DSP Implementation of FIR Filters

Overview

- LTI Systems
- FIR filter design methods
- DSP implementation using circular buffers
- Compiler optimization
- Lab 3: FIR Filtering
### Linear Time Invariant (LTI) Systems

A system with input $x(n)$ and output $y(n)$ is LTI if

\[
\alpha_1 x_1(n) + \alpha_2 x_2(n) \rightarrow \alpha_1 y_1(n) + \alpha_2 y_2(n)
\]

\[
x(n - k) \rightarrow y(n - k)
\]

An LTI system is completely characterized by its impulse response

\[
\begin{aligned}
\delta(n) & \rightarrow \text{LTI System} \rightarrow h(n) \\
y(n) &= \sum_{k=-\infty}^{\infty} x(k) h(n - k) = \sum_{k=-\infty}^{\infty} h(k) x(n - k)
\end{aligned}
\]

### Sinusoidal Steady-State Response

Consider passing a complex exponential through an LTI system:

\[
x(n) = e^{j\omega_0 n}
\]

Convolution sum gives us

\[
\begin{aligned}
y(n) &= \sum_{k=-\infty}^{\infty} h(k) e^{j\omega_0 (n-k)} \\
&= e^{j\omega_0 n} \sum_{k=-\infty}^{\infty} h(k) e^{-j \omega_0 k} \\
y(n) &= H(e^{j\omega_0}) e^{j\omega_0 n}
\end{aligned}
\]

Complex exponentials are eigenfunctions of LTI systems

\[
e^{j\omega_0 n} \rightarrow H(e^{j\omega_0}) e^{j\omega_0 n}
\]
$H(e^{j\omega})$ is the frequency response of the system:

$$H(e^{j\omega}) = \sum_{n=-\infty}^{\infty} h(n)e^{-j\omega n} \triangleq A(\omega)e^{j\theta(\omega)}$$

so we have

$$y(n) = H(e^{j\omega})e^{j\omega_0 n} = A(\omega_0)e^{j[\omega_0 n + \theta(\omega_0)]}$$

where

Amplitude Response: $A(\omega) = |H(e^{j\omega})|$  
Phase Response: $\theta(\omega) = \text{arg}\{H(e^{j\omega})\}$  

$2\pi$-periodic

FIR Filter Design

- References

- Methods
  - Window Method
  - Parks-McClellan (optimum equiripple filters)
Window Method

To design an N-tap FIR filter with ideal transfer function

\[
H_d(e^{j\omega}) = \begin{cases} 
1 & \omega \leq \omega_c \\
0 & \omega_c < |\omega| \leq \pi 
\end{cases}
\]

1. Take the inverse DTFT to obtain desired impulse response

\[
h_d(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} H_d(e^{j\omega}) e^{j\omega n} d\omega = \frac{\sin(\omega_c n)}{\pi n}
\]

2. Delay \(h_d(n)\) and truncate to obtain a causal FIR filter

\[
h(n) = h_d\left(n - \frac{N-1}{2}\right) w(n)
\]
31-Tap LPF with $\omega_c = \pi/4 \approx 0.785$

- Rectangle
- Hamming
- Blackman

FIR Filters
- Lack precise control over critical frequencies
- No obvious way to determine filter length needed to meet specs
- Use `WINDOW.EXE` or Matlab (fir1, fir2, firrcos, hamming, etc.)

**Optimum Equiripple FIR Filters**

- Typically filters are described by a brickwall specification

\[
1 - \delta_1 \leq A(\omega) \leq 1 + \delta_1 \quad \text{Passband Regions}
\]

\[
A(\omega) \leq \delta_2 \quad \text{Stopband Regions}
\]

LPF specs (linear gain) LPF specs (dB loss)
• Design viewed as opt problem which seeks to min weighted error between actual and desired responses in specified bands

• Optimum in the sense that
  – Weighted error is dist. evenly over pass and stop bands
  – Maximum error is minimized

• Solution by Parks and McClellan

FIR Filters

Filter design programs

• DOS: REMEZ87.EXE returns filter coefficients when given
  – Edge frequencies
  – Filter order (specifies absolute error)
  – Weights (determine how error is distributed over the bands)

• Matlab:
  – remezord(): Determines filter order which meet the specs
  – remez(): Similar to REMEZ87.EXE

FIR Filters
**DSP Implementation of FIR Filters**

For an $N$-Tap FIR Filter

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n - k) = \sum_{k=n-N+1}^{n} x(k)h(n - k)$$

we have the following block diagram representation

```
   x(n)   z^-1   z^-1   z^-1   z^-1
        h(0)   h(1)   h(2)   h(N-2) h(N-1)
        ⊕      ⊕      ⊕      ⊕      y(n)
```

A brute force approach shifts values in memory as shown above.

<table>
<thead>
<tr>
<th>Array Index</th>
<th>Filter Coef (hr)</th>
<th>Input Buffer (xbuf)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n = 3$</td>
<td>$n = 4$</td>
</tr>
<tr>
<td>0</td>
<td>$h(3)$</td>
<td>$x(0)$</td>
</tr>
<tr>
<td>1</td>
<td>$h(2)$</td>
<td>$x(1)$</td>
</tr>
<tr>
<td>2</td>
<td>$h(1)$</td>
<td>$x(2)$</td>
</tr>
<tr>
<td>3</td>
<td>$h(0)$</td>
<td>$x(3)$</td>
</tr>
</tbody>
</table>

$$y(n) = \sum_{k=0}^{3} h_r(k)x_{buf}(k)$$

$$x_{buf}(0) = x_{buf}(1)$$

$$\vdots$$

$$x_{buf}(3) = \text{new sample}$$
A better approach remembers the oldest sample and overwrites it.

<table>
<thead>
<tr>
<th>Array Index</th>
<th>Filter Coef (hr)</th>
<th>Input Buffer (xbuf)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$h(3)$</td>
<td>$x(0)^o$</td>
</tr>
<tr>
<td>1</td>
<td>$h(2)$</td>
<td>$x(1)$</td>
</tr>
<tr>
<td>2</td>
<td>$h(1)$</td>
<td>$x(2)$</td>
</tr>
<tr>
<td>3</td>
<td>$h(0)$</td>
<td>$x(3)$</td>
</tr>
</tbody>
</table>

\[ y(n) = \sum_{k=0}^{3} hr(k)xbuf(oldest + k) \]

\[ xbuf(oldest) = \text{new sample} \]

\[ \text{oldest} = \text{oldest}++ \]

Need to make sure we don’t go outside the array.

---

**FIR Filters**

**Convolution Using a Circular Buffer**

<table>
<thead>
<tr>
<th>Index</th>
<th>Filter Coef. hr</th>
<th>Circular Buffer xcirc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$h[N - 1]$</td>
<td>$x[n - oldest + 1]$</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>newest</td>
<td>\vdots</td>
<td>$x[n]$</td>
</tr>
<tr>
<td>oldest</td>
<td>\vdots</td>
<td>$x[n - N + 1]$</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>$N - 1$</td>
<td>$h[0]$</td>
<td>$x[n - oldest]$</td>
</tr>
</tbody>
</table>

\[ y(n) = \sum_{k=0}^{N-1} hr(k)xcirc((oldest + k) \mod N) \]

**FIR Filters**
Circular Buffers Using C

```c
xcirc[oldest] = newsample; /* Put sample buffer */
oldest = (oldest + 1) % N; /* Circularly inc ptr */
y = 0;
for (k = 0; k < N; k++) /* Convolution sum */
    y += hr[k]*xcirc[(oldest + k) % N];
```

- **oldest** must be $↑ \mod N$ rather than $↓$ because the C mod operation does not force negative numbers into the range \{0,...,N − 1\}.
- The % operator calls a library function which might be very inefficient. Try estimating the number of filter taps that can be realized by the following code.

More Efficient MOD Operator

```c
int oldest;
int x_index;
float y;

xcirc[oldest++] = newsample;
if(oldest == N) oldest = 0;

y = 0.0;
x_index = oldest;
for (k = 0; k < N; k++){
    y += hr[k]*xcirc[x_index++];
    if(x_index == N) x_index=0;
}
```
Compiler Optimization

- To implement the longest possible FIR filter, you will need to optimize your code and select the appropriate compiler options.

- **Project -> Build Options -> Basic**
  - Select **No Debug** (removes `-g`):
    - Debug info is useful when initially creating code, but reduces amount of optimization.
  - Select **File Optimization** (adds `-o3`):
    - Highest level of opt including loop unrolling and software pipelining. File level characteristics used to improve perf.
  - Select **Speed Most Critical** (removes `-ms`):
    - Allows tradeoff between code size and speed.
  - Select appropriate **Program Level Opt** (adds `-pm -op#`)

- **Project -> Build Options -> Advanced**
  - Select appropriate **RTS Modifications** option (adds `-ol#`)
  - Select **No Bad Alias Code** (adds `-mt`)

- **Project -> Build Options -> Preprocessor**
  - Under define symbols add `CHIP_6701`

- Since the filter coefficients will not change, declare the array `hr[n]` as `const` and initialize it in the declaration.

- You can tell the compiler that a given pointer is the only pointer to access a block of memory with the **restrict** qualifier. This will aid the pipelining process.

- Select **670x** as the target processor (`-mv6700`) so the compiler will generate code that utilizes the floating point capabilities of the hardware instead of slower software implementations.

FIR Filters
• Where it might be ambiguous, use the \#MUST\_ITERATE pragma to tell the compiler the minimum and maximum number of times a loop will run.

• For more information on code optimization refer to
  – *C6000 Optimizing Compiler User’s Guide* (spru187i)
  – *C6000 Programmer’s Guide* (spru198f)
  – *C6x Code Development Flow* (spra518)
  – *Memory Alias Disambiguation on the C6000* (spra658)

---

**Lab 3: FIR Filtering**

1. Measure & sketch the EVM frequency response for \( F_s = 16 \) kHz

2. Consider an FIR filter with impulse response

\[
h[n] = \begin{cases} 
\frac{1}{3} & 0 \leq n \leq 2 \\
0 & 3 \leq n \leq M - 1 
\end{cases}
\]

   (a) Implement this filter for \( M = 3 \) and a 16 kHz sampling rate, and sketch the frequency response.

   (b) Increase \( M \) and include the zero multiplies in your code. The filter response should not change. Use this to determine the maximum filter length you can implement.

   (c) Change the sampling frequency to 32 kHz and repeat.
3. Use `remezord()` and `remez()` in Matlab to design an equiripple FIR filter meeting the following specifications

\[
\begin{align*}
A(f) &= 1 + \delta_1 \\
1 &- \delta_1 \\
\delta_2 &
\end{align*}
\]

\[f_1, f_2, f\]

\[F_s = 16 \text{ Hz}, f_1 = 2.5 \text{ kHz}, f_2 = 3.2 \text{ kHz}, \delta_1 = 0.001, \delta_2 = 0.005\]

4. Use `fir1()` in Matlab to design an LPF with \(f_c = 2.8 \text{ kHz}\), a rectangular window, and the same length as in (3).