GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

ECE 2025 Fall 2001

Lab #12: Digital Communication: FSK Modem

Date: 26–30 Nov. 2001

This is *the official* Lab #12 description.

You should read the Pre-Lab section of the lab and do all the exercises in the Pre-Lab section before your assigned lab time. You **MUST** complete the online Pre-Post-Lab exercise on Web-CT at the beginning of your scheduled lab session. You can use MATLAB and also consult your lab report or any notes you might have, but you cannot discuss the exercises with any other students. You will have approximately 25 minutes at the beginning of your lab session to complete the online Pre-Post-Lab exercise. The Pre-Post-Lab exercise for this lab includes some questions about concepts from the previous Lab report as well as questions on the Pre-Lab section of this lab.

The Warm-up section of each lab must be completed **during your assigned Lab time** and the steps marked *Instructor Verification* must also be signed off **during the lab time**. After completing the warm-up section, turn in the verification sheet to your TA.

Forgeries and plagiarism are a violation of the honor code and will be referred to the Dean of Students for disciplinary action. You are allowed to discuss lab exercises with other students and you are allowed to consult old lab reports but the submitted work should be original and it should be your own work.

The lab report for this lab will be **informal.** Discuss your results from section 4.

The lab report will be **due during the week of 3–6 Dec. at the beginning of your lab**. Next week, you will have to demonstrate a working system for the FSK demodulator and decoder.

---

1 Introduction

The goal of this lab is to understand a simple modem, the Frequency Shift Keying (FSK) Modem, referred to by the International Telecommunications Union (I.T.U.) as V.21. Here is a quick recap of the operation of the v.21 FSK modem. The V.21 modem communicates 1's and 0's by sending either a 1650 Hz tone or a 1850 Hz tone, respectively, for 1/300 sec. Thus the overall data rate is 300 bits/second (one bit is sent in 1/300-th of a second). Even though 300 bps is quite slow in comparison to the theoretical maximum of 56 kilobits per second over a phone line, the V.21 protocol is still used in almost every modem call, because receiving it is so simple. A V.21 modem call can be received without using difficult techniques such as equalizers, cancellers and matched filters. Furthermore, it can be received accurately even in the presence of a significant amount of noise. For these reasons, V.21 is used to perform the initial *handshake* between two modems, meaning that V.21 is a way to communicate some basic startup and control information between the two modems. You can hear the V.21 modem tones at home when your V34, V.90, V.92 phone line modem or fax machine starts a phone call. V.21 is also used to transmit caller ID information over the phone line.

1.1 Demodulator

The receiver for V.21 must determine which of the two tones is present, and must make this decision every 1/300-th of a second. Furthermore, the receiver must *synchronize* with the bit interval, meaning that it must learn where the starting and ending times of each bit are located. This synchronization is essential to making reliable “0-1” decisions because the transition times must be avoided. A block diagram of the FSK V.21 demodulator is given in Fig. 1. Each of the main sections will be described in more detail below.

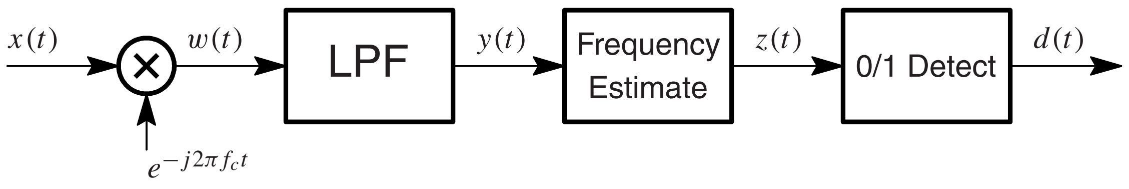

Figure 1: Block diagram of the FSK V.21 demodulator.

1.1.1 Mixing

A basic operation that most modems need to perform is frequency shifting of the input signal. According to the frequency-shifting property of the Fourier Transform, this can be done by simply multiplying the input signal by a complex exponential.

$$w(t) = x(t)e^{-j2\pi f_c t}$$

This effect can best be understood by thinking of $x(t)$ as a sum of complex exponentials and observing what happens to each individual frequency component.

$$x(t) = e^{j2\pi f_1 t} + e^{j2\pi f_2 t}$$

When $x(t)$ is multiplied by the given complex exponential at $f = -f_c$ Hz, the exponents simply add and the resulting $w(t)$ is as follows:

$$w(t) = e^{j2\pi(f_1 - f_c)t} + e^{j2\pi(f_2 - f_c)t}$$

Note that the new frequencies are simply the old frequencies shifted down by f_c, i.e., $f_1 - f_c$ and $f_2 - f_c$. For V.21 FSK, we need to choose f_c so that the original frequencies of 1650 Hz and 1850 Hz are shifted to ± 100 Hz.

1.1.2 Low Pass Filter Design

You should already be familiar with some methods for designing FIR low pass filters with given specifications on the passband and stopband edges, as well as constraints on the ripple characteristics in the pass and stop bands. Such designs can be carried out in MATLAB with the `filtdemo` GUI, or with the M-file called `hammfilt`. Since the digital filter is typically used to filter signals that are sampled analog signals, it is important to know how the bandedges of the digital filter can be expressed in terms of the desired analog cutoff frequencies. This can be done if the sampling frequency f_s is known. For example, if we want to have a lowpass filter with an *effective* cutoff frequency of 2000 Hz, and we are using a digital filter running at $f_s = 8000$ Hz, then the digital filter must have its cutoff frequency[1] at $\hat{\omega} = 2\pi(2000/8000) = \frac{1}{2}\pi$.

In the FSK V.21 system, the frequency shifting of the mixer will generate spectrum lines that must be removed by filtering. If we demand that these unwanted spectrum components must be reduced in magnitude by a factor of 100, then we have given the specifications on the stopband ripple. A reduction by a factor of

[1]Remember that the frequency response of a digital filter, $H(e^{j\hat{\omega}})$, is a function of the frequency variable $\hat{\omega}$ that runs from $\hat{\omega} = -\pi$ to $\hat{\omega} = +\pi$.

100 means that the stopband ripple must be less than 0.01, or -40 dB. The passband, on the other hand, must be made wide enough so that the desired frequency components will go through the LPF will little or no change. Since the LPF's frequency response will have a passband ripple, we will use a specification on the passband of 1 dB, which forces the passband magnitude to lie between 0.89 and 1.12.

1.1.3 Simulation Rate

The MATLAB implementation of the FSK V.21 modem is a *simulation.* This means that even though we seem to have continuous-time signals such as $x(t)$ and $w(t)$, we actually use sampled versions in the MATLAB program. For this simulation, we will choose the input sampling frequency to be

$$\boxed{f_s = 9000 \text{ Hz}} \tag{1}$$

This value is chosen to be rather high so that the input signal will appear to be continuous if we use the MATLAB `plot()` command. In Fig. 2 the continuous-time signals have been replaced with their sampled versions, e.g., $x(t)$ becomes $x[n]$, $w(t)$ becomes $w[n]$, and so on. The complex exponential used in the mixer must also be converted from $e^{-j2\pi f_c t}$ to a discrete-time signal $e[n]$.

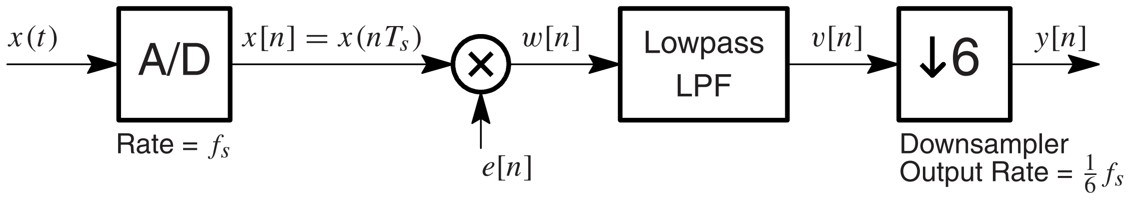

Figure 2: FSK system simulated as a discrete-time system at a sampling rate of f_s. The LPF is a digital filter capable of anti-alias filtering so that the output sampling rate can be lowered by a factor of 6.

1.1.4 Downsampling: Anti-Alias Filtering

The FSK bit rate is 300 bits per second, so it would be very inefficient to operate the entire system at a rate of $f_s = 9000$ Hz.[2] It turns out that it is relatively easy to lower the sampling rate as the signal moves through the different processing stages in Fig. 1. The first opportunity to change the sampling rate is at the output of the lowpass filtering stage. In order to understand how to do this, consider the concept of *rate changing.* The sampling theorem implies that a given discrete-time signal $w[n]$ can be *downsampled* from f_s to f_s' without aliasing effects, if the corresponding continuous-time signal $w(t)$ contains no frequencies above $\frac{1}{2}f_s'$. This downsampling condition can be imposed by removing these frequencies with a *digital* low pass filter (LPF). Such a filter is called a digital *anti-aliasing* filter. Once this digital anti-aliasing filter has been used, the signal can be downsampled without any significant aliasing effects.

Consider the following numerical example: A signal $w(t)$ has been sampled at $f_s = 9000$ Hz to give $w[n]$. We want to downsample $w[n]$ to get $y[n]$ so that $y[n]$ has samples of $w(t)$ at a rate of $f_s' = 1500$ Hz, i.e., a 6:1 downsampling. In order to do this and have no aliasing, the analog signal cannot have any frequency components above $\frac{1}{2}f_s' = 750$ Hz. We cannot enforce this bandlimit on $w(t)$, but we can filter $w[n]$ to remove any high frequency components. Therefore, we need a digital lowpass filter running at $f_s = 9000$ Hz whose effective stopband is 750 Hz and above. When converted to the $\hat{\omega}$ frequency domain,

[2]For hardware implementation on Digital Signal Processing (DSP) chips, the number of multiplications and additions per second is a crucial measure of performance. A lower sampling rate or lower computation rate leads to simpler implementations that can be cheaper or consume less power.

the stopband cutoff frequency of this digital *anti-aliasing filter* will be $\hat{\omega}_s = 2\pi(750/9000) = \pi/6$. The output of the digital anti-aliasing filter, $v[n]$, can then be downsampled by taking every sixth sample, i.e., $y[n] = v[6n]$.

1.1.5 Slicing

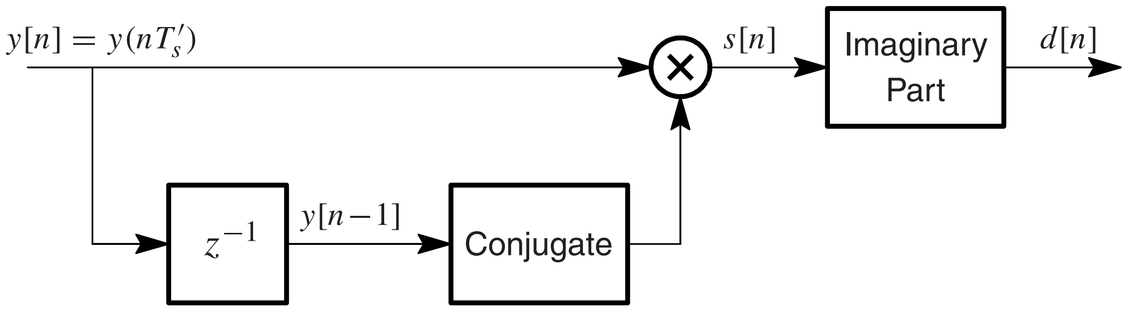

Figure 3: Frequency estimation in a dual-frequency FSK system can be performed with a *slicer.* When the input signal is a single complex exponential signal at either $+f_{\text{in}}$ Hz or $-f_{\text{in}}$ Hz, the sign of the output signal is positive or negative, respectively.

Another basic operation of most modems (and DSP systems) is to measure the frequency of a received tone. This could be accomplished by optimal filtering algorithms such as matched filters designed to enhance the tones of interest. However, if you are guaranteed to be looking at only one tone at a given time and the noise is not severe (both of which are true for V.21), then there is a faster method that can be employed to save computation when measuring frequency.

Slicing is defined as follows:

$$s[n] = y[n](y[n-1])^* \tag{2}$$

It is a non-linear operation, but in the case where $x[n]$ is a single tone input, $y[n] = Ae^{j\hat{\omega}_0 n}$, the output of the slicer (2) reduces to

$$s[n] = (Ae^{j\hat{\omega}_0 n})(A^* e^{-j\hat{\omega}_0 (n-1)})$$

After adding the exponents the output simplifies to

$$s[n] = |A|^2 e^{j\hat{\omega}_0}$$

If the objective were to determine the frequency $\hat{\omega}_0$, then it is sufficient to take the imaginary part

$$d[n] = \Im m\{s[n]\} = \Im m\{|A|^2 e^{j\hat{\omega}_0}\} \qquad \Longrightarrow d[n] = |A|^2 \sin(\hat{\omega}_0)$$

and use $d[n]$ to calculate the $\text{ArcSin}(\cdot)$ to get an estimate of $\hat{\omega}_0$. However, the FSK V.21 system is even simpler than that, because we only need to decode two cases: a zero or a one. When the frequency is negative we have a “1”, and when $\hat{\omega}_0 > 0$ we have a “0.” In addition, the sign of $\hat{\omega}_0$ is the same as the sign of $|A|^2 \sin(\hat{\omega}_0)$, so we only need to check the sign bit of $d[n]$ to perform the decoding. In MATLAB, it is possible to extract the sign of a number; consult `help sign.`

As will be seen in the final implementation of this lab, the recovery of the V.21 signal will reduce to discriminating between a $+100$ Hz tone and a -100 Hz tone. At this point our signal will be sampled at $f_s' = 1500$ Hz and taking the imaginary part of $s[n]$, the slicer output, will provide an easy way to determine whether a 1650 Hz or 1850 Hz tone was originally present.

Thus we can define $b[n]$ as the bits decoded from the slicer output:

$$b[n] = \begin{cases} 0 & \text{when } d[n] \geq 0 \\ 1 & \text{when } d[n] < 0 \end{cases} \tag{3}$$

f_{original}	f_{mixed}	$\lvert A\rvert^2 \sin \hat{\omega}_0$	Bit Received
1650 Hz	-100 Hz	Negative	1
1850 Hz	100 Hz	Positive	0

Table 1: FSK Decoding Rule

1.2 Decoding the Bits

After the mixing and filtering blocks in the FSK receiver, we must try to decode the message bits. There are three steps: synchronizing with the bit interval, finding the preamble that marks the beginning of the message, and grouping the bits into 8-bit sets to be converted into ASCII characters.

1.2.1 Decision Making, an Easy Baud Lock

The most challenging part of the decoding process is synchronizing the receiver to the start times of the bit intervals. The method for synchronization involves transmitting a preamble to the actual message, and having the receiver look for the alternating bit pattern at the beginning of that preamble. For our purposes we will use a preamble that consists of sending the length-20 bit pattern `[01010101010111111111]` as the prefix to the actual message. The first twelve bits in this pattern give eleven transitions between zero and one. The next eight bits, which are all ones and decode as the number 255, mark the beginning of the actual message. There are actually nine ones at the end of the sync pattern, and these are used as a flag to indicate that data mode is about to start.

The eleven consecutive zero-one transitions are used to synchronize. The sampling rate of the slicer output is 1500 Hz, so a 300 bps signal with alternating zeros and ones will make the zero-one transition every five samples. In other words, the signal $d[n]$ in Fig. 3 will change sign every fifth sample. The problem is that we don't know the index where the first zero-one change happens.

Example: If the first zero-one transition happens at $n = 7$, then the subsequent changes are expected to occur at $n = 12, 17, 22, \ldots$. However, if the first zero-one transition is at $n = 744$, then the following zero-one jumps are at $n = 749, 754, 759, \ldots$.

Therefore, the synchronization problem involves two steps: first, how do we use the slicer output bits $b[n]$ to *detect* the eleven consecutive zero-one transitions? Second, how do we *estimate* the index of the last transition? Once we know the indices where the zero-one transitions occur, we can pick the middle of the bit intervals as the location where the decoding decisions should be made so that we will be safely away from the transition indices.

1.2.2 Detection of Eleven Consecutive Jumps

We can process the bit signal $b[n]$ to produce a signal that is non-zero only when there is a jump. There are two ways to do this: (1) form the exclusive-OR of $b[n]$ and $b[n+1]$ (see `help xor` in MATLAB), or (2) filter $b[n]$ with a first-difference to get $\lvert b[n+1] - b[n]\rvert$. In either case, the resulting signal will be one whenever there is a jump (0 to 1, or 1 to 0).

We know that the eleven zero-one jumps are at the beginning, so we can search forward from $n = 0$, but we don't know how far we will have to search. However, the eleven alternating zeros and ones should occur within an interval of length 55, and the distance from the first transition to the last should be exactly 50. We can use MATLAB's `find` function to generate a list of indices where the jumps occur. From this list of indices we can form the following simple detection criterion based on the first and last transitions: *determine the first subset of eleven indices where the first and last index are separated by exactly 50.* This simple test might be sufficient, but if you want to make the test more robust, it would also be necessary to check the other nine transitions to make sure that they are all spaced by five.

1.2.3 Robust Estimate of Jump Locations

A complication arises when the signal is noisy because the random fluctuations caused by the noise might produce a false jump. For this reason, we need a robust method to check that the jumps are all spaced by 5. We might be willing to accept an occasional spacing of 4 or 6 when noise perturbations influence the transitions, but we should reject any other spacings. Thus, once the detect-eleven algorithm generates a candidate interval, the jump estimation proceeds as follows:

Let $\{n_1, n_2, n_3, \ldots, n_{11}\}$ be the eleven indices in the candidate interval. Although we might assume that $n_{11} - n_1$ is equal to 50, it is not necessary when using the test proposed below. If these indices were all spaced by 5, then the differences $n_{11} - n_1$, $n_{11} - n_2$, $n_{11} - n_3$, ... $n_{11} - n_{10}$, would all be divisible by 5 The modulo operator[3] can be used to check this, because we should have

$$(n_{11} - n_k) \bmod 5 = 0 \qquad \text{for } k = 1, 2, 3, \ldots, 10$$

Thus a robust jump test is

$$\sum_{k=1}^{10} |(n_{11} - n_k) \bmod 5| < N_0$$

where N_0 would be chosen as a small number such as 1, 2 or 3 to allow for slight perturbations in the transition indices.

Once this test is passed, we can use n_{11} as the index of the last jump, which should precede the nine 1's that mark the beginning of the actual message.

1.2.4 Ready to Decode

Once we know the location of the last jump in the sequence of eleven, then we can set up for decoding. We want to make "bit decisions" in the middle of the bit interval, so we should add 3 to n_{11}, because n_{11} is the index of a transition. Furthermore, n_{11} should be the index prior to the last "1" in the preamble of alternating zeros and ones, so we must also add 5 to move over to the next bit. Taken together, $n_{11} + 8$ should be the location of the first one in the group of eight ones at the end of the sync pattern that are used as a flag to indicate that message is about to start.

At this point everything is synchronized. Since we only need to examine the value in the middle of the bit interval, it is possible to downsample once more, this time it would be downsampling by a factor of 5. In other words, after this downsampling you will have:

$$a[\ell] = b[n_{11} + 8 + 5\ell] \qquad \text{for } \ell = 0, 1, 2, \ldots$$

Each $a[\ell]$ represents one bit of the message. The first eight bits should be ones, indicating the "start of message flag." After verifying the eight consecutive ones, you can get all the ASCII characters of the message by grouping the bits together, eight at a time. the decoder continues to extract characters until it encounters the "end of message flag" which is also a string of eight or more consecutive ones.

The eight ones at the beginning of the message are used to synchronize the 8-bit sets that will be converted to ASCII. If this "byte synch" is not aligned correctly, then all the characters will be translated

[3]Use `help mod` in MATLAB

incorrectly. Therefore, it is crucial to find the nine consecutive ones. If there is an error made in finding n_{11}, the system will fail, unless the decoder is robust. This could happen when the signal is noisy. One way to do a robust detection of the consecutive ones is to use the index $n_{11} + 3$ as the starting index for a search that attempts to find the nine consecutive ones, i.e., don't assume that the bits starting at $n_{11} + 8$ are guaranteed to be ones.

2 Pre-Lab Exercises

2.1 Binary Stream to ASCII

The FSK decoder involves the operation of converting 8-bit patterns to ASCII. The MATLAB functions `bin2dec` and `char` can convert a bit stream to ASCII. Consider the following example:

```
bb = [0,1,0,0,0,0,0,1], char(bin2dec(char(bb+abs('0'))))
```

In order to understand this example, you might want to see what each `char` and `bin2dec` operator produces. Modify the example above so that it can do more than one ASCII character. Complete the loop below:

```
inbits = round(rand(1,64));   %-- 64 random bits become 8 random characters
outchar = [];
for kk=1:8:length(inbits)
   bb = inbits( ?? : ?? );      %<=====FILL in code here
   outchar = [outchar, ???????? ];    %<=====FILL in code here
end
outchar
```

2.2 Mixing Moves the Spectral Lines

The input signal to the mixer is a real sinusoid, so it has spectral lines at either ± 1650 Hz, or ± 1850 Hz. The mixer multiplies by a complex exponential, $e^{-j2\pi(1750)t}$. For both input cases, determine the location of *all* the spectral lines after mixing.

2.3 LPF Design

After mixing, the signal of interest is either $+100$ Hz or -100 Hz. Since the sampling rate is $f_s = 9000$ Hz, the desired passband of the LPF can be calculated in the $\hat{\omega}$ domain: call the result $\hat{\omega}_p$. The stopband of the digital LPF must be chosen to remove the extra spectral lines at the output of the mixer. Determine the stopband cutoff frequency that will be needed to remove these extra spectral lines. Design the digital FIR lowpass filter that will have these bandedges. Make the stopband ripple less than -40 dB, and the passband ripple less than 1 dB. The FIR filter that will meet these specs is extremely short — determine its length, i.e., number of filter coefficients.

If you use `filtdemo`, then you can work directly with the analog frequencies, once you enter $f_s = 9000$ Hz into the GUI. On the other hand, if you use `hammfilt()` you will need to enter the cutoff frequency for the lowpass filter using an $\hat{\omega}$ value that is in between $\hat{\omega}_p$ and $\hat{\omega}_s$ (half way is a good guess, but you will probably have to use trial and error to get an acceptable design).

3 Warm-up

For the exercises in this warm-up, you will need the FSK encode/modulator from Lab #11. If your own version is not yet ready, then you can use the FSK encoder/modulator called `qdfsk()` which can be downloaded from WebCT in the ZIP file called `FSK_Lab12.zip`. Look for the file called `qdfsk.p`.

3.1 More Filter Design

The digital lowpass filter (from Sect. 2.3) can be redesigned so that it can also be used as an anti-aliasing filter for downsampling the signal. Consult Section 1.1.4 in the Introduction for a discussion of changing the sampling rate from 9000 Hz to 1500 Hz. In this section, determine the filter specs needed to perform correct anti-aliasing. With respect to the LPF designed in the Pre-Lab in Sect. 2.3, all specs can remain the same except for the stopband edge which must be changed to satisfy the anti-aliasing condition.

(a) Design the digital LPF that will act as an anti-aliasing filter. If you use `filtdemo`, then you can enter the bandedges directly, and also you will be working directly with the analog frequencies. On the other hand, if you use `hammfilt()`, you will need to enter a cutoff frequency for the lowpass filter using an $\hat{\omega}$ value that is in between $\hat{\omega}_p$ and $\hat{\omega}_s$ (try $\frac{1}{2}(\hat{\omega}_p + \hat{\omega}_s)$ as a first guess and then do trial and error).

(b) Use `xx = qdfsk('Tech',300,9000)` to create an FSK signal for a bit stream. This encoder runs at 300 bps and uses a sampling rate of 9000 Hz. If you have your own working FSK generator, you can use that M-file instead.

(c) Process the FSK signal created in the previous part through the mixer (with $f_c = 1750$ Hz) and the lowpass filter from part (a). Make a plot of the output signal in the time domain. Label the horizontal axis with time in seconds. Since the output is complex-valued, plot the real part only.

Instructor Verification (separate page)

(d) Carry out the downsampling operation and then replot the signal. Show that the plot is essentially unchanged. Note that correct labeling of the horizontal axis requires a new time vector because the sampling rate is six times lower.

(e) It is possible to view the spectrogram of the input and output of the lowpass filter, but some care is needed. First of all, for the 300 bps signal, an extremely short window length is needed when calling `specgram`, e.g., `specgram(xx,32,fs)`, because the bit duration is only 30 samples at $f_s = 9000$ Hz. Secondly, the `specgram` plot for a complex input signal extends from 0 to f_s, so negative frequency components actually show up at high frequencies. For example, -2000 Hz would show up at 7000 Hz when $f_s = 9000$ Hz.

3.2 Slicer Implementation

The objective of this section is to implement the slicer and show that it gives a constant output when the input is a single-frequency complex exponential.

(a) For the input to the slicer, use the filtered output signal from the previous section. Or, if you are unsure of that output, make a test signal by synthesizing $y[n]$ as a single complex exponential:

$$y[n] = e^{j\hat{\omega}_1 n}$$

Find the correct value for $\hat{\omega}_1$ from $f_s = 1500$ Hz and $f_1 = 100$ Hz.

(b) Note that $y[n-1]$ can easily be obtained from $y[n]$ by with a delay of one sample. In the interest of keeping the vectors the same size, the `filter()` command can be used to create $y[n-1]$:

```
yy_delayed = filter([0 1],1,yy)
```

(c) Now $s[n]$ (the slicer output) can be obtained by using the `conj()` function in MATLAB. Plot the imaginary part of the slicer output `ss` and compare this to the predicted value which is a constant:

$$\sin(2\pi(100)/1500)$$

Instructor Verification (separate page)

4 Lab Exercises

In the Warm-up section, you should have completed the implementation of all parts of the FSK demodulator, so the only thing left is the "decoder" that will synchronize on the bits and group them into 8-bit bytes.

4.1 Transmitter/Modulator

For this lab, you can use your own FSK transmitter, or an FSK transmitter will be provided for you in the M-file called `qdfsk.` The transmitter/modulator must be set to run at a sampling rate of 9000 Hz. A realistic simulation would use 8000 Hz because phone lines are 8000 Hz, but 9000 Hz keeps our sampling rate an integer multiple of 300 which is our symbol/bit rate. One useful test message is `'@U '` because it contains runs of consecutive zeros, consecutive ones, and also alternating zeros and ones.

4.2 Synchronize on the Bits

Use the FSK Modulator to produce a test signal for a very short message, and run that signal through the mixer, LPF and slicer to produce the output $b[n]$ which are the detected zeros and ones obtained from $d[n]$. For testing the synchronization algorithm that you will write, you will concentrate on the front end of $b[n]$ where the preamble is found. Since $b[n]$ is sampled at $f_s' = 1500$ Hz, each bit interval lasts for five samples.

The goal here is to find the transitions (or jumps) in $b[n]$, which also correspond to sign changes in $d[n]$. This could be accomplished with a `for` loop or a piece of smart vectorized code.[4] Consult Section 1.2.2 for methods of processing $b[n]$ to looks at the jumps. Ideally these transitions should be exactly 5 samples apart. For example, if you use the difference operator to produce the jumps and they occur at samples $n = 8$, 13, 18 etc., then the actual transition occurs somewhere between sample 8 and sample 9. If we call the jump location 8.5, then the middle of the next bit interval will be at $8.5 + \frac{1}{2}(5)$, or $n = 11$, because there are 5 samples per bit interval. All other bit decisions can be made at $n = 11 + 5\ell$

4.3 M-file for Baud Lock

Write an M-file called `jumplast` that will find the index where the final zero-one transition in a group of eleven occurs. Use the ideas from Section 1.2.2 and the previous section.

The comments below describe the calling sequence for `jumplast`. Test your code with the input signal `bb = [zeros(1,77),cos(0.2*pi*(0:60))>0];`

[4]There are cases where `for` loops in MATLAB are extremely useful. For example, when writing code that will be translated from MATLAB to C or Assembly for a DSP, a MATLAB `for` loop will provide an answer key for checking your C/Assembly-code.

```
function jindex = jumplast( bb, P )
%JUMPLAST   find the location of the last 0-1 jump in a group of eleven
%
%       jindex = jumplast( bb, P )
%
%       bb = input signal consisting of bits: 0’s and 1’s
%        P = number of samples per bit interval
%   jindex = index of the last transition. If the transition is from
%            0 to 1, then jindex is at the end of the "0". Thus, you
%            must add 3 to get to the middle of the next bit interval
```

NOTE: if you are unable to produce a working function for this module of the decoder, you can use the M-file called `qdjump.m` which is in the ZIP archive `FSK_Lab12.zip` which can be obtained from WebCT.

4.4 Get the Message Text

The final step of the decoder is to turn the bit patterns back into text. Consult Section 2.1 for the useful MATLAB functions needed to do this efficiently. Recall that the FSK transmitted data is organized as follows:

Sync Pattern	Start Flag	DATA	End Flag
`010101010101`	`11111111`	`message bits`	`1111111111111111`

Three test messages are supplied in the ZIP file `FSK_Lab12.zip`. For the first one (`FSK1test.mat`), only the data bits are given; the second one (`FSK2test.mat`) contains all of $b[n]$ including the preamble. You should be able to verify that you have correctly recovered the data for both of these. You can use the first test to check that your “bits to ASCII” code is working. The second one will allow you to test the “baud lock synchronizer.

Then you are ready to run the entire FSK demodulator/decoder to recover a message from $w[n]$ which is an FSK signal having the same data format as shown above, but this time you will have to demodulate the signal to get the data unlike the first two test messages that gave you the transmitted data bits. This is the signal `FSK3test.mat`. Please note that the DATA portion of each message is actually text, so you should get something that is readable as a sentence in English. If you correctly group the received data into bytes (8 bits), and convert each byte into its ASCII character `(help char)` you will be able to read the ECE-2025 messages.

4.5 Others

Be sure to include the following in your lab:

1. Plots of the `specgram` or sketches of the spectrum for the signals into the mixer, out of the mixer, and after the LPF.

2. A stem plot of $b[n]$ showing where the decisions will be made, based on the synchronization that found the middle of the bit interval.

3. A description of your technique for synchronizing the bits and bytes, based on your knowledge of the training signal and the data flags.

4. A comparison of the received bits to the transmitted pattern for a simple case. These ought to be identical when there were no distortions, which is the (unrealistic) condition for this lab.

5 Supplementary Thoughts

In the past we have given a project on AM radio, which is one of the oldest forms of analog communication. In this lab we built a modem, a digital form of communication, which is generally speaking far superior to analog communication. The magic of digital is that if the bits are recovered correctly (and there are a variety of advanced techniques to achieve this), then all the distortions of the channel, noise, and modem imperfections along the way are effectively eliminated. In this lab, the digital magic occurs when the decision is made to convert the slicer output $d[n]$ into $b[n]$ which is either a 0 or a 1 bit. A cell phone is a good example of the magic of digital. In spite of the horrible channel between you and the cell tower (which is changing as you cruise down the interstate while talking on your cell phones) and all the ambient noise, the data bits can be heavily encoded and recovered properly, thus achieving digital quality.

Lab #12
ECE-2025
Fall-2001
INSTRUCTOR VERIFICATION PAGE

For each verification, be prepared to explain your answer and respond to other related questions that the lab TA's or professors might ask. Turn this page in at the end of your lab period.

Name: ____ Date of Lab: __________

Part 3.1(c): Show the output from the lowpass filter that was designed to do anti-aliasing when down-sampling from $f_s = 9000$ Hz to $f_s' = 1500$ Hz. In addition, sketch the filter specifications (in the space below) that you used to create the filter.

Verified:____________________ Date/Time:___________

Part 3.2(c): Demonstrate that the slicer will give a constant output when the input is a single complex exponential.

Verified:____________________ Date/Time:___________