The Sampling Process and Evaluation of Difference Equations

Digital Signal Processing (DSP) is centered around the idea that you can convert an analog signal to a digital signal that is discrete in both time and amplitude, process the resulting digital signal with a computer, and convert the digital result back to an analog signal (usually). A typical DSP system consists of the blocks shown below.

The sample and hold, represented above by a switch, samples an analog signal and holds the sampled value long enough so that the A/D converter can make a conversion of the signal. The sampling frequency $f_s = 1/T$. Thus the computer receives a new sample of the data every $T$ seconds so that any numeric processing that it is doing must be done in that amount of time or less.

Typically the computer evaluates a difference equation of the form

$$y_k = b_0 u_k + b_1 u_{k-1} + \cdots + b_m u_{k-M} - a_1 y_{k-1} - a_2 y_{k-2} - \cdots - a_N y_{k-N}$$

In this equation $k$ is the time counting variable so that samples are taken at $0, T, 2T, 3T \ldots kT \ldots$. The variable $y_k$ is the output variable at the time $kT$. The variables $y_{k-1}, y_{k-2}, \ldots y_{k-N}$ are the previous output variables. The variables $u_k, u_{k-1}, \ldots u_{k-M}$ are the input variables both past and present. The coefficients $a_x$ and $b_x$ are constants.

The computer program which evaluates this difference equation might look something like this in pseudocode.

```
Initialize Variables
DO Forever
    Call AtoD(Vi)              ;Get a sample from the A to D
    Vo = {difference equation} ;Output Vo to D to A
    CallDtoA(Vo)
    Vol = Vo                    ;Reset the value of the old variable.
    Wait for T seconds to pass
Loop
End
```
The loop runs forever and the loop timing is typically governed by a polled or interrupt
driven timer. The processing time for the difference equation evaluation is the most time
consuming part of the process. It is this processing time which determines the fastest
sampling period since samples cannot be brought in faster than the machine can process
them.

Looking at the difference equation we see that it consists of a sequence of multiplies and
adds. Such operations are a mainstay of DSP systems and nearly all specialized DSP
chips have a single hardware unit which does a multiply and add operation which is
called a MAC for multiply and accumulate.

A few general purpose processors have become fast enough to do DSP algorithms and
some have added MAC units as part of the CPU make up. The ARM7 processor is one
such CPU that has a MAC unit (in addition to its barrel shifter). The ARM Cortex M0
does not have a MAC unit.

In assembly language the multiply and accumulate instruction looks like this
\[
\text{mla } rd, rm, rs, rn
\]
This instruction multiplies does the following $rd \leftarrow rm \times rs + rn$

Unfortunately, the Keil C compiler does not make use of the mla instruction when it
evaluates a difference equations but creates code which does the multiplies and as
separate assembly operations. So to take advantage of the mac unit on the ARM7 you
have to write assembler code.

Creating a digital filter using MatLab

While this is not a course on DSP (for that you need to take EE 311), we can use Matlab
as a tool to create useful difference equations which represent digital filters. We will
look at two types: Butterworth filters and Chebyshev filters.

Butterworth filters in MatLab

A Butterworth low pass filter is maximally flat in the passband, has a relatively long
transition band, and is monotonic in the stop band. In Matlab the relevant function is
\[
\text{[num den] = butter(N, wn);} \hspace{1cm} \text{num and den are the numerator and denominator polynomials in z for the}
\text{Butterworth filter, N is the filter order, and wn is the normalized cutoff frequency. (The}
\text{cutoff frequency is normalized so that fs/2 = 1.)}
\]

For example,

\begin{verbatim}
N = 3; \hspace{1cm} \%filter order
fs = 11050; \hspace{1cm} \%sample frequency
wn = 3000/(fs/2); \hspace{1cm} \%normalized cutoff frequency
[num den] = butter(N, wn); \hspace{1cm} \%calculate filter
disp(num); \hspace{1cm} \%display numerator and denominator
disp(den);
\end{verbatim}

gives

\[9-2\]
which translates into the following transfer function in z

\[ H(z) = \frac{y(z)}{u(z)} = \frac{0.2025z^3 + 0.6075z^2 + 0.6075z + 0.2025}{z^3 + 0.2477z^2 + 0.3496z + 0.0228} \]

To translate this into a difference equation we first multiply numerator and denominator by \( z^{-3} \). This gives

\[ H(z) = \frac{y(z)}{u(z)} = \frac{0.2025 + 0.6075z^{-1} + 0.6075z^{-2} + 0.2025z^{-3}}{1 + 0.2477z^{-1} + 0.3496z^{-2} + 0.0228z^{-3}} \]

Cross multiply this equation to get

\[ y(z) + 0.2477z^{-1}y(z) + 0.3496z^{-2}y(z) + 0.0228z^{-3}y(z) = 0.2025u(z) + 0.6075z^{-1}u(z) + 0.6075z^{-2}u(z) + 0.2025z^{-3}u(z) \]

Noting that the inverse z transform of \( y(z) \) \( \rightarrow \) \( y_k \) and the inverse transform of \( z^{-x}y(z) \) \( \rightarrow \) \( y_{k-x} \) we can transform this equation to the time domain and solve for \( y_k \). This gives:

\[ y_k = 0.2025u_k + 0.6075u_{k-1} + 0.6075u_{k-2} + 0.2025u_{k-3} - 0.2477y_{k-1} - 0.3496y_{k-2} - 0.0228y_{k-3} \]

This equation can be implemented on the ARM Cortex M0 board in floating point mode or, if it is rescaled, in integer mode. Integer mode is faster but is sometimes difficult to scale without producing overflow. Floating point mode is slower but overflow is seldom a problem.

We can also use MatLab to look at the expected frequency response of this filter by plotting the filter gain vs. frequency. The following Matlab code does this.

```matlab
%Butter1.m
N = 3;   % filter order
fs = 11050;  % sample frequency
wn = 3000/(fs/2);  % normalized cutoff frequency
[num den] = butter(N, wn);  % calculate filter
disp(num);  % display numerator and denominator
disp(den);
% freqz finds the frequency response for 512 points between 0 and \( \frac{fs}{2} \)
[H f] = freqz(num, den, 512, fs);
figure(1);clf;
plot(f, abs(H));
xlabel('frequency in Hz');
ylabel('filter gain');
title('3rd Order Butterworth filter');
```
Note that the cutoff frequency is at the point where the gain is .707 (the half power point).

**Chebyshev filters in MatLab**

A Chebyshev low pass filter is has some ripple in the passband, has a faster transition band than does the Butterworth filter, and is monotonic in the stop band. In Matlab the relevant function is

\[ (num\ den) = \text{cheby1}(N, \ Rdb, \ wn); \]

In this equation num and den are the numerator and denominator polynomials in z for the type 1 Chebyshev filter, N is the filter order, Rdb is the passband ripple in decibels, and wn is the normalized passband edge frequency. (The passband edge frequency is normalized so that fs/2 = 1.) Here is a Matlab program to calculate and plot a 3\textsuperscript{rd} order Chebyshev low pass filter.

```
%Cheby1000.m
N = 3;                       %filter order
fs = 11025;                  %sample frequency
wn = 1000/(fs/2);            %normalized passband edge frequency
ripple = 0.05;               %pass band peak to peak ripple
rDB = -20*log10(1-ripple);   %convert ripple to decibels
[num den] = cheby1(N, rDB, wn); %calculate filter
disp(num);                   %display numerator and denominator
disp(den);
%freqz finds the frequency response for 512 points between 0 and fs/2
[H f] = freqz(num, den, 512, fs);
figure(1);clf;
plot(f, abs(H));
xlabel('frequency in Hz');
ylabel('filter gain');
title('3rd Order Chebyshev filter');
```
The numerator and denominator polynomials are
0.012444695267244  0.037334085801731  0.037334085801731  0.012444695267244
1.000000000000000  -2.074201526685092  1.653663536335353  -0.479904447512313
The resulting frequency plot for the filter is shown below.

Class exercise 9-1: The program ChebF3.c implements the 3rd order Chebyshev filter above as a digital filter running on the ARM Cortex M0 processor. Set up the debugger so that there is a sinusoid on ain0. Run this program and use the logic analyzer to observe Port 1.9 and the input on ain0. Verify that the sample frequency of 11025 Hz is correct.
This program implements a filter in floating point
This is for ARM Cortex M0 processor board.
Numerator
0.01244695267244  0.037334085801731  0.037334085801731  0.01244695267244
Denominator
1.000000000000000 -2.074201526685092  1.653663536335353 -0.47990447512313
H(Z) = 0.012446Z^3 + 0.03733Z^2 + 0.0373Z + 0.012445
Z^3 + -2.074201Z^2 + 1.65367Z^2 - 0.47990
This filter was designed in MatLab as a 3rd order Chebyshev filter with
fpass = 1000;
Rp = 0.05;N = 3;
The PWM is implemented with an interrupts and is glitch free.

const float b0 = 0.012444695267244;
const float b1 = 0.037334085801731;
const float b2 = 0.037334085801731;
const float b3 = 0.012444695267244;
const float a1 = -2.074201526685092;
const float a2 = 1.653663536335353;
const float a3 = -0.479904447512313;
unsigned int yOut, flag;
int main(void)
{
unsigned int uInt;
float u, y;
float u1, u2, u3;
float y1, y2, y3;
NVIC_ISER |= (1 << 17); //Enable 16-bit timer 1 interrupt
PDRUNCFG &=(1 << 4); //Power up A/D
SYSAHBCLKCTRL |= (1 << 13); //ADC Clock enable
IOCON_RPIO0_11 &= 0xFFFFFF78;
IOCON_RPIO0_11 |= (1 << 1);
AD0CR = 0x0B01;
//*************** Set up PWM
SYSAHBCLKCTRL |= (1 << 8); //Enable clock for 16-bit timer 1
IOCON_PI01_9 |= 1; //Selects match function so Pin_9 is PWM
TMR16B1PR = 1; //Prescale register. Divide P Clock by 2 = 24 MHz
//1/24MHz = 0.04167usec. 1/11025Hz = 90.703usec. 90.703/0.04167 = 2176.7
// Set match register to 2177
TMR16B1MR3 = 2177; //Match register 3. TR and Pin_9 Reset
TMR16B1MCR |= (1 << 10); //Causes TimerCounter to be reset if match on MR3
TMR16B1MCR |= (1 << 9); //Causes interrupt on match to MR3

TMR16B1PWMC |= 1; //Enable PWM on Channel 1
TMR16B1TCR |= 1;    //Enable TimerCounter to run

// Do Filter
while(1)
{
    AD0CR |= (1 << 24); // Start conversion
    while(AD0DR0 < 0x7FFFFFFF); // Wait for done bit
    uInt = (((AD0DR0 & 0xFFC0) >> 6)); // Data in from A/D
    u = (float)uInt; // /1024.0;
    y = b0*(u + u3) + b1*(u1 + u2);
    y = y - a1*y1 - a2*y2 - a3*y3;
    yOut = ((int)(y*1024.0));
    y3 = y2; // Shift variables for next iteration
    y2 = y1;
    y1 = y;
    u3 = u2;
    u2 = u1;
    u1 = u;
    flag = 1;
    while(flag); // Wait for interrupt
}

// void TIMER16_1_IRQHandler(void)
{
    TMR16B1IR = 8; // Clear interrupt due to MR0
    TMR16B1MR0 = yOut; // Load the value of yOut from the main program
    flag = 0;
}