ECE423
Real-Time DSP Laboratory 2:
Signals and Systems on the TMS320C6711

Name1
Name2
Date

Introduction

The purpose of this lab is to explore the effects of a Low Pass filter (Anti-Aliasing Filter) following the onboard codec when sampling and reconstructing a square wave in a straight wire manner. The straight wire (A/D feed to D/A) code will then be modified to implement a simple difference equation and the characteristics of the filter created will be explored. Then the lab will utilize the PCM 3003 daughter card and the generation of phase offset sinusoids and how to display them on an oscilloscope using the XY mode. Whereas the onboard mono codec is capable of a 8kHz sampling rate, the PCM3003 is a stereo codec set to sample at 24 kHz but capable of sampling up to 72 kHz. A program to add N sinusoids will be examined to show the effects of interference. Then a program to show the subtle beating effects of adding two sinusoids just slightly off in frequency from one another will be written and examined.

Part 1: straight wire.c

This first section of the lab focuses on the C program *straight wire.c* which was downloaded from the ECE423 class website and imported into CCS. The code in *straight wire.c* is shown in figure 1 below with all lines of code numbered.

straight wire.c

```
01. // This project uses support files generated by Rulph Chassaing
02. // Comm routines included in C6xdskinit.c
03.
04. interrupt void c_int11()              // interrupt service routine
05. {
06.  output_sample(input_sample());       // Take the input from the codec and feed it back out
07.  return;                              //return from interrupt
08. }
09.
10. void main()                           // Main body of code
11. {
12.   comm_intr();                        // initialize DSK, codec, McBSP for interrupts
13.  while(1);                            // wait for an interrupt to occur (Infinite loop)
14. }
```

Figure 1 Line numbered code for straight wire.c

This program sets up the DSK, codec and McBSP for interrupts and then waits for the 8 kHz interrupts. As the interrupts are occurring at an 8 kHz rate, samples are taken from the A/D converter and then sent immediately to the D/A converter to simulate a digital straight wire.

Assignment 1:

Sampling at 8KHz has its limitations. The sampling theorem states that the sampling frequency for a give signal must be at least 2 times higher than any frequency of interest in the signal. If the sampling frequency becomes any less than 2x the signal frequency,

then aliasing occurs. Therefore, the highest frequency at which a signal can be successfully sampled is $f_s/2$ and is dubbed the Nyquist Frequency.

Using straight wire.c to sample a square wave and changing the frequency f_0 from 300Hz to 1.3 kHz, results in a square wave changing into a sine wave on the oscilloscope. This behavior can be explained from both a frequency-based and a time-based perspective.

The Fourier series expansion of a square wave is a series of odd harmonic sine waves with diminishing amplitude versus frequency as shown in Equation 1. Figure 2 shows the fourier series of a 500Hz square wave. The Square wave looks mostly square due to the addition of three higher frequency harmonics. Figure 3 shows the FFT of a 1KHz sine wave sampled at f_s=8KHz. The DSK's codec has a low-pass filter at ~3.6KHz, so the only part of the FFT that will make it through to the oscilloscope is the 1 and 3 kHz information from the sine wave. The square wave edges are more rounded because the filter has stopped most of the higher ordered harmonics.

If

$$f(x) = 2\left[H(\frac{x}{L}) - H(\frac{x}{L} - 1) \right] - 1$$

the Fourier series is

$$\frac{4}{\pi} \sum_{n=1,3,5,\ldots}^{\infty} \frac{1}{n} \sin(\frac{n\pi x}{L}) \qquad (1)$$

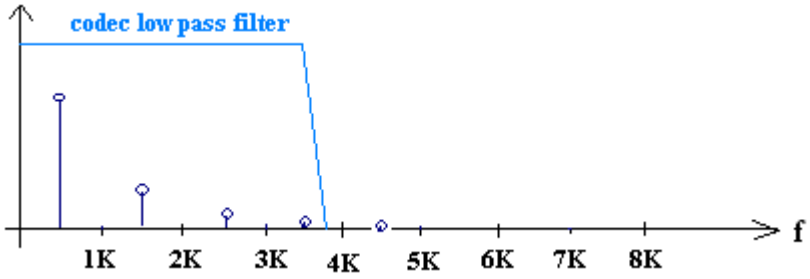


Figure 2 FFT of 500 Hz square wave sampled at 8KHz. Only four sine waves from the square wave's fourier series are passed by the LPF. The fifth at 4500 Hz is filtered out, slightly reducing the sharp edges of the square wave.

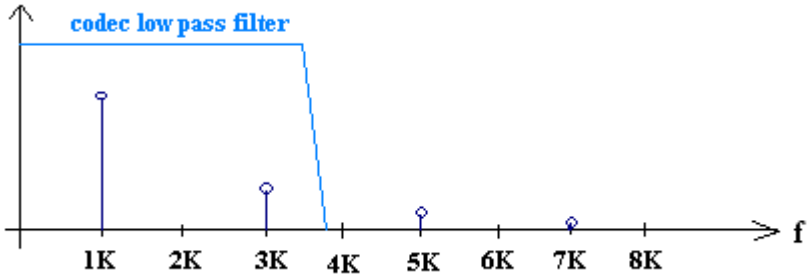


Figure 3 FFT of 1.0 KHz square wave sampled at 8KHz. Only two sine waves from the square wave's fourier series are passed by the LPF. The square waves edges are more rounded.

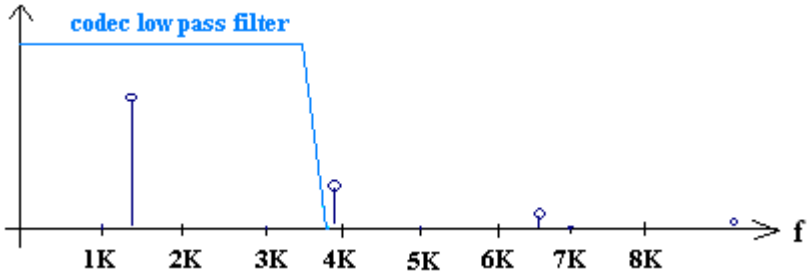


Figure 4 FFT of 1.3 KHz square wave sampled at 8KHz. Only a single sine wave from the square wave's fourier series is passed by the LPF.

Figure 4 shows the FFT of a 1.3KHz square wave. Only the 1.3 kHz sine wave of the square wave fourier series makes it through the Low Pass filter. The third harmonic frequency of the square wave fourier series has moved beyond the cutoff of the LPF and now only a pure sine wave due to the fundamental frequency is shown on the oscilloscope.

Assignment 2a:

For assignment 2 part a, the code from straight_wire.c was modified to add the variable *vgain* to allow the adjustment for unity gain. Note on line 6 that the variable is of data type float so that non-integer values can be used. The other modification is in line 11 where the *vgain* variable is multiplied times the *cur_sample* variable and then the entire sample value is cast to data type short before being outputted to the codec.

straight_wire.c modified to add unity gain variable

```
1.  // This project uses support files generated by Rulph Chassaing
2.  // Comm routines included in C6xdskinit.c
3.  // Modified by
4.
5.  short cur_sample;          // variable to store current sample
6.  float vgain=1.0;           // variable to allow adjustment for unity gain
7.
8.  interrupt void c_int11()                    // interrupt service routine
9.  {
10. cur_sample=input_sample();                  //retrieve current sample
11. output_sample((short)(vgain*cur_sample));   // scale sample by gain and output
12. return;                                     //return from interrupt
13. }
```

```
14.
15. void main()                                          // Main Loop
16. {
17. comm_intr();          // initialize DSK, codec, McBSP for interrupts
18. while(1);             // wait for an interrupt to occur (Infinite Loop)
19. }
```

Figure 5 Line-numbered code for straight_wire.c with modifications on lines 6 and 11 to add a unity gain adjustment variable.

Using a 1KHz, 500mVpp sine wave from the function generator, with the variable *vgain* set to 1.0, the amplitude observed on the scope was 953mVpp. To find the unity gain factor that we need to ensure that what we see on the scope is the same 500mVpp that we input with the function generator, we use

$$vgain = \frac{500mV}{953mV} = 0.525 \qquad (2)$$

Setting *vgain* to this value of 0.525 served to make the output on the scope have an amplitude of 518mVpp, which is close to 500mVpp (within the scope measurement error).

Assignment 2b:

When the variable gain multiplier (*vgain*) was set to a value of 2, the output waveform on the oscilloscope was clipped when the function generator output reached 740mVpp. Multiplying 740mV by 2 corresponds roughly to the 1.65Vp maximum of the DSK codec.

Assignment 2c:

Figure 6 (below) shows the modifications made to straight_wire.c in order to make each sample output be the difference of the current sample and the previous sample. This code takes each input sample of the incoming waveform from the function generator and rather than outputting that sample directly, the code subtracts off the previous sample from the current sample before it is outputted. Lines 5 and 6 of the code show the declaration of short variables for the current and previous samples. On line 10, as soon as the interrupt is called the input sample is stored as the *cur_sample*. Line 11 outputs the current sample minus the previous sample. Line 12 saves off the current sample into the previous sample variable so that the next time the interrupt is called, the previous sample will be the current sample from this interrupt. The only time this value (*prev_sample*) is unknown is the very first time the interrupt is called. At this time, the *prev_sample* variable will be some trash value and the only reason this might be bad is if the user cares about a glitch on the first outputted sample. The rest of the code is identical to the original straight_wire.c program.

straight_wire.c modified for difference filter

```
1. // This project uses support files generated by Rulph Chassaing
2. // Comm routines included in C6xdskinit.c
```

```
3.  // Modified by
4.
5.  short cur_sample;            // variable to store current sample
6.  short prev_sample;           // variable to store previous sample
7.
8.  interrupt void c_int11()                    // interrupt service routine
9.  {
10. cur_sample=input_sample();                  //retrieve current sample
11. output_sample(cur_sample-prev_sample);     // Output current sample minus previous sample
12. prev_sample=cur_sample;              // save off the current sample as the next previous sample
13. return;                                     //return from interrupt
14. }
15.
16. void main()                                         // Main Loop
17. {
18. comm_intr();                        // initialize DSK, codec, McBSP for interrupts
19. while(1);                           // wait for an interrupt to occur (Infinite Loop)
20. }
```

Figure 6 Line-numbered code for straight_wire.c modified to create a difference filter.

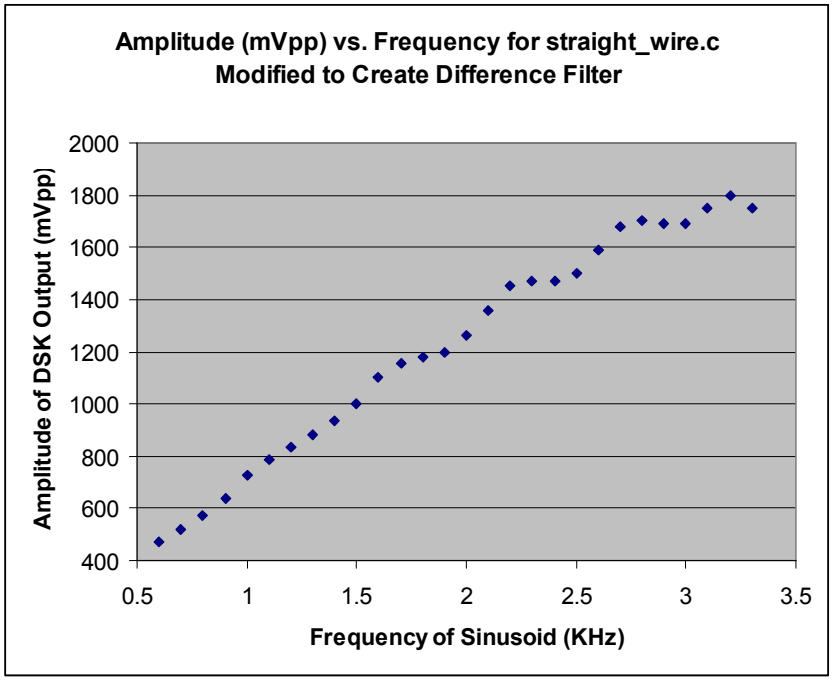


Figure 7a: Experimental Bode plot using sine wave. The plot shows that subtracting the previous sample from the current sample creates a high-pass characteristic.

When the Amplitude versus Frequency data is plotted from part 2C as shown in Figure 7a, it shows the amplitude of the sine wave goes up with frequency. This shows the difference equation implemented in Figure 6 reacts as a High Pass filter to a frequency of 3.4 kHz. It begins to show a dip above this frequency which could be attributed to the LPF following the codec.

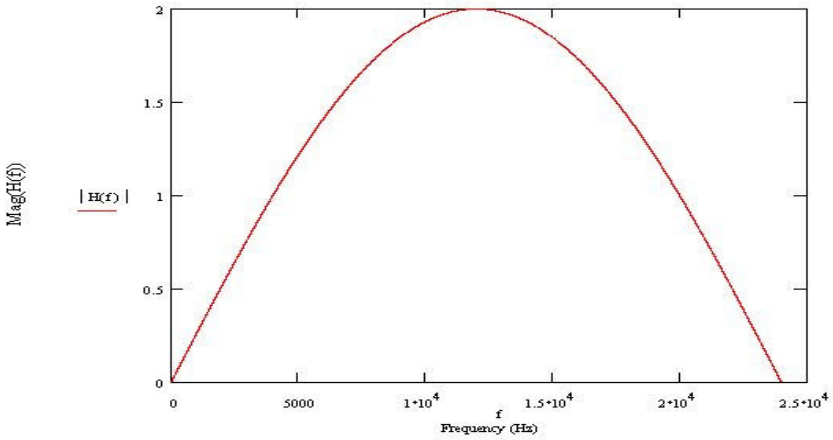
$j := \sqrt{-1}$ Definition of imaginary axis j

$f_s := 24000$ Sampling frequency of audio card

$t_s := \frac{1}{f_s}$ Sampling Period

$H(f) := 1 - \exp\left(-j \cdot 2 \cdot \pi \cdot f \cdot t_s\right)$ Frequency response of difference filter

$f := 1, 5 .. 24000$



Note: The magnitude response repeats periodically, but due to the bandwidth of the audio codec, only one period (or a fraction of a period) is seen empirically.

Figure 8b: Matlab calculated Bode plot.

Figure 7b plots the magnitude response for the FIR filter $H(z) = 1 - z^{-1}$. The FIR filter behaves as a High Pass filter up to the 3.6 kHz cutoff frequency of the LPF following the codec.

Assignment 3:

The code of figure 8 is the program rp_100Hz.c which was downloaded from the class website. The program generates a rotating phasor that is rotating at a rate of 100Hz by sending the real and imaginary parts of the phasor to the left and right output channels respectively.

rp_100Hz.c

```
1.  // This project uses support files generated by Rulph Chassaing
2.  // Comm routines included in C6xdskinit_pcm.c
3.
4.  #include <math.h>                          // Include math library
5.  #define PI 3.14159265359                    // define the constant PI
6.  typedef struct {float real,imag;} COMPLEX;   // define complex number exp_value
7.  short sample_period=240;                     // sinusoid period in samples
8.  short ctr;                                  // loop counter
9.  float angle;                                // angle for cosine function
10. COMPLEX phasor;                             // define phasor as a complex variable type
11. float Fs = 24000.0;                         // irrelevant since jumper in 3-4
12.
```

```
13. interrupt void c_int11()                    // interrupt service routine
14. {
15. angle = 2.0*PI*ctr/sample_period;           // calculate the angle
16. phasor.real=20000*cos(angle);               // real part = cos(w nts)
17. phasor.imag=20000*sin(angle);               // imag part = sin(w nts)
18. output_left_right_sample((short)phasor.real, (short)phasor.imag);   // output each sine value
19. if (ctr < sample_period-1) ++ctr;           // increment counter (0 through 239)
20. else ctr = 0;                               // reset counter if necessary
21. return;                                     // return from interrupt
22. }
23.
24. void main()                                 // Main body of code
25. {
26. ctr=0;                                      // initialize counter
27. comm_intr();                                // initialize DSK, codec, McBSP, interrupts
28. while(1);                                   // wait for an interrupt to occur (Infinite loop)
29. }
```

Figure 9 The line-numbered code for the rp_100Hz.c

In the case of rp_100Hz.c, the frequency and amplitude of both the real and imaginary parts of the phasor are the same, so when this program was loaded onto the DSK and the output of channel 1 was scoped vs. the output of channel 2, a perfect circle was observed on the scope.

Figure 9 (below) shows the code for modified to produce Lissajous figures. The main changes center on making the amplitude and angle of the real and imaginary part of the phasors independent. Thus there are now two sample periods (*sample_period1* and *sample_period2*), two counter variables (*ctr1* and *ctr2*), two angle variables (*angle1* and *angle2*), two phase variables (*phase1* and *phase2*), and finally, two amplitude variables (*Amplitude1* and *Amplitude2*).

rp_100Hz.c – Modified to produce Lissajous figure

```
1.  // This project uses support files generated by Rulph Chassaing
2.  //Comm routines included in C6xdskinit_pcm.c
3.
4.  #include <math.h>                            // Include math library
5.  #define PI 3.14159265359                    // define the constant PI
6.  typedef struct {float real,imag;} COMPLEX;  // define complex number exp_value
7.  short sample_period1=240;
8.  short sample_period2=240;                   // sinusoid period in 10. samples
9.  short ctr1;
10. short ctr2;                                 // loop counter
11. float angle1;                               // angle for cosine function
12. float angle2;                               // angle for cosine function
13. COMPLEX phasor;                             // Define variable phasor as type COMPLEX
14. float Fs = 24000.0;                         // irrelevant since jumper in 3-4
15. float Amplitude1 = 1.0;                     //amplitude for sinusoid1
16. float Amplitude2 = 2.0;                     //amplitude for sinusoid2
17. float phase1 = 30;                          //phase offset for sinusoid 1
18. float phase2 = 45;                          //phase offset for sinusoid 2
19.
20. interrupt void c_int11()                    // interrupt service routine
21. {
22. angle1 = 2.0*PI*ctr1/sample_period1+phase1;   //Compute the angle for sinusoid1
```

```
23. angle2 = 2.0*PI*ctr2/sample_period2+phase2;          //Compute the angle for sinusoid2
24. phasor.real=Amplitude1*10000*cos(angle1);                    // real part = cos(w nts)
25. phasor.imag=Amplitude2*10000*cos(angle2);                    // imag part = sin(w nts)
26. output_left_right_sample((short)phasor.real, (short)phasor.imag);  // output each sine value
27. if (ctr1 < sample_period1-1) ++ctr1;                  // increment counter (0 through 239)
28. else ctr1 = 0;
29.
30. if (ctr2 < sample_period2-1) ++ctr2;                  // increment counter (0 through 239)
31. else ctr2 = 0;                                  // reset counter if necessary
32. return;                                        // return from interrupt
33. }

34. void main()                                    // Main loop
35. {
36. ctr1=0;                                       // initialize counter
37. ctr2=0;                                              // initialize counter
38. comm_intr();                                    // initialize DSK, codec, McBSP
39. while(1);                                            // wait for an interrupt to occur
40. }
```

Figure 10 Line numbered code for rp_100Hz.c modified to produce Lissajous figures.

The discrete-time equation that we are plotting with this program is

$$A_1 \cos(2\pi\omega_1 t_s + \phi_1) + jA_2 \cos(2\pi\omega_2 t_s + \phi_2) \qquad (3)$$

Using lissajous.m in matlab and values of A_1=2, A_2=1, f_1=0, and f_2=45deg, we produced the following figure 10 on the oscilloscope.

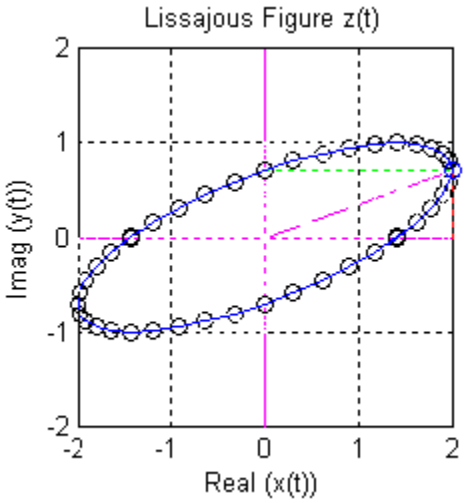


Figure 10 –Lissajous Figure

With our modifications in figure 9 we were able to produce a Lissajous figure on the scope that was the same as that of figure 10. Initially it looked wrong, but we found that we had the left and right channels swapped. Once we got them connected correctly to the scope, the correct figure was observed. By changing the phases and amplitudes of the two sinusoids, the width and rotation of the Lissajous figure could be varied. Also, we found

that changing the relative frequencies of the two sinusoids would created some really interesting figures (like figure eights and things of that nature).

Assignment 4:

Figure 11 below shows the line numbered code for interference.c.

interference.c

```
1.  // This project uses support files generated by Rulph Chassaing
2.  // Comm routines included in C6xdskinit_pcm.c
3.
4.  #include <math.h>
5.  #define PI 3.14159265359                    // define the constant PI
6.  short sample_period=12;                     // sinusoid period in samples
7.  short ctr;                                  // loop counter
8.  short phase[2];                             // theta in degrees (holds two values)
9.  short out_value;                            // value sent to codec
10. float angle;                                // angle for cosine function
11. float wnts;                                 // omega * n * ts - used for current angle
12. float Fs = 24000.0;                                          // irrelevant since jumper in 3-4
13.
14. interrupt void c_int11()                    // interrupt service routine
15. {
16.  int i;                                     // used in for loop
17.
18.  // create interference pattern
19.  // use amplitude 30000/N to prevent overflow in codec, where N=2
20.  wnts = 2.0*PI*ctr/sample_period;           // current angle (w/out phase)
21.  out_value=0;
22.  for (i=0; i<2; i++)
23.  {
24.  angle = wnts + phase[i]*PI/180;           // current angle (with phase)
25.  out_value += (30000/2)*cos(angle);         // cos(w nts + theta_k) k in {1,2}
26.  }
27.
28. output_left_sample(out_value);              // output each sine value
29.  if (ctr < sample_period-1) ++ctr;          // increment counter (0 through 47)
30.  else ctr = 0;                              // reset counter
31.  return;                                    // return from interrupt
32. }
33.
34. void main()
35. {
36.  phase[0]=0;                                // theta 1
37.  phase[1]=45;                               // theta 2
38.  ctr=0;                                     // initialize counter
39.   comm_intr();                              // initialize DSK, codec, McBSP
40.   while(1);                                 // wait for an interrupt to occur
41. }
```

Figure 11 Line numbered code for interference.c

Line # 8 of the code shows the declaration of an array of two values of data type short. These values will be phase[0], the phase of the first cosine wave and phase[1], the phase of the second cosine wave. Line # 16 of the code declares an integer variable called *i*

that is used as the index variable used in the for loop inside the interrupt (starting at line 22). Line # 20 creates the argument for the cosine function without the phase. This argument follows the structure

$$wnts = 2\pi \frac{ctr}{sample\_period} \qquad (4)$$

where the variable *ctr* is initialized to zero at the beginning of the program run (line 38), and then is incremented by one every time the interrupt is called. Once the *ctr* variable reaches the *sample_period* which, in this case, is set to 12 (line 6), it is reset to zero (lines 29 – 32). So, the angle without the phase shift (*wnts*) starts at zero then increments by $\pi/6$ until it reaches 2π where it resets back to zero and starts the incrementing over again in a cyclical manner. In line # 21, every time the interrupt is called, the output sample value is initialized to zero before the next loading of the output sample value. Lines 22 through 26 form a for loop in which the two interfering cosines are created. The first time through the loop, the cosine angle is set on line 24 to

$$angle_0 = wnts + phase[0] * \frac{\pi}{180} \qquad (5)$$

The $\pi/180$ term is just to convert the phase from degrees to radians. Then on line 25 the *out_value* for the sample out is set to

$$\frac{30000}{2} * \cos(angle_0) \qquad (6)$$

where the angle is given by line 24. The next time through the loop, the angle changes to

$$angle_1 = wnts + phase[1] * \frac{\pi}{180} \qquad (7)$$

and on line 25, the *out_value* becomes

$$\frac{30000}{2} * \cos(angle_0) + \frac{30000}{2} * \cos(angle_1) \qquad (8)$$

The third time through, nothing happens because the loop index is now 2 and the Boolean *i<2* in line 22 makes the loop drop out. Finally, lines 36 and 37 initialize the two phase values (*phase[0]* and *phase[1]* to 0 degrees and 45 degrees respectively at the beginning of the program. This way the two phases will be known when running the program without setting these variables in the watch window.

The discrete –time equation for the interference pattern that is being created by interference.c is as follows:

$$\frac{30000}{2} * \cos(\frac{\pi N}{6} + \frac{\pi}{180} phase[0]) + \frac{30000}{2} * \cos(\frac{\pi N}{6} + \frac{\pi}{180} phase[1]) \qquad (9)$$

The program interference.c was run on the DSK and the output observed and compared to the output graphs of the matlab program interference_pattern.m. The magnitude graph (lower left) of the matlab program showed that for a phase difference of 45 degrees ($\pi/4$), the amplitude of the interference wave should be 1.8475/2=0.92. So, the amplitude should be at 92% of the maximum when the phase difference is 45 degrees. Using the DSK output and the scope, when the two phases (phase[0] and phase[1]) are both set to zero, the resulting wave has peak-peak amplitude of 2.18V. When the phases are then set to 0 and 45, the amplitude changes to 2.016V. Since 2.016/2.18=0.92 or 92% this shows good agreement between the matlab magnitude plot and the output observed on the scope. The table of figure XX below shows a comparison of the matlab magnitude and phase values vs. those observed on the scope through the DSK.

Phase[0]	Phase[1]	Normalized Matlab Magnitude	Normalized DSK Magnitude
0	0	1	1
0	45	0.9237	0.9248
0	90	0.7071	0.7064
0	180	0	0 (flatline)
26	263	0.4771	0.4872

The table above shows good agreement between the matlab magnitudes vs. phase difference of the two cosines, but what about the resulting phase of the interference pattern. It is hard to see phase on the scope without a trigger source, thus the phase will just be discussed intuitively. The phase of the resulting interference pattern is just the average of the two original cosine phases.

$$\theta_{\text{int}\,erference} = \frac{\theta_1 + \theta_2}{2} \qquad (10)$$

This observation was confirmed by looking at the bottom right graph of the matlab output of interference_pattern.m.

Assignment 5:

The task at hand for this assignment was to modify the program interference.c such that it will implement the following equation:

$$A\cos(\omega t) + A\cos(\omega t + 1\theta) + A\cos(\omega t + 2\theta) + \ldots + A\cos(\omega t + (N-1)\theta) \qquad (11)$$

Figure 12 (below) shows the code from interference.c modified per the above equation.

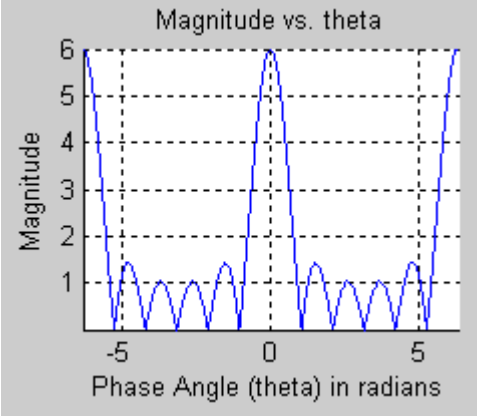
```
1.  // This project uses support files generated by Rulph Chassaing
2.  // Comm routines included in C6xdskinit_pcm.c
3.
4.  #include <math.h>
5.  #define PI 3.14159265359                  // define the constant PI
6.  short sample_period=12;                   // sinusoid period in samples
```

```
7.  short ctr;                          // loop counter
8.  short phase = 45;                   // theta in degrees (holds two values)
9.  float out_value;                    // value sent to codec
10. float angle;                        // angle for cosine function
11. float wnts;                         // omega * n * ts - used for current angle
12. float Fs = 24000.0;                                   // irrelevant since jumper in 3-4
13. short N = 3;            // index for arbitrary number of cosines
14. short Amp = 5000;       // amplitude of each cosine
15.
16. interrupt void c_int11()            // interrupt service routine
17. {
18. int i;                          // used in for loop
19. // create interference pattern
20. // use amplitude 30000/N to prevent overflow in codec, where N=2
21. wnts = 2.0*PI*ctr/sample_period;            // current angle (w/out phase)
22. out_value=0;
23. for (i=0; i<N; i++)
24. {
25. angle = wnts + i*phase*PI/180;      // current angle (with phase)
26. out_value += Amp*cos(angle);        // cos(w nts + theta_k) k in {1,2}
27. }
28.
29. //output interference pattern
30. output_left_sample((short)out_value);           // output each sine value
31. if (ctr < sample_period-1) ++ctr;           // increment counter (0 through 47)
32. else ctr = 0;                       // reset counter
33. return;                             // return from interrupt
34. }
35.
36. void main()
37. {
38. ctr=0;                          // initialize counter
39. comm_intr();                        // initialize DSK, codec, McBSP
40. while(1);                           // wait for an interrupt to occur
41. }
```

Figure 12 Line numbered code for interference.c modified to implement Equation (11)

The modifications to the interference.c program to create the sum of N cosine waves with amplitude A and N even spaced phases are as follows. The array variables *phase[0]* and *phase[1]* were removed and a single phase variable was created (line 8). A new variable *N* was declared and used in line 23 to build the sum of the N cosines. Additionally, in line 25, the angle was adjusted to use the loop index *i* for calculating the phase of the sinusoid. Lastly, a new amplitude variable Amp was declared in line 14 and was programmed at a value low enough to avoid overdriving the codec.

When observing the DSK output of this program with N=6, the amplitude on the scope vs. phase (θ) corresponded to the magnitude vs. θ plot of figure 11 in the lab 2 notes. Specifically, when θ was set to 60 degrees, the scope flat-lined and this corresponded to the first magnitude null in the figure 11 plot. And, when Q was set to 87 degrees a relative, local maximum was observed on the scope. This corresponded to the first peak in the figure 11 plot.



Assignment 6:

The program below defines two 'beat' frequencies f1 and f2 which will be summed together. Every time a interrupt is generated, two independent angles and offsets are calculated for the two sinusoids of different frequencies. These sinusoids are then summed together and outputted to the codec.

When the codec's output is observed on the oscilloscope, an amplitude pulsating sine wave is observed. The rate of the pulsation increases as the difference between the two frequencies f1 and f2 is increased. When listening to earphones plugged into the output, the result is akin to the throbbing sound heard from a passing dual engine airplane.

beat.c

```
1.  // This project uses support files generated by Rulph Chassaing
2.  // Comm routines included in C6xdskinit_pcm.c
3.
4.  #include <math.h>
5.  #define PI 3.14159265359          // define the constant PI
6.  float f1=200;                    // freq of 1st sinusoid for beat pattern
7.  float f2=205;                    // freq of 2nd sinusoid for beat pattern
8.  float out_value;                 // value sent to codec
9.  float angle1;                   // angle for 1st sinusoid
10. float angle2;                   // angle for 2nd sinusoid
11. float offset1=0;                // offset for 1st sinusoid
12. float offset2=0;                // offset for 2nd sinusoid
13. float Fs = 24000.0;                                          // irrelevant since jumper in 3-4
14. short Amp = 15000;              // amplitude of each sinusoid
15.
16. interrupt void c_int11()        // interrupt service routine
17. {
18. offset1 = 2.0*PI*f1/Fs;    //setup the 1st sinusoid angle
19. angle1=angle1+offset1;
20.
21. offset2 = 2.0*PI*f2/Fs;   //setup the 2nd sinusoid angle
22. angle2=angle2+offset2;
23.
24. out_value = Amp*cos(angle1)+Amp*cos(angle2);  //sum the 2 sinusoids of frequencies f1 and f2
25.
```

```
26. // output interference pattern
27. output_left_sample((short)out_value);   // output each sine value
28. if (angle1 > 2*PI)                  // reset angle1 if > 2*PI
29. angle1 -= 2*PI;                    // angle = angle1 - 2*PI
30. if (angle2 > 2*PI)                  // reset angle2 if > 2*PI
31. angle2 -= 2*PI;                    // angle = angle2 - 2*PI
32. return;                          // return from interrupt
33. }
34.
35. void main()
36. {
37. comm_intr();                          // initialize DSK, codec, McBSP
38. while(1);                            // wait for an interrupt to occur
39. }
```

Figure 13 Line numbered code for beat.c

Conclusion

Assignment 1 demonstrates how a low pass filter following a codec can cut out the higher frequency components of a signal. This is most apparent when sampling a square wave and only the fundamental fourier series frequency of the square wave is passed resulting in a pure sine wave output.

Assignment 2a shows how the straight wire.c code can be modified with a software gain value such that unity gain is achieved from input to output of the codec. When the gain was adjusted to a value of 2 for assignment 2b, the output would begin clipping with an input of one half the maximum output value. This shows that proper gain levels through each level must be properly maintained through each gain stage. Assignment 2c shows that implementing a simple difference equation of the current sample minus the previous sample will create a high pass filter. If one imagines the input moving infinitely slowly, then the difference will be zero, if one imagines the input and output moving rapidly (HighFrequency) between samples, then the difference (IE output) will be large.

Assignment 3 shows that when the real and imaginary components of a rotating phasor are displayed on a oscilloscope in XY mode, a circle will be generated.

Assignment 4 shows that a Lissajous figure (ellipse) could be generated by modifying the rotating phasor code in assignment 3 to include different amplitude and phases for the real and imaginary components of the rotating phasor.

Assignment 5 demonstrates that constructive interference could be obtained with the addition of six sinusoids of a given frequency and phase offset. This effect is similar to the gain increase of a phase array antenna using quarter wave elements and proper spacing or time delay for the phase addition.

Assignment 6 shows that the addition of two frequencies will produce the original two frequencies plus the sum and difference of the original frequencies. This effect is similar to a mixer circuit used in a heterodyne circuit. The difference frequency produces the beating effect. The frequency of the beating decreases as the two frequencies approach each other. This effect is used to tune a guitar string to a pitch fork by adjusting the tension of the guitar string while minimizing the audio beating sound between the pitch fork and guitar string.