EECE 2510 – Circuits and Signals: Biomedical Applications

Lab 16: Digital Processing of the ECG

Sec 2

Introduction:

Now that you have some ECG signals recorded using the analog portion of the ECG system, combined with the A/D converter, you are ready to work in the digital world to improve and extract information from your signal. Although you filtered out many of the high frequencies before A/D conversion, you may be able to do a more complete job in the digital realm. In addition, because 60 Hz interference is within the bandwidth of your ECG signal, it is not easy to filter it out of the signal before A/D conversion. We may be able to do better by manipulating the digital signal. Finally, extracting information such as the heart rate or the variability of the heart rate may be useful in clinical settings. You should complete as many of these tasks as you have time for (not necessarily in this order).

1. Build a digital low-pass filter to reduce the noise in the ECG signal.

2. Remove the 60 Hz interference by using a digital notch filter centered at 60 Hz.

3. Automatically detect the heart rate in the ECG signal.

Background: As a reminder, in class we talked about the fact that digital filters are typically implemented via linear constant coefficient difference equations:

We specify the filter via two sets of weights, *{a}* and *{b}* and then compute filter outputs y[n[from inputs x[n] as :

$$y[n] = -(a_1 y[n-1] \ldots a_N y[n-N]) + b_0 x[n] + b_1 x[n-1] \ldots b_m x[n-M]$$

Note that the output coefficients a_k on the right side have minus signs so that they will have + signs on the left hand side --- this will matter when using the Matlab *filter* function we will describe below.

As described in class, this equation corresponds to an *algorithm*. (Actually, there are many many ways to actually implement this, but here we are talking conceptually, at a high level.) To see this, suppose that

1) we start computing outputs for n=0, and

2) we know that both y[n] = 0 and x[n] = 0 for n<0.

The input signal x(n) can be any discrete signal, as long as it is 0 for n<0.

Plugging into

$$y[n] = -(a_1 y[n-1] \ldots a_N y[n-N]) + b_0 x[n] + b_1 x[n-1] \ldots b_m x[n-M]$$

1. We can compute the first output, at n=0, y[n=0] = b_0x[0] (since all other terms are zero since they correspond to terms for which n<0.

2. Then we can compute the next output value, y[n=1] = -a_1y[0] + b_0x[1] + b_1x[0] (since again all other terms =0).

Plugging in from Step 1 we get y[n=1] = -a_1 b_0x[0] + b_0x[1]+ b_1x[0].

3. Then the third value, y[n=2] = -a_1y[1] -a_2y[0] + b_0x[2] + b_1x[1]+b_2x[0], for which we could substitute in the expressions we already have above for y[0] and y[1] .

4, 5, 6, …. We can just keep going to compute as many values of y[n] as we want.

Thus we see that we can recursively compute values of the output y[n] for as many outputs as we like, based on the filter coefficients and the input values.

Matlab implements this algorithm (calculated in a more sophisticated way) in a function called, appropriately enough, *filter*. The Matlab function *filter*, as we will use it, takes as its inputs two vectors of the coefficients from the difference equation,

one for the input, [b_0 b_1 … b_M] and

one for the output [1 a_1 a_1 …. a_N],

that describe the filter.

These vectors, in fact, are what are returned by the filter design functions we will describe and use below.

See *help filter* in Matlab and ask if you are not sure how to use it ---- but it really is straightforward. You just supply the two vectors of coefficients (being sure to keep the order straight of which vector is the numerator and which is the denominator) and the vector of inputs, to *filter,* and it returns the filtered output.

Part I: Digital Filters from Analog Designs

It turns out that there are families of digital filters than can be designed, and implemented in Matlab, using the same polynomial prototype design approaches that are used for analog design. Here you will design and study digital Butterworth filters.

(As an aside, technically the system functions for digital filters are usually written as functions of a complex variable "z" rather than "s", corresponding to something called the "Z Transform". The Z Transform plays a role for discrete time systems corresponding to that played by the Laplace Transform for continuous time systems. Frequency selective filters in discrete time can be designed in terms of the Z Transform polynomials, just as was done with the "s" polynomials for analog filters.)

1.1: Use *help* in Matlab to learn how to use *buttord* and *butter* and then design a digital Butterworth filter with the following specifications:

1. pass band edge corresponding to 50Hz,
2. stop band edge corresponding to 80Hz,
3. pass band attenuation of no more than 2dB, and
4. stop band attenuation of at least 30 dB.

Note that "design" means obtaining vectors of *a* and *b* coefficients to use with the *filter* program. The coefficients describe the denominator (*a*) and numerator (*b*) polynomials in the frequency response and also the corresponding weights in the difference equation – as with CT differential equations and the Fourier Transform, these sets of coefficients are (very conveniently!) the same!

Set the sampling rate to the rate you used to acquire your ECGs.
NOTE: You may need to use what you have learned about how the sampling rate translates continuous time to discrete time frequencies (in the notation we have used, transforming *f* to *F)*. Also you need to know that Matlab describes discrete

frequencies so that 0 to 1 in Matlab routines maps to 0 to π in DT radian frequencies or 0 to ½ in DT per-sample frequencies**.** (And remember that half he sampling rate maps to π or ½.
Putting this all together means, for example, that if you wanted a band edge at 100 Hz with a sampling rate of 2000 samples / second you would specify it to Matlab routine as `100/1000 = 0.1`.
Ask if this is not clear to you.

As per our normal coding guidelines, be sure to use names you can remember for the two coefficient vectors returned by *butter* as we will use them in the next section to actually implement this filter and filter an acquired ECG signal.

1.2: Use the Matlab routine *freqz* to plot the magnitude and phase of the frequency response of the digital filter you just designed. Check to be sure that your design meets the specifications. (Note that if you run *freqz* with no outputs and it will produce the plots we are looking for.) What do you notice about the phase plot? In particular, over which frequency ranges, outside of sudden jumps, does the phase change most rapidly (and non-linearly)? Do you think this might affect the relationship between the output and the input? (Hint: is this change in the pass band of the filter?) Also, can you think what the jumps in the phase might correspond to? (Hint: how big are the jumps in radians or degrees?)

1.3 Filter some of your acquired ECGs with the digital Butterworth filter you designed using the Matlab *filter* function. Plot the output and compare it to the original acquired signal. What do you observe? You many see a large artifact at the start of the filtered signal which you may want to avoid including in your plot. Ask if you are not sure how to do this. You can also acquire ECG signals from both before and after the low pass filter in your circuit and try running your filter on both and compare results.

Part II. **Designing Digital Filters Directly**

There are many ways to design digital filters directly from frequency specifications in the discrete frequency domain without using an analog prototype design method. You can learn more about this in the Linear Systems and Digital Signal Processing classes later on. Here we will experiment with one design, known as “Parks-McClellan” (after the people who first derived this method) or “equiripple” (you will soon see why this name is appropriate) filters. The Matlab routines we will use for the design are *firpmord*, to find the required order and other key parameters

from the specifications, and *firpm* to actually design the filter from those parameters.

If you run *help firpmord* you will see that it requires four sets of input variables. The first two parameters are 1) a vector of key "break" frequencies and 2) a vector of desired amplitudes at those frequencies. As an example, for a low pass filter with a band pass edge at a frequency corresponding to 250 Hz with a given sampling frequency and a stop band edge corresponding to 350 Hz, the first vector would be [250 350] and the second one [1 0]. **Why do you think the second vector is [1 0**]? Explain in your notebook, or ask if you don't understand. The third vector gives desired tolerances in each of those bands. Also note that if you do not supply a sampling rate as the fourth argument, the help says that the default value is 2.

The parameters returned by the order design routine, *firpmord*, are used directly as the input to the filter design routine *firpm*. *firpm* has only output variable (one vector of coefficients) as this type of filter has only a numerator (or input weight) set of coefficients --- in other words the denominator polynomial is equal to 1, equivalent to all the *a* coefficients being equal to 0 except a_0 which equals 1. In terms of the difference equation this implies that there are no "recursions" on past output values, just a direct weighting of the input signal. It turns out that this type of filter, known as Finite Impulse Response (FIR), has some significant advantages in certain situations. Again, ask if you are curious to learn more.

2.1: Design a Parks-McClellan filter using these routines to low-pass filter your signal with the same band specifications as with the Butterworth filter. Play with the tolerances in the third argument to try to meet the specifications you used for the Butterworth filter. Use *freqz* to check your design. Note here you need to supply the number "1" for the input parameter "a" in *freqz*.

Why do you think this is called an "equiripple" design? What do you notice about phase response? Can you explain the jumps in the phase response here?

Filter your ECG signal(s) with your Parks-McClellan design using *filter*. Here, as with *freqz* you need to specify the "1" in the input for the 2nd input parameter.

Try different design specifications for the Parks-McClellan design and record your results in your notebook. Note that if you make the filter too "good" in some sense (cutoff frequency too low, stopband ripple too small, transition too fast), it will begin to distort the signal. Ask if you have questions.

2.2: Using the same methods, design a 60 Hz notch filter to remove the 60 Hz interference. This is likely to result in signal artifacts, but see what happens and discuss.

Part III. Estimating Heart Rate from your ECGs

3.1. Process your acquired ECGs in Matlab to estimate an average heart rate. You can do this any way you would like. One approach is using threshold detection (or another method of your choice) to count the heartbeats in a certain interval and determine a heart rate. Another would be to use the FFTof the heart signal to determine the heart rate, writing an automated program in MATLAB to try find the frequency of the lowest frequency peak. If you get this working you may want to acquire a longer run of ECGs and then you can study how stable the heart rate is --- for example, find the average heart rate and its standard deviation, or try plotting the beat-to-beat interbeat interval as a function of "time" (where time here means beats).

You do not need to hand anything in for this lab. Do have the TAs sign off on whatever you get done.