Real-Time DSP Lab 2: Signal Generation and Aliasing

EE3221 Digital Signal Processing
Electrical Engineering and Computer Science Department
Milwaukee School of Engineering
Last Update: 30 November 2017

Contents

1 Overview

A common task in many applications involving DSP is signal (or waveform) generation. DSP techniques allow for precise creation and control of synthesized waveforms. However, certain constraints imposed by the processing techniques and hardware must be considered. Through this exercise you will gain experience generating waveforms in both simulation and on real-time DSP hardware.

2 Matlab Simulation

The following task uses MATLAB [1] to investigate sampled waveforms. MATLAB is available for download through the MSOE network. The provided code, with minor modification, can be run on the freely available GNU Octave [2].

Sampled versions of signals (sequences) can be quickly and easily generated in MATLAB. The MATLAB listing in Figure 1 shows an example of creating a sequence that represents samples of a 200Hz sinusoid.

```matlab
clear all
close all

fs = 8000;           % sample rate is 8000 samples/second
Ts = 1/fs;           % time between samples is inverse of sample rate
t_max = 0.01;        % max time is 0.01 seconds
n_max = t_max/Ts;    % determine largest index
n = 0:1:n_max;       % generate index vector

%% Generate samples of sinusoid
frequency = 200;        % in Hz
x1 = sin(2*pi*frequency*(Ts*n));

%% Plot discrete sequence
figure
stem(n,x1)

%% Visualize continuous signal
figure
plot(n*Ts,x1)
```

Figure 1: MATLAB script for generating samples of a sinusoid

By changing the value of the variable `frequency`, you can easily modify the sequence so that it represents the samples of a different sinusoid. However, we must be careful not to violate the Nyquist Theorem. That is, if the frequency of the signal we attempt to generate equals or exceeds $f_s/2$, where f_s is the sampling frequency, aliasing will occur. Thankfully, at least in MATLAB, we can also easily modify the sample rate to accommodate the frequencies of signals we may want to generate.

Question 1: Modify the script above according to the values of `fs` and `frequency` shown in the table below. In each case, observe the plots and interpret the results. You may need to adjust `t_max` in order to better visualize the waveforms.

value assigned to `fs`	value assigned to `frequency`
8000	1000
8000	2000
8000	4000
8000	7000
16000	7000

Which cases result in aliasing?

Other types of signals can also be easily generated in MATLAB. Figure 2 shows a script for generating samples of a square wave.

```
clear all
close all

fs = 8000;           % sample rate is 8000 samples/second
Ts = 1/fs;           % time between samples is inverse of sample rate

%% Generate samples of a square wave
x1 = [1*ones(1,8) -1*ones(1,8)];
n = 1:length(x1);

%% Plot discrete sequence
figure
stem(n,x1)

%% Visualize continuous signal with samples superimposed
figure
hold on
plot(n*Ts,x1)        % plot visualization of continuous signal
plot(n*Ts,x1,'o')    % plot sample points as open circles
```

Figure 2: MATLAB script for generating samples of a square wave

Question 2: Assume the samples correspond to one period of a square wave. What is the frequency of the square wave?

3 Real-Time Signal Generation - Sinusoids of Arbitrary Frequency

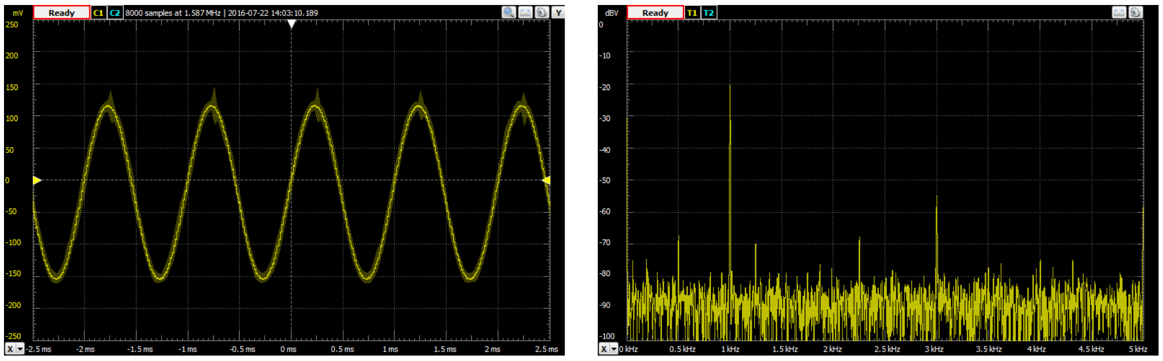
The program listing `sine_intr.c` shown in Figure 3 generates sinusoidal waveforms on the real-time DSP hardware. The program structure is identical to the examples of the previous laboratory assignment. That is, the program is interrupt-based and operates at a sampling frequency of 8kHz. However, notice that the ADC samples are never used. Instead, the output samples are based on calls to the `sin()` function. The frequency of the signal being generated is controlled by the variable `frequency`. The DAC will reconstruct a continuous-time signal based on the sample values.

```c
// sine_intr.c

#include "audio.h"

void I2S_HANDLER(void) {  /****** I2S Interruption Handler *****/
  const int sampling_freq = 8000;
  const float32_t frequency = 1000.0;
  const float32_t amplitude = 2000.0;
  const float32_t theta_increment = 2*PI*frequency/sampling_freq;
  static float32_t theta = 0.0f;
  int16_t audio_chR=0;
  int16_t audio_chL=0;

  audio_IN = i2s_rx();
  audio_chL = (audio_IN & 0x0000FFFF);
  audio_chR = ((audio_IN >>16)& 0x0000FFFF);

  theta += theta_increment;
  if (theta > 2*PI) theta -= 2*PI;
  audio_chL = (int16_t)(amplitude*sin(theta));
  audio_chR = audio_chL;

  audio_OUT = ((audio_chR<<16) & 0xFFFF0000) | (audio_chL & 0x0000FFFF);
  i2s_tx(audio_OUT);
}

int main(void) {
  audio_init(hz8000, line_in, intr, I2S_HANDLER);
  while(1){}
}
```

Figure 3: Listing of program `sine_intr.c`

To ensure accurate signal generation, a portion of the calculation in `sine_intr.c` must be carried out using floating point arithmetic. Notice that variables `theta_increment` and `theta` are of type `float32_t`. Because the values ultimately sent to the DAC must be integers, note the type cast to `int16_t` when assigning variable `audio_chL`.

Connect an oscilloscope to one channel of the headphone output on the Cypress FM4 board and observe the signal in both the time and frequency domains (Math FFT function or Spectrum Analyzer). Verify the 1kHz sinusoidal output. Your results should resemble those shown in Figure 4.



(a) (b)

Figure 4: Analog output generated by `sine_intr.c` in (a) time domain and (b) frequency domain.

Question 3: Modify the script above according to the values `frequency` shown in the table below. In each case, capture images of the output signal in both the time and frequency domain and interpret the results.

value assigned to `frequency`
1500
2641
3500
4500
7000

One final comment regarding `sine_intr.c`. The fact that the program used the math library function `sin()` to compute output samples is somewhat unusual for real-time DSP. Calls to trigonometric functions such as `sin()` are computationally expensive and are typically avoided in real-time programs. Instead, the sample values are usually pre-computed and then stored in look-up tables. We will see such an example in the next section of the lab.

4 Real-Time Signal Generation - Lookup Table Method

The C source file `sine_lut_intr.c` listed in Figure 5 generates a sinusoidal signal using a lookup method. It operates as follows.

An eight point lookup table is initialized in the array `sine_table` such that the values of the array correspond to the expression

$$\texttt{sine\_table}[\mathrm{i}] = 10000 \sin(2\pi i/8) \tag{1}$$

The values in the array `sine_table` are samples of exactly one cycle of a sinusoid.

Function `main()` is identical to the listings described in Laboratory 1.

When an interrupt occurs, `I2S_HANDLER` is called. Sample values read from `sine_table` are written into both channels of the DAC. Variable `sine_ptr` is incremented to point to the next value in the array.

```c
// sine_lut_intr.c

#include "audio.h"

void I2S_HANDLER(void) {     /****** I2S Interruption Handler*****/
  const int loop_size = 8;
  const int16_t sine_table[loop_size] = {0, 7071, 10000, 7071, 0, -7071, -10000, -7071};
  static int sine_ptr = 0;

  int16_t audio_chR=0;
  int16_t audio_chL=0;

  audio_IN = i2s_rx();
  audio_chL = (audio_IN & 0x0000FFFF);
  audio_chR = ((audio_IN >>16)& 0x0000FFFF);

  audio_chL = sine_table[sine_ptr];
  audio_chR = sine_table[sine_ptr];
  sine_ptr = (sine_ptr+1) % loop_size;

  audio_OUT = ((audio_chR<<16) & 0xFFFF0000) | (audio_chL & 0x0000FFFF);
  i2s_tx(audio_OUT);
}

int main(void) {
  audio_init (hz8000, line_in, intr, I2S_HANDLER);
  while(1){}
}
```

Figure 5: Listing of program `sin_lut_intr.c`

The resulting waveform on the left and right audio channels will be a 1kHz sinusoidal signal. The 1kHz frequency is due to the eight samples per cycle output at a rate of 8kHz (8000 samples per second). The signal will be a sinusoid since the WM8731 codec contains a low-pass reconstruction filter that interpolates between output samples to give a smooth analog signal.

Question 4: Modify `sin_lut_intr.c` to use a sampling rate of 48kHz by changing the parameter `hz8000` to `hz48000` in the call to `audio_init()`. What will be the new frequency of the sinusoid? Justify your answer. Include oscilloscope and spectrum analyzer captures of the generated sinusoid.

Question 5: Modify `sin_lut_intr.c` to generate a 2kHz sinusoid using a sampling rate of 8kHz. Include oscilloscope and spectrum analyzer captures of the generated sinusoid. Also include in your answer the lines of code you modified.

Question 6: Repeat the previous problem, but this time generate a 1.5kHz sinusoid.

5 Real-Time Signal Generation - Square Waves

Consider the sample sequence $\{10000, 10000, 10000, 10000, -10000, -10000, -10000, -10000\}$. The sequence represents the samples of one period of a square wave. Since there are eight samples in one period, using a sampling frequency of 8kHz would result in a 1kHz square wave.

Edit program `sine_lut_intr.c`, replacing the sample values representing a sinusoid in array `sine_table` with the sequence given above. That is, replace the relevant line in the source file with

```
int16_t sine_table[LOOP_SIZE] = {10000, 10000, 10000, 10000, -10000, -10000, -10000, -10000};
```

Run the program and observe the output signal in the time-domain on the oscilloscope.

Question 7: How would you describe the time-domain representation of the square wave? Is it a square wave? How does its shape compare to the square wave generated using MATLAB?

Now, observe the output signal in the frequency-domain using the spectrum analyzer.

Question 8: Using the frequency information, precisely describe the shape of the waveform. How does this result correspond to the Fourier Transform (or Fourier Series) of a square wave?

References

[1] Mathworks, http://www.mathworks.com/.

[2] Gnu Octave, https://www.gnu.org/software/octave/.

[3] ARM University Worldwide Education Program. ARM-based Digital Signal Processing Lab-in-a-Box, 2014.