REAL-TIME DSP LABORATORY2:
Signals and Systems on the TMS320C6713 DSK

Contents

Note: Starred sections contain assigned tasks to be written up in the report.

1 Introduction

In this lab, we lay the foundations for implementing digital filters in DSP hardware. The trick is to digitize analog signals, filter them digitally, and then convert them back to analog. Therefore we will study

- I/O via the codec,
- system implementations, and
- coding and implementing algorithms using complex numbers.

2 Sampling And Reconstruction

The concepts of sampling, reconstruction, and rate conversion have been covered in your undergraduate course in *Signals and Systems*. If you are unfamiliar with these concepts, refer to [3] on the web. References [6] and [8] offer a more thorough treatment of this subject and are often used in undergraduate and graduate DSP courses.

2.1 Discrete-Time Systems

A discrete-time signal is a sequence of complex number $\{x[n], n = 0, \pm 1, \pm 2, \ldots\}$. If $x[n]$ is obtained from samples of a continuous-time signal $x(t)$ every t_s seconds, then we use the notation $x[n] = x(nt_s)$.

Sampling

A simplified version of sampling a continuous-time signal is shown in Figure 1 (adapted from [3]).

Figure 1: Continuous Time to Discrete-Time Conversion

Here, a continuous-time signal enters an analog lowpass filter (LPF), which is referred to as an *anti-aliasing* filter. The purpose of this is to essentially bandlimit the signal to the Nyquist frequency or half-sampling frequency ($f_s/2$). This essentially bandlimited signal is then sampled at rate $f_s = 1/t_s$ to produce a discrete-time signal. This means that the high frequency artifacts of the continuous-time signal will be lost. This is the cost of managing aliasing. In practice, the anti-aliasing filter will have a cutoff frequency at roughly 90% of the Nyquist frequency and use the other 10% for roll-off [3]. For a sampling rate of $f_s = 8$KHz, an anti-aliasing filter with cutoff frequency 3.6KHz, which is 90% of $f_s/2$, would be appropriate. Later we will see that the DSK uses a more sophisticated sampling scheme, called delta-sigma conversion; this simplifies the design of the analog anti-aliasing filter at the input to the ADC, as well as the smoothing filter at the output of the DAC.

2.2 Digital Systems

A digital signal is a discrete-time signal that has been resolved to a finite set of amplitudes (quantized) and assigned a bit pattern known as a *finite-length word*[1]. This process of converting a continuous-time signal into a collection of finite-length words is referred to as *coding*. In the case of the on-board codec, a sampled continuous-time signal is coded into 16-bit finite-length words at the rate of one word each .125ms for a word rate of 8KHz. (NB: This idea of coding is a "symbol-to-bit" mapping. We will return to this idea in a future communications lab.) See Figure 2 below to get an idea of how this works.

When we refer to *digital* signals, we think of discrete values at discrete instances of time, but what we have in hardware is a set of bits. In the case of Figure 2, we have the following sequence of 3-bit finite-length words: `000`, `000`, `001`, `001`, `110`, `101`, `011`, `010`, `111`, `000`, `000`.

[1] In the context of TI's C6000 series of DSPs, we refer to 32-bits of data as a *word*, 16-bits as a *half-word*, and 8-bits as a *byte*. In terms of digital signals, we will refer to quantized samples as *finite-length words* regardless of the number of bits in each finite-length word.

2.3 Reconstruction

In real-time DSP, the digital to analog reconstruction (decoding) conceptually goes as follows: the sequence of digital words enters the DAC, which outputs an analog stair-step changing at the sample rate f_s. A DAC which operates this way is called a zero-order-hold DAC. Finally, an analog LPF is used to smooth the stair-step function.

2.4 Mixed Analog And Discrete-Time Systems

In DSP, we have in general two types of inputs (analog and digital) and two types of outputs (analog and digital). Figure 3 below illustrates this. DSP uses all three blocks to achieve the effects of the fourth: code the input signal with the ADC (SW block), digital filter the digitized signal (SE block), and reconstruct the filtered analog signal with the DAC (NE block). This achieves the effects of an analog filter in the NW block.

In this lab, we have explored a system that had an analog input (signal generator) and an analog output (oscilloscope) using a codec. For these types of systems, we will always use a codec to interface the DSP chip. In the last lab, we saw an example of a digital input (samples of a sine wave stored in a table) that we sent through the codec to our analog output (either speakers or oscilloscope). In future labs, we will study experiments where we have an analog input which we will store digitally (digital recording). These three types of systems will be implemented under the real time constraints that we explored in the last lab. The fourth type of system (digital input/digital output) will not be implemented by itself in real time, but rather within a system that communicates in real-time with the analog world. An example of this would be to read in an analog signal, convert it to digital (analog in, digital out), process the digital signal (digital in, digital out), then output the signal via the codec (digital in, analog out).

3 TLV320AIC23 Stereo Audio CODEC

Located onboard the DSK is the TLV320AIC23 codec (coder/decoder) [**?**]. The coder part is implemented using a sampler, delta-sigma modulator, and a decimation filter. The output rate from the coder is 8000 to 96000 words per second (f_s = 8KHz to f_s = 96KHz), with word size of 16,20,24, and 32 bits. In this course, we will configure the codec to use 16 bit word size.

3.1 Sampling

Internally, the sampler is over-sampling at rate 250-384 times f_s, depending on f_s. This is a common practice when using delta-sigma modulators. A delta-sigma modulator takes the current sample and compares it with the previous sample. If the current sample is larger (or smaller) than the previous sample, it makes the current sample the previous sample plus (or minus) some fixed amplitude δ. In our case, we have a 1-bit delta-sigma modulator which means that the current sample (16-bit finite-length word) will be the previous sample plus (or minus) a binary 1 assuming that the current sample is larger (or smaller) than the previous sample. By

over-sampling, a good representation of the original signal is captured at each sample point, since the signal, which is essentially band-limited, will not change by more than 1 bit, or quantization level, between each sample point.

An additional advantage of over-sampling is that images of the input signal spectrum created by sampling are shifted to very high frequencies. For example, with $f_s = 8$KHz and an oversampling ratio of 250, the first image occurs at 2MHz, so the anti-aliasing filter need only rolloff at 90% of 1MHz, which is much easier to implement than a filter for a Nyquist-rate converter (sampling ratio=1), which would require a 4KHz rolloff.

Only every 250th to 384th sample point is needed to accurately represent the bandlimited signal, so a decimation filter (LPF running at 8KHz) is used to decrease the word rate to the desired rate of 8KHz. These 16-bit finite-length words are then transmitted to the CPU via serial port 0 on the DSP chip. NB: The CPU will treat these 16-bit finite length words as 16-bit signed integers (-32768 to +32767).

3.2 Reconstruction

For reconstruction of the discrete-time signal, the block diagram of Figure 4 is implemented. (NB: This method is commonly used in CD players and is also used in the DSK codec. See reference [3].)

In Figure 4, the discrete-time signal is artificially increased in rate by a factor of M (zero-fill upsampling). Here $M = 250$ to 384, as with the coder (ADC). Then, a high rate discrete-time LPF, $P(z)$, is used to interpolate the $M-1$ newly inserted samples. Next, the signal is converted from a discrete-time signal to a continuous-time signal by a DAC and finally, a smoothing filter (*anti-imaging* filter) is used to create a continuous-time signal that was represented by the samples $x[n]$ Usually, the smoothing filter will be a LPF with the same characteristics as the anti-aliasing filter used previously. As with the ADC, the higher output rate of the DAC, combined with the interpolating filter, considerably simplifies the analog LPF (the "smoothing filter") on the output of the DAC.

The operation of this practical method for reconstruction is illustrated in Figure 5 below. For a mathematical description of digital-to-analog reconstruction of ideal bandlimited signals, see [3].

4 Sampling and Reconstruction using the Codec: A Programming Example

Real-time digital signal processing involves logical operations on an incoming stream of finite-length words to produce a processed output stream. Figure 6 illustrates this idea.

In most DSP applications, the *DSP core* of Figure 6 will contain local program and data memory on the chip that is used to store the algorithm being implemented, local and global variables, and processed data. In the first experiment, we will implement the "DSP core" block as a straight wire, which means that only an identity map will be implemented between the input and output of the DSP core. No local or global variables or processed data will be stored in memory, and no signal processing will be done. Only the program instructions will be stored in the program memory.

4.1 DSP Core as a Straight Wire Using the Codec*

Create a new workspace for the Lab02 projects and create a project for the straight wire exercise by cloning a sine_gen project. Rename this project with a descriptive name, e.g. u:\DSP\lab_02\straight_wire). Replace the file sine_gen*.c file with the file `straight_wire.c` from the class webpage. See Figure 7 for a listing of `straight_wire.c`.

Open the file `straight_wire.c`. Notice that the file has only the global variable f_s, no local variables, and that whatever is read in from the codec is immediately sent back to the codec via the command
`output_sample(input_sample());`.
What about type declarations? The declarations for `output_sample()` and `input_sample()` are contained in the header file `DSK6713_aic23.h`.

Compile and load this program onto the DSK and run it. Connect the signal generator to the left channel (white BNC connector) of one stereo audio cable and plug the other end into the *line-in* jack on the DSK (`J303`). Plug the second cable into the *line-out* jack on the DSK (`J304`) and both BNC connectors into the oscilloscope, with the white BNC in INPUT 1 out-jack on the DSK. Set the signal generator to a 1KHz sinusoid with amplitude 1Vpp (peak-to-peak voltage), and observe a 1KHz sinusoid on INPUT 1 of the oscilloscope. Now switch the signal generator to the *red* BNC connector and notice that the output signal moves to INPUT 2 of the oscilloscope.

Notice that in both cases the actual peak-to-peak voltage of the signal is greater than 1V (about 1.8V). This is due to the fact that the resistance of the DSK is greater than the resistance to which the signal generator has been matched[2].

Change the peak-to-peak voltage and the frequency on the signal generator and observe the effects on the oscilloscope. If the voltage input to the codec exceeds 3.3V, then the codec will be overdriven. When the codec is overdriven, the output on the codec will reveal either amplitude

[2] To see this, consider the Thevenin equivalent circuit of a signal generator, with internal resistance 50Ω matched to the load R_L. Then the output will be $V_o = \frac{R_L}{50+R_L} V_T$, where V_T is the actual (Thevenin) voltage supplied by the signal generator and V_o is the voltage that you set on the signal generator. Now, if $R_L = 50\Omega$, which is what the signal generator expects, then $V_o = \frac{1}{2} V_T$, which means that $V_T = 2V_o$. In the case of the on-board codec, $R_L \gg 50\Omega$, so the actual output voltage, V, is $V \approx V_T = 2V_o$.

clipping of the input sine wave, or two's complement overflow effects (a sudden change in sign or some other weird effect). If the voltage input is too small (around 100mV or less), then coupling effects from the internal hardware will be observed. For frequency, keep in mind that there is a lowpass filter that cuts off frequencies above 4KHz. Vary the frequency of the sinusoid on the signal generator and observe that sinusoids above 4KHz are suppressed. (NB: You should notice that sinusoids around 4KHz are decreased in amplitude, but not fully suppressed. This is due to the fact that the anti-aliasing filter is not ideal and therefore has a non-zero roll-off around 4KHz.)

Assignment

1. To observe the effects of the anti-aliasing filter, use a square wave as the input. Use square waves between 200Hz and 2KHz. Use your knowledge of the Fourier series expansion of a square wave to explain what you see. Then use the lowest range on the signal generator and input a sine wave in the range of 10-50Hz. What does the shape of the wave suggest regarding how the signal is coupled into the codec input?Also, listen to this signal and explain what you hear. Give as much intuition as possible. For a review of Fourier series, see [5].

Assignment

2. Make the following modifications to the program `straight_wire.c`. Comment on your results and include the code used.

   (a) Introduce a global variable to store the incoming sample and then send the sample to the codec. Use a 500mV_{pp}, whose frequency is less than 2.5KHz. Observe the sinusoid on the oscilloscope and record the output peak-to-peak voltage. Use this value for the value of V_{pp}^{ug} in part c. NB: the ug superscript stands for *unity gain*.

   (b) Multiply the current input sample by some non-trivial scalar such as 2. How does this affect the range of input voltages that you can use on the signal generator and not overdrive the codec?

   (c) Create another global variable and use it to store the previously read sample. Now, output the current sample (unscaled) minus the previous sample (unscaled). Use sinusoids with a 500mV_{pp} as your input. This sure ensure that you do not overdrive the codec (e.g. if the input is a 2KHz sinusoid, the output voltage should be twice the "unity gain" voltage from part a). Include a copy of your C code and explain your code line-by-line. How does this filter affect the amplitude of sinusoids at various frequencies? Sketch the Bode plot for this filter (HINT: Plot the (scaled) output voltage, $\text{V}_{pp}/\text{V}_{pp}^{ug}$ versus the sinusoid frequency in Hertz). What type of filter is this? NB: This is the implementation of the difference equation $y[n] = x[n] - x[n-1]$, which corresponds to the FIR filter $H(z) = 1 - z^{-1}$. Calculate and plot the the magnitude of the frequency response $H(e^{j2\pi ft_s}) = 1 - e^{-j2\pi ft_s}$ versus frequency in Hertz. How does this compare to the Bode plot from before? Explain the discrepancies.

4.2 Processing Two Data Channels

In the previous section, we saw that the output of the codec when using function `output_sample()` is a 32-bit binary number that contains two 16-bit finite-length words: one for the left channel (least significant 16-bits) and one for the right channel (most significant 16-bits). In this section, we look at functions which allow us to manipulate the left and right channels individually and do operations such as switch the left input channel to the right output channel.

The files `dsk6713_aic23.h` and `dsk6713_bsl.lib` contain the prototypes and implementations of a number of functions used to interface the DSP chip to the codec:

- `input_left_sample()`: Reads the 16-bit finite-length word from the codec containing samples of the left channel.
- `input_right_sample()`: Reads the 16-bit finite-length word from the codec containing samples of the right channel.
- `input_sample()`: Reads a 32-bit binary number from the codec, 16 bits for each channel.
- `output_left_sample(short out_data)`: Sends the 16-bit signed integer `out_data` to the codec, containing samples for the left channel only.
- `output_right_sample(short out_data)`: Sends the 16-bit signed integer `out_data` to the codec, containing samples for the right channel only.
- `output_sample(int out_data)`: Sends the 32-bit binary number `out_data` to the codec, containing a 16-bit sample for each channel. Note: To use `output_sample()`, you have to do the work of correctly packing two 16-bit words (one for each channel) into one 32-bit word. In most applications, it will be easier to use the next function.
- `output_left_right_sample(short left_channel, short right_channel)`: Takes the 16-bit binary numbers in left_channel and right_channel and correctly packs them into a 32-bit binary number, which is sent to the codec.

The terms *signed integer* and *finite-length word* are used interchangeably, since each 16-bit finite-length word is coded as a 16-bit signed integer of type `short`. We will continue to use these terms interchangeably through this course.

4.2.1 Coding One Complex Phasor On Two Real Channels

Often times in electrical engineering, we use complex signals to represent systems. This is done to help with intuition and to give a clean and concise representation of what is actually happening. In practice, we implement these complex representations by using two channels, which represent the real and imaginary parts of the complex signal. Euler's identity is $e^{j\theta} = \cos(\theta) + j\sin(\theta)$, which is the equivalent polar/Cartesian representation of a complex number with magnitude 1 and phase θ. This can be generalized to a rotating phasor (time-varying complex number) of the form $re^{j\omega t} = r\cos(\omega t) + jr\sin(\omega t)$, which has magnitude r and is rotating counterclockwise at rate ω radians per second in the complex plane. If ω is negative, then it rotates clockwise in the complex plane. The complex plane is a plane where the horizontal axis represents the real part of the complex number and the vertical axis represents the imaginary part of the complex number (see Figure 8 below). Real numbers lie on the real axis (imaginary part = 0) and imaginary numbers lie on the vertical axis (real part = 0). The field of real numbers and the field of imaginary numbers are just subsets of the field of complex numbers. See [2] and [4] on the webpage for a more complete discussion of complex numbers and rotating phasors.

4.2.2 Rotating Phasor Example *

Create a new project `rotating_phasor` and replace the top-level C source file with the file `rp_100Hz.c`) from the webpage. Examine the file `rp_100Hz.c`.

Line 6 of Figure 9 defines a structure named `COMPLEX`. This structure contains two float variables, `real` and `imag`, which are used to store the real and imaginary parts of a complex number. The keyword `typedef` is short for *type definition* and `struct` is short for *structure*. The keywords `typedef` and `struct` are standard C commands and are recognized by any ANSI C compiler. See [7] for a more in-depth explanation. Line 10 creates the global variable `phasor`, which is defined by the structure `COMPLEX`. This program implements the following rotating phasor: $e^{j2\pi 100t} = \cos(2\pi 100t) + j\sin(2\pi 100t)$, where the real part of $e^{j2\pi 100t}$ is $\cos(2\pi 100t)$ and the imaginary part is $\sin(2\pi 100t)$. This phasor is generated in C code by sending samples of $e^{j\theta_o n}$, $\theta_o = 2\pi\frac{f_o}{f_s}$, to the codec at rate $f_s = 24$KHz. In this example, $\theta_o = 2\pi\frac{f_o}{f_s} = \frac{2\pi}{N}$, where N is the value stored in the variable `sample_N=240` (see line 7 in Figure 9). This means that we are generating a rotating phasor that is rotating at rate $f_o = \frac{f_s}{N} = \frac{24000}{240} = 100$Hz, hence the name `rp_100Hz`. Lines 16 and 17 in Figure 9 show how each item (or channel) in the two channel variable `phasor` is accessed. Line 18 sends each channel to the codec. The input arguments to the function `output_left_right_sample()` are typecast as `(short)`. Typecasting converts the two values `phasor.real` and `phasor.imag` from single-precision floating point numbers to 16-bit signed integers. This is needed since the codec is expecting two 16-bit signed integers (one for each channel). The function `output_left_right_sample()` will pack the two 16-bit signed integers into a single 32-bit binary number that will be sent to the codec.

Compile and load this program onto the DSK. Plug the stereo jack into the out-jack on the DSK and connect the two BNC plugs to the two inputs on the oscilloscope. Observe two sinusoids 90^o out of phase on the oscilloscope. Once you have these, set up the '`xy`' deflect mode on the oscilloscope. You should see a circle.

The x and y channels, $x =$ `phasor.real` and $y =$ `phasor.imag`, are exactly the inphase and quadrature signals that are used in a quadrature modulator or demodulator. We will study these in a lab on modulation and demodulation.

Assignment

3. A Lissajous figure is a time-varying complex number of the form $z(t) = A_1\cos(\omega t + \phi_1) + jA_2\cos(\omega t + \phi_2)$. Download the demo `lissajous.m` from the webpage and run it in MATLAB to get a feel for how Lissajous figures are created. Create a project in CCS that produces a Lissajous figure when viewed on an oscilloscope in the '`xy`' mode. (NB: You can copy the files from `rp_100Hz` into a new directory and then just edit the C code in `rp_100Hz.c` to save time.) Give the discrete-time equation for $z[n]$ that you are generating in hardware. Comment on how $z[n]$ is coded in hardware and how this will generate a Lissajous figure. Test your program in both MATLAB and on hardware for the following specifications: $\omega = 2\pi 100$ (as before), $A_1 = 2$, $A_2 = 1$, $\phi_1 = 0$, and $\phi_2 = 45^o = \pi/4$ rad/sec. Choose other interesting values of ω, A_1, A_2, ϕ_1, and ϕ_2 and implement the Lissajous figure using both MATLAB (software) and the DSK board (hardware).

4.2.3 Interference Patterns*

Two or more signals may be added at a linear sensor or antenna. Depending on the relative phase, or time alignment, of the signals, they may add constructively or destructively. A fading radio transmission is a consequence of destructive interference of signals that are misaligned in time by virtue of multi-path propagation. A large gain in a filter is a consequence of constructive interference of signals that are aligned in time [4].

Let's elaborate by considering the addition, or interference, of two signals of the same amplitude and frequency, but different phases (or delays). Consider $x(t) = A\cos(\omega t + \theta_1) + A\cos(\omega t + \theta_2)$. It is easy to show using phasors and simple algebra that $x(t) = A\cos(\omega t + \theta_1) + A\cos(\omega t + \theta_2) = 2A\cos(\psi/2)\cos(\omega t + \theta)$ where $\theta = \frac{\theta_1 + \theta_2}{2}$ is the average phase and $\psi = \theta_1 - \theta_2$ is the phase difference. The interference effect is this: when $\psi = 0$, then $\cos(\psi/2) = 1$ and the output, $x(t)$, is doubled in amplitude. For $\psi = \pi$, then $\cos(\psi/2) = 0$ and the output is zeroed. For in-between values of the phase difference, the gain is between 0 and 2. Download the MATLAB file `interference_pattern.m` from the webpage for this lab, and run it to get a feel for how these interference patterns appear for various phase differences [4].

Create a project `interference` and copy in the C source code file `interference.c` from the webpage and open it in CCS. Examine the listing of `interference.c` in Figure 10. This project generates an interference pattern for the sum of two cosines with frequency 2KHz and initial phases $\theta_1 = \pi/4$ and $\theta_2 = 0$.

Assignment

4. Study the code in `interference.c`. Using the help menu in CCS, explain the purpose of the following lines of code: 8, 16, 20, 21, 22 through 26, 36, and 37. Give the discrete-time equation for the interference pattern that is being generated in this program. Build and run this project on the DSK. Download the MATLAB file `interference_pattern.m` and run it with $\theta_1 = 45^o$ and $\theta_2 = 0^o$. Refer to the bottom two graphs of the output. These graphs give the relative amplitude and phase of the interference pattern for different phase differences $\psi = \theta_1 - \theta_2$. Use a Watch Window to change the phases θ_1 and θ_2. In order to access θ_1 and θ_2 in a watch window, enter the variable names `phase[0]` and `phase[1]`, respectively. Change the values of θ_1 and θ_2 and used the phase difference to make sense of the bottom two graphs given by `interference_pattern.m`. Do these graphs accurately represent how the phase difference affects the amplitude and phase of the interference pattern? Give as much intuition as possible.

We now wish to generalize the previous development by considering the interference of N signals, evenly phased in time:

$$x(t) = A\cos(\omega t) + A\cos(\omega t + \theta) + A\cos(\omega t + 2\theta) + \ldots + A\cos(\omega t + (N-1)\theta) \tag{1}$$

Using simple algebra, phasors, and a finite geometric sum formula, it can be shown that $x(t)$ is

$$x(t) = A\sum_{k=0}^{N-1}\cos(\omega t + k\theta) = A\frac{\sin(N\theta/2)}{\sin(\theta/2)}\cos(\omega t + \frac{(N-1)}{2}\theta). \tag{2}$$

Figure 11 shows the sum of 6 evenly phased cosines for $\theta = 0, \pi/4, \pi/2, 3\pi/4, \pi, 5\pi/4, 3\pi/2$, and $7\pi/4$, and the interference pattern (amplitude and phase offset) associated with a given θ.

Assignment

5. Modify the previous program `interference.c` so that it implements eqn(1) for an arbitrary N and θ. Give the discrete-time equation for the interference pattern that you will be generating. Test your program with $N = 6$ and $\theta = \pi/4$. Use a Watch Window to change N and θ. Be sure not to overdrive the codec (see lines 17 and 21 of `interference.c`). Note what changes were made to `interference.c` and comment on your results. Pay particular attention to the output when $N = 6$. Relate the output that you see on the oscilloscope for various values of theta with the bottom two graphs of Figure 11.

   (NB: These kind of interference patterns are observed when multiple antennas, which are evenly spaced, are used at a radar or communication receiver.)

4.2.4 Beating Between Tones*

In the section on interference, we dealt with signals that had the same amplitude and angular frequency, but different phases. Now, we are going to allow the angular frequencies to differ and study the phenomenon of beating. As with the interference experiment, we are going to assume that the amplitudes of the signals are equal. Then the sum of two signals with different frequencies is equal to $y(t) = A\cos(\omega_1 t) + A\cos(\omega_2 t)$ [4]. Using simple algebra and phasors, the following formula may be derived

$$y(t) = A\cos(\omega_1 t) + A\cos(\omega_2 t) = 2A\cos(\frac{\omega_1 - \omega_2}{2}t)\cos(\frac{\omega_1 + \omega_2}{2}t). \tag{3}$$

Equation (3) shows that the sum of two cosines with frequencies ω_1 and ω_2 is equivalent to the product of two cosines. The frequency of one cosine is the average of the two frequencies $\frac{\omega_1+\omega_2}{2}$ and the frequency of the other is the difference, or beat, frequency $\frac{\omega_1-\omega_2}{2}$. Download the file `beating_tones.m` from the webpage and run it in MATLAB to see various examples of the beating between tones [4]. This can be simulated on the DSK by generating two sinusoids in C code with have different frequencies and observing the output on the oscilloscope.

Assignment

6. Create a C program that generates a beating tone pattern by summing up two cosines of the same amplitude but with different frequencies. Give the discrete-time equation for the beating pattern that you will be generating in this program. Use a Watch Window to change the frequency of each of your cosines. Observe the beat pattern on the oscilloscope and listen to the output for various beating patterns. Comment on your results. Give as much intuition as possible. (HINT: It is easier for the human ear to hear the beating between tones at lower frequencies. Try listening to the beat pattern from summing a 200Hz cosine and a 205Hz cosine.)

5 End Notes

We have now laid the foundation for digital signal processing in DSP hardware. In the next lab, we will explore the DSP assembly code, and in subsequent labs, we will proceed to implement various Finite Impulse Response (FIR) filters, Infinite Impulse Response (IIR) filters, and Fast Fourier Transf6orm pairs (FFT and IFFT pairs). These filters and transforms will become invaluable tools for designing, analyzing, and simulating communication systems in future labs.

References

[1] Rulph Chassaing. *DSP Applications Using C and the TMS320C6x DSK*. Wiley, New York, 2002.

[2] Colorado State University, Fort Collins, CO. *Signals and Systems Laboratory 1: Introduction to MATLAB and Complex Numbers*, 2001.

[3] Colorado State University, Fort Collins, CO. *Signals and Systems Laboratory 10: Sampling, Reconstruction, and Rate Conversion*, 2001.

[4] Colorado State University, Fort Collins, CO. *Signals and Systems Laboratory 2: Phasors, Interference, Beating Between Tones, and Lissajous Figures*, 2001.

[5] Colorado State University, Fort Collins, CO. *Signals and Systems Laboratory 5: Periodic Signals and Fourier Series*, 2001.

[6] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing.* Prentice Hall, Uper Saddle River, NJ, 1989.

[7] Greg Perry. *Absolute Beginners Guide to C.* Pearson Education, 1984.

[8] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications.* Prentice Hall, Uper Saddle River, NJ, 1996.

[9] Steven A. Tretter. *Communication Design Using DSP Algorithms: With Laboratory Experiments for the TMS320C6701 and TMS320C6711.* Kluwer Academic/Plenum Publishers, New York, 2003.

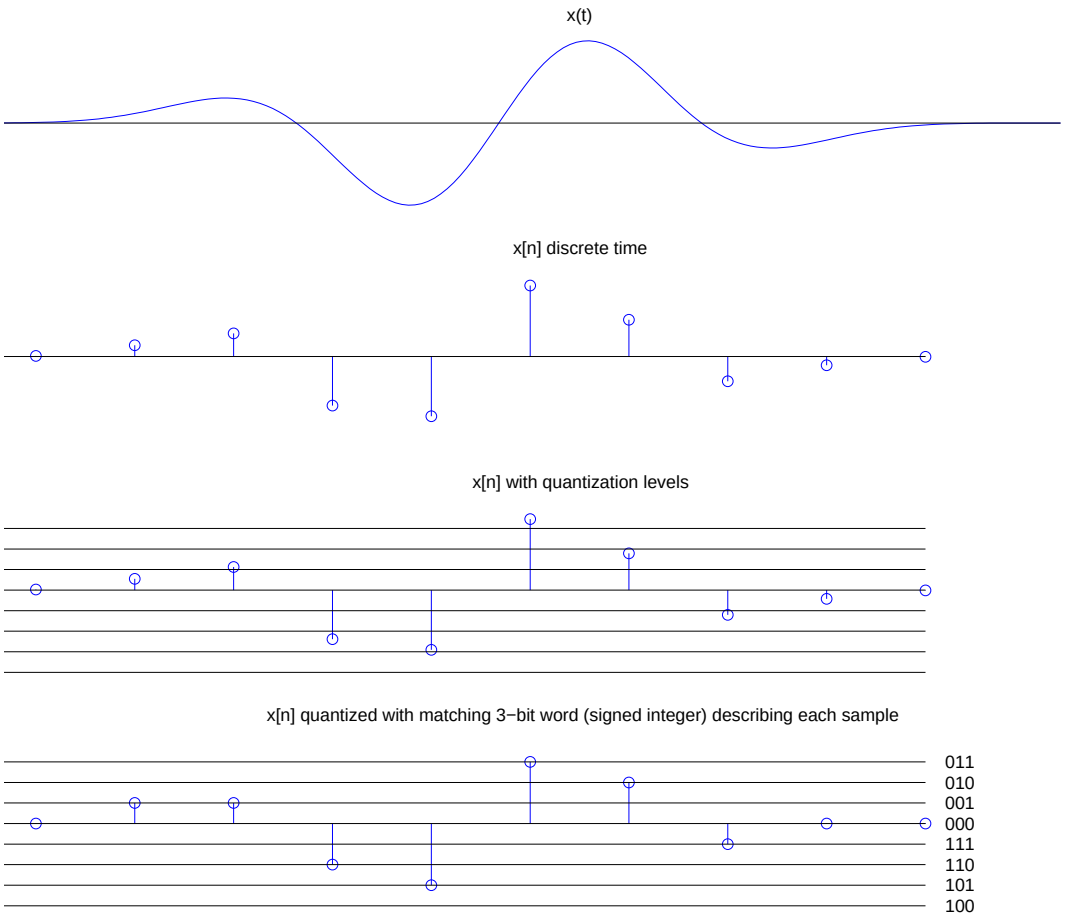

Figure 2: Analog to Digital Conversion with 2's compliment 3-bit signed integer assignments

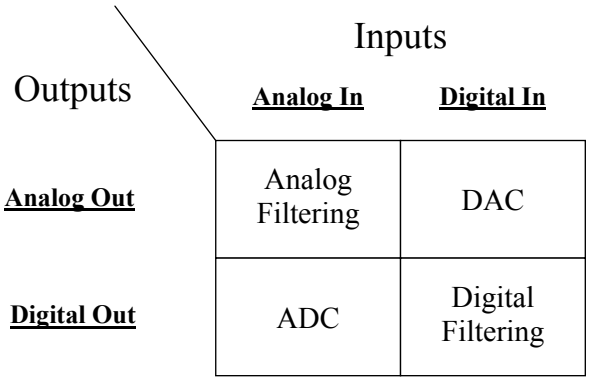

Figure 3: Mixed Analog and Digital Systems

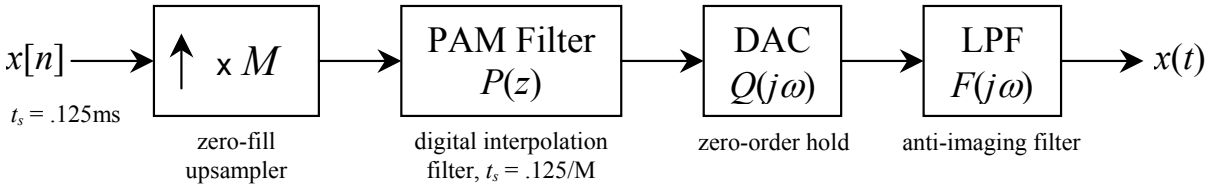

Figure 4: Discrete-Time to Continuous Time Reconstruction

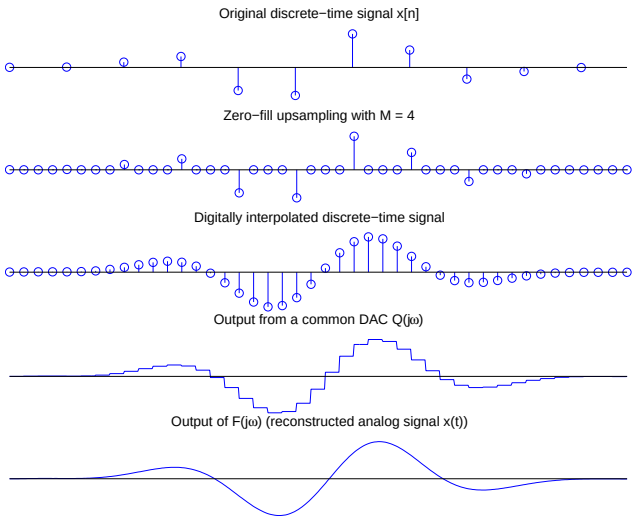

Figure 5: Analog Reconstruction of a Discrete-time Signal

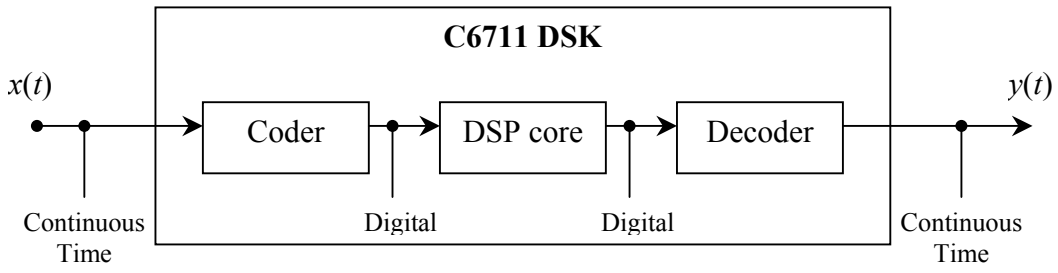

Figure 6: DSP Block Diagram

```
/*1 */ // This project uses support files generated by Rulph Chassaing
/*2 */ // Comm routines included in C6xdskinit.c
/*3 */
/*4 */ #include "dsk6713_aic23.h"
/*5 */ Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;    // sampling frequency of codec
/*6 */
/*7 */ interrupt void c_int11()             // interrupt service routine
/*8 */ {
/*9 */  output_sample(input_sample());
/*10*/  return;                             //return from interrupt
/*11*/ }
/*12*/
/*13*/ void main()
/*14*/ {
/*15*/   comm_intr();                       // initialize DSK, codec, McBSP for interrupts
/*16*/   while(1);                          // wait for an interrupt to occur
/*17*/ }
```

Figure 7: Listing of `straight_wire.c`

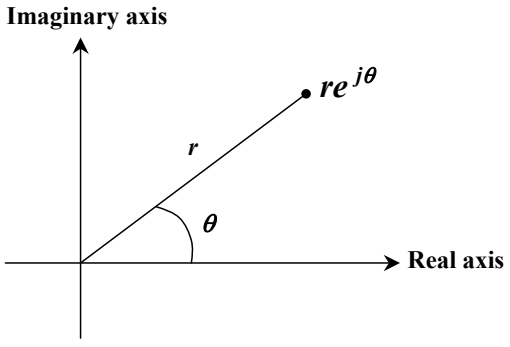

Figure 8: $re^{j\theta}$ on the Complex Plane

```
/*1 */ // This project uses support files generated by Rulph Chassaing
/*2 */ // Comm routines included in dsk6713_bsl.lib
/*3 */ #include "dsk6713_aic23.h"
/*4 */ #include <math.h>
/*5 */ #define PI 3.14159265359                          // define the constant PI
/*6 */ typedef struct {float real,imag;} COMPLEX;
/*7 */                                                  // define complex number exp_value
/*8 */ short sample_period=240;                         // sinusoid period in samples
/*9 */ short ctr;                                       // loop counter
/*10*/ float angle;                                     // angle for cosine function
/*11*/ COMPLEX phasor;
/*12*/ Uint32 fs=DSK6713_AIC23_FREQ_24KHZ               // Define sampling frequency
/*13*/
/*14*/ interrupt void c_int11()                         // interrupt service routine
/*15*/ {
/*16*/  angle = 2.0*PI*ctr/sample_period;
/*17*/  phasor.real=20000*cos(angle);                   // real part = cos(w nts)
/*18*/  phasor.imag=20000*sin(angle);                   // imag part = sin(w nts)
/*19*/  output_left_right_sample((short)phasor.real, (short)phasor.imag);
/*20*/                                                  // output each sine value
/*21*/  if (ctr < sample_period-1) ++ctr;               // increment counter (0 through 239)
/*22*/  else ctr = 0;                                   // reset counter if necessary
/*23*/  return;                                         // return from interrupt
/*24*/ }
/*25*/
/*26*/ void main()
/*27*/ {
/*28*/   ctr=0;                                         // initialize counter
/*29*/   comm_intr();                                   // initialize DSK, codec, McBSP
/*30*/   while(1);                                      // wait for an interrupt to occur
/*31*/ }
```

Figure 9: Listing of `rp_100Hz.c`

```
/*1 */ // This project uses support files generated by Rulph Chassaing
/*2 */ // Comm routines included in dsk6713_bsl.lib
/*3 */ #include <math.h>
/*4 */ #include "dsk6713_aic23.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*/ Uint32 fs = DSK6713_AIC23_FREQ_24KHZ;
/*13*/
/*14*/ interrupt void c_int11()               // interrupt service routine
/*15*/ {
/*16*/  int i;                                // used in for loop
/*17*/  // create interference pattern
/*18*/  // use amplitude 30000/N to prevent overflow in codec, where N=2
/*19*/  wnts = 2.0*PI*ctr/sample_period;      // current angle (w/out phase)
/*20*/  out_value=0;
/*21*/  for (i=0; i<2; i++)
/*22*/  {
/*23*/    angle = wnts + phase[i]*PI/180;     // current angle (with phase)
/*24*/    out_value += (30000/2)*cos(angle);  // cos(w nts + theta_k) k in {1,2}
/*25*/  }
/*26*/
/*27*/  //output interference pattern
/*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 10: Listing of `interference.c`

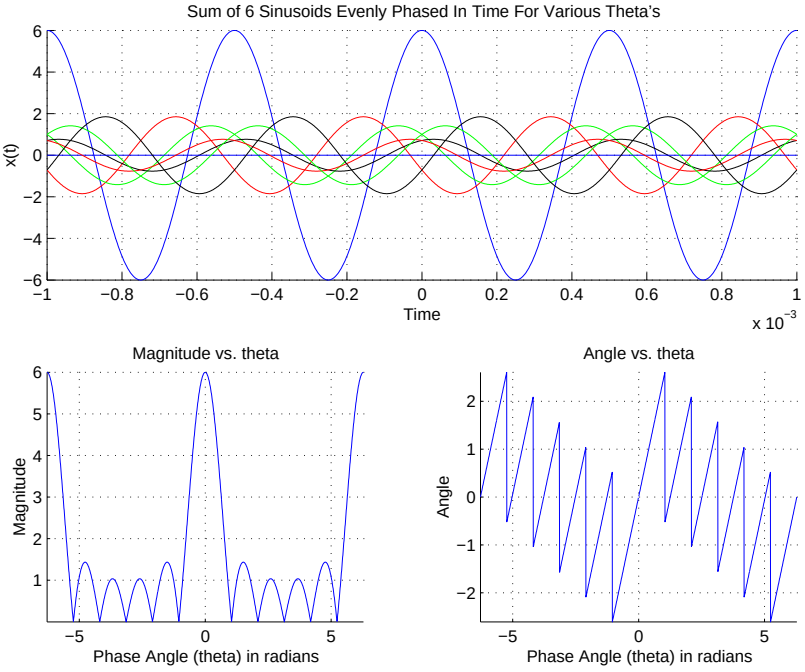

Figure 11: Interference Pattern For $N = 6$ Evenly Phased Cosines