EE 113D: Digital Signal Processing Design

Lab #1
Waveform Generation and Measurement

Names: Erik Hodges & Ivan Manan

Due Date: October 10, 2018

Step 1

The first part of the lab involved generating a 250 Hz sine wave. The sampling frequency was taken at 8 kHz. A look-up table was provided with 32 discrete points that described one cycle of a 250 Hz sine wave.

Figures 1.1 and 1.2 displays the screenshots of the oscilloscope. These screenshots demonstrate the before and after effects of doubling the output amplitude (*i*changed the gain from 20 to 40).

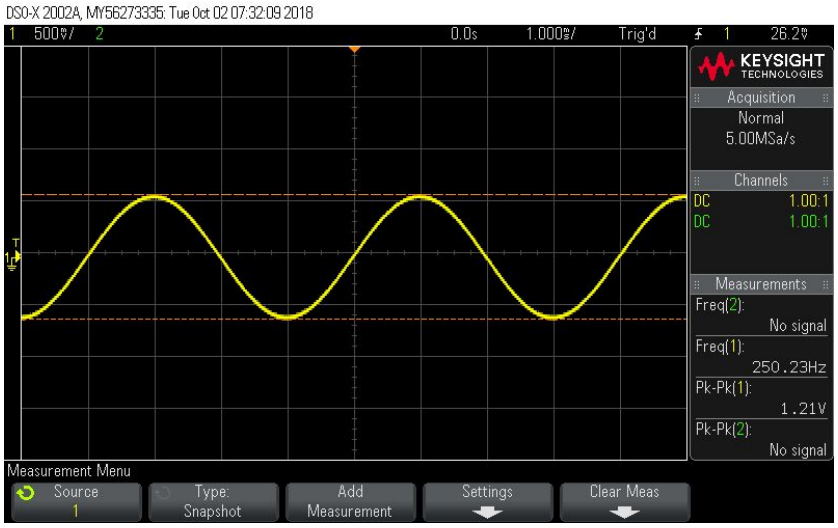

Figure 1.1 Original Sine Wave with a Gain of 20

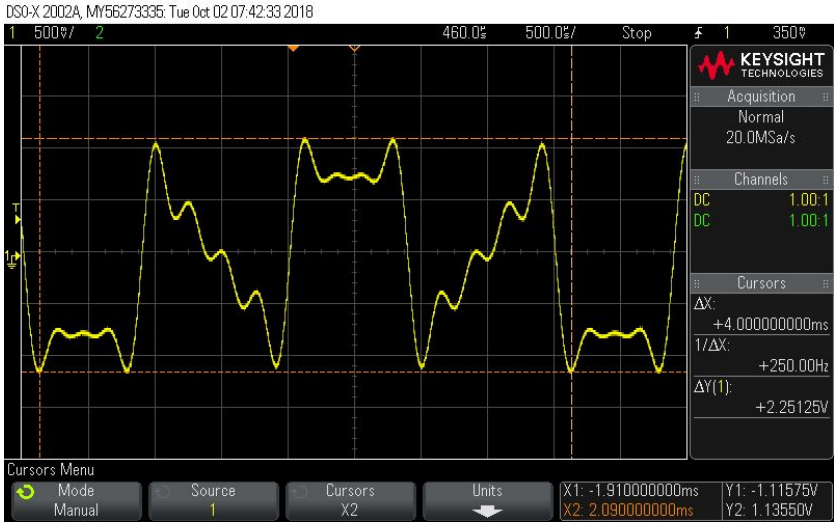

Figure 1.2: Altered Sine Wave with a Gain of 40

Notice the disparity between the two images in Figure 1. When the original sine wave was altered to have a gain of 40, the plot became distorted. This was because the LCDK is a 32-bit machine. Since the gain is 40, the sine table was scaled to a higher amplitude. However, some values in the sine table multiplied by a gain of 40 exceed the value of 32,678. As a result, integer overflow occurs, subsequently distorting the 250 Hz sine wave to be unpredictable.

The code fragments responsible for the higher-amplitude sinusoid are highlighted in yellow.

```
#include "L138_LCDK_aic3106_init.h"
#include "evmomapl138_gpio.h"

#define LOOPLENGTH 32

int16_t sine_ptr = 0;
int16_t sine_table[LOOPLENGTH]={0, 195, 383, 556, 707, 831, 924, 981,
                          1000, 981, 924, 831, 707, 556, 383, 195, 0,
                          -195, -383, -556, -707, -831, -924, -981,
                          -1000, -981, -924, -831, -707, -556, -383, -195};

int16_t gain = 40;

interrupt void interrupt4(void) // interrupt service routine
{
      int16_t left_sample;

      left_sample = (sine_table[sine_ptr]*gain);
      sine_ptr = sine_ptr + 1;
      sine_ptr = sine_ptr % LOOPLENGTH;

      output_left_sample(left_sample);

      return;
}
int main(void)
{
L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB,LCDK_LINE_INPUT);
while (1);
}
```

Step 2

In this step we will control the frequency of the generated sine wave using the following relationship:

$$\frac{Sample\ Rate\ (samples/s)}{Lookup\ Table\ Parameter\ (samples/cycle)} = Output\ Frequency\ (cycles/s)$$

To produce a 1000 Hz sine wave, the sampling rate was increased by a factor of 4 from the rate of the 250 Hz sine wave. This produced 4 cycles in the same amount of time that 1 cycle took previously, thereby increasing the frequency by a factor of 4 and giving us a 1000 Hz sine wave.

250 Hz Sine Wave → 8000 (samples/s) / 32 (samples/cycle) = 250 (cycles/s)
1000 Hz Sine Wave → 32000 (samples/s) / 32 (samples/cycle) = 1000 (cycles/s)

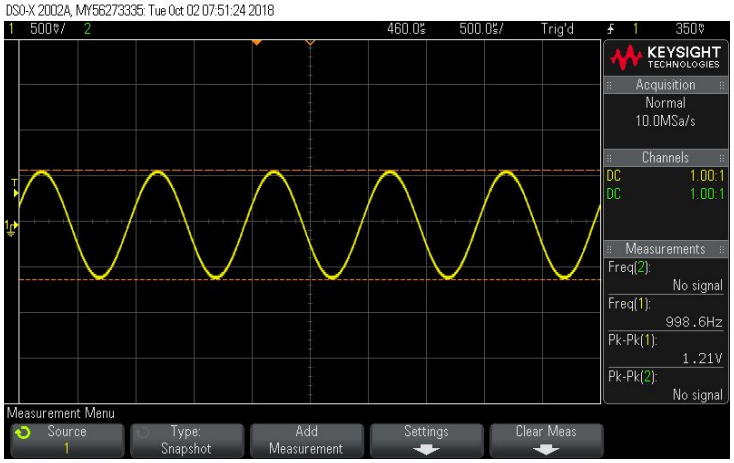

Figure 2.1: Screenshot of the Oscilloscope for a 1000 Hz Sine Wave

To produce a 1400 Hz sine wave, a new sequence was created that produced 7 cycles in 40 samples. When outputted at a sample rate of 8000 Hz, this produced a 1400 Hz sine wave. (See equation below)

1400 Hz Sine Wave → 8000 (samples/s) / (40/7) (samples/cycle) = 1400 (cycles/second)

Altering the index of the lookup table will cause the output of the frequency to change. This is because the generated frequency was determined by the equation

$$f_{gen} = F_s/samples.$$

Consequently, increasing the number of samples will cause the output frequency to decrease. The LOOPLENGTH and the sine_table in the code had to be adjusted in order to produce the desired number of cycles.

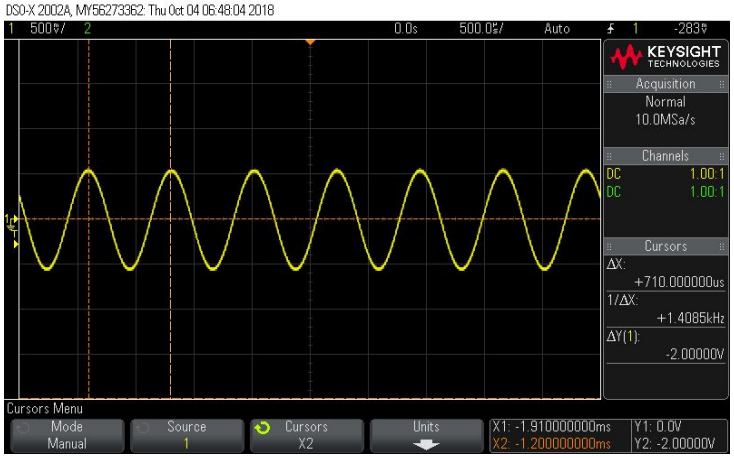

Figure 2.2: Screenshot of the Oscilloscope for a 1400 Hz Sine Wave

```
#include "L138_LCDK_aic3106_init.h"
#include "evmomapl138_gpio.h"

#define LOOPLENGTH 40

int16_t sine_ptr = 0;
int16_t sine_table[LOOPLENGTH]={0,
            891,
            809,
            -156,
            -951,
            -707,
            309,
            988,
            588,
            -454,
            -1000,
            -454,
            588,
            988,
            309,
            -707,
            -951,
            -156,
            809,
            891,
            0,
            -891,
            -809,
            156,
            951,
            707,
            -309,
            -988,
```

```
            -588,
            454,
            1000,
            454,
            -588,
            -988,
            -309,
            707,
            951,
            156,
            -809,
            -891

};

int16_t gain = 20;

interrupt void interrupt4(void) // interrupt service routine
{
      int16_t left_sample;

      left_sample = (sine_table[sine_ptr]*gain);
      sine_ptr = sine_ptr + 1;
      sine_ptr = sine_ptr % LOOPLENGTH;

      output_left_sample(left_sample);

      return;
}
int main(void)
{
L138_initialise_intr(FS_8000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB,LCDK_LINE_INPUT);
while (1);
}
```

Step 3

A sinusoidal function could be made using a difference equation instead of a lookup table. A 5 kHz sinusoid was made using the below difference equation,

$$y(n) = x(n) + 2 \bullet cos(\theta) \bullet y(n-1) - y(n-2).$$

The resulting sinusoid has an oscillating frequency of

$$f = f_s(\frac{\theta}{2\pi}),$$

where f_s is the sampling frequency of the D/A converter. In this case, the sampling frequency is 48,000 samples/second. Figure 3 depicts the 5 kHz sine wave made by the difference equation.

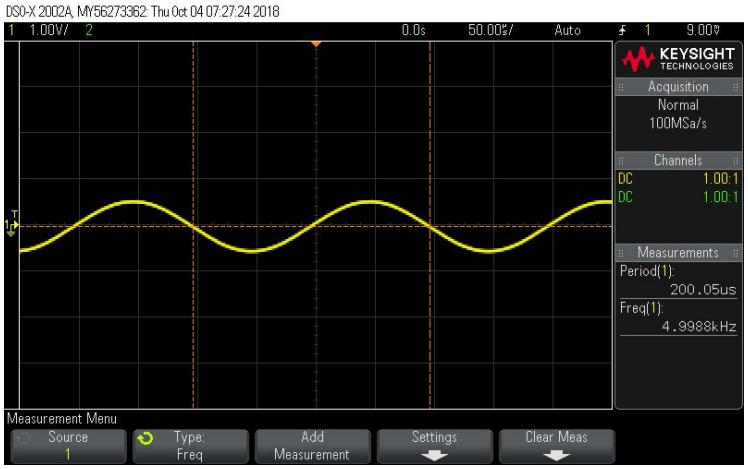

Figure 3.1 A 5 kHz Sine Wave Formulated by the Difference Equation

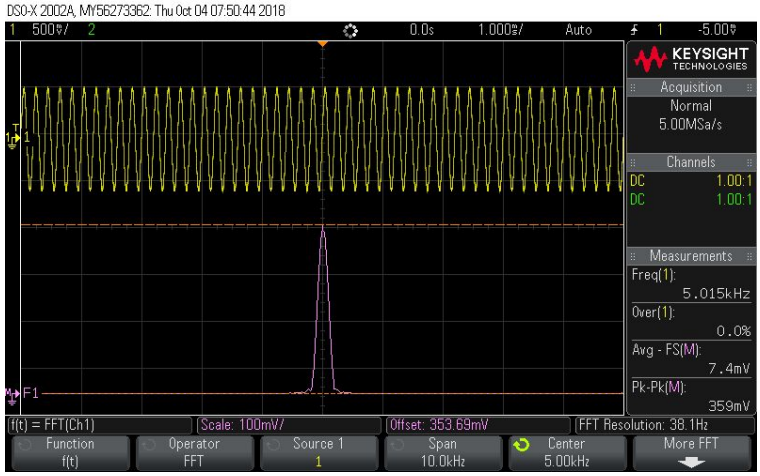

Figure 3.2 A 5 kHz Sine Wave with Numerous Cycles. The pink curve displays the FFT within a Hanning Window.

Notice in Figure 3 that the FFT within the Hanning Window is a single peak at 5 kHz. One important consideration when using the Oscilloscope FFT is that the scope only takes the FFT of the waveform shown on the screen, so it is impossible to take the FFT of an infinitely long signal using the oscilloscope. If the time scale of the FFT in Figure 3.2 was adjusted such that the screen displayed fewer cycles, then the FFT would have a broader spectrum. By changing the time base to display fewer cycles of the sine wave, the FFT converges to the FFT of a DC signal, which is a delta function at 0 Hz. As the time window shrinks, the peak should become broader and shorter and a peak should begin to develop at 0 Hz. On the other hand, manipulating the time base to display more cycles narrows the FFT. By changing the time base to display more cycles, the FFT approaches the FFT of an infinitely long sinusoidal function, which is a delta function at 5 kHz.

```
#include "evmomapl138_gpio.h"
#include <math.h>

int16_t gain = 20;
float pi;
float theta;
float yn1 = 0; // y(n-1)
float yn2 = 0;
float x1;
float yn1_coeff;
int16_t left_sample;


interrupt void interrupt4(void) // interrupt service routine
{
      float y;
      y = x1 + yn1_coeff*yn1 - yn2;
      yn2 = yn1;
      yn1 = y;
      left_sample = (int16_t)(y*20000);
      output_left_sample(left_sample);
      x1 = 0;
      return;
}

int main(void)
{
      pi = 3.14159;
      theta = 5*pi/24;
      x1 = sin(theta);
      yn1_coeff = 2*cos(theta);
L138_initialise_intr(FS_48000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB,LCDK_LINE_INPUT);
while (1);
}
```

Step 4

Using the difference equation method in Step 3, we generated a sum of 3 sine waves at 2.5 kHz, 3.5 kHz, and 5.8 kHz. We used a sampling rate of 48 ksps. The resulting signal and its FFT are shown below. Note that the FFT has 3 peaks at 2.5 kHz, 3.5 kHz, and 5.8 kHz, which is what we expect. The time domain waveform looks like a 5.8 kHz waveform modulated by a lower frequency envelope.

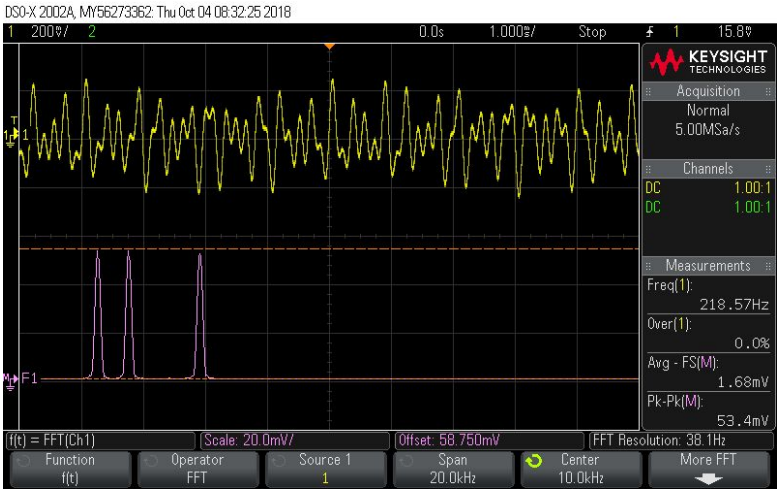

Figure 4.1 Original Three-Frequency Signal and its FFT

Next, a fourth sinusoid at 19.5 kHz was added onto the waveform. The Nyquist frequency of the DAC is half of the sample rate, or 24 kHz. Since our new sinusoid is at a lower frequency than the Nyquist frequency, we expect there to be no aliasing. As expected, in Figure 4.2, we see peaks at 2.5 kHz, 3.5 kHz, 5.8 kHz, and 19.5 kHz. With the newly added frequency, the time domain waveform now looks like a 19.5 kHz waveform modulated by a lower frequency envelope.

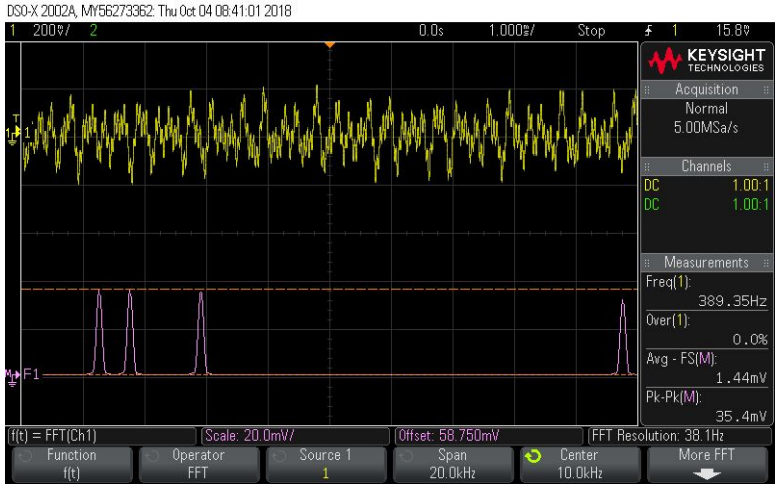

Figure 4.2 Waveform with Noise and FFT at 48 Ksps

Next, we reduced the sampling rate from 48 ksps to 24 ksps. This reduces the Nyquist frequency to 12 kHz. We now expect any frequencies between 12 kHz and 24 kHz to be mirrored about the Nyquist frequency into the range of frequencies from 0 Hz to 12 kHz. This is called aliasing. Since 19.5 kHz is 7.5 kHz greater than the Nyquist frequency, we expect it to be mirrored onto the frequency 7.5 kHz less than the Nyquist frequency,

$$12 \text{ kHz} - 7.5 \text{ kHz} = 4.5 \text{ kHz}.$$

As expected, in Figure 4.3, we no longer see a peak at 19.5 kHz and instead see a peak at 4.5 kHz. Because of aliasing, the DAC actually outputs a 4.5 kHz waveform instead of a 19.5 kHz waveform, so the output once again looks like a 5.8 kHz waveform modulated by a lower frequency envelope.

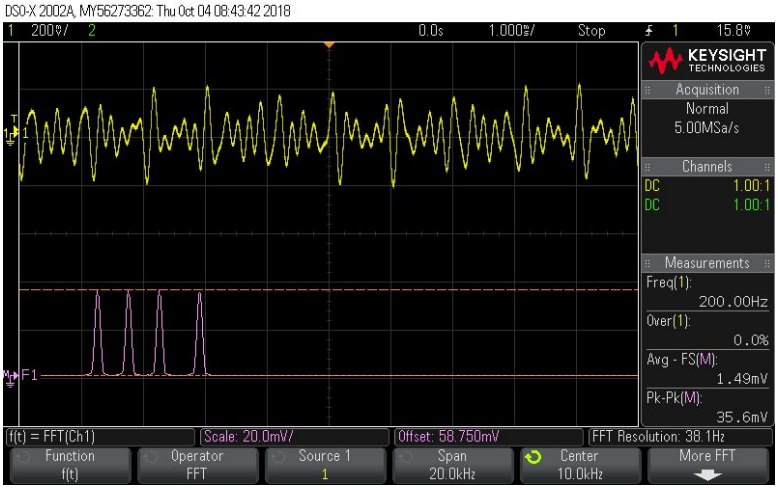

Figure 4.3 Waveform with Noise and FFT at 24 Ksps

In conclusion, the highest frequency contained in the signal should always be less than the Nyquist frequency, which is half of the sample rate. Otherwise, aliasing will occur in the output of the DAC.

```
#include "L138_LCDK_aic3106_init.h"
#include "evmomapl138_gpio.h"
#include <math.h>

int16_t gain = 20;
float pi;
float x1, x2, x3, x4;
float theta1, theta2, theta3, theta4;
float coeff1, coeff2, coeff3, coeff4;
```

```
int16_t left_sample;

float yn1_2500 = 0;
float yn2_2500 = 0;
float yn1_3500 = 0;
float yn2_3500 = 0;
float yn1_5800 = 0;
float yn2_5800 = 0;
float yn1_19500 = 0;
float yn2_19500 = 0;

float theta(float f) {
      return f * 2 * pi / 24000;
}

float waveform(float yn1, float yn2, float coeff) {
      return coeff * yn1 - yn2;
}

interrupt void interrupt4(void) // interrupt service routine
{
      float y1, y2, y3, y4;
      y1 = x1 + waveform(yn1_2500, yn2_2500, coeff1);
      yn2_2500 = yn1_2500;
      yn1_2500 = y1;

      y2 = x2 + waveform(yn1_3500, yn2_3500, coeff2);
      yn2_3500 = yn1_3500;
      yn1_3500 = y2;

      y3 = x3 + waveform(yn1_5800, yn2_5800, coeff3);
      yn2_5800 = yn1_5800;
      yn1_5800 = y3;

      y4 = x4 + waveform(yn1_19500, yn2_19500, coeff4);
            yn2_19500 = yn1_19500;
            yn1_19500 = y4;

      x1 = 0;
      x2 = 0;
      x3 = 0;
      x4 = 0;

      float y = y1 + y2 + y3 + y4;

      left_sample = (int16_t)(y*2000);
      output_left_sample(left_sample);
      return;
}
int main(void)
{
      pi = 3.14159;
      theta1 = theta(2500);
```

```
      theta2 = theta(3500);
      theta3 = theta(5800);
      theta4 = theta(19500);
      x1 = sin(theta1);
      x2 = sin(theta2);
      x3 = sin(theta3);
      x4 = sin(theta4);

      coeff1 = 2*cos(theta1);
      coeff2 = 2*cos(theta2);
      coeff3 = 2*cos(theta3);
      coeff4 = 2*cos(theta4);
L138_initialise_intr(FS_24000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB,LCDK_LINE_INPUT);
while (1);
}
```