this case) and the output is passed through a 3 point moving average with equation $y[n] = \frac{0.8 \times x[n-1] + 1.2 \times x[n] + x[n+1]}{3}$. (*Note*: Coefficients were chosen arbitrarily and other coefficients could have been chosen.)



Graph 1. This graph shows the filtered and unfiltered outputs.

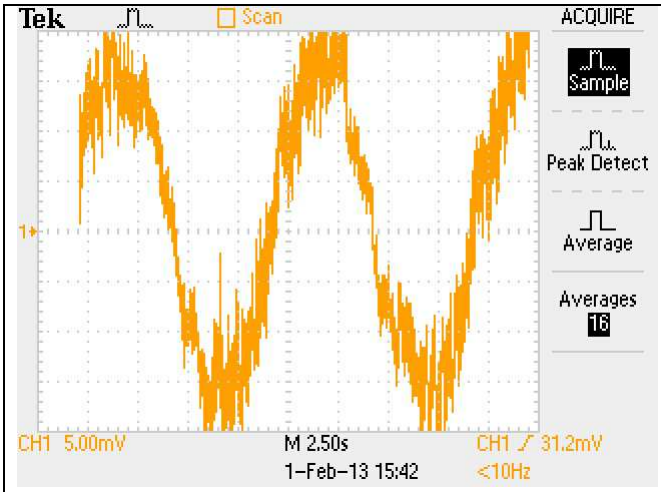
Table 3 and Graph 1 show how a moving average filter at the output creates an output with more discrete steps, thus increasing the resolution and creating a smoother output signal.

However in practice the moving average solution does not work. Indeed steps are very difficult to notice at frequencies supposedly concerned (see Graph 2 where the steps are only really noticeable if inferred from the theoretical values). The other problem with the moving average filter would be that at higher frequencies the signal would be "slowed down" as a moving average is equivalent to a low pass filter. An easy solution would be to only activate the filter when generating below $\frac{sampling\ frequency}{\#\ of\ samples}[Hz]$.

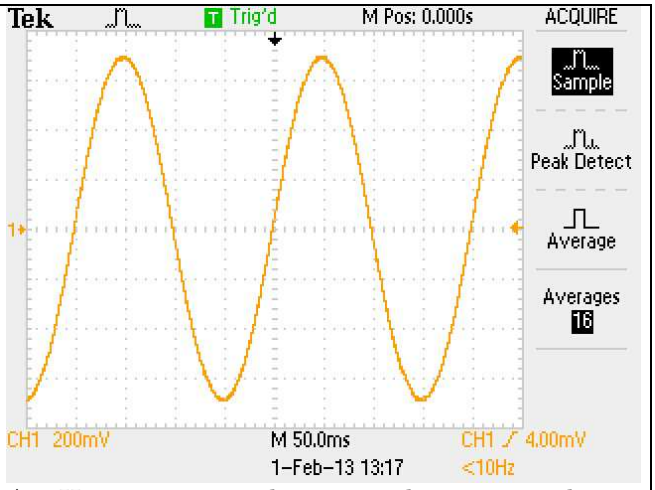
Thus we conclude that while there are different ways to improve resolution, at least theoretically, the two proposed ones, a quarter sinewave and a moving average filter, do not show any noticeable improvement at normal operating frequencies (i.e. a filter has been observed to possibly help at 0.1Hz by making the noise appear less obvious however the signal is so corrupted by the high pass filter at 0.1Hz that the effect is barely noticeable). Thus quantisation error is not the main limiting factor of resolution but rather sampling frequency.

4. Scope Traces

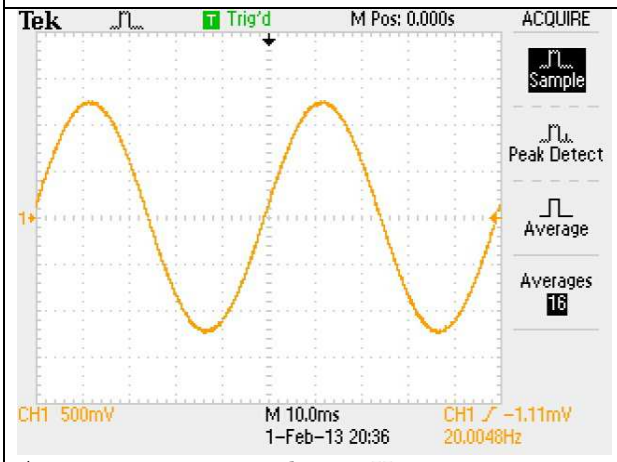
The following set of scope traces was taken at a sampling frequency of 8000Hz and the output sine waves range from 0.1Hz to 4 kHz (the Nyquist frequency).



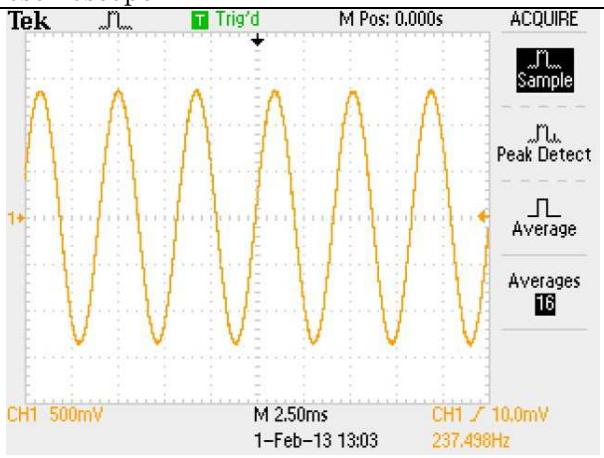
This is a 10 second period sinewave. While it is very noisy we notice that the code still works at very low frequencies.



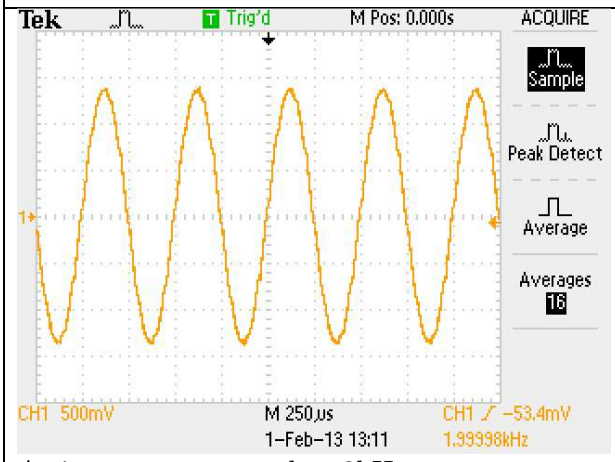
A 5Hz sinewave. This time there is nearly no noise and it is difficult to see the quantisation steps, even by increasing the scale on the oscilloscope.



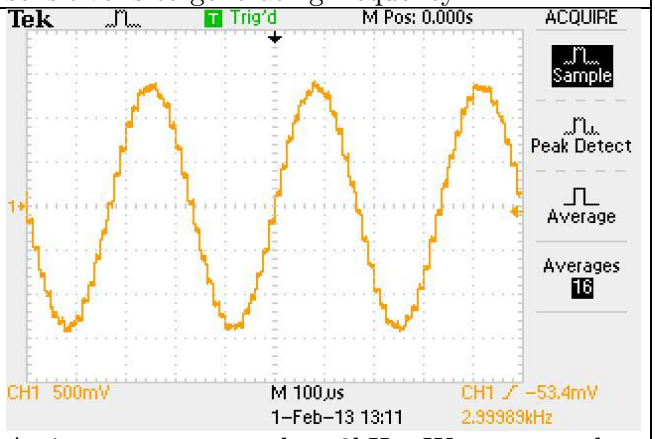
A sinewave generated at 20Hz.



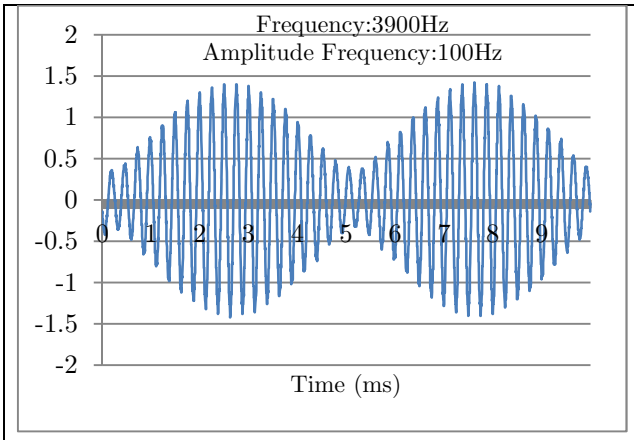
A sinewave generated at 237.5Hz to see how sensitive is to generating frequency.



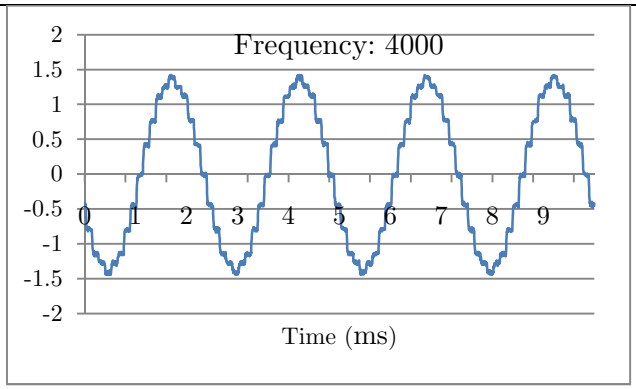
A sinewave generated at 2kHz.



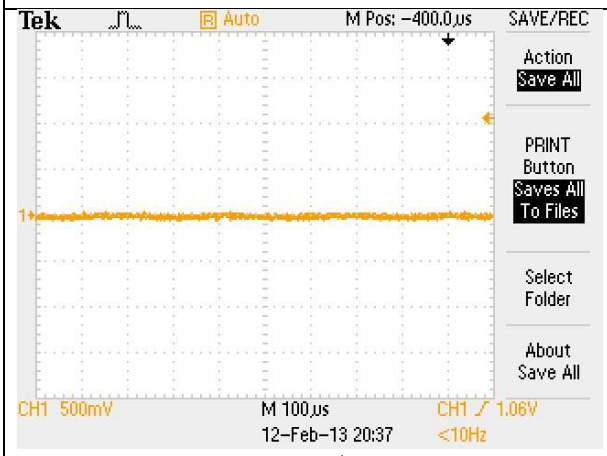
A sinewave generated at 3kHz. We can see that the generated sinewave starts to exhibit signs of "stepiness"



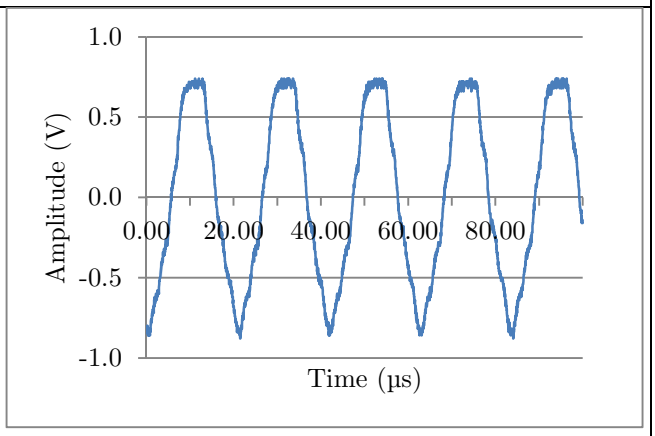
Here the sinewave generated at 3900Hz shows sign of amplitude modulation with a frequency of 100Hz.



At 4000Hz we notice steps in the output. These are not the steps due to table value access but rather seem to be related with the output filter or interpolator as they occur at a frequency around 9600Hz.



With the correct offset (more explanation as to why later) we get non-existent signal for 4000Hz



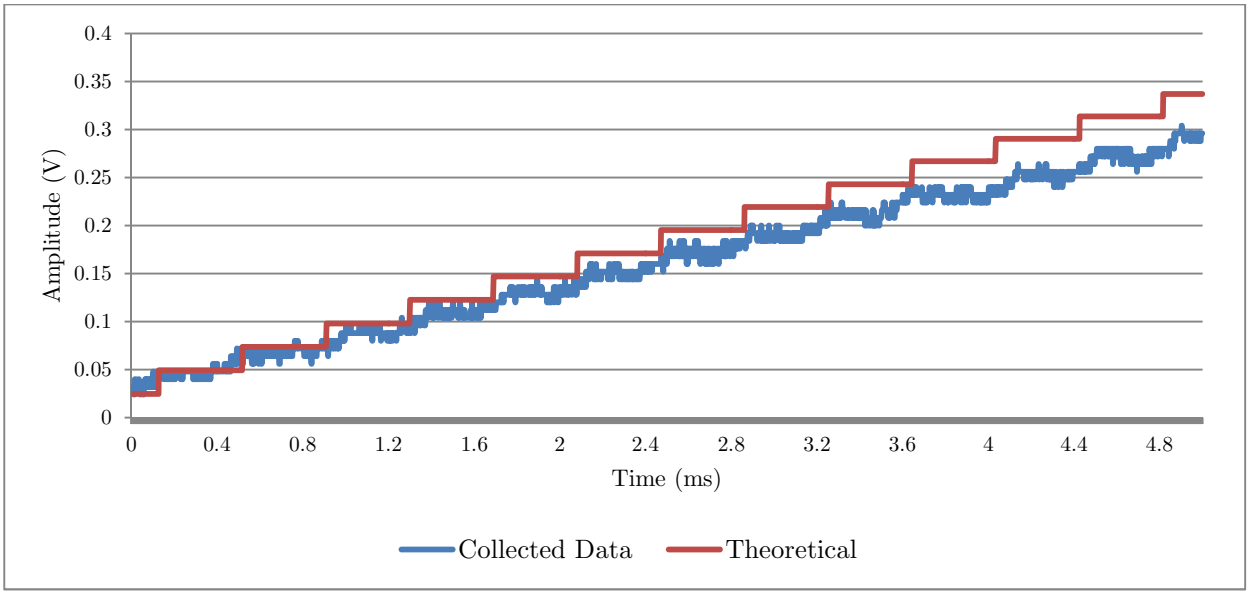
At 48000Hz, with a 96000Hz sampling frequency, i.e. at Nyquist frequency, we observe that the wave breaks down into what looks like a cutoff triangle wave.

5. Discussion on performance

a. Lower bound

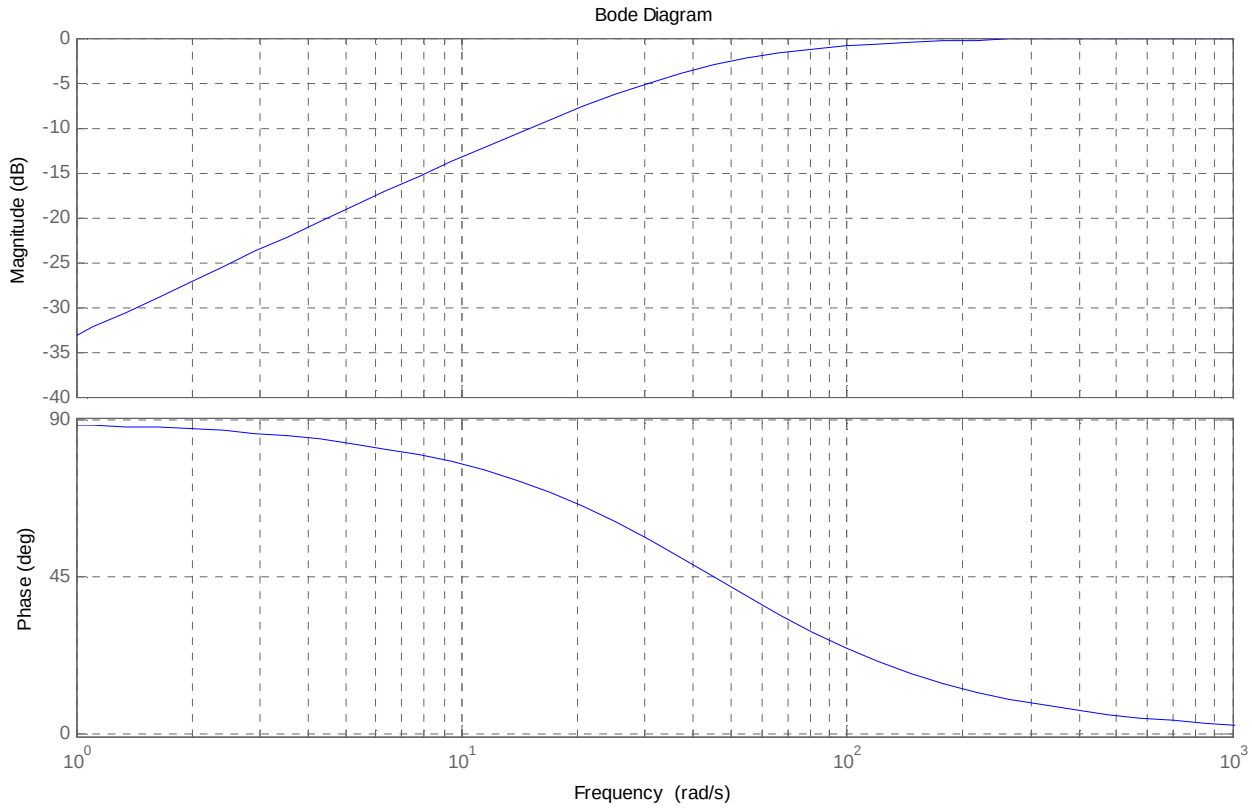
We can define lower bound in two ways, frequency below $\frac{sampling\ frequency}{\#\ of\ samples}[Hz]$ or simply below the 10Hz range.

Below $\frac{sampling\ frequency}{\#\ of\ samples}[Hz]$, which for 8000 Hz and 256 samples is 31.25Hz, we can notice steps in the signal, as Graph 2 illustrates. These small but noticeable steps are due to the limited number of samples available at the time of generation. As suggested earlier a moving average filter, for example, would help avoid this issue. However the signal seems to be so much corrupted by noise that it is unclear if the moving average solution would actually help.



Graph 2. This data shows collected data against theoretical data for a 10 Hz signal with a sampling frequency of 8000Hz and a table of 256 samples.

Starting from frequencies of about 10Hz and under we notice the signal amplitude decreasing. This is due to the low pass filter at the audio output and this can be understood in more detail from Graph 3 which shows how around under 20Hz magnitude loss becomes significant. The filter's main aim is to remove DC bias and achieves this while also compromising low frequency performance.



Graph 3. Bode plot of the output high-pass filter, using values provided in the datasheet.

b. Upper bound

Upper bound is defined as being near or at Nyquist frequency.

Multiple effects can be observed at high frequencies, also defined as upper bound operation. The first is amplitude modulation which occurs just around Nyquist range. The second effect is the change in amplitude at Nyquist frequency due to index offset. The third effect is related to how discrete steps in the output become more obvious. Finally we can also comment on the mode of operation beyond Nyquist.

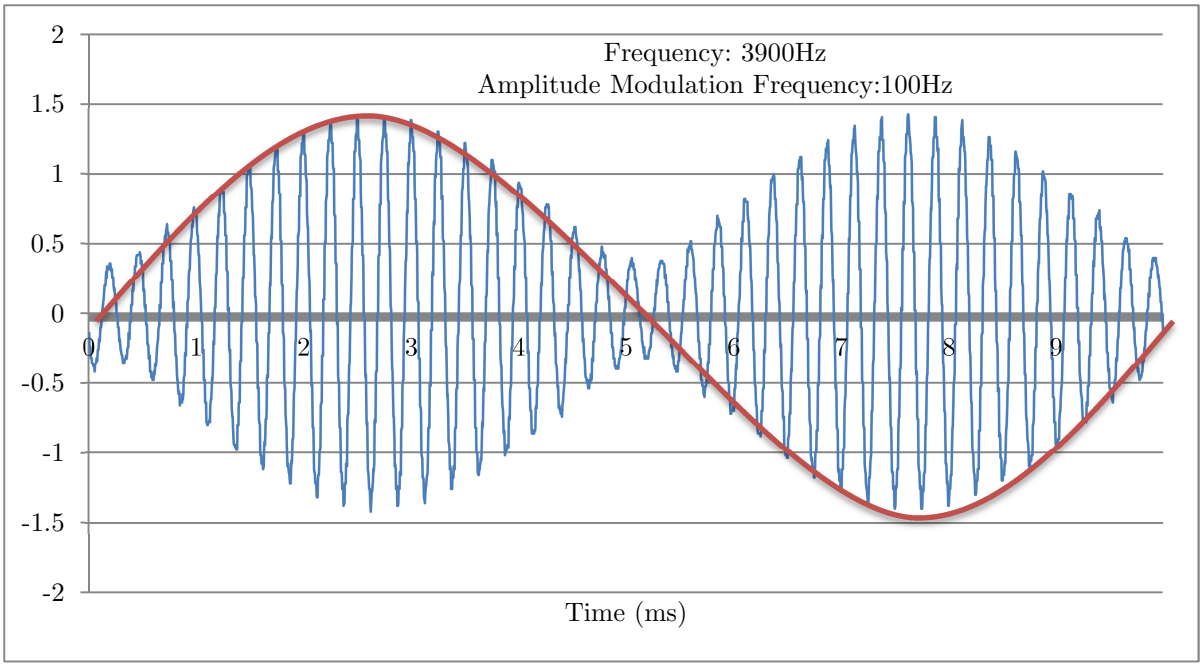
i. Amplitude Modulation

This effect becomes noticeable near Nyquist frequency and Graph 4 provides a visual example of this effect. The reason for this happening is that the discrete signal gets reflect in frequency domain at multiple locations. In the case of 3900Hz sampled at 8000hz this gets reflected to 4100Hz. Thus the output we get is:

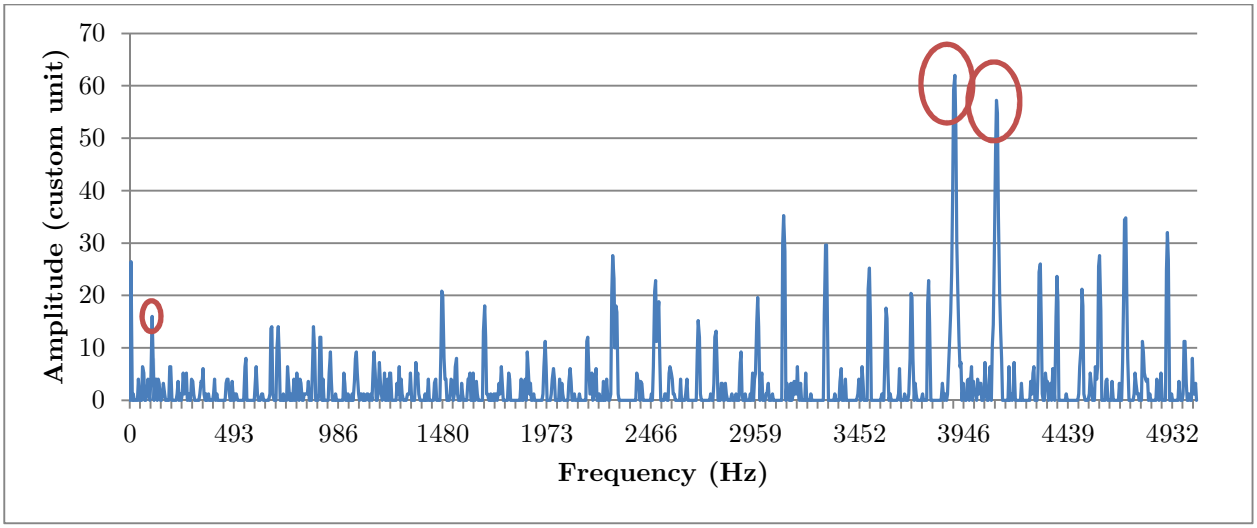
$$\cos(3900 \times 2\pi \times t) + \cos(4100 \times 2\pi \times t) = 2\cos(4000 \times 2\pi \times t)\cos(100 \times 2\,\pi \times t)$$

Which we see is composed of 2 frequencies and that the envelope observed in Graph 4 is 100Hz. In reality the output is slightly different as the amplitude of both waves is different, i.e. the equation resembles something more of the type $A\cos(2\pi ft) + B\cos\left(2\pi\left(f_{samp} - f\right)t\right)$ where $A > B$ near Nyquist frequency and as the generated frequency deviates from the Nyquist frequency we get $A \gg B$. This is as a result of the input filter which removes high frequency components but cannot reliably do so around Nyquist.

Graph 6 shows a FFT of a sinewave at 3900Hz and the various other frequency components.



Graph 4. Graph of collected data for a sinewave generated at 3900Hz at a sampling frequency of 8000Hz. The red line represents the envelope of the amplitude.

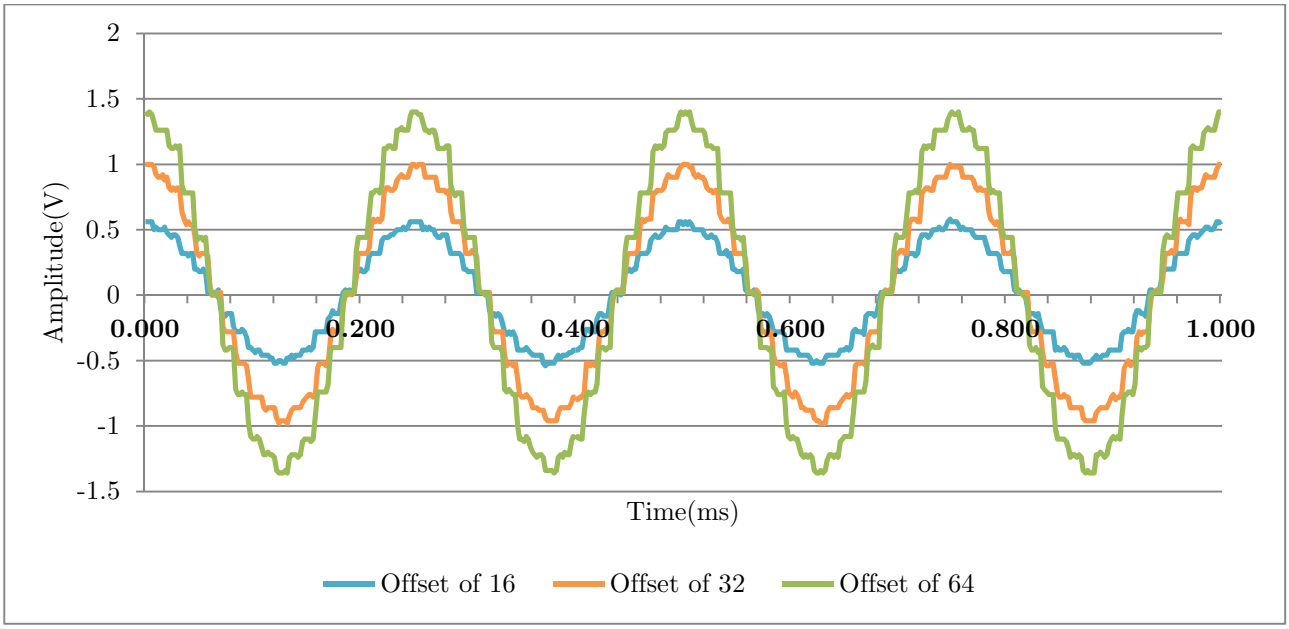


Graph 5. Graph of collected data for a FFT of a sinewave generated at 3900Hz at a sampling frequency of 8000Hz. The red bubbles circle the frequencies of interest, 100Hz, 3900Hz and 4100Hz.

ii. Amplitude change at Nyquist

At Nyquist frequency we notice the amplitude is seemingly random whenever we move to it and sometimes the amplitude can be zero, as shown earlier in scope traces. The reason for this is due to the indexing.

In the case of having 256 samples we would get an interval of $\frac{256}{2} = 128\ samples$. In this case if the first cell accessed was 0 the next cell would be 128 with the one after being 0 (due to looping back) and so on. What we can observe is that both cell 0 and 128 contain 0 ($\sin\left(2\pi \times \frac{0}{256}\right) = \sin\left(2\pi \times \frac{128}{256}\right) = 0$), implying that the output has to be 0 at Nyquist. This conclusion is not entirely correct as it relies on the starting value to be 0, this being the key to understanding this problem. Indeed with a starting value of 64 (i.e. $\frac{256}{4}$) gives us two values of 1 and -1. Thus the discrete output values can range from 0 to 1, which Graph 6 illustrates.



Graph 6. Examples of amplitude variation at Nyquist frequency from different offsets. Data collected for a signal at 4000Hz generated with a sampling frequency of 8000Hz.

iii. Output steps

We also notice that the output is a well formed sinewave despite only 2 values existing as possible outputs. This is due to the interpolator in the DSK which joins up two points in a sinusoid manner, this also explains why the steps are about 96 kHz as this is the maximum rate of the DSK interpolator (we can calculate this by knowing that the average step level time is 5.6 samples long on average with each sample being separated by about 0.00198ms according to the oscilloscope logs, thus we have $\frac{1}{5.6\times0.00198ms} = 90.19kHz \approx 96Khz$).

iv. Slew rate

We notice that at 96 kHz sampling frequency very high frequencies do not always display perfect sinewave. The reason for this is the filter of the DSK which imposes a slew rate by having a capacitor. This slew rate is calculated to be around $0.190\ \frac{V}{\mu s}$ which is a very reasonable slew rate (considering the slew rate for a high voltage linear ramp is typically $0.1\frac{V}{\mu s}$.

v. Beyond Nyquist

Beyond Nyquist frequency (i.e. half the sampling rate) the generated frequency is $f_{samp} - f$ but with a $\pi\ [rad]$ phase shift. This is due to the cells effectively being accessed in reverse order.

6. Full code with comments

```
/*************************************************************************
                        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 outputing 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 3.14159265358979323846264338327950288419712//3.141592653589793
#define SINE_TABLE_SIZE 256//256//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                  */\
                        /*****************************************************************/
};

// 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;

// 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 carefull that you do not set it above the current nyquist frequency! */
float sine_freq = 1000.0;

//define sinetable
float table[SINE_TABLE_SIZE];

//index, a global float variable
float index = 0;

/****************************** 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 sine 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));

}
/********************************* sine_init() **********************************/
void sine_init(void)
{
        int i;//index variable
        //loop to populate each value such that the table contains a full sinewave
        for (i = 0; i < SINE_TABLE_SIZE; i++)
        {
                table[i] = sin(2*i*PI/(SINE_TABLE_SIZE));
        }
        return;
}
/********************************* sinegen() ************************************/
float sinegen(void)
{
        //index is a global float variable thus it does not suffer from quantisation error
        //step increase is calculate logically by knowing that we have a SINE_TABLE_SIZE
```

```
        //and if we want to output a frequency that is (sampling_freq/sine_freq) times the
        //sampling_freq we logically divide SINE_TABLE_SIZE by the ratio (sampling_freq/sine_freq)
        index += (SINE_TABLE_SIZE)/(sampling_freq/sine_freq);
        //In the following case we are attempting to effectively do the following operation:
        //index %= SINE_TABLE_SIZE
        //however this is not possible with a float therefore we simply
        //substract the SINE_TABLE_SIZE when the index is over this value.
        if (index >= SINE_TABLE_SIZE)
        {
                index -= SINE_TABLE_SIZE;
        }
        //finally as an array can only be accessed by an integer we typecast it as an
        //integer to access the correct index
    return(table[(int)index]);

}
```