EE 113D: Digital Signal Processing Design

Lab #2
Fast Fourier Transform

Names: Erik Hodges & Ivan Manan

Due Date: October 22, 2018

Step 2

In Step 2 of the lab, the function generator provided a 3 kHz, 0.5 V_{pp} waveform as input for the LCDK. Likewise, the LCDK received the input and redirected the same input as its output. The below screenshot of the oscilloscope displays the waveforms from the function generator (green) and the LCDK (yellow). Notice the delay of the output from the LCDK. This is the processing delay of the LCDK.

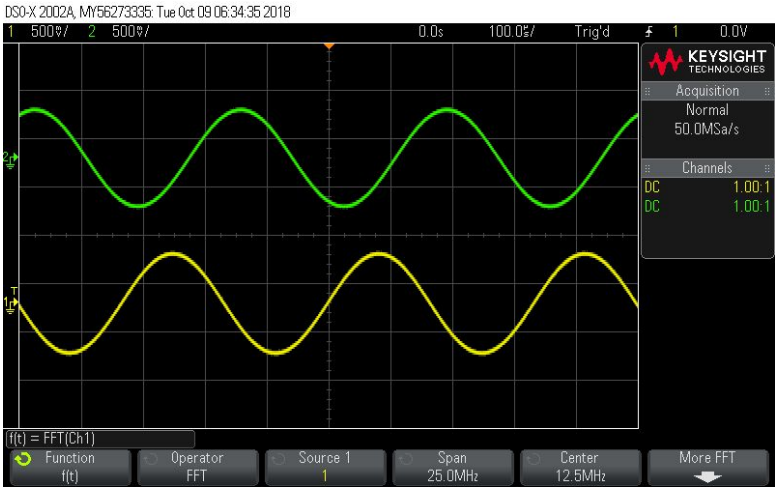

Figure 2.1 Outputs of the Function Generator (Green) and the LCDK (Yellow)

Step 4

The next portion of the lab involved taking the same input from the function generator and having the LCDK compute the 1024-point FFT. The sampling frequency was 48 Ksamples/s. The source code template was provided by the TA.

The array containing the signal's FFT will have complex numbers, $a + bj$. The magnitude for a single point of the FFT was calculated as:

$$|X(k)| = \sqrt{(a)^2 + (b)^2} .$$

Figure 4.1 displays the FFT magnitude plot for a 3 kHz sine wave signal. Likewise, Figure 4.2 displays the FFT magnitude plot for a 4 kHz sine wave signal.

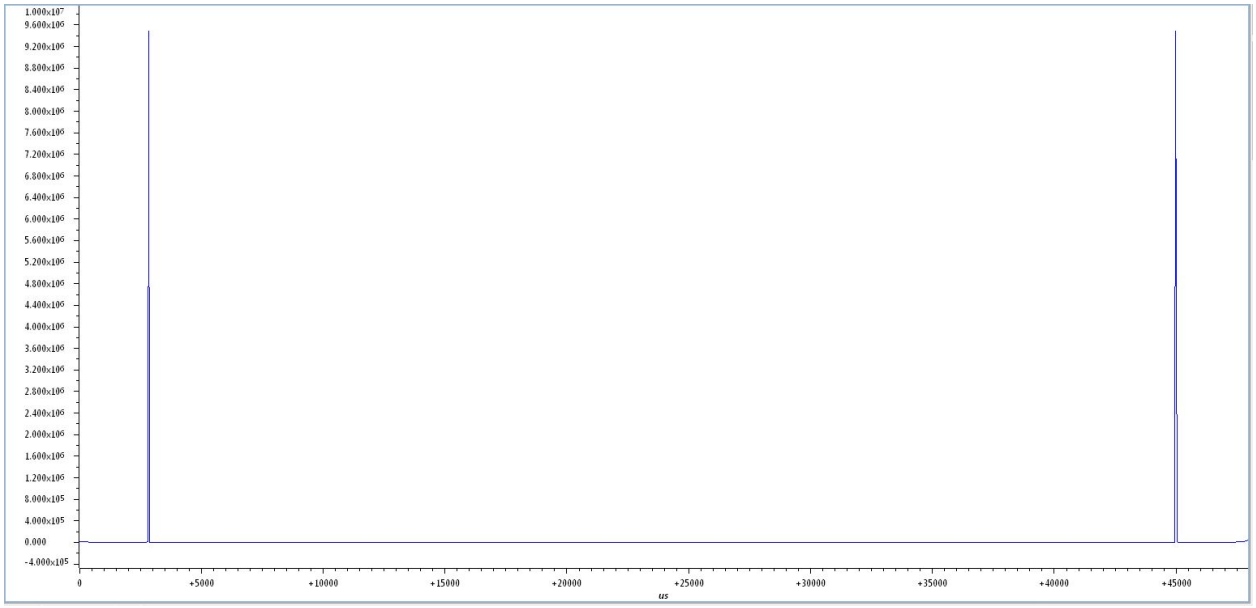

Figure 4.1 The FFT Magnitude Plot for a 3 kHz Sine Wave Signal

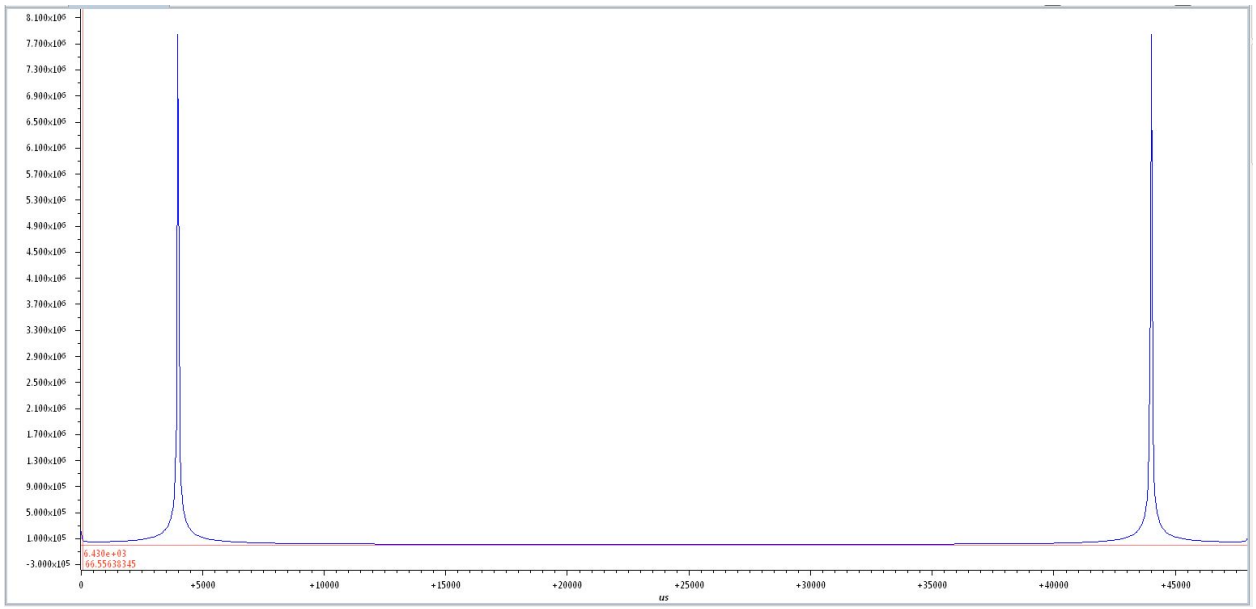

Figure 4.2 The FFT Magnitude Plot for a 4 kHz Sine Wave Signal

The FFT magnitude plots in Figures 4.1 and 4.2 each have two peaks. However, the primary difference between the two FFT magnitude plots is the location of the peaks. In Figure 4.1, since the input for the LCDK was a 3 kHz sine wave, the first peak was located at 3000 Hz. Likewise in Figure 4.2, since the input for the LCDK was a 4 kHz sine wave, the first peak was located at 4000 Hz.

Since the FFT is periodic, a second peak appears at 45000 Hz in Figure 4.1. The second peak is analogous to a peak at -3000 Hz. Likewise in Figure 4.2, the second peak appears at 44000 Hz, analogous to a peak at -4000 Hz.

Step 5

The IFFT transforms the FFT into the time domain signal, which is a 3 kHz sine wave. The real and imaginary parts of the IFFT were separated and displayed in Figures 5.1 and 5.2.

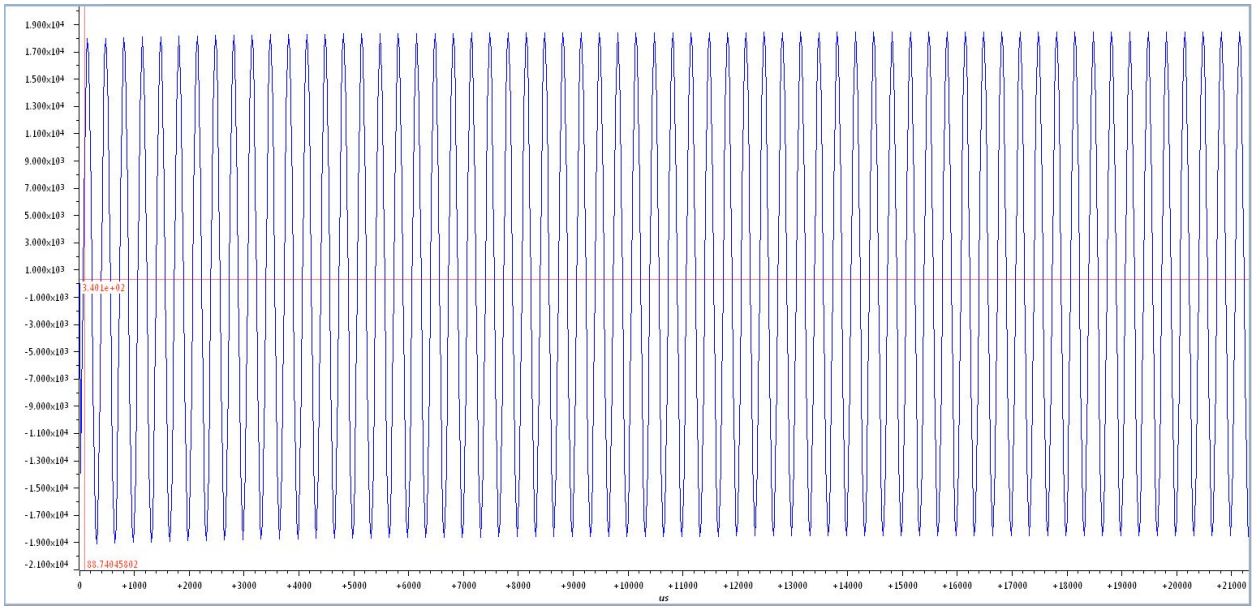

Figure 5.1 Real Part of the IFFT

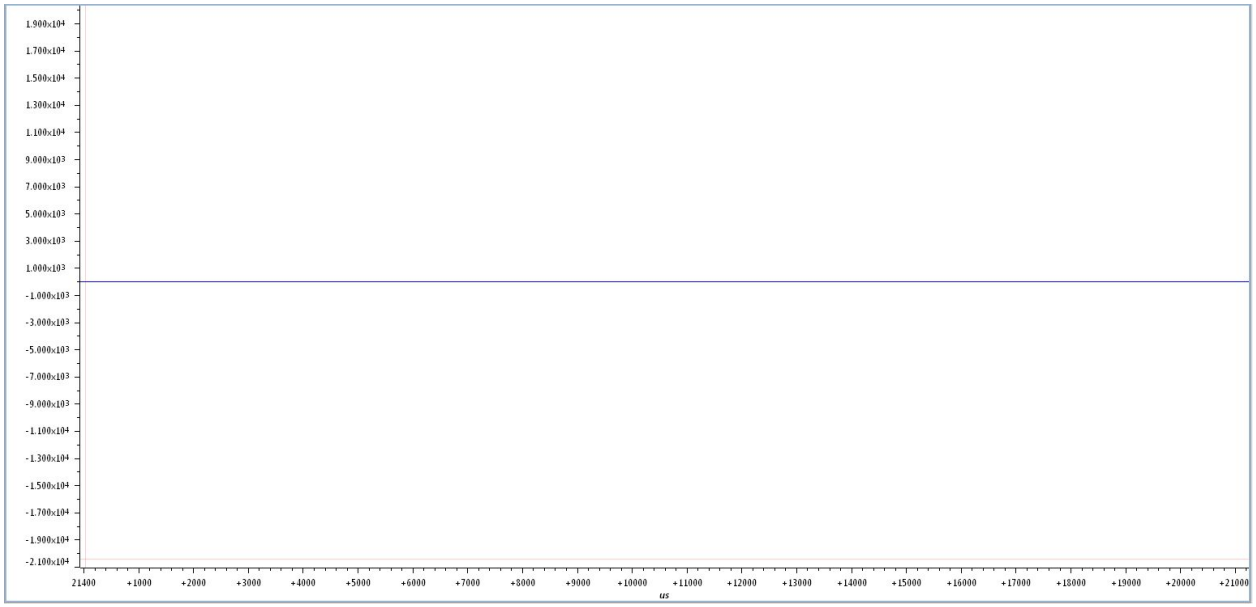

Figure 5.2 Imaginary Part of the IFFT

Notice the difference between Figures 5.1 and 5.2. The original signal was entirely real, with no imaginary part. As a result, Figure 5.1 displays the original sine wave. Figure 5.2 displays a constant value at an amplitude of approximately $-1.00 \bullet 10^3$, which is approximated as zero. This slight discrepancy is likely due to random noise.

The code below was used to output the real and imaginary parts of the IFFT. The yellow highlighted segments displays the code responsible for this.

```
#include "L138_LCDK_aic3106_init.h"
#include "evmomapl138_gpio.h"
#include <stdint.h>
#include <math.h>
#include <ti/dsplib/dsplib.h>


// Global Definitions and Variables
#define PI 3.14159265358979323
#define N 1024

int16_t my_arr[N];
float magnitude_arr[N];

int idx = 0;
int flag = 0;

// IFFT X real and imag
float x_real[N];
float x_imag[N];

/* Align the tables that we have to use */

// The DATA_ALIGN pragma aligns the symbol in C, or the next symbol declared
in C++, to an alignment boundary.
// The alignment boundary is the maximum of the symbol's default alignment
value or the value of the constant in bytes.
// The constant must be a power of 2. The maximum alignment is 32768.
// The DATA_ALIGN pragma cannot be used to reduce an object's natural
alignment.

//The following code will locate mybyte at an even address.
//#pragma DATA_ALIGN(mybyte, 2)
//char mybyte;

//The following code will locate mybuffer at an address that is evenly
divisible by 1024.
//#pragma DATA_ALIGN(mybuffer, 1024)
//char mybuffer[256];
#pragma DATA_ALIGN(x_in,8);
int16_t x_in[2*N];
```

```
#pragma DATA_ALIGN(x_sp,8);
float   x_sp [2*N];
#pragma DATA_ALIGN(y_sp,8);
float   y_sp [2*N];
#pragma DATA_ALIGN(w_sp,8);
float   w_sp [2*N];

// brev routine called by FFT routine
unsigned char brev[64] = {
    0x0, 0x20, 0x10, 0x30, 0x8, 0x28, 0x18, 0x38,
    0x4, 0x24, 0x14, 0x34, 0xc, 0x2c, 0x1c, 0x3c,
    0x2, 0x22, 0x12, 0x32, 0xa, 0x2a, 0x1a, 0x3a,
    0x6, 0x26, 0x16, 0x36, 0xe, 0x2e, 0x1e, 0x3e,
    0x1, 0x21, 0x11, 0x31, 0x9, 0x29, 0x19, 0x39,
    0x5, 0x25, 0x15, 0x35, 0xd, 0x2d, 0x1d, 0x3d,
    0x3, 0x23, 0x13, 0x33, 0xb, 0x2b, 0x1b, 0x3b,
    0x7, 0x27, 0x17, 0x37, 0xf, 0x2f, 0x1f, 0x3f
};

// The seperateRealImg function separates the real and imaginary data
// of the FFT output. This is needed so that the data can be plotted
// using the CCS graph feature
float y_real_sp [N];
float y_imag_sp [N];

separateRealImg () {
    int i, j;

    for (i = 0, j = 0; j < N; i+=2, j++) {
        y_real_sp[j] = y_sp[i];
        y_imag_sp[j] = y_sp[i + 1];
    }
}

// Function for generating sequence of twiddle factors
void gen_twiddle_fft_sp (float *w, int n)
{
    int i, j, k;
    double x_t, y_t, theta1, theta2, theta3;

    for (j = 1, k = 0; j <= n >> 2; j = j << 2)
    {
        for (i = 0; i < n >> 2; i += j)
        {
            theta1 = 2 * PI * i / n;
            x_t = cos (theta1);
            y_t = sin (theta1);
            w[k] = (float) x_t;
            w[k + 1] = (float) y_t;

            theta2 = 4 * PI * i / n;
            x_t = cos (theta2);
```

```
            y_t = sin (theta2);
            w[k + 2] = (float) x_t;
            w[k + 3] = (float) y_t;

            theta3 = 6 * PI * i / n;
            x_t = cos (theta3);
            y_t = sin (theta3);
            w[k + 4] = (float) x_t;
            w[k + 5] = (float) y_t;
            k += 6;
        }
    }
}

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

  // Input from ADC (Line IN)
  left_sample = input_left_sample();

  // Your code here
  if(idx<N){
      // Input is being read sample by sample real part in even indices,
imaginary in odd.
    x_in[2*idx]=left_sample;
    x_in[2*idx+1]=(float)0.0;


      // Variable idx is global and its value is kept
      idx++;
  }

  // Output to DAC (Line OUT)
  output_left_sample(left_sample);
  if (idx >= 1024)
        flag = 1;
  return;
}

void calculate_magnitude() {


      // Even indices indicate real
      // Odd indices indicate imaginary
      int i;
      for (i = 0; i < 1024; i++) {
            //magnitude_arr[i] = sqrt(output[i]*output[i] + output[i+1] *
output[i+1]);
            magnitude_arr[i] = sqrt(y_real_sp[i]*y_real_sp[i] + y_imag_sp[i]
* y_imag_sp[i]);
      }
}
```

```
void plot_ifft() {
	int k;
	int counter = 0;

	// REAL
	for (k = 0; k < 2048; k += 2) {
		x_real[counter] = x_sp[k];
		counter++;
	}

	counter = 0;
	// ODD
	for (k = 1; k < 2048; k += 2) {
		x_imag[counter] = x_sp[k];
		counter++;
	}
}

int main(void)
{

L138_initialise_intr(FS_48000_HZ,ADC_GAIN_0DB,DAC_ATTEN_0DB,LCDK_LINE_INPUT);


  // SAMPLE CODE: USE OF FFT ROUTINES
  while(1) {

        if (flag == 1)
              break;
  }
      // Copy input data to the array used by DSPLib functions
  int n;
      for (n=0; n<N; n++)
      {
        x_sp[2*n]   = x_in[2*n];
        x_sp[2*n+1] = x_in[2*n+1];
        my_arr[n] = x_sp[2*n];
      }

      // Call twiddle function to generate twiddle factors needed for FFT and
IFFT functions
  gen_twiddle_fft_sp(w_sp,N);

  // Call FFT routine
  DSPF_sp_fftSPxSP(N,x_sp,w_sp,y_sp,brev,4,0,N);



  // Call routine to separate the real and imaginary parts of data
  // Results saved to floats y_real_sp and y_imag_sp
  separateRealImg ();
```

```
// Call the inverse FFT routine
DSPF_sp_ifftSPxSP(N,y_sp,w_sp,x_sp,brev,4,0,N);

// END OF SAMPLE CODE



calculate_magnitude();
plot_ifft();

while(1);
}
```

Step 6

The next portion of this lab involves zeroing out part of the original signal in order to simulate an application in which we take the FFT of a sequence with length that is not a power of 2. This was done by generating 900 samples of a sinusoid followed by 124 zeros. This is the same as multiplying the time domain signal by a square pulse in the time domain. In the frequency domain, the analogous operation is convolution with a sinc function, resulting in the ripples around the peaks. One interesting thing is that, since the FFT of the non-zero-padded 3000 Hz sine wave had very low spectral leakage, each peak was very close to a delta function as in Figure 4.1. Since convolving a function with a delta function gives back the original function, the ripples around the peak resemble a perfect sinc function, as displayed in Figure 6.1. However, the 4000 Hz sine wave displayed large spectral leakage, so the sinc pattern is distorted, as evident in Figure 6.2.

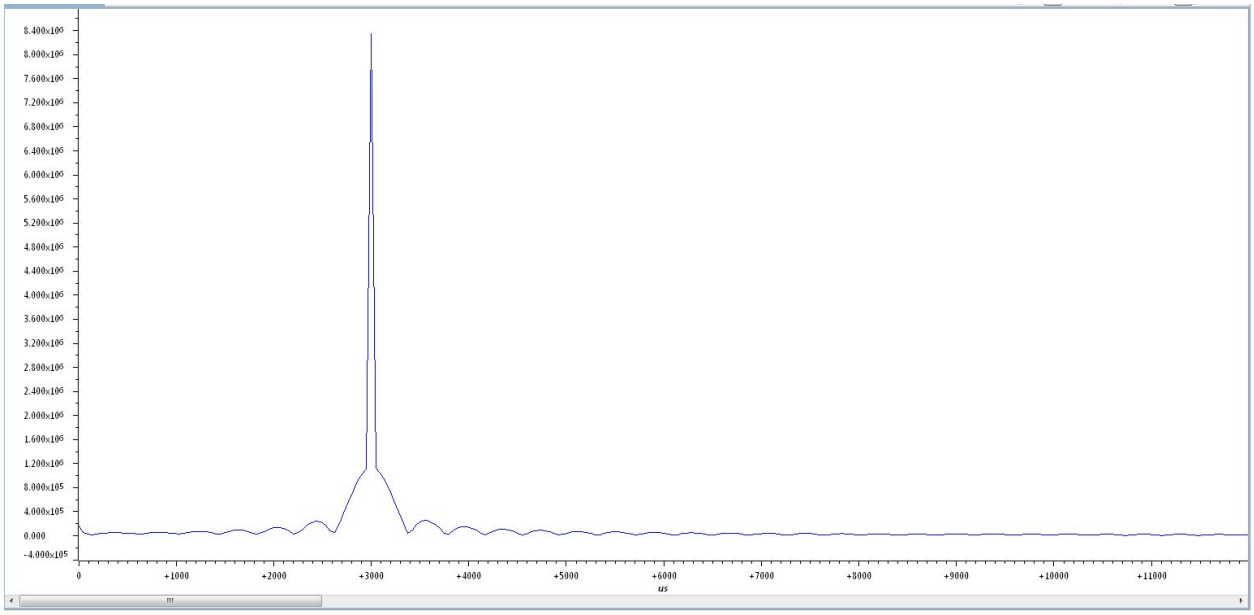

Figure 6.1 The FFT of the 3 kHz Sinusoid with Added Zeros

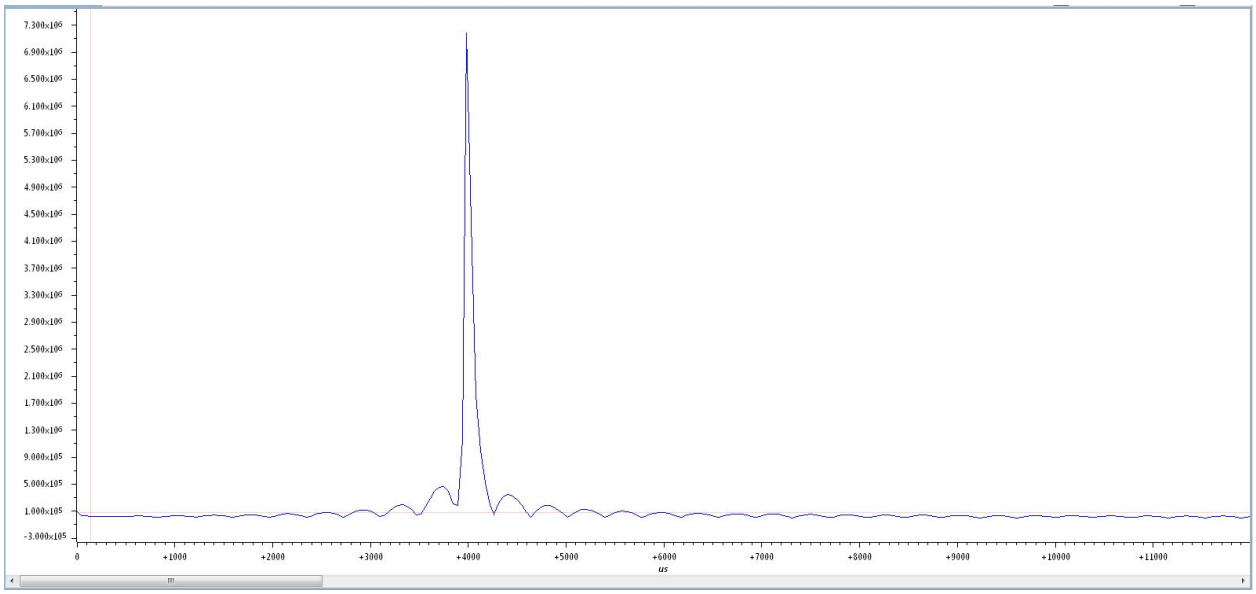

Figure 6.2 The FFT of the 4 kHz Sinusoid with Added Zeros

Step 7

The LCDK took the FFT of an incoming square wave and outputted a copy of the square wave to the oscilloscope, which took an FFT of the output. Viewing the FFT taken by the LCDK in Figure 7.1, it is evident that the function generator produced a square wave by overlaying various sinusoids with frequencies of 1000 Hz, 3000 Hz, 5000 Hz, 7000 Hz and so on. Since the peaks are sharp, it is assumed that the function generator is producing each individual sinusoid accurately. However, looking at the screenshot of the oscilloscope in Figure 7.2, it is clear that the LCDK's DAC is not capable of producing a square wave of the same quality as the function generator. In the time domain, the waveform exhibits Gibbs Phenomenon, which occurs when the DAC cannot generate high enough frequencies to produce a sharp edge. To compensate, the DAC has to start the transition in output voltage early and has to overshoot the final voltage. In the screenshot, we can clearly see a small voltage spike immediately before and after each edge of the square wave. The ripple in the waveform is also a result of Gibbs Phenomenon. The spectral leakage in the oscilloscope FFT is probably from a combination of the inferiority of the LCDK when compared to the function generator as well as the short length of the time window on the oscilloscope.

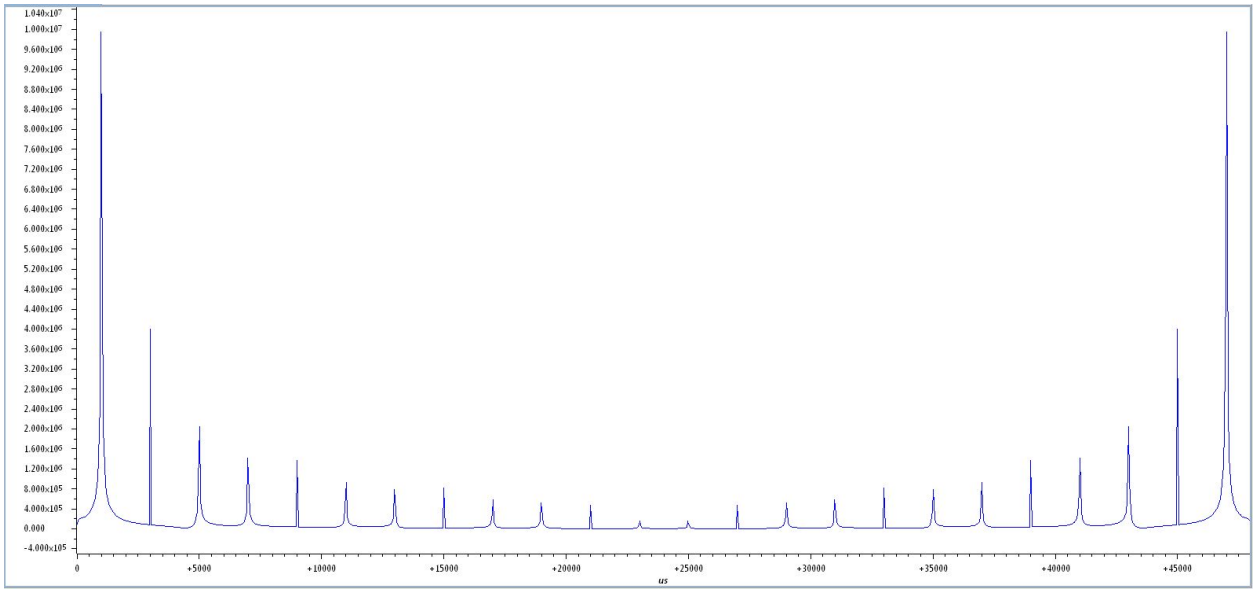

Figure 7.1 Computer Monitor Screen Shot of the FFT taken by the LCDK

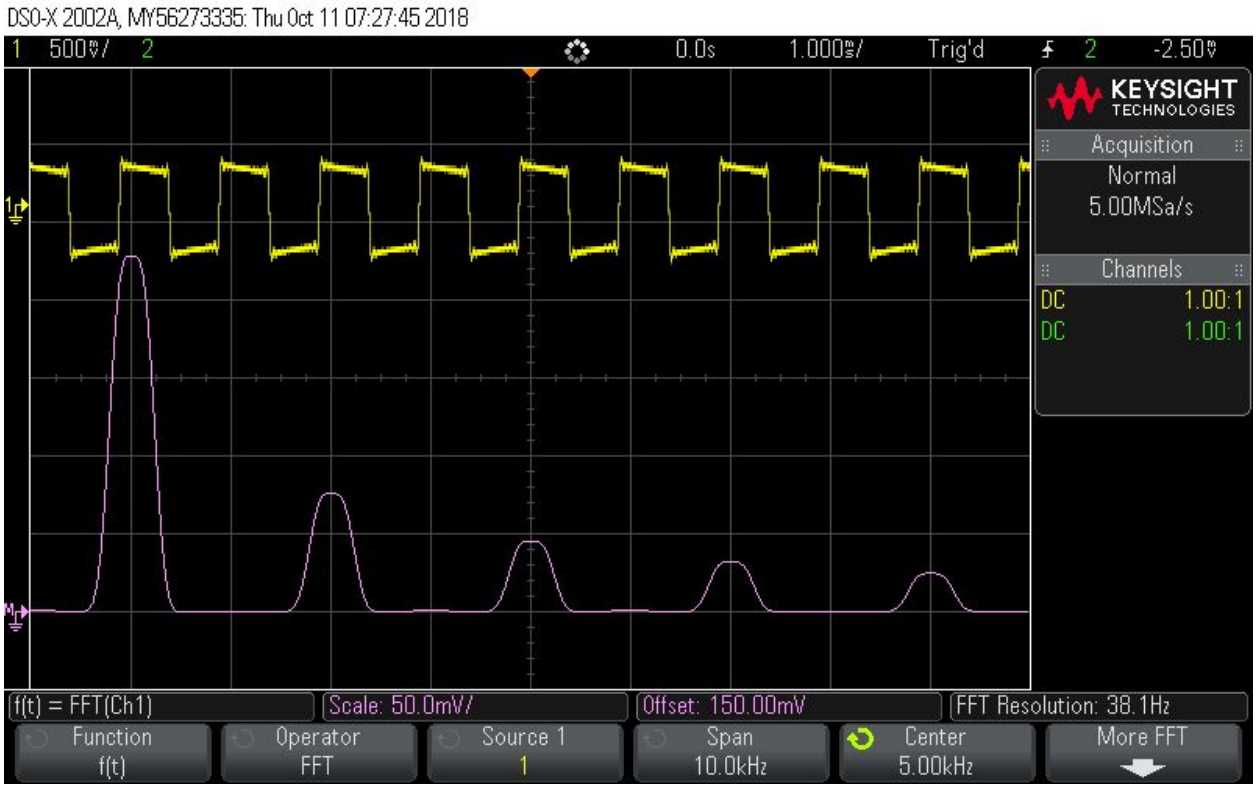

Figure 7.2 Oscilloscope Screen Shot of the FFT of the Square Wave

Step 8

The 2D FFT of a square of ones gives a sinc pattern running along the x and y directions. This corresponds to a constant field intensity across the aperture. Since the distribution is a square, the pattern is identical in both the x and y direction.

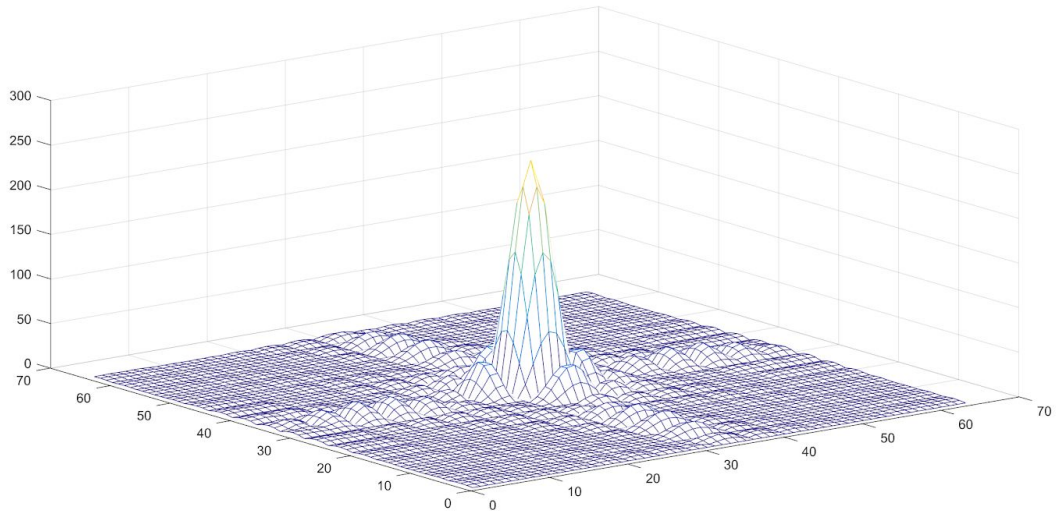

Figure 8.3 Magnitude Plot of a 2D FFT with a 16x16 Matrix of One's

When aperture changed to a 24 x 16 rectangle, the direction with 24 ones becomes narrower since the FFT of a longer square pulse gives a narrower sinc function. This makes sense because large aperture antennas are more directional than small aperture antennas, so the axis of a wider aperture should exhibit a more directional beam pattern. Since the aperture is larger overall, the peak in the center is higher since the antenna is more directional overall.

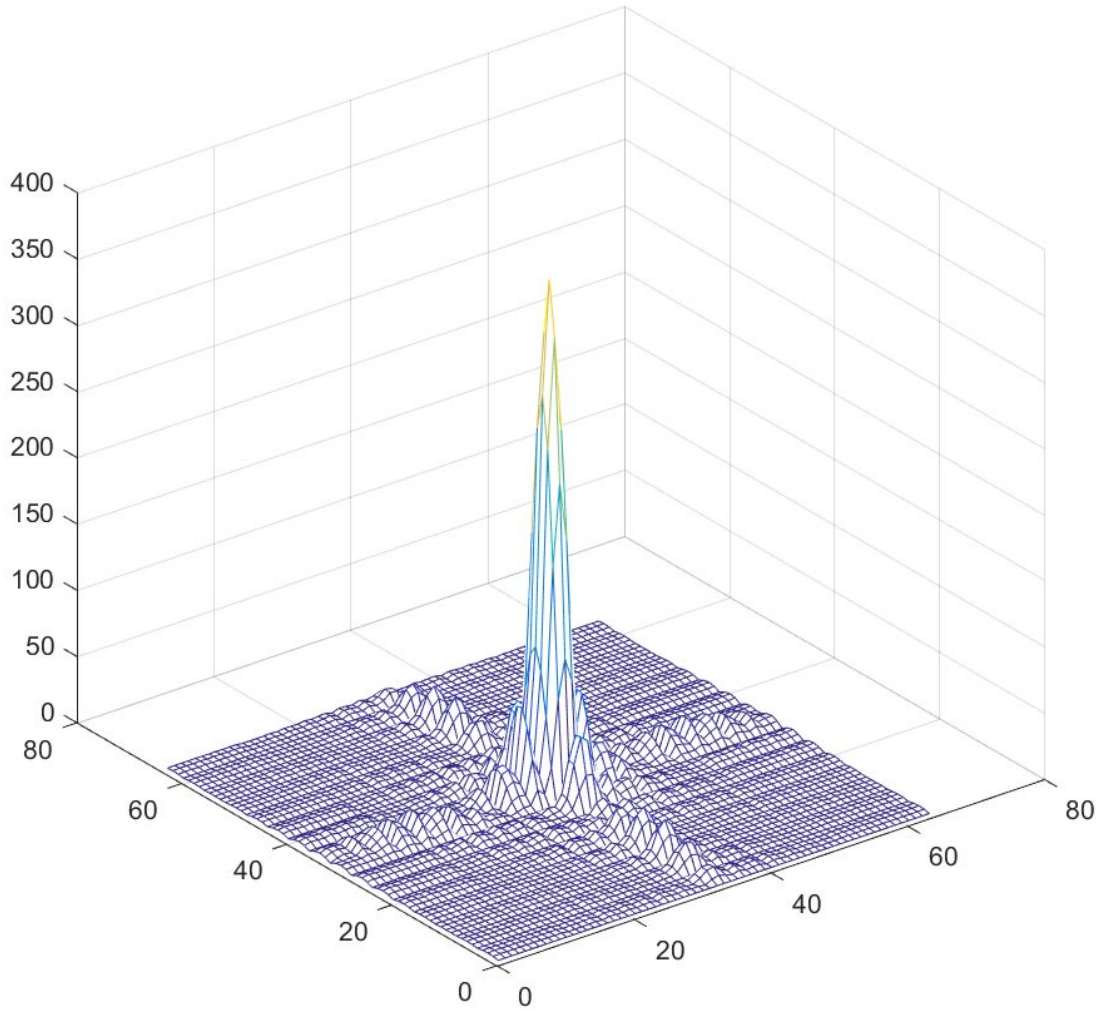

Figure 8.3 Magnitude Plot of a 2D FFT with a 24x16 Matrix of One's

When the intensity is sinusoidally distributed along each row, the intensity drops to zero at each each end of the aperture in one direction but remains constant in the other direction. Along the direction that remains constant, the FFT should still look like a sinc function. Along the sinusoidal direction, the time domain signal is equivalent to a cosine multiplied by a square pulse. In the frequency domain, this corresponds to a sinc function convolved with two offset delta functions of opposite signs. Therefore, the direction that varies sinusoidally should have a wider beamwidth. This makes sense physically because the sinusoidal distribution effectively shrinks the aperture of the antenna along this direction.

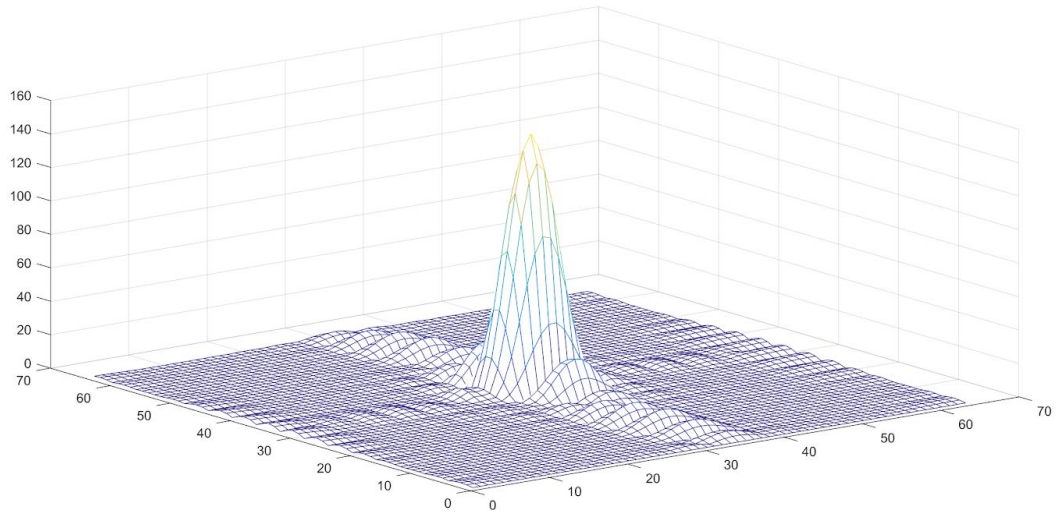

Figure 8.3 Magnitude Plot of a 2D FFT with a Sine Wave