1 Lecture Outline

Reading: Chapter 7 Digital Filter Realizations

- Linear Delay Line (Section 4.2.1)
- Circular Buffer (Section 4.2.4)
- C Implementation of FIR Digital Filters
- C Implementation of Sample-by-Sample Processing

2 Linear Delay-Line

In the implementation of digital filters, we must efficiently implement delay chains or delay-line buffers as in the example below.

Figure 1: Orfanidis p. 153 Figure 4.2.6 delay line buffer.

In this example, the internal filter states at time n of the delay line are given by [for notational simplicity, we write $w_i(n)$ as w_i]

$$\begin{aligned} w_0 &= x(n) \\ w_1 &= x(n-1) \\ w_2 &= x(n-2) \\ w_3 &= x(n-3). \end{aligned} \tag{1}$$

The states of the delay line buffer are updated for time $n+1$ and given by

$$\begin{aligned} w_0 &= x(n+1) \\ w_1 &= x(n) \\ w_2 &= x(n-1) \\ w_3 &= x(n-2). \end{aligned} \tag{2}$$

Figure 2: Internal filter states in linear delay line.

If each filter state is stored in memory, we see that the data are shifted forward in memory while the addresses are fixed. In the C programming language, the state update is given by

```
for (i=D; i>=1; i--)
   w[i] = w[i-1];
```

For convenience, we have a delay C function:

Figure 3: Orfanidis p. 151 delay.c function.

As an example, using the delay function is seen below.

Figure 4: Orfanidis p. 152 usage of delay function.

3 Circular Buffer

An alternative way to update the internal states is to use a *circular delay-line buffer*. Instead of shifting the data forward while holding the buffer addresses fixed, the data are kept fixed and the addresses are shifted backwards in the circular buffer. The relative movement of data versus addresses remains the same.

To understand this, consider first the conventional linear delay-line buffer case, but wrap it around in a circle, as shown below for $M = 3$. Going from time n to $n + 1$ involves shifting the content at each memory address counter-clockwise into the next memory address. The addresses for $\{w_0, w_1, w_2, w_3\}$ remain the same, but now they hold the shifted data values, and the first memory address w_0 receives the next input sample $x(n+1)$. The pointer p always points to the current input sample and stays fixed at the first memory address or starting address of the state vector.

Figure 5: Orfanidis p. 165 Figure 4.2.11 wrapped linear delay line buffer.

By contrast, in the circular buffer arrangement shown below, instead of shifting the data counter-clockwise, the buffer addresses are decremented, or shifted clockwise once, so that w_3 becomes the new beginning address of the circular buffer and will hold the next input $x(n+1)$.

Figure 6: Orfanidis p. 166 Figure 4.2.12 modulo-$(M+1)$ circular delay-line buffer.

But, whereas in the linear case the starting address is always fixed and pointing to w_0, as shown in Fig. 4.2.11, the starting address in the circular case is back-shifted from one time instant to the next. To keep track of this changing address, the pointer p always points to the current input, as shown in Fig. 4.2.12.

At each time instant, the w-address pointed to by p gets loaded with the current input sample. After computing the current output, the pointer p is decremented circularly so that for the next time instant, we may overwrite the oldest sample with the newest one. Figure 4.2.13 shows the position of the pointer p at successive sampling instants. The pointer p is restricted to lie within the pointer range of the linear buffer w, that is, $w \leq p \leq w + M$.

Figure 7: Orfanidis p. 166 Figure 4.2.13 successive positions of address pointer p, repeating modulo-$(M+1)$.

To make sure that p and its shifts $p+i$ always stay within the address space of w, they must be wrapped modulo-$(M+1)$. This is accomplished by the following routine wrap.c. Note that *p is passed by reference so we can change it (it serves as both an input and output)

(p. 169) **wrap()**

```
1  void wrap(int M, float *w, float **p)
2  {
3      if (*p < w || *p > (w + M))
4          *p = w + (*p - w) % (M + 1);
5
6      if (*p - w < 0)
7          *p += M+1;  // stupid C modulus operator
8  }
```

This functions operates as follows. Assume M is the order of the array, i.e. length of array is $M+1$, *w is a pointer to the base address of the array, and *p is a pointer into the array. Since *p is modified by wrap() it must be passed by reference, i.e. a pointer to a pointer **p. This routine circularly wraps pointer p relative to the array w by determining if *p has gone past the end of the array (or before the beginning of the array) and wraps it modulo M. Since the C modulo operator can't wrap for negative values we take this account into account with the second IF statement.

We combine decrementing pointer p with modulo wrapping in the following routine delay.c

(p. 174) **cdelay()**

```
1  void cdelay(int D, float *w, float **p)
2  {
3      (*p)--;          // decrement pointer and...
4      wrap(D, w, p);   // ...wrap modulo-(D+1)
5  }
```

This functions operates as follows. Assume D is the order of the array, i.e. length of array is $D+1$, *w is a pointer to the base address of the array, and *p is a pointer into the array (p is an address). Since *p is modified by cdelay() it must be passed by reference, i.e. a pointer to a pointer **p. Since we want *p to point to the oldest sample or state, this routine simply decrements the pointer to the new, oldest state and wraps the pointer back around for circular addressing.

4 FIR Program

The FIR program computes the convolution (or inner product) between a vector of filter coefficients and an input vector containing the most recent N samples. The function cfir() implements the convolution assuming the input samples are stored in a circular queue.

(p. 170) **cfir()**

```
1  #include "util.h"
2
3  float cfir(int M, float *h, float *w, float **p, float x)
4  {
5      int i;
6      float y=0.0;
7
8      **p = x;     // read input sample
9
10     // compute filter output, y = \sum_{i=0}^{M} h_k w_k
11     for (i=0; i<=M; i++)
12     {
13         y+= (*h++) * (*(*p)++);
14         wrap(M, w, p);
15     }
16
17     // update filter states, i.e. wk(n+1) = w{k-1}(n)
18     cdelay(M, w, p);     // p -> wm
19
20     return y;    // output sample
21 }
```

This functions operates as follows. Assume M is the filter order, i.e. length of filter is $M+1$, *h is a pointer to the base address of the filter coefficients, *w is a pointer to the base address of the input queue, *p is a pointer to the oldest sample in the input queue passed by reference, i.e. **p, and x is the newest sample. The routine first replaces away the oldest sample in the queue with the newest, i.e. **p = x so that *p now points to the newest sample. Next we accumulate a sum of products between filter coefficients and the appropriate input sample. Each time through the loop we advance the pointer to the coefficients and the pointer to the input samples. Of course each time the input sample pointer is incremented we must wrap the pointer. Next, with cdelay() we point to the new, oldest sample. Finally, we return this output sample.

5 C Implementation of FIR Filter

Prof. De Leon has implemented two C passcodes which process signals sample-by-sample. One passcode reads, writes input, output respectively samples from text file while the other other passcode reads, writes input, output respectively samples from the soundcard using the PortAudio API.

In this example, we will read an input text file containing interleaved right/left samples from a stereo audio signal. These samples will be filtered (processed) by a simple two coefficient FIR filter. The output samples will be written to an output text file.

main.c

```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

#include "user_data.h"

int main (void)
{
    float inputRightSample, inputLeftSample, outputRightSample, outputLeftSample;
    FILE *inputFilePtr, *outputFilePtr;

    inputFilePtr = fopen("input.txt","r");  // open input file for reading
    if(inputFilePtr == NULL)     // make sure the input file exists...
    {
        perror("Error");     // if not message...
        exit(1);     // and exit
    }
    outputFilePtr = fopen("output.txt","w");     // open output file for writing

    /******************/
    /* Initialization */
    /******************/
    initialize_program();

    /*****************************************************************************/
    /* Main loop - process right, left input samples; get right, left output samples */
    /*****************************************************************************/
    while (fscanf(inputFilePtr, "%f%f", &inputRightSample, &inputLeftSample) != EOF)
    {
        process_signal(inputRightSample, inputLeftSample, &outputRightSample, &outputLeftSample);
        fprintf(outputFilePtr, "%f\n%f\n", outputRightSample, outputLeftSample);
    }

    fclose(inputFilePtr);   // close input file
    fclose(outputFilePtr);  // close output file

    return 0;
}
```

user_data.h

```c
#ifndef USER_DATA
#define USER_DATA

void initialize_program(void);
void process_signal(float inputRight, float inputLeft, float *outputRight, float *outputLeft);

#define FILTERORDER 1

// extern variables are seen by *all* functions in *all* source files, i.e. global variables
// initialization of extern variables in initialize_program.c
extern float coeffs[FILTERORDER+1]; // FIR filter coefficients
extern float states[FILTERORDER+1]; // filter states all initialized to zero
extern float *oldestStatePtr;       // oldest state pointer

#include "util.h"

#endif
```

util.h

```c
#ifndef UTIL_H
#define UTIL_H

// function prototypes
float ccan(int M, float *a, float *b, float *w, float **p, float x);
void cdelay(int D, float *w, float **p);
float cfir(int M, float *h, float *w, float **p, float x);
```

```
8  float impulse(void);
9  float plain(int D, float *w, float **p, float a, float x);
10 float tap(int M, float *w, float *p, int i);
11 float tdl(int M, float *w, float **p, int N, float *gain, int *delta, float x);
12 void wrap(int M, float *w, float **p);
13
14 #endif
```

initialize_program.c

```
1  #include "user_data.h"
2
3  /**************************************/
4  /* Global variable initializations here */
5  /**************************************/
6  float coeffs[FILTERORDER+1]={0.5,0.5};  // FIR filter coefficients
7  float states[FILTERORDER+1]={0};        // filter states all initialized to zero
8  float *oldestStatePtr=states;          // oldest state pointer
9
10 void initialize_program(void)
11 {
12
13 }
```

process_signal.c

```
1  #include "user_data.h"
2
3  void process_signal(float inputRight, float inputLeft, float *outputRight, float *outputLeft)
4  {
5      /*******************************/
6      /* Process right channel sample */
7      /*******************************/
8      *outputRight = cfir(FILTERORDER, coeffs, states, &oldestStatePtr, inputRight);
9
10     /******************************/
11     /* Process left channel sample */
12     /******************************/
13     *outputLeft = inputLeft;
14 }
```