Abstract

In the first experiment of this laboratory, we examine a source code example of a cascaded biquad IIR filter implemented in fixed-point arithmetic. The second task deals with zero-input limit cycles in fixed-point IIR filters.

1 Theoretical Overview

1.1 Limit Cycles

A limit cycle\(^1\) is an isolated closed trajectory in the state-space of a system (see \[1\]). Whenever the system’s state advances on a closed trajectory, the system exhibits oscillations. Limit cycles are inherently nonlinear phenomena—they can’t occur in linear systems. Of course, a linear system of at least second order can produce oscillations (e.g., a linear filter with a pole on the unit circle), but their closed trajectories are not isolated (neighboring trajectories are closed too—the amplitude of the oscillation of a linear system is set entirely by the initial conditions).

When we implement an IIR filter (feedback loops) with finite-precision arithmetic, we will get a nonlinear dynamic system, and therefore oscillations may occur even when the (quantized) filter coefficients yield a stable (linear) filter. We distinguish between

1. limit cycles due to overflow (modulo behavior) after accumulation and

2. limit cycles due to quantization (truncation or rounding) after multiplication or accumulation.

The first produces oscillations with high amplitudes, whereas the second exhibits oscillations with low amplitudes, which are disturbing especially if no input signal is applied to the filter (zero input). Fortunately, limit cycles of the first type can be avoided in second-order systems by using a saturation characteristic. For more information refer to \[2\].

\(^{1}\)ger: Grenzzyklus
1.2 Data Types and Word Lengths

Make sure that you are familiar with data types to read assembly code. The C/C++ data types and word lengths are as follows:

- bool: 8 bits (top 7 bits ignored)
- char: 8 bits
- short: 16 bits
- int: 32 bits
- long: 32 to 64 bits
- long long: 64 bits
- float: 32 bits
- double: 64 bits
- long double: 96 bits
- pointer: 32 bits

1.3 Assembly Routine \texttt{iir\_cas4}( )

You will need the code of the following assembly routine in experiment 1: fixed-point implementation of a cascade form IIR filter. Before attending this laboratory unit, make sure that you are familiar with right-shifts, numerical formats, and biquad filters.

```c
void iir_cas4( int n_cas, short *coeffs, int *states, int *io)
{
    int k0, k1, i;
    for( i = 0; i < n_cas; i++)
    {
        k0 = coeffs[4*i+1] * (states[2*i+1] >> 16) + coeffs[4*i+0] * (states[2*i+0] >> 16) + io[0];
        io[0] = coeffs[4*i+3] * (states[2*i+1] >> 16) + coeffs[4*i+2] * (k0 >> 16) + k0;
        states[2*i+1] = k0;
        k1 = coeffs[4*i+1] * (states[2*i+0] >> 16) + coeffs[4*i+0] * (k0 >> 16) + io[1];
        states[2*i+0] = k1;
    }
}
```

Figure 1: \textit{C equivalent of the assembly routine iir\_cas4.}
1.4 Single-Sideband Modulation Using a Hilbert Transformer

We use the Hilbert transformer to generate the analytic version \( x_a[n] \) of a real-valued signal \( x[n] \). The Fourier transform of an analytic signal is zero for all negative frequencies:

\[
X_a(e^{j\theta}) = 0, \quad -\pi \leq \theta < 0. \tag{1}
\]

Consequently, \( x_a[n] \) must be a complex-valued sequence, because real-valued signals have Hermitian (conjugate symmetric) Fourier transforms:

\[
X(e^{j\theta}) = X^*(e^{-j\theta}).
\]

The (ideal) Hilbert transformer is an allpass filter that shifts the phase by 90 degrees. Its frequency response is

\[
H_{hil}(e^{j\theta}) = \begin{cases} 
-j, & 0 < \theta < \pi \\
j, & -\pi < \theta < 0
\end{cases} \tag{2}
\]

and the (infinite and anticausal) impulse response is

\[
h_{hil}[n] = \begin{cases} 
0, & n \text{ even} \\
\frac{2}{\pi n}, & n \text{ odd}
\end{cases} \tag{3}
\]

In this laboratory, we use an FIR approximation of the transformer by windowing the shifted infinite impulse response with \( w[n] \) (e.g., a Kaiser window). Note, this implementation yields a linear-phase filter and we have to compensate for it’s delay to synchronize real and imaginary parts in order to get the proper analytic signal (see Figure 2).

In frequency-division multiplexing, modulating analytic signals instead of the real-valued ones halves the requirement of bandwidth. This kind of amplitude modulation is called single-sideband modulation.

In the common (double-sideband) modulation, the real-valued signal \( x[n] \) modulates the amplitude of the also real-valued sinusoidal carrier \( \cos(\theta_0 n) \):

\[
y_{dsb}[n] = x[n] \cos(\theta_0 n) = \frac{1}{2} x[n] e^{j\theta_0 n} + \frac{1}{2} x[n] e^{-j\theta_0 n}. \tag{4}
\]

The Fourier transform of \( y_{dsb}[n] \) is illustrated in Figure 3(b).

For single-sideband modulation, we take the analytic signal \( x_a[n] \) and shift it to the frequency \( \theta_0 \) by multiplication with the also analytic carrier \( e^{j\theta_0 n} \). Finally, we take the real part

\[\footnotesize{\text{In order to facilitate simple demodulation, we assume the signal } x[n] > 0. \text{ Thus, we ensure that the signal is uniquely represented by the (always positive) envelope of the modulated carrier. To achieve this requirement, a DC offset can be added to } x[n] \text{ before modulation.}}\]
of this shifted analytic signal:

\[ y_{ssb}[n] = \text{Re}\{x_a[n]e^{j\theta_0n}\} = \text{Re}\left\{\left(\text{Re}\{x_a[n]\} + j\text{Im}\{x_a[n]\}\right)\left(\cos(\theta_0n) + j\sin(\theta_0n)\right)\right\} = \text{Re}\{x_a[n]\}\cos(\theta_0n) - \text{Im}\{x_a[n]\}\sin(\theta_0n). \]  

(5)

The Fourier transform of \(y_{ssb}[n]\) is illustrated in Figure 3(c) and a block diagram of the last equation in Figure 4.

Figure 3: Double-sideband modulation (b) versus single-sideband modulation (c) of the signal in (a).

Figure 4: Single-sideband modulator according to Equation (5).
2 Practical Part

Experiment 1
Fixed-Point Implementation of a Cascade Form IIR Filter

Equipment: –
Software: –

1. Examine the C code in Figure 1. This is the C equivalent of the hand-optimized assembly routine iir_cas4() taken from an older version of TI’s C62x DSPLIB.

   (a) Draw a signal flow graph of a part of the routine’s code shown in Figure [1] consider lines 4 to 7 only. Which filter is it?

   (b) Draw the block diagram of the filter represented by the code inside the routine’s for-loop.

   (c) Explain the meaning of a state in the given architecture.

   (d) What is the state space of the given digital filter and what’s the space’s dimension if the filter’s order is one?

2. Answer the following questions:

   (a) How many multiplications does the routine perform per input/output-sample? How many samples does the routine process during an iteration (an instant of time)?

   (b) Which numerical operation is equivalent to a 16-bit right-shift?

   (c) Why is it necessary to shift the states in this example?

   (d) Which numerical format (number of bits for the integer portion and for the fractional portion) is expected for the filter coefficients and for the input/output signal?

   

<table>
<thead>
<tr>
<th>N-1</th>
<th>N-2</th>
<th>N-3</th>
<th>N-4</th>
<th>N-5</th>
<th>...</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>...</td>
<td>v</td>
<td>w</td>
</tr>
</tbody>
</table>

   (e) Sometimes, non-linear effects and interferences may change the state of one or more bits accidently. The system may become unstable. What is the numerical range of the filter coefficients to ensure a stable filter behavior by considering the numerical format of the coefficients given in this experiment? Hints: (1) Write down the transfer function of the biquad filter and examine the denominator; (2) Assume complex conjugated poles and the numerical range of the filter’s coefficients calculated in (d).

   (f) Draw the area in the z-plane where poles can be placed. Does this routine represent an implementation of a general IIR filter?

   (g) How many bits do you have to consider at most for the right-shift to place poles anywhere inside the unit circle?

---

3Note that a filter-architect may choose between different implementations, e.g., FIR/IIR, DF1/DF2, etc.

4How many bits does the result of a summation and a multiplication of two 16-bit numbers exhibit? Which numerical formats exhibit both binary numbers?


**Experiment 2**

Limit Cycle due to ...?

Equipment: PC, Raspberry Pi, headphones
Software: NetBeans, download filt2.zip from the course website and unzip it on your workstation

1. Plug the output of the PC soundcard (or portable media player) to the Raspberry Pi line–in and the headphones to the Raspberry Pi line–out.

2. In NetBeans, open the project provided in the extracted folder.

3. Edit the filter coefficients in iir2.cpp according to the following table ($Q_{15} = 2^{15}$ is already defined).

<table>
<thead>
<tr>
<th>b(0)</th>
<th>b(1)</th>
<th>b(2)</th>
<th>a(1)</th>
<th>a(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_{15}$</td>
<td>0</td>
<td>0</td>
<td>$-Q_{15} \times \frac{3}{4}$</td>
<td>$Q_{15} \times \frac{3}{4}$</td>
</tr>
</tbody>
</table>

Draw an exact block-diagram of this system. What type of filter would you get when setting $a(1) = a(2) = 0$?

4. Build the program, load it to the Raspberry Pi, and run it. Provide an signal to the input and listen to the output. If you can hear a non-distorted output signal we can assume the filter works properly.

5. Edit the file iir2.cpp again. Set the initial conditions to

<table>
<thead>
<tr>
<th>y(-2)</th>
<th>y(-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-Q_{15} \times \frac{4}{5}$</td>
<td>$Q_{15} \times \frac{4}{5}$</td>
</tr>
</tbody>
</table>

6. Rebuild the program and load it to the Raspberry Pi. **Before** you run the program, ensure the amplitude of the provided input signal is zero (e.g., use the soundcard’s mixer to set the volume) and ensure you do not wear the headphones. What can you acoustically observe at the output? Use an oscilloscope to specify the audible artifact.

7. Continuously increase the amplitude (volume) of the provided input signal. Does this change the program’s behavior?
   
   Answer whether these samples can be the zero-input response of a stable linear system or not (explain). Analyze the source code in iir2.cpp and find out whether there is a nonlinearity that causes the observed behavior.

8. Edit the file iir2.cpp again. Try to find a way to get your program to show the output signal (after the filtering), either by printing them or by using the debug feature. How can you connect the values of the output signal to what you can see when examining them with the oscilloscope?
Experiment 3
Hilbert Filter and Single-Sideband Modulator

Equipment: PC
Software: MATLAB

1. Open Matlab and create a new script.

2. Compute the impulse response $h[n]$ of a length $M = 127$ Hilbert filter using Equation (3). Hint: $h[n]$ has to be symmetric around zero.

3. What is a common problem with this (truncated) implementation? How can you improve the implementation?

4. Now use a Kaiser window with $\beta = 2.63$ and apply it to the response of your Hilbert filter. You can use the Matlab function `kaiser()` to create the window function $w[n]$. Use `freqz()` to compare the response with and without windowing.

5. In general, what is a fast way to implement filtering? What do you have to be careful of?

6. Try to implement your own Hilbert filter and compare the output with the output obtained when using the Matlab function `hilbert()`. Hint: Be careful of the additional time delay needed to synchronize the real and imaginary part for your own implementation (see Figure 2).

7. Why is the additional delay needed?

8. Use a Hilbert filter to perform single-sideband modulation according to Equation (5) on the signal

   \[ x[n] = 1 + \cos(\theta n). \]  

   You can choose $\theta$ at will, but be careful to set $\theta_0$ (the sideband center frequency as indicated in Figure 5) to a suitable value to be able to see the effects of the modulation.

9. Examine the spectrum of the input signal $x[n]$ and the resulting single-sideband modulated signal $y_{ssb}[n]$.

References

