ECE 4680 DSP Laboratory 6: Signal Generation Using DDS

Due Date: _____

Introduction and Background

Signal processing systems, in particular communications systems, need to generate signals in addition to processing them. Text Chapter 5, entitled *Signal Generation*, drills down on this important topic. Signal generation requirements might be for sinusoids, pulse type signals, pseudo-random data, or noise waveforms, to name a few. In this lab the focus will be on the generation of sinusoidal signals using what is known as direct digital synthesis (DDS) [1]–[3]. Two applications, audio special effects and a communications receiver for frequency modulation (FM) are described later in this document. The audio special effects application makes of the time varying delay introduced in Lab 4. The FM receiver implements complex frequency translation of the input signal in order for demodulation to be performed at *complex baseband*.

Direct Digital Synthesis

In the introduction and setup of the FM4 you had a chance to play with wavetable lookup as a simple means of signal generation. Two other means of sinusoidal signal generation are: (1) direct digital synthesizer (DDS) and (2) the digital resonator. The digital resonator uses an IIR filter with poles located on the unit circle that is excited by an impulse to start the oscillation. The focus here is the DDS technique, as it is quite popular in communications transmitter and receivers and audio special effects.

The Voltage Controlled Oscillator as Motivation

The DDS is motivated by the voltage controlled oscillator (VCO), which is used as sinusoidal signal generator in analog electronics [1]. A VCO has output frequency, $f_i(t)$, that is proportional to the input control voltage, $e(t)$ plus the *quiescent frequency* f_0.Working from the VCO block diagram of Figure 1, you have

$$\phi(t) = 2\pi K_v \int_{-\infty}^{t} e(\lambda) d\lambda \tag{1}$$

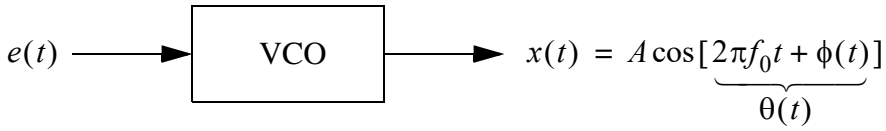


Figure 1: VCO high level block diagram.

with VCO gain constant K_v having units of Hz/v.The total phase of the VCO output, $\theta(t)$, is related to the instantaneous frequency of the VCO as

$$f_i(t) = \frac{1}{2\pi}\frac{d}{dt}\theta(t) = f_0 + K_v e(t) \tag{2}$$

where f_0 is the VCO quiescent frequency. From the above equations you can now draw the behavioral level block diagram of Figure 2.

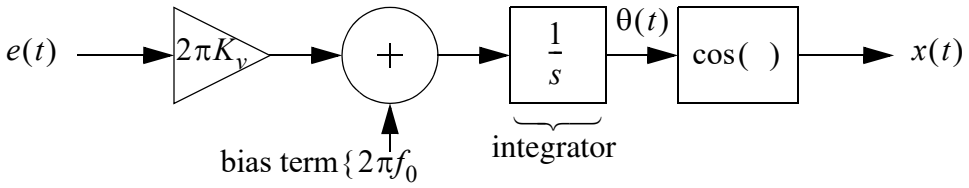


Figure 2: VCO behavioral level block diagram.

Note when $e(t) = 0$ the instantaneous frequency becomes $f_i(t) = f_0$ Hz.

Converting the VCO Model to the Discrete-Time Domain

In the discrete-time domain consider a sampling rate of $f_s = 1/T$, with T being the sampling period or spacing. You have

$$x[n] = x(nT) = \cos[2\pi f_0 nT + \phi(nT)], \tag{3}$$

where the discrete-time phase is $\phi[n] = \phi(nT)$.The integrator is replaced by an accumulator when you consider approximating the integral via rectangular areas. This is shown in Figure 3.

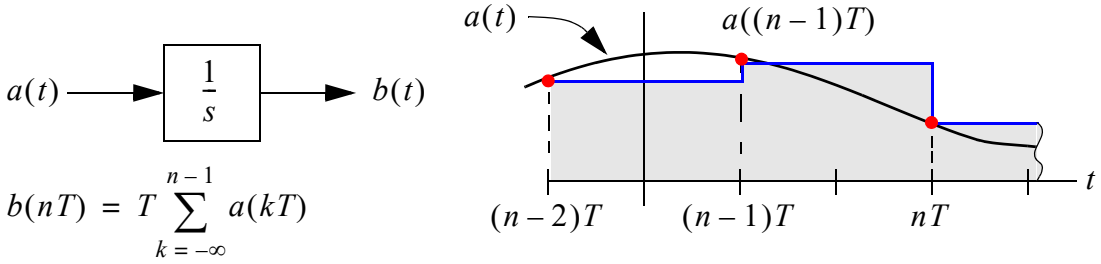


Figure 3: Discrete-time integration approximation using the left endpoint for a.

With the integration above understanding you can write

$$\begin{aligned}\phi[n] = \phi(nT) &= 2\pi K_v T \sum_{k=-\infty}^{n-1} e(nT) \\ &= 2\pi K_v T \sum_{k=-\infty}^{n-1} e[n] = k_v e[n-1] + \phi[n-1]\end{aligned} \tag{4}$$

where $k_v = 2\pi K_v T$ rad/sample. The recursive form for $\phi[n]$ in the last line of (4) motivates the

discrete-time form of the VCO shown in Figure 4.

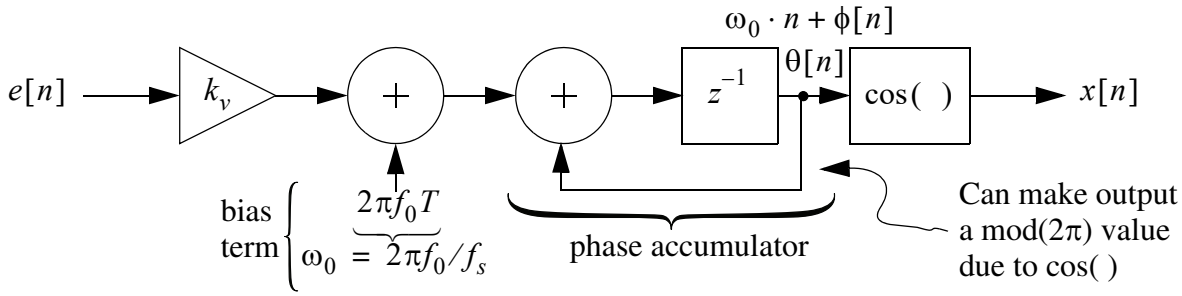


Figure 4: Discrete-time VCO block diagram.

As a fixed frequency generator let $e[n] = 0$ and set f_0 or $\omega_0 = 2\pi f_0 / f_s$ to the desired quiescent frequency $0 < f_0 < f_s/2$. This behavior is similar to the analog VCO. Note the discrete-time VCO is sometimes referred to as a *numerically controlled oscillator* (NCO), but more typically is the idealized mathematical form of the DDS. The output equation for the NCO/DDS when $e[n] = 0$ is

$$x[n] = \cos(2\pi f_0 / f_s \cdot n) = \cos(\omega_0 n) \tag{5}$$

where you can also write that

$$\begin{aligned} \theta[n] &= \omega_0 n \\ \theta[n+1] &= \omega_0 (n+1) = \omega_0 n + \omega_0 = \theta[n] + \omega_0 \end{aligned} \tag{6}$$

C-Code Implementation Using Floats

On the FM4 a near ideal DDS implementation is possible since floating-point arithmetic is available via the `math.h` library (included automatically in the ZIP file sample projects). Specifically what this means is that on-the-fly calculation of sine and cosine using `sinf()` and `cosf()` is possible. The code snippets below are for a DDS built in portable C (here using GCC on Windows 10 bash shell):

```
// Portable DDS Prototype
#include <stdio.h>
#include <math.h>
//define ARM compatible data type
typedef float float32_t;
typedef double float64_t;
typedef int int32_t;

#define NSAMPLES 10000
#define f0 14000  // Hz

int main(void)
{
    FILE *fp;
    int n;
    float32_t x[NSAMPLES];
```

```
    float64_t y[NSAMPLES];
    //DDS variables for fs = 48000 Hz
    float32_t delta = 0.0001308996938995747; // 2*pi/48000
    float32_t twopi = 6.283185307179586;
    float32_t angle = 0.0;

    // Output results to a file
    fp = fopen("DDS_test.txt", "w");
    for (n = 0; n < NSAMPLES; n++) {
        // DDS phase accumulator
        angle += delta*f0;
        if (angle >= twopi) angle -= twopi;
        // On-the-fly sin/cos rather than look-up table (LUT)
        x[n] = cosf(angle); // single precision
        y[n] = cos(twopi*f0/48000.0*n); //double precision compare
        // Write float and double results to a file for reading into
        // file for reading into a Jupyter notebook.
        if (n == 0) {
          fprintf(fp, "DDS float32_t output data as n, x\n");
          fprintf(fp, "%d, %6.16f, %6.16f\n", n, x[n], y[n]);
        }
        else if (n == NSAMPLES-1) {
          fprintf(fp, "%d, %6.16f, %6.16f", n, x[n], y[n]);
        }
        else {
          fprintf(fp, "%d, %6.16f, %6.16f\n", n, x[n], y[n]);
        }
    }
    fclose(fp);
    printf("Done!\n");
    return 0;
}
```

First off from the above code snippet you see that the DDS quiescent frequency is determined by choosing

$$\omega_0 = 2\pi \cdot \frac{f_0}{f_s}, 0 < f_0 < f_s/2 \tag{7}$$

A serious drawback of the on-the-fly calculation is the time it takes to execute a trig function call. On the FM4 with full optimization, a call to `cosf()` takes about 0.73 µs, while a call to `arm_cos_f32()` takes about 0.41 µs. Both measured using GPIO timing. Recall the cycle time is 1/200 MHz or 5 ns (0.005 µs).

Practical Implementation Considerations

Most DDS implementations, in particular ASIC and FPGA forms, utilize fixed-point arithmetic. Finite precision impacts include:

- Bit width of the accumulator; controls the ultimate frequency precision or smallest frequency step size via $\Delta f = f_s/2^{B_{\text{acc}}}$ Hz, where f_s is the sampling rate and B_{acc} is the accumulator bit width
- The size of the sine/cosine look-up-table (LUT) is 2^{B_w}, where B_w is the bit width of the

table address, typically reduced from the accumulator bit width, that is $B_w < B_{\text{acc}}$

- The storage precision or bit width of the sine/cosine values is B_{cos}

A modified DDS block diagram, that includes finite precision attributes is shown in Figure 5. For

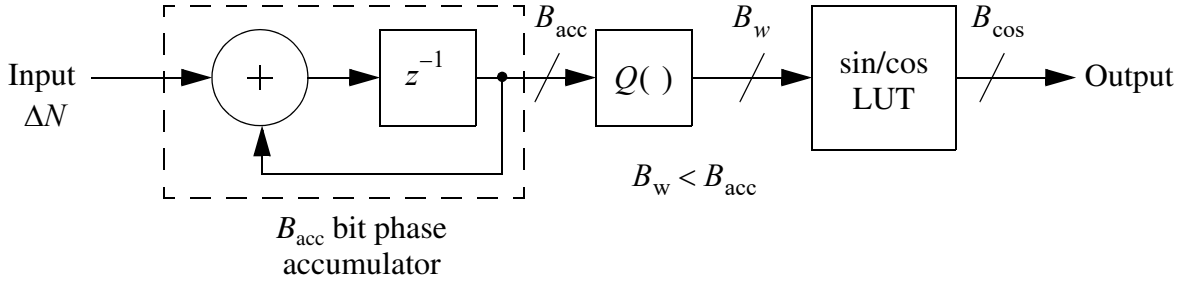


Figure 5: Finite precision DDS block diagram.

an accumulator input step size ΔN (an integer value), the DDS output frequency is

$$f_0 = \frac{f_s}{2^{B_{\text{acc}}}} \cdot \Delta N \text{ Hz} \tag{8}$$

or from a design standpoint

$$\Delta N = \frac{f_0 \cdot 2^{B_{\text{acc}}}}{f_s}. \tag{9}$$

Note in [3] and elsewhere f_s is referred to as the DDS clock frequency, f_{clk}.

A Python simulation of this system is contained in the module `dds.py` via the function `DDS()`:

```
def DDS(f0,fs,N_samps,Bcos,Bacc,Bw):
    """
     x,a_out,n = DDS(f0,fs,N_samps,Bcos,Bacc,Bw)
     ///////////////// Inputs //////////////////////
          f0 = desired output frequency
          fs = sampling frequency
     N_samps = number of samples to simulate
        Bcos = bit width of cos/sin values
        Bacc = bit width of the accumulator
          Bw = bit width of the LUT address
     ///////////////// Outputs ///////////////////
           x = output signal
       a_out = accumulator normalized to a [0,1) float value
           n = time index

    Mark Wickert November 2016
    """
    n = np.arange(N_samps-1)
    x = np.zeros(len(n))
    a_out = np.zeros(len(n))
    a = 0
    w = 0
    theta = 0
    for k in range(len(n)):
```

```
        #x[k] = cos(2*np.pi*a)
        x[k] = ssd.simpleQuant(cos(2*np.pi*w/2**Bw),Bcos,1,'none')
        a += round(f0/fs*2**Bacc)
        if a >= 2**Bacc:
            a -= 2**Bacc
        w = round(a/2**(Bacc-Bw))
        a_out[k] = w/2**Bw
    return x, a_out, n
```

The Jupyter notebook '`DDS and Applications.ipynb`', found in the python folder of the Lab6 project ZIP, contains among other things the examples described below. To see the DDS simulation in action suppose $f_s = 48\,\text{kHz}$ and consider 32 bits (very large) for all of the bit widths and compare that to the case $B_{\text{acc}} = 32, B_w = 16$ and $B_{\text{cos}} = 16$ in Figure 6.

```
x32_32_32, a_out, n = dds.DDS(14,48.0,10000,32,32,32)
x14_32_14, a_out, n = dds.DDS(14,48.0,10000,16,32,16)
subplot(211)
psd(x32_32_32,2**12,48);
title(r'Bit Widths: $B_{\cos} = 32$, $B_{acc} = 32$, $B_{w} = 32$')
ylabel(r'PSD (dB)')
xlabel(r'Frequency (kHz)')
ylim([-140,10])
subplot(212)
psd(x14_32_14,2**12,48);
title(r'Bit Widths: $B_{\cos} = 16$, $B_{acc} = 32$, $B_{w} = 16$')
ylabel(r'PSD (dB)')
xlabel(r'Frequency (kHz)')
ylim([-140,10])
tight_layout()
```

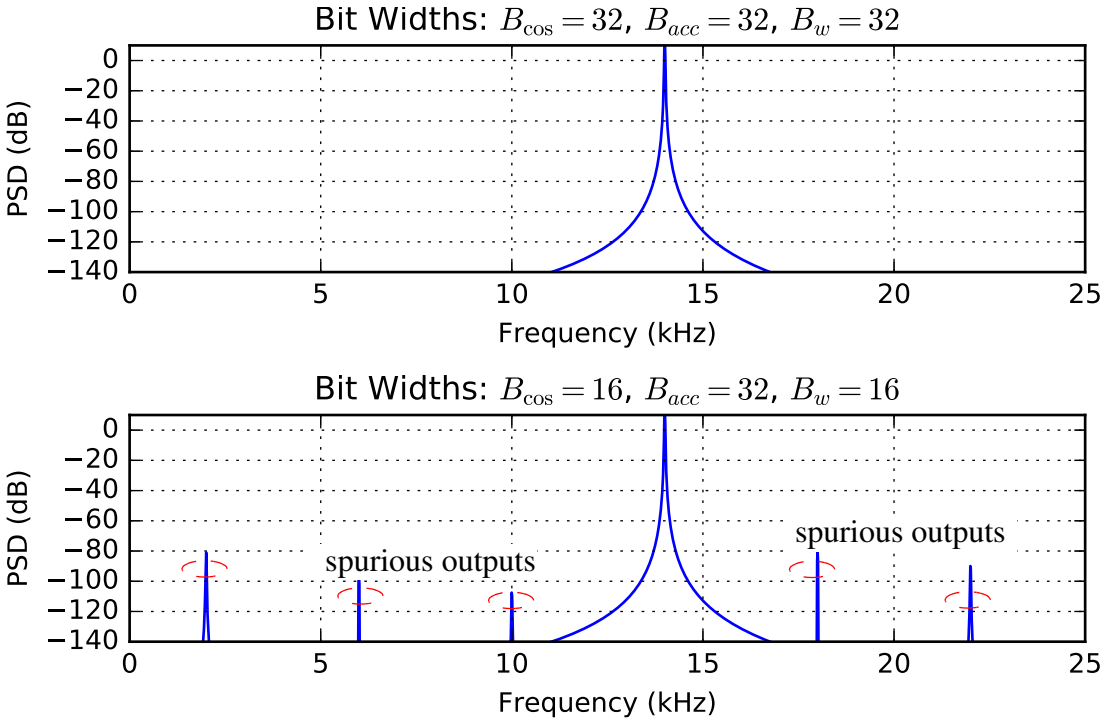


Figure 6: DDS output at 14 kHz with $f_s = 48\,\text{kHz}$ and bit widths (a) $B_{\text{acc}} = B_w = B_{\text{cos}} = 32$, and (b) $B_{\text{acc}} = 32, B_w = 16$, and $B_{\text{cos}} = 16$.

The idealized result (32 bits everywhere) is shown in Figure 6a along with reduced bit width results in Figure 6b. As the bit width is reduced spurious outputs (spurs) start to appear. This is a result of the finite precision arithmetic involved in the design. Spurious outputs are one of the main issues with the DDS. In the example above you see that with enough precision in the algorithm, spurious outputs can be down 100 dB or more from the desired output.

Further discussion of DDS spurs can be found in the appendix of this document. The 32-bit accumulator output for $f_0 = 14$ kHz and $f_s = 48$ kHz is shown in Figure 7. As expected the waveform

```
x32_32_32, a_out, n = dds.DDS(14,48.0,10000,32,32,32)
plot(n[:50],a_out[:50])
plot(n[:50],a_out[:50],'r.')
title(r'Bit Widths: $B_{\cos} = 32$, $B_{acc} = 32$, $B_{w} = 32$')
ylabel(r'Normalized Phase Accumulator')
xlabel(r'Sample Index $n$')
grid();
```

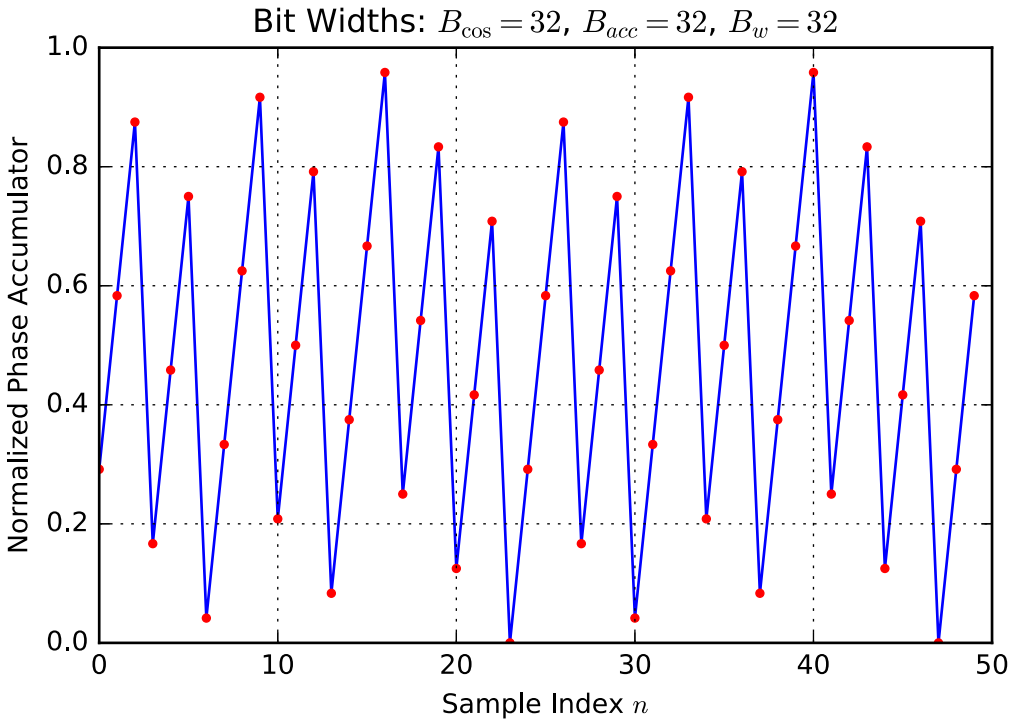


Figure 7: DDS accumulator output for $f_0 = 14$ kHz with $f_s = 48$ kHz and bit widths $B_{\text{acc}} = B_w = B_{\cos} = 32$.

is a sampled ramp as the phase of the cosine is advancing linearly with the sample index.

The primary focus of this lab is to study a float32_t DDS implementation, see

What to Expect with an FM4 Implementation

When you implement the DDS using `float32_t` (follow the C example above) on the FM4 you

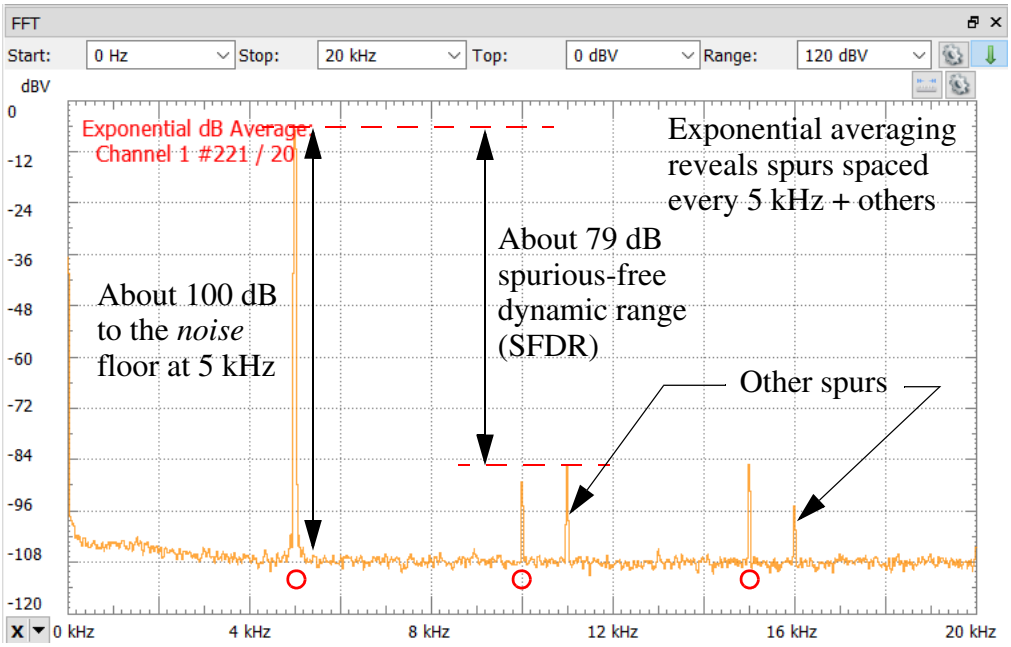


Figure 8: DDS output at 5 kHz from the FM4 with $f_s = 48$ kHz using real-time floats via `arm_cos_f32()`.

will first need to scale the float `cosf()` value to some integer value inside the 16-bit signed integer range, e.g., `15000*cosf()` works well, assuming you keep the GUI slider interface in place, e.g.,

```
xL = (int16_t) (FM4_GUI.P_vals[0] * 15000*arm_cos_f32(angle));
```

Upon running the code and setting the frequency f_0 to 5 KHz you should obtain results similar to those shown in Figure 8. In this case you see a combination of harmonic outputs relative to the 5 kHz signal and non-harmonic spurs. The noise floor is mostly due to the Analog Discovery itself. Exponential averaging in dB is used to reveal spurs sitting just above the noise floor.

Before moving into actual lab tasks a couple of application examples are introduced to get you prepared for some of the lab problems/exercises coming up.

Applications

DDS application domains explored in this lab are audio special effects and communication receiver complex frequency translation.

Audio Special Effects

Audio special effects is a popular real-time DSP application area. Looking at [4] Chapter 10, you find a Chapter 10 entitled *Guitar Special Effects*. A special effect of interest in this lab is flanging[1], which is formed using a variable length delay line and a sinusoidal waveform generator as

1. https://en.wikipedia.org/wiki/Flanging.

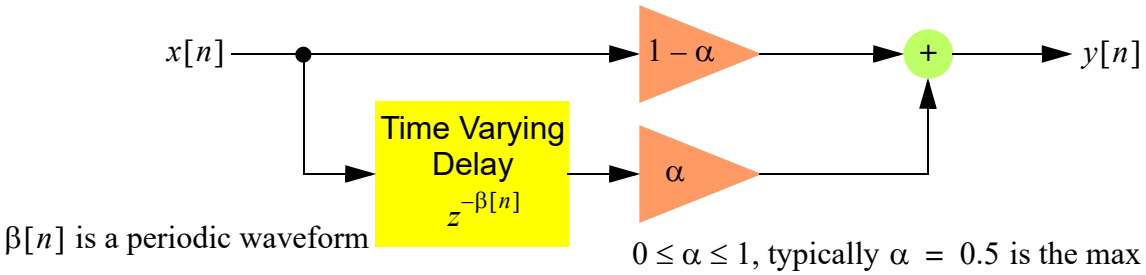


Figure 9: Time varying delay from Lab 4 configured to form the audio special effect known as flanging.

shown in Figure 9. The sinusoidal generator comes into play by making $\beta[n]$ a cosine signal of the form

$$\beta[n] = \frac{D}{2}\left(1 + \cos\left[2\pi\frac{f_0}{f_s}n\right]\right) = D_p\left(1 + \cos\left[2\pi\frac{f_0}{f_s}n\right]\right), \tag{10}$$

where D is the delay line length, f_0 is a low frequency, around 1 Hz or less, and f_s is the sampling rate. The peak delay, $D_p = D/2$, is the quantity controlled in the problem section. The constant $0 \le \alpha \le 1$ controls the fraction of the time delayed signal that mixes with the direct signal. Typically a 50/50 mix, which occurs when $\alpha = 0.5$ is the maximum value, but in testing it is nice to be able to turn the direct path off by setting $\alpha = 1.0$. The time varying delay imposes a Doppler frequency shift, that is also time varying itself, on the input signal. When the two signals are mixed together you hear a *swooshing* sound. If f_0 is too large the Doppler frequency shift makes the effect more like listening to badly tuned instruments. The Wiki page describes other variations of the basic block diagram. The delay generator may also be a triangle, sawtooth, or periodic exponential wave shape. In the lab exercises you will explore varying both D_p and f_0.

Understanding the Doppler Shift and Spectrum

For the special case of a sinusoidal input signal, i.e.,

$$x[n] = A_m\cos\left(2\pi\frac{f_m}{f_s}n\right) \tag{11}$$

where A_m and f_m are the amplitude and analog frequency of the sinusoid respectively. Assuming that the direct path is turned off, $\alpha = 1$, you have

$$\begin{aligned} y[n] &= A_m\cos\left(2\pi\frac{f_m}{f_s}(n - \beta[n])\right) \\ &= A_m\cos\left\{2\pi\frac{f_m}{f_s}\left[n - D_p\left(1 + \cos\left(2\pi\frac{f_0}{f_s}n\right)\right)\right]\right\} \end{aligned} \tag{12}$$

A Doppler frequency behavior exists as $\beta[n]$ is a function of time n, thus making it possible for the time axis n to compress of expand. In the discrete-time domain the first backwards difference is like taking the derivative in the continuous-time domain. Let

$$\psi[n] = 2\pi\frac{f_m}{f_s}(n - \beta[n]) \tag{13}$$

be the total phase of the cosine. The Doppler frequency in Hz/sample is contain within the first backwards difference upon scaling by the sampling frequency

$$\begin{aligned}\frac{\psi[n] - \psi[n-1]}{2\pi} \cdot f_s &= f_m - f_s(\beta[n] - \beta[n-1]) \\ &= f_m - D_p f_s\left(\cos\left(2\pi\frac{f_0}{f_s}n\right) - \cos\left(2\pi\frac{f_0}{f_s}(n-1)\right)\right)\end{aligned} \tag{14}$$

The Doppler shift is thus

$$f_D = D_p f_s\left[\cos\left(2\pi\frac{f_0}{f_s}n\right) - \cos\left(2\pi\frac{f_0}{f_s}(n-1)\right)\right] \text{ Hz/sample} \tag{15}$$

Beyond the Doppler view in (15), you can also consider (12) as sinusoidal frequency modulation (FM) [5]. To match the analysis of [5] assume that the delay is symmetrical about zero and further assume that $-\cos(\,)$ is replaced by $+\sin(\,)$. With these changes you can write

$$y[n] = \text{Re}\{e^{j2\pi f_m \cdot n/f_s} \cdot e^{j\beta_{\text{FM}}\sin(\omega_0(n/f_s))}\} \tag{16}$$

where β_{FM} is equivalent to the modulation index found in sinusoidal phase/frequency modulation [5] is given by

$$\beta_{\text{FM}} = 2\pi f_m \cdot \frac{D_p}{f_s} = 2\pi f_m \cdot D_p T,\ T \tag{17}$$

where $T = 1/f_s$ is the sampling period. The spectrum of $y[n]$ or in reality $y(t)$ as captured at the output of the DAC, can be found by comparing (15) to sinusoidal frequency modulation in [5]. To make the connection replace n/f_s with t, as if to convert from continuous to discrete variables. Note this is an approximation, but for $f_m, f_0 \ll f_s$ the approximation is better. Now (15) becomes

$$\begin{aligned}y(t) &= \text{Re}\{e^{j2\pi f_m \cdot t} \cdot e^{j\beta_{\text{FM}}\sin(2\pi f_0 t)}\} \\ &= A_m \sum_{n=-\infty}^{\infty} J_n(\beta_{\text{FM}})\cos[(2\pi f_m + n2\pi f_0)t]\end{aligned} \tag{18}$$

where $J_n(\,)$ is the Bessel function of the first kind having order n. The spectrum of $y(t)$, the DAC output, is a line spectrum centered about f_m with sidebands spaced at $\pm nf_0$.

A simulation model in Python (available in the "`DDS and Application.ipynb`") is compared

with measurements taken on the FM4 using the Analog Discovery in Figure 10.

```
import scipy.special as special
```

```
def flanging_sinusoid_spec(fm,f0,Dp,N_lines,fs):
    """


    Mark Wickert November 2016
    """
    beta = 2*pi*fm*Dp/fs
    print('beta = %4.2f' % beta)
    n_lines = arange(-N_lines,N_lines+1)
    f = n_lines*f0
    Y = zeros(len(f))
    for k, n_linesk in enumerate(n_lines):
        Y[k] = abs(special.jn(n_linesk,beta))
    f += fm
    return f, Y
```

```
f, Y = flanging_sinusoid_spec(5000,20,40,100,48000)
```

```
beta = 26.18
```

```
f_AD,SV1_AD,SV2_AD = loadtxt('flanger_spec_fm5k_Dp100_f020_fs48.csv',
                             delimiter=',',skiprows=6,unpack=True)
f, Y = flanging_sinusoid_spec(5000,20,100,100,48000)
```

```
beta = 65.45
```

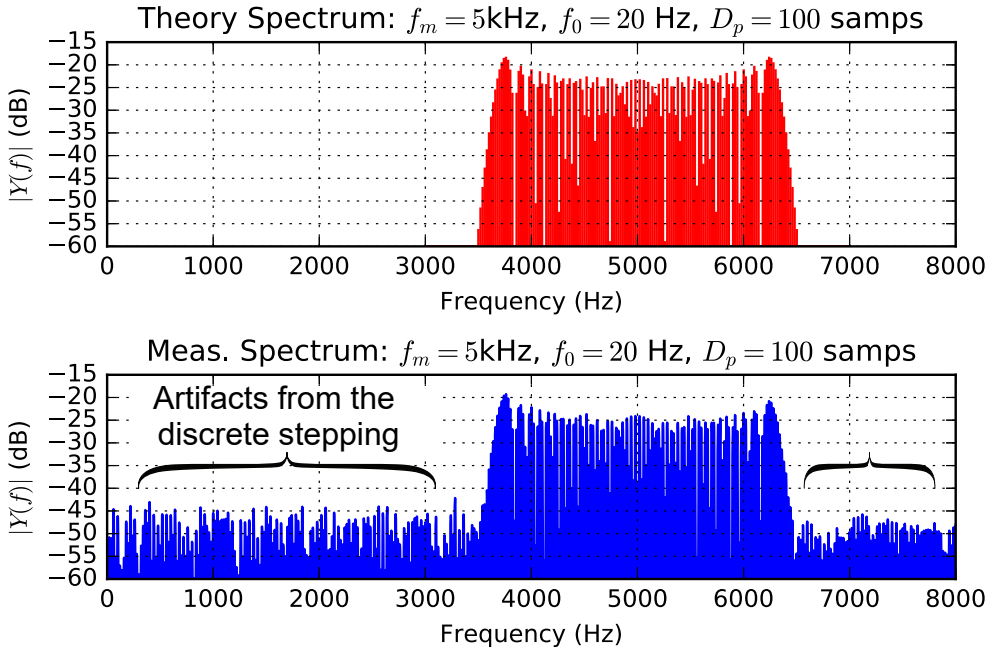


Figure 10: The induced Doppler spectrum, like the spectrum of a frequency modulated carrier for $f_m = 5\,\text{kHz}$, $f_0 = 20\,\text{Hz}$ and $D_p = 100\,\text{samples}$.

Application: FM Communications Receiver

A very important application of the DDS is in a communications receiver utilizing complex frequency translation. The sine and cosine signals of a DDS form a complex sinusoid that is used to frequency translate a input/received signal, e.g., from 30 kHz to baseband ($f = 0$). Demodulation is then performed on the now complex signal to *recover* the message signal.

Receiving System Details

Suppose the signal of interest is a frequency modulated (FM) carrier waveform given by

$$x_c(t) = A_c \cos\left[2\pi f_c t + 2\pi f_d \int^t m(\lambda)d\lambda\right]. \tag{19}$$

You study this waveform in detail in ECE 4625/5620, Communications Systems I. The signal carrier or center frequency is f_c Hz. The information or message carried by the signal is $m(t)$. The message signal can be analog voice or music, or digitally encoded information. A generic frequency spectrum for the FM signal $x_c(t)$ is shown in Figure 11. Mathematically the spectrum of

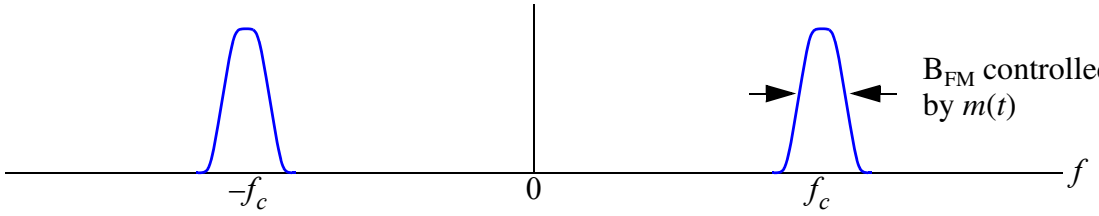


Figure 11: Spectrum of an FM carrier centered at f_c Hz.

$x_c(t)$ takes the form

$$X_c(f) = \frac{1}{2}[X_{\text{BB}}(f-f_c) + X_{\text{BB}}(f+f_c)] \tag{20}$$

where $X_{\text{BB}}(f)$ is the complex baseband spectrum corresponding to $x_c(t)$. Note $x_{\text{BB}}(t)$ is really just

$$x_{\text{BB}}(t) = A_c \exp\left[j2\pi f_d \int^t m(\lambda)d\lambda\right] \tag{21}$$

and

$$x_c(t) = \text{Re}\left\{x_{\text{BB}}(t)e^{j2\pi f_c t}\right\}. \tag{22}$$

To get your hands on $x_{\text{BB}}(t)$ all you need to do is complex frequency translate either to the left or right by f_c Hz and lowpass filter to one half the FM RF bandwidth, B_{FM}

$$x_{\mathrm{BB}}(t) \;=\; \mathrm{LP}\left\{ x_c(t) \cdot e^{\pm j2\pi f_c t} \right\}. \tag{23}$$

In the lab tasks you will chose to make the translation frequency negative to move the spectrum to the left. If in doubt, recall from Fourier transform theory that

$$\mathrm{FT}\{x(t)e^{j2\pi f_0 t}\} \;=\; X(f-f_0) \tag{24}$$

where $X(f) \;=\; \mathrm{FT}\{x(t)\}$. The second step is to demodulate the message $m(t)$ from $x_{\mathrm{BB}}(t)$. In communication systems you learn that a *frequency discriminator* of some sort is required. Here you use a DSP implementation that fits well with the overall receiver architecture.

DSP Receiver Implementation

A DSP based receiver utilizing the capabilities of the FM4 board is shown in Figure 12. The main

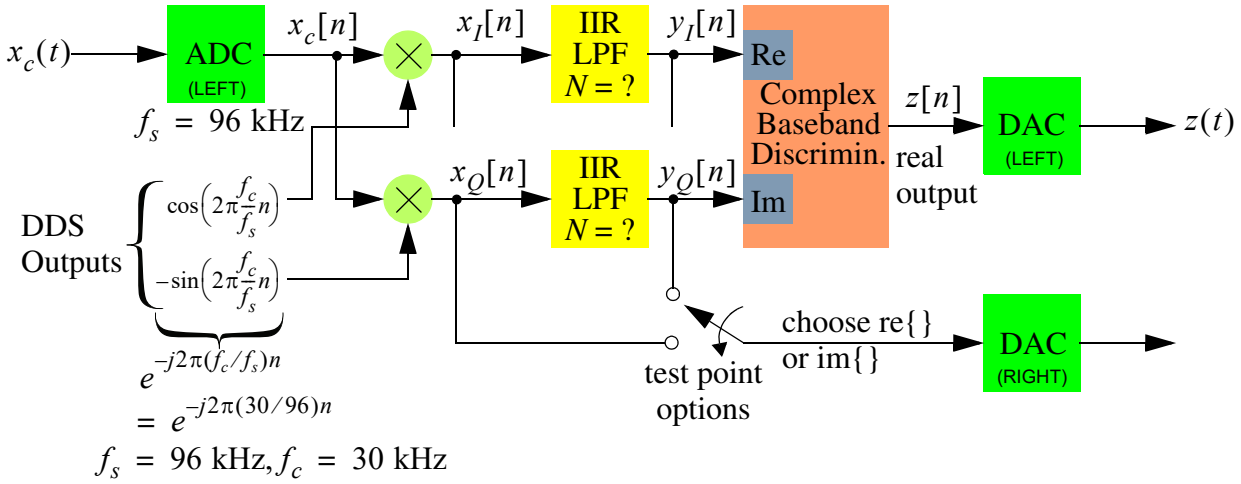


Figure 12: FM4 FM receiver block diagram.

signal flow passes from put to output using the left audio channel. The sampling rate is set to 96 ksps, which is the fastest rate supported by the audio codec.

Since the complex frequency translation is performed on the sampled input signal, $x_c[n]$, there are a some differences due to spectral images. Consider Figure 13 to help visualize how sampling

followed by complex frequency translation and a lowpass filter achieves the intended result.

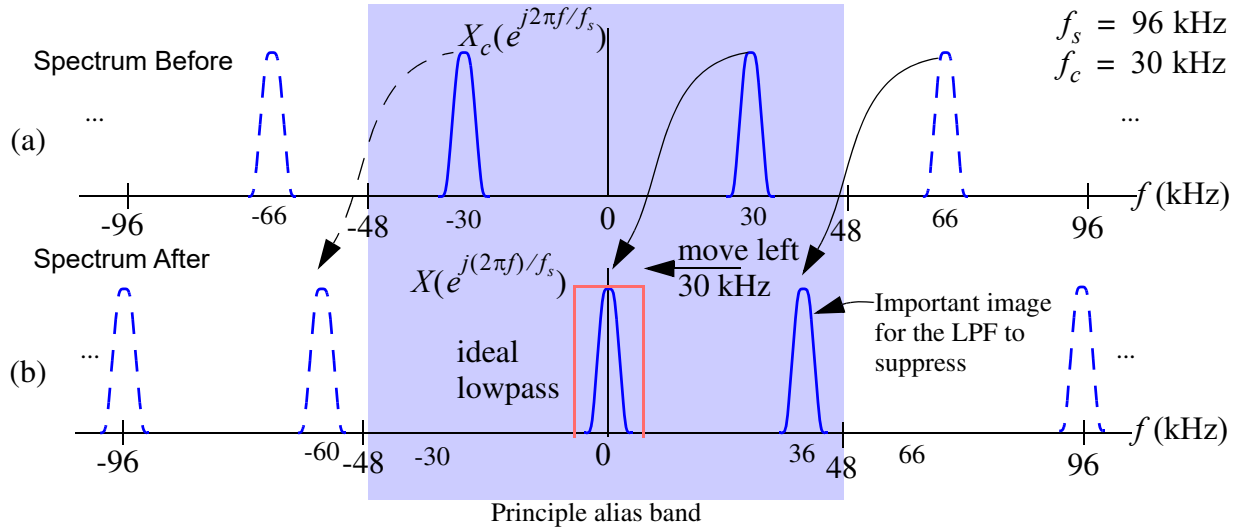


Figure 13: The $x_c(t)$ spectrum (a) following sampling and (b) following frequency translation when $f_s = 96$ kHz and $f_c = 30$ kHz (shift left).

Note that the spectrum of a complex signal is not symmetrical (in magnitude) about $f = 0$. Communications applications of real-time DSP typically involve complex signals.

Complex Baseband Discriminator

The complex baseband signal $y[n]$ for the case of an FM input signal, is of the form

$$y[n] = A_c e^{j\phi[n]} = A_c\{\cos(\phi[n]) + j\sin(\phi[n])\} = y_I[n] + jy_Q[n]. \tag{25}$$

Note $\phi[n] = \tan^{-1}(x_Q[n]/x_I[n])$. The frequency discriminator seeks to find the derivative of the phase $\phi[n]$. In the continuous-time domain the derivative of the inverse tangent function is

$$z(t) = \frac{d\phi(t)}{dt} = \frac{y_I(t)y_Q'(t) - y_Q(t)y_I'(t)}{y_I^2(t) + y_Q^2(t)}. \tag{26}$$

A discrete-time approximation to the above derivative is

$$z[n] = \frac{y_I[n]\cdot(y_Q[n] - y_Q[n-1]) - y_Q[n]\cdot(y_I[n] - y_I[n-1])}{y_I^2[n] + y_Q^2[n]}. \tag{27}$$

Here the denominator of (27) can be omitted since the magnitude squared of the FM signal is a constant.

Python Simulation

To verify the receiver design approach before writing any C-code a simulation is good practice. The Jupyter notebook "`DDS and Applications.ipynb`" contains such as model. The starting pointis generating a real FM signal centered at 30 KHz. The message signal will be a 1 kHz sinusoid deviating the 30 kHz carrier by 2.0 kHz peak. There are various ways to generate FM, but it

easy to use a modified version of `DDS()`:

```
def FM_gen(m,f0,fs,real_output = True):
    """
    x,a_out = FM_gen(m,f0,fs)
        m = message signal, amp. sets Df in fs units
       f0 = carrier frequency
       fs = sampling rate
        x = output waveform
    a_out = [0,1] accumulator output

    Mark Wickert November 2016
    """
    a_out = mod(cumsum(f0/fs + m/fs),1) # avoid for loop
    x = exp(1j*(2*pi*a_out)) # start with complex
    if real_output:
        x = x.real
    return x, a_out
```

The complex baseband discriminator is implemented in the function `discrim.m`:

```
def discrim(x):
    """
    disdata = discrim(x)
    where x is an angle modulated signal in complex baseband form.

    Mark Wickert
    """
    X=np.real(x)        # X is the real part of the received signal
    Y=np.imag(x)        # Y is the imaginary part of the received signal
    b=np.array([1, -1]) # filter coefficients for discrete derivative
    a=np.array([1, 0])  # filter coefficients for discrete derivative
    derY=signal.lfilter(b,a,Y)  # derivative of Y,
    derX=signal.lfilter(b,a,X)  #    "          X,
    disdata=(X*derY-Y*derX)/(X**2+Y**2)
    return disdata
```

The function `DDS()` is used to generate a complex sinusoid for frequency translation and a filter coefficients (and later a filter coefficients header file) are obtained using the functions

```
import iir_design_helper as IIR_d
import coeff2header as c2h
b, a, sos = IIR_d.IIR_lpf(5000, 6000, 0.5, 80, 96000,'cheby1')
# or
sos = signal.butter(10, 2*10/96,output='sos')
c2h.IIR_sos_header('FM_LPF1.h',sos)
```

The complete simulation is as follows:

```
n = arange(100000)
m = cos(2*pi*1000/96000*n) # 1000 Hz sinsoid message
xc, a_out = FM_gen(2000*m,30000,96000) # 2000 Hz peak deviation
x = xc*exp(-1j*2*pi*30/96*n) # complex frequency translate
y = signal.lfilter(b,a,x) # lowpass filter x = xI + j xQ
z = dds.discrim(y) # Send through discriminator
N0 = 500 # point to display
```

The input spectrum, the frequency translated spectrum, and the filtered spectrum are shown as subplots in Figure 14.

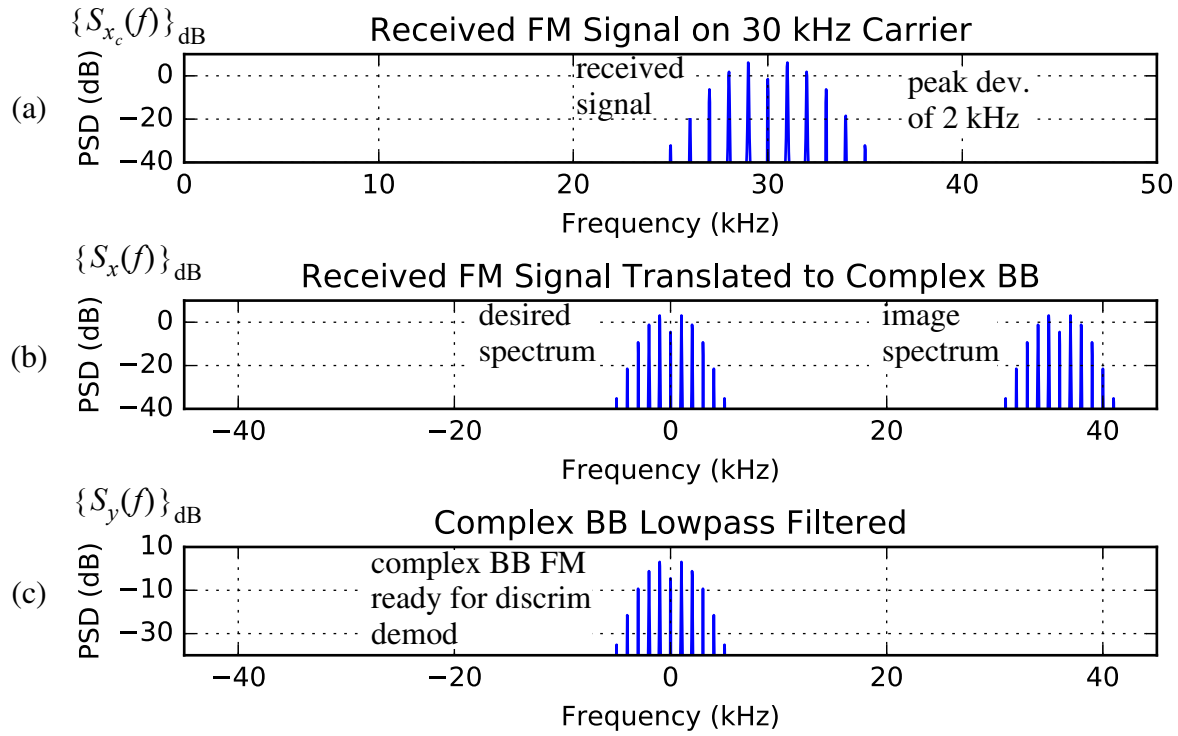


Figure 14: The spectra at (a) the input, (b) following complex frequency translations, and (c) after the lowpass filter, in the Python simulation of the FM receiver.

The final discriminator output signal $z[n]$ is given in Figure 14.

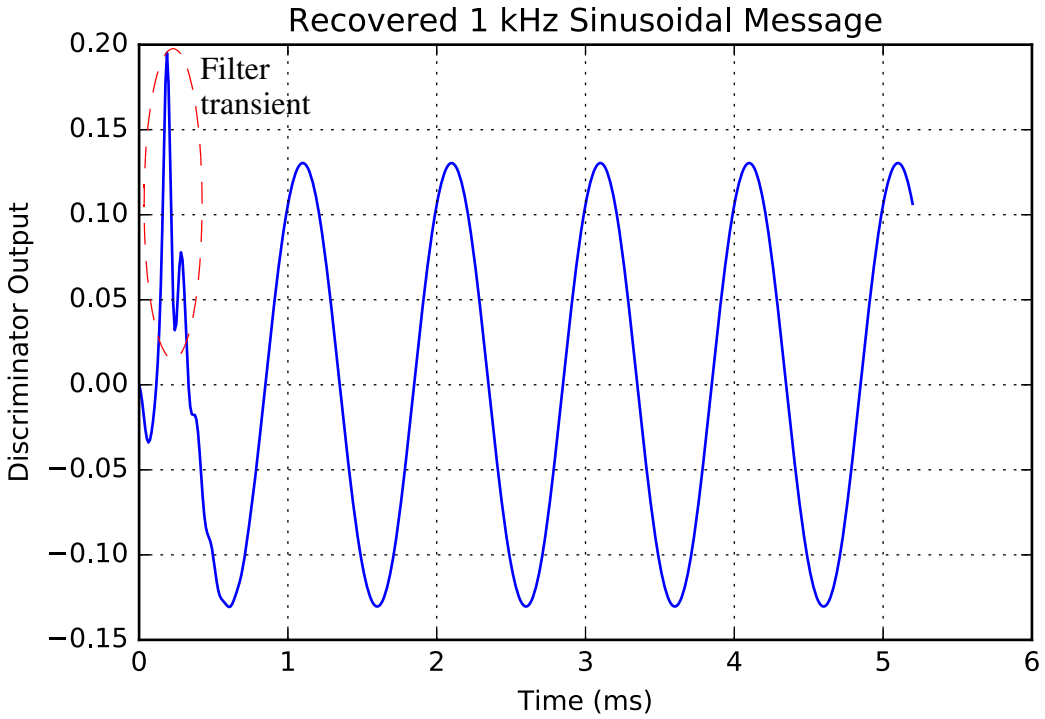


Figure 15: The 1 kHz message signal recovered at the output of the discriminator in the Python simulation.

The model results look good. The system can now be implemented on the FM4 with added confidence.

FM4 Sample Outputs

A complete implementation on the FM4 is tested using the Analog Discovery and its FM signal source and scope/spectrum analysis capabilities. An FM signal with a sinusoidal message at 1 kHz and 30 kHz carrier is configured as shown in Figure 16.

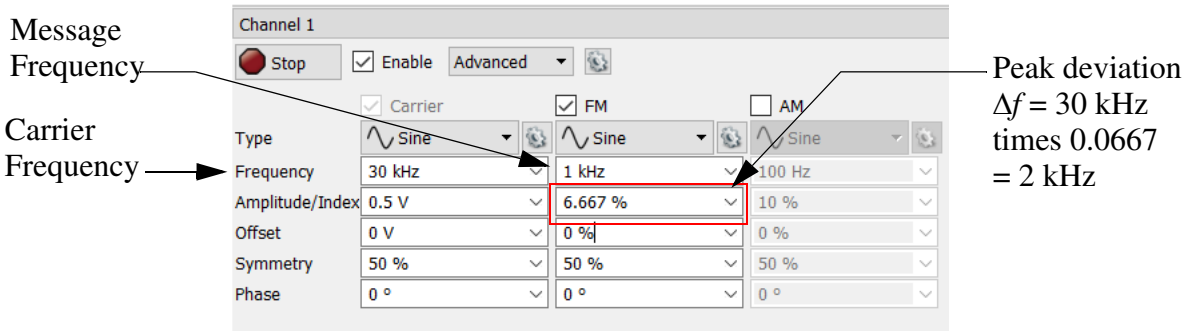
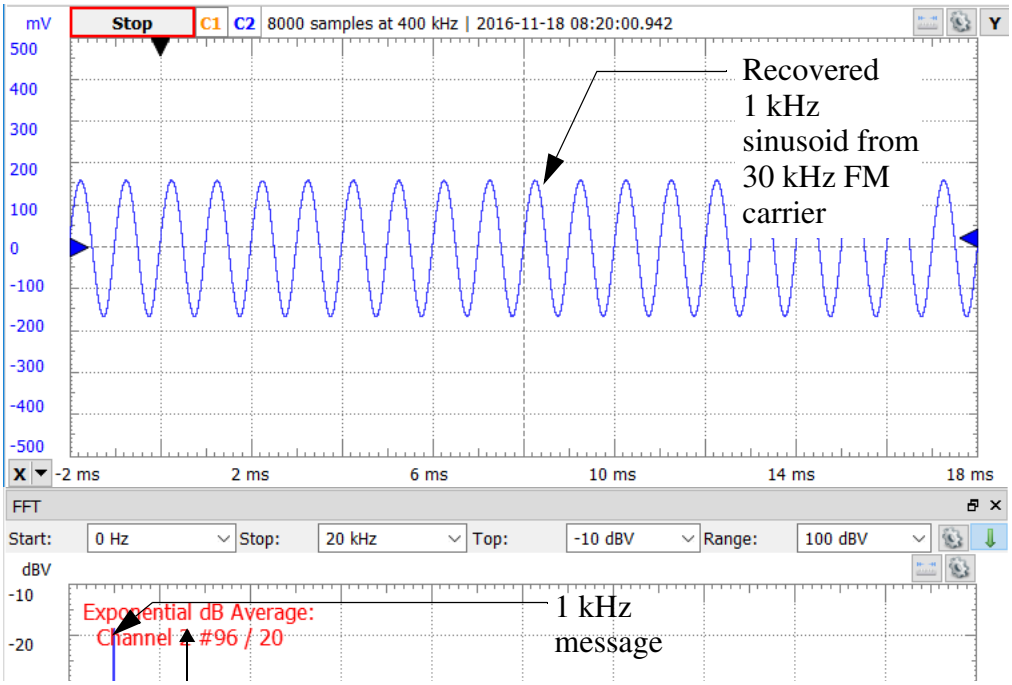


Figure 16: Setting up the function generator in Analog Discovery to produce an FM carrier at 30 kHz with a 1 kHz sinusoidal message at peak deviation of 0.0667 x 30 kHz = 2.0 kHz.

The signal captured at the output of the complex baseband discriminator is shown in Figure 17.



The spectrum is clean with the exception of harmonics at the message frequency. This is expected due to the known nonlinear behavior of the discriminator algorithm.

Expectations

When completed, submit a lab report which documents code you have written and a summary of your results. Screen shots from the scope and any other instruments and software tools should be included as well. I expect lab demos of certain experiments to confirm that you are obtaining the expected results and knowledge of the tools and instruments.

The ZIP package `Lab6.zip`, contains a complete Keil project for the lab and in the Python notebook you will find the supporting Jupyter notebook, "`DDS and Applications.ipnb`", the Python module `dds.py`, a wave file, `Music_Test.wav`, for checking out flanging, and other supporting files, e.g., filter design functions and CSV files created by the Analog Discovery.

Problems

1. **DO f2019** Implement the DDS code described at the bottom of p. 3 on the FM4. The task boils down to porting the C code from GCC code example given earlier to the Cortex-M. You will want to define the accumulator variables as globals and the updates to the accumulator, including the call to `cosf()/arm_cos_f32()`, will be inside the codec ISR. Choose the sampling frequency to be 48 kHz. Setup a GUI slider to allow the frequency to step from 1 Hz to 20 kHz in 1 Hz steps. Note I recommend using the `arm_sin_f32()/arm_cos_f32()` functions over `sinf()/cosf()`, as they execute faster.

   The lab instructor will ask you to demo this via the spectrum analyzer (Agilent 4395A in spectrum analyzer mode) and/or the scope.

2. **DO f2019** Experimentally find the ISR service time of the simple DDS of Problem 1 using the digital I/O technique first described in Lab 3.

3. **OMIT f2019** In Problem 1 the basic DDS calculation is of the form

   ```
   xL = (int16_t) (FM4_GUI.P_vals[0] * 15000*arm_cosf32(angle));
   ```

   An efficient DDS uses a LUT in place of this on-the-fly cosine calculation. In this problem, rather than implementing an actual LUT, you will emulate one by doing some fixed-point conversions in the argument of `arm_cos_f32()`. To start with note that the 15000 scales the `arm_cos_f32()` value to lie near the max amplitude range of an `int32_t` when the GUI scaling is taken into account (recall [-32768, 32767]). The only fixed-point quantization that is taking place presently is casting the cosine values from `float32_t` to `int32_t` as they are output to the audio codec (specifically the DAC).

   To emulate the LUT consider a rework of the original code to set up a float accumulator running from [0, 1), but quantized in the argument of `arm_cos_f32()`. The code below increments the accumulator `a` by $0 < f_0/f_s < 1$ (in code `f0_fs`), for $f_0/f_s = 1000/48000$. The argument of `arm_cos_f32()` is $2\pi \cdot Q_W(f_0/f_s)$, where $Q_W(\ )$ is a W bit quantizer implemented as

$$Q_W(x) = \text{int32}[x \cdot 2^W] \cdot 2^{-W} \quad . \tag{28}$$

Note int32[ ] represents casting a `float32_t` value to an `int32_t` integer. The returned float value of (28) is again a float because 2^{-W} is represented as a `float32_t` constant. The returned value is however quantized. The modified DDS c-code is given below:

```
// DDS variables for fixed 1 kHz and fs = 48000 Hz
float32_t twopi = 6.283185307179586;
float32_t acc = 0;
float32_t f0_fs = 0.020833333333333; // f0/fs = 1000/48000
float32_t a_scale_p = 1<<W; // emulate a table size of 2^W entries
float32_t a_scale_m = 1.0f/(1<<W); // using these scaling constants

...

void PRGCRC_I2S_IRQHandler(void)
{
  union WM8731_data sample;
int16_t xL, xR;
float32_t x, y;
int16_t a_int16;

  gpio_set(DIAGNOSTIC_PIN,HIGH);
// Get L/R codec sample
  sample.uint32bit = i2s_rx();

// Do more processing on LEFT and RIGHT channels
x = (float32_t) sample.uint16bit[LEFT];
//x = (float32_t) (rand_int32()>>4);

// DDS phase accumulator using integers
 a_int16 = (int16_t)(acc*a_scale_p + 0.5f);
 y = arm_cos_f32(twopi*a_int16*a_scale_m);
 acc += f0_fs;
 if (acc >= 1) acc -= 1.0f;

// No processing done to the right channel

// Return L/R samples to codec via C union with slider gain
xL = (int16_t) (FM4_GUI.P_vals[0] * 15000*y);
xR = (int16_t) (FM4_GUI.P_vals[1] * sample.uint16bit[RIGHT]);
sample.uint16bit[LEFT] = xL;
sample.uint16bit[RIGHT] = xR;
  i2s_tx(sample.uint32bit);

  NVIC_ClearPendingIRQ(PRGCRC_I2S_IRQn);

gpio_set(DIAGNOSTIC_PIN,LOW);
}

...
```

a) Using the spectrum analyzer mode of the Agilent 4395A, characterize the spectrum

quality of the 1 kHz output signal for `W=16`. The quantity of interest is the spurious free dynamic range (SFDR) as shown in Figure 8. The SFDR measures the maximum dynamic range between the signal of interest and any adjacent spurs. The ISR code `DDS_ISRs_p3.c` has the frequency set to 1 kHz. Keep that setting for all measurements.

b) Repeat part (a) for `W=12`.

4. **Do f2019** Implement the audio flanging system of Figure 9 using the DDS design of Problem 1 to implement the time delay control oscillator. The starting point will be the variable delay code from Lab 4. For example:

```
...
#define N_buff 481

int32_t rand_int32(void);
int16_t pmod(int16_t a, int16_t b);

// Create (instantiate) GUI slider data structure
struct FM4_slider_struct FM4_GUI;

//Parameters used by circular buffer and extracting a delayed sample
int16_t ptr, delay;
float32_t circbuf[N_buff];

void PRGCRC_I2S_IRQHandler(void)
{
  union WM8731_data sample;
  int16_t xL, xR;
  float32_t x, y;

  gpio_set(DIAGNOSTIC_PIN,HIGH);
  // Get L/R codec sample
  sample.uint32bit = i2s_rx();

  // Do more processing on LEFT and RIGHT channels
  x = -(float32_t) sample.uint16bit[LEFT];

  // Begin buffer processing
  // Write new input over oldest buffer value
  circbuf[ptr] = x;
  // Calculate delay index working backwards
  delay = (int16_t) FM4_GUI.P_vals[2];
  y = circbuf[pmod(ptr - delay,N_buff)];
  // Update ptr to write over the oldest value next time
  ptr = (ptr + 1) % N_buff;

  // Return L/R samples to codec via C union with slider gain
  xL = (int16_t) (FM4_GUI.P_vals[0] * y);
  xR = (int16_t) (FM4_GUI.P_vals[1] * (x));
  sample.uint16bit[LEFT] = xL;
  sample.uint16bit[RIGHT] = xR;
  i2s_tx(sample.uint32bit);

  NVIC_ClearPendingIRQ(PRGCRC_I2S_IRQn);
```

```
  gpio_set(DIAGNOSTIC_PIN,LOW);
}

int main(void)
{
      int16_t m;
      // Initialize the slider interface by setting the baud rate (460800 or 921600)
      // and initial float values for each of the 6 slider parameters

  init_slider_interface(&FM4_GUI,460800, 1.0, 1.0, 10.0, 0.0, 0.0, 0.0);

  // Initialize the delay buffer with zeros
  for (m = 0; m <= N_buff; m++) {
    circbuf[m] = 0.0f;
  }
  // Initialize the circular buffer pointer
  ptr = 0;
  delay = 10;
...
}
```

The design requirements on the flanger parameters are: delay mix gain, $0 \le \alpha \le 1$, delay oscillation frequency range, $0.1 \le \beta \le 10$ Hz, and peak delay tunable over $10 \le D_p \le 200$ samples. The variable delay thus needs to contain at least 400 states. Padding the delay length up 500 samples is a good starting point. Use the left in and out channels for $x[n]$ and $y[n]$ respectively. Send a scaled version of the delay control signal to the right channel output, e.g.,

```
xR = (int16_t) (FM4_GUI.P_vals[1] * (delay*10));
```

Note delay is created as in `int16_t` type.

a) As an initial test of the flanger input a 5000 Hz sinusoidal signal and observe the spectrum of the output with $D_p = 50$, $f_m = 5$ kHz, and $f_0 = 20$ Hz. Set the gain mix slider $\alpha = 1$ so hat only the delay output is sent to the codec left channel. Observe the FM-like spectrum on the spectrum analyzer and compare it with the theoretical spectrum calculated using the examples found in the provided "`DDS and Applications.ipnb`" Jupyter notebook. Once you have good agreement between theory and measured, you can move on to experience flanging in part (b).

b) Use the supplied wave file `Music_Test.wav` to output audio from the PC headphone jack into the Line-in jack of the FM4. The program material is a keyboard/synthesizer vamp with a simple rhythm section. I suggest you keep the flanging rate f_0 between 0.1 and 1 Hz. Try different mix values α and peak delay D_p. Demo to your instructor. If you have other audio tracks that in your opinion reveal the flanging effect more clearly, go for it. You also need to experience what happens when f_0 goes above 1 Hz. Comment on what you hear.

5. **OMIT f2019 (Lab demo?)** In this problem you will explore the FM receiver design of Part II. As an FM signal source you will use the internal FM capability of the Agilent 33250 function generator or the two channel generator capability of the MSOX6004A scope found

at your lab bench. Your lab instructor will help you with the set-up of the generator.

a) Write C code to implement the FM receiver described in Figure 12. As a starting point find the file `FM_Demod_ISRs.c` as the starting point.

b) Configure the Agilent 33250 to produce a sinusoidal FM signal having $f_c = 30\,\text{kHz}$, a modulation frequency of 1 kHz, and a peak frequency deviation in the range of 1 to 3 kHz. Start with a 2 kHz deviation as discussed in the background section.

c) Experiment with mistuning the frequency of the DDS relative to the known FM signal carrier at 30 kHz. How far above and below 30 kHz can you tune the DDS without resulting in a heavily distorted demodulated 1 kHz sinusoid?

References

[1] Michael Rice, *Digital Communications: A Discrete-Time Approach*, Prentice Hall, New Jersey, 2009.

[2] Analog Devices MT-85 Tutorial, *Fundamentals of Direct Digital Synthesis (DDS)*, 2009, `http://www.analog.com/static/imported-files/tutorials/MT-085.pdf`.

[3] Xilinx LogiCore, *DS246 Product Specification v5.0*, April 28, 2005. `http://www.xilinx.com/support/documentation/ip_documentation/dds.pdf`.

[4] Thad Welch, Cameron Wright, and Michael Morrow, *Real-Time Digital Signal processing from MATLAB to C with the TMS320C6x DSPs* second edition, CRC Press, 2012.

[5] R. Ziemer and W. Tranter, *Principles of Communications*, seventh edition, Wiley, 2014.

Appendix

Dealing with Spurs

Spurs are a known fact of DDS implementations. Design techniques to mitigate spurs are described in [1],[2], and [3]. The root cause of spurs is the fact that the quantizer $Q(\ )$ introduces phase error

$$\delta\theta[n] = \hat{\theta}[n] - \theta[n]. \tag{29}$$

The accumulator output driving the LUTs is of the form

$$\hat{\theta}[n] = \theta[n] + \delta\theta[n]. \tag{30}$$

In the generation of a complex sinusoid (sin and cos), i.e., $e^{j\hat{\theta}[n]} = \cos(\hat{\theta}[n]) + j\sin(\hat{\theta}[n])$, you can write

$$\begin{aligned} \hat{e^{j\theta[n]}} &= e^{j\theta[n]} \cdot e^{j\delta\theta[n]} = e^{j\theta[n]}\{\cos(\delta\theta[n]) + j\sin(\delta\theta[n])\} \\ &= e^{j\theta[n]} \cdot \{1 + j\delta\theta[n]\} \end{aligned} \tag{31}$$

assuming the error is small. The error phase is also a periodic ramp signal so it effectively modulates $\theta[n]$ via the term $j\delta\theta[n]e^{j\theta[n]}$. This action produces sidebands at frequency offsets from the fundamental frequency (4 kHz in the case of Figure 6b).

One mitigation approach is to inject a small random dithering signal at the input to the quantizer as shown in Figure 18. The idea is that the dithering signal disrupts the periodicity and replaces it with a dominant random phase error [1]. The corresponding error spectrum is transformed from

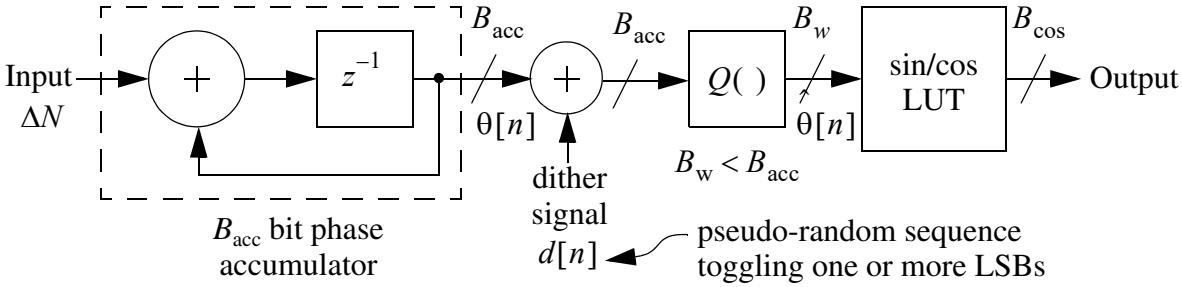


Figure 18: Finite precision DDS with dithering to mitigate spurs.

spectral lines to a flat noise-like spectrum across the entire spectrum. The spectrum now has a noise floor, but the spurs are gone and/or reduced. The module `dds.py` contains `DDS_dither()`:

```
def DDS_dither(f0,fs,N_samps,Bcos,Bacc,Bw):
    """
     x,a_out,n = DDS(f0,fs,N_samps,Bcos,Bacc,Bw)
     ///////////////// Inputs //////////////////////
          f0 = desired output frequency
          fs = sampling frequency
     N_samps = number of samples to simulate
        Bcos = bit width of cos/sin values
        Bacc = bit width of the accumulator
          Bw = bit width of the LUT address
     ///////////////// Outputs //////////////////////
           x = output signal
       a_out = accumulator normalized to a [0,1) float value
           n = time index

    Mark Wickert November 2016
    """
    n = np.arange(N_samps-1)
    x = np.zeros(len(n))
    a_out = np.zeros(len(n))
    a = 0
    w = 0
    theta = 0
    for k in range(len(n)):
        #x[k] = np.cos(2*np.pi*a)
        x[k] = ssd.simpleQuant(np.cos(2*np.pi*w/2**Bw),Bcos,1,'none')
        a += round(f0/fs*2**Bacc)
        if a >= 2**Bacc:
```

```
            a -= 2**Bacc
        a = a + np.random.randn(1)*2**(Bacc - Bw - 3); # Dithering added here
        w = round(a/2**(Bacc-Bw))
        a_out[k] = w/2**Bw
    return x, a_out, n
```

Reworking the results of Figure 6, you now have the results shown in Figure 19.

```
x16_32_16, a_out, n = dds.DDS(14,48.0,10000,16,32,16)
x16_32_16d, a_out, n = dds.DDS_dither(14,48.0,10000,16,32,16)
subplot(211)
psd(x16_32_16,2**12,48);
title(r'$B_{\cos} = 16$, $B_{acc} = 32$, $B_{w} = 16$, no dithering')
ylabel(r'PSD (dB)')
xlabel(r'Frequency (kHz)')
ylim([-140,10])
subplot(212)
psd(x16_32_16d,2**12,48);
title(r'$B_{\cos} = 16$, $B_{acc} = 32$, $B_{w} = 16$, dithering')
ylabel(r'PSD (dB)')
xlabel(r'Frequency (kHz)')
ylim([-140,10])
tight_layout()
```

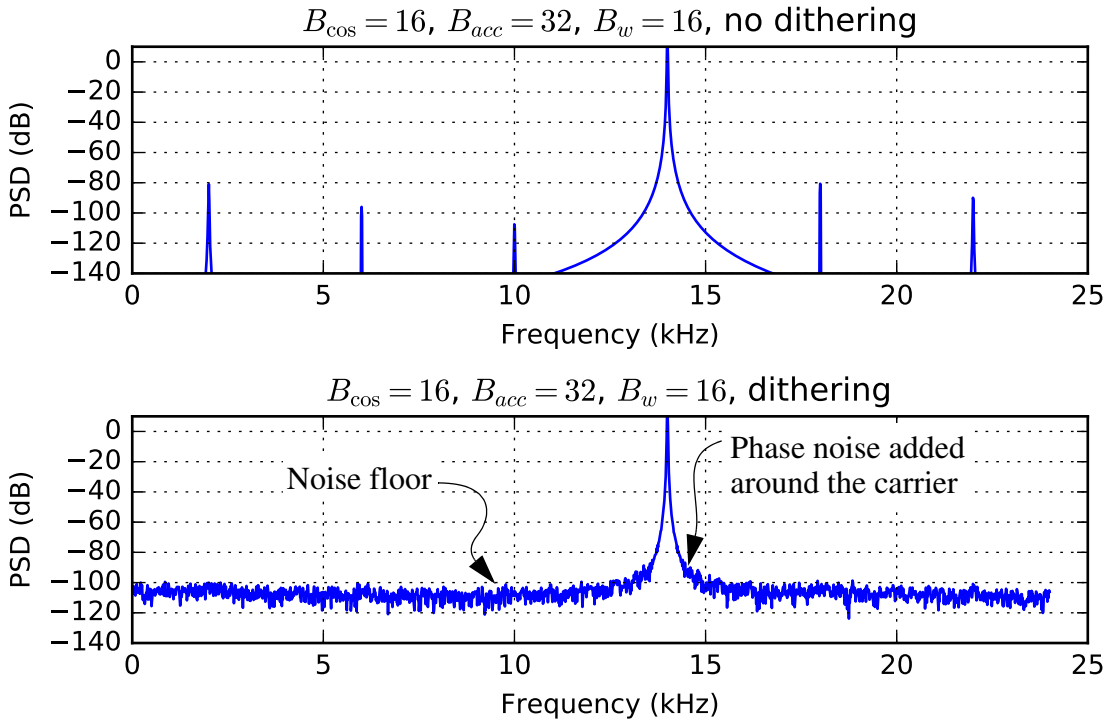


Figure 19: DDS output at 14 kHz for $f_s = 48$ kHz, $B_{\text{acc}} = 32$, $B_w = 16$, and $B_{\cos} = 16$ without (above) and with (below) dithering.

Listing for `dds.py`

```
"""
```

```
Direct Digital Synthesis Simulation

Mark Wickert November 2016

Development continues!
"""

"""
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
"""

import numpy as np
import ssd
from scipy import signal

def DDS(f0,fs,N_samps,Bcos,Bacc,Bw):
    """
     x,a_out,n = DDS(f0,fs,N_samps,Bcos,Bacc,Bw)
     ///////////////// Inputs //////////////////
          f0 = desired output frequency
          fs = sampling frequency
     N_samps = number of samples to simulate
        Bcos = bit width of cos/sin values
        Bacc = bit width of the accumulator
          Bw = bit width of the LUT address
     ///////////////// Outputs //////////////////
           x = output signal
       a_out = accumulator normalized to a [0,1) float value
           n = time index

    Mark Wickert November 2016
    """
    n = np.arange(N_samps-1)
    x = np.zeros(len(n))
    a_out = np.zeros(len(n))
    a = 0
    w = 0
    theta = 0
    for k in range(len(n)):
        #x[k] = np.cos(2*np.pi*a)
        x[k] = ssd.simpleQuant(np.cos(2*np.pi*w/2**Bw),Bcos,1,'none')
        a += round(f0/fs*2**Bacc)
        if a >= 2**Bacc:
            a -= 2**Bacc
        w = round(a/2**(Bacc-Bw))
        a_out[k] = w/2**Bw
    return x, a_out, n
```

```
def DDS_dither(f0,fs,N_samps,Bcos,Bacc,Bw):
    """
     x,a_out,n = DDS(f0,fs,N_samps,Bcos,Bacc,Bw)
     ///////////////// Inputs //////////////////////
          f0 = desired output frequency
          fs = sampling frequency
     N_samps = number of samples to simulate
        Bcos = bit width of cos/sin values
        Bacc = bit width of the accumulator
          Bw = bit width of the LUT address
     ///////////////// Outputs //////////////////////
           x = output signal
       a_out = accumulator normalized to a [0,1) float value
           n = time index

    Mark Wickert November 2016
    """
    n = np.arange(N_samps-1)
    x = np.zeros(len(n))
    a_out = np.zeros(len(n))
    a = 0
    w = 0
    theta = 0
    for k in range(len(n)):
        #x[k] = np.cos(2*np.pi*a)
        x[k] = ssd.simpleQuant(np.cos(2*np.pi*w/2**Bw),Bcos,1,'none')
        a += round(f0/fs*2**Bacc)
        if a >= 2**Bacc:
            a -= 2**Bacc
        a = a + np.random.randn(1)*2**(Bacc - Bw - 3); # Dithering added here
        w = round(a/2**(Bacc-Bw))
        a_out[k] = w/2**Bw
    return x, a_out, n


def DDS_float(f0,fs,N_samps,Bcos=32):
    """
     x,a_out,n = DDS_float(f0,fs,N_samps,Bcos,Bacc,Bw)
     ///////////////// Inputs //////////////////////
          f0 = desired output frequency
          fs = sampling frequency
     N_samps = number of samples to simulate
        Bcos = float cos size: 32 or 64
     ///////////////// Outputs //////////////////////
           x = output signal
       a_out = accumulator normalized to a [0,1) float value
           n = time index

    Mark Wickert November 2016
    """
    n = np.arange(N_samps-1)
    x = np.zeros(len(n))
    a_out = np.zeros(len(n))
    a = 0
    pi32 = np.float32(np.pi)
    pi64 = np.float64(np.pi)
    theta = 0
    for k in range(len(n)):
```

```
        if Bcos == 32:
            x[k] = np.cos(a,dtype=np.float32)
            a += np.float32(2*pi32*f0/fs)
            if a >= 2*pi32:
                a -= 2*pi32
            a_out[k] = a/(2*pi32)
        else:
            x[k] = np.cos(a,dtype=np.float64)
            a += np.float64(2*pi64*f0/fs)
            if a >= 2*pi64:
                a -= 2*pi64
            a_out[k] = a/(2*pi64)
    return x, a_out, n


def discrim(x):
    """
    disdata = discrim(x)
    where x is an angle modulated signal in complex baseband form.

    Mark Wickert
    """
    X=np.real(x)        # X is the real part of the received signal
    Y=np.imag(x)        # Y is the imaginary part of the received signal
    b=np.array([1, -1]) # filter coefficients for discrete derivative
    a=np.array([1, 0])  # filter coefficients for discrete derivative
    derY=signal.lfilter(b,a,Y)  # derivative of Y,
    derX=signal.lfilter(b,a,X)  #    "          X,
    disdata=(X*derY-Y*derX)/(X**2+Y**2)
    return disdata

def FM_gen(m,f0,fs,real_output = True):
    """
    x,a_out = FM_gen(m,f0,fs)
        m = message signal, amp. sets Df in fs units
       f0 = carrier frequency
       fs = sampling rate
        x = output waveform
    a_out = [0,1] accumulator output

    Mark Wickert November 2016
    """
    a_out = np.mod(np.cumsum(f0/float(fs) + m/float(fs)),1) # avoid for loop
    x = np.exp(1j*(2*np.pi*a_out)) # start with complex
    if real_output:
        x = x.real
    return x, a_out
```