

**CPE 367 – Final Project – v6**  
**Design and Analysis of a DTMF Decoding System – Dr F DePiero**

**Background**

This final project provides an opportunity to design a system that includes varied signal processing stages and that uses integer-based DSP. You will have an opportunity to experiment with your own creative variations to the design. You should work on this project individually and turn in your own final report. You may consult with other students during the project; but you should create all your own Python code and perform your own testing. This project provides an opportunity to succeed in a design project that is not completely defined, and to resolve uncertainties using analysis techniques, experimentation and creative investigation.

The application area we will investigate is the touchtone dialing system known as Dual Tone Multi Frequency, or DTMF. This is a standard in the telecom industry. You hear these tones when dialing a landline phone or when selecting options (“Press 1 for pizza”). We will look at opportunities to increase the communication rate. A faster rate would permit calls to be established more quickly and could enable new applications. When attempting to increase the communication rate, we will continue to observe the DTMF standard. By respecting this standard, your results may be compatible with legacy equipment.

**Table 1. DTMF Tone Frequencies and Symbols (1,2,3, ... C,D).**

1209 Hz	1336 Hz	1477 Hz	1633 Hz	
1	2	3	A	697 Hz
4	5	6	B	770 Hz
7	8	9	C	852 Hz
*	0	#	D	941 Hz

Table 1 shows the frequencies and associated symbols used in the DTMF system. When a user presses a button on the phone keypad, two tones are produced. One tone designates the row and the other the column. Together these two tones specify a symbol. Note that A-B-C-D are not used on a standard phone. Your system will listen for pairs of tones and determine which symbol is present.

As the receiver processes the incoming signal, it continually tries to determine the proper symbol. In general, a challenge associated with digital communications occurs during the transition from one symbol to the next. The symbol errors during these transitions are referred to as inter-symbol interference, or ISI. Minimizing ISI is key to improving data communication rates because detecting a symbol more quickly permits the time associated with each symbol to be shorter. We will use ISI as a performance measure for your design. Data sets have been prepared that include samples of the DTMF audio signal along with the proper symbol associated with that symbol. A Python class, “cpe367\_sig\_analyzer,” is available that supports reading these signals, storing your own signals, plotting, and computing ISI. In addition to ISI, you will need to examine your plots to ensure that the symbol detection errors are largely confined to the transition times between symbols. Because your approach will likely include nonlinear processing blocks, a Python implementation will facilitate the optimization of the system design.

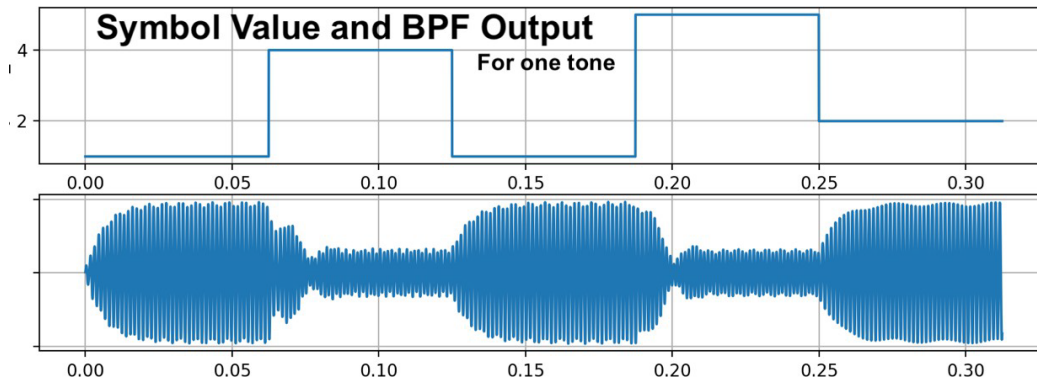


Figure 1. The symbol value (1,2,4,5) and the output of the 697 Hz BPF.

## Symbol Value and Envelope Signals

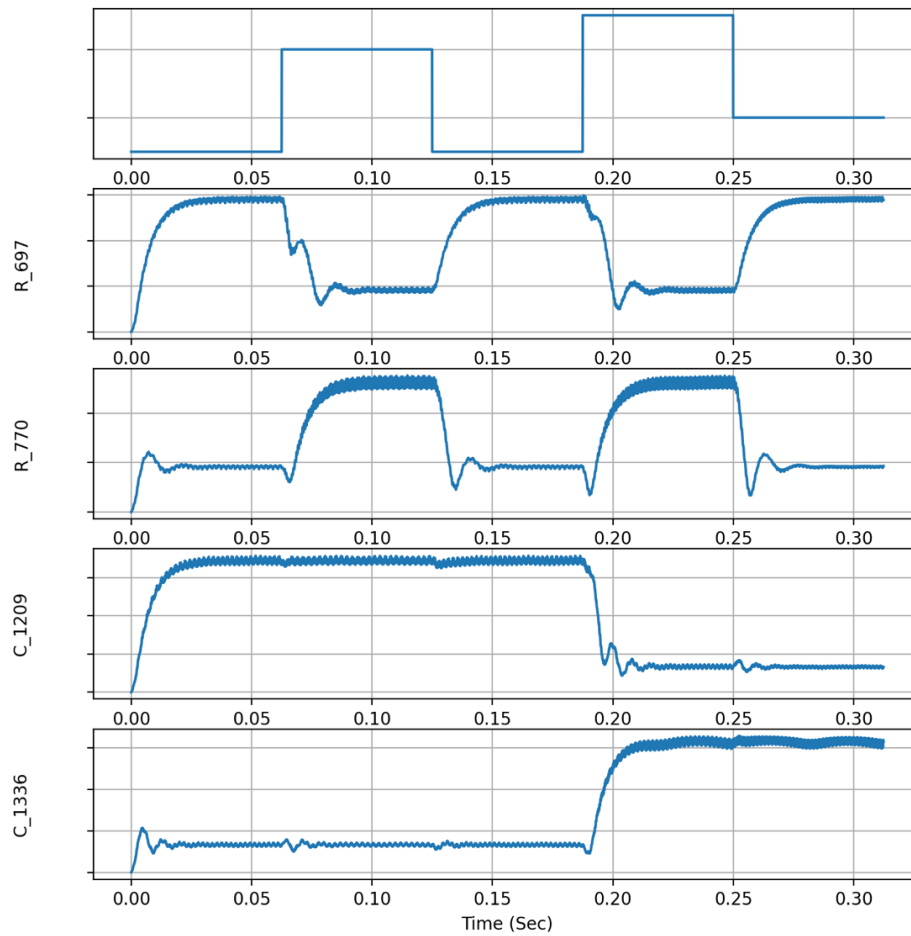


Figure 2. The symbol value and envelope signals for each tone

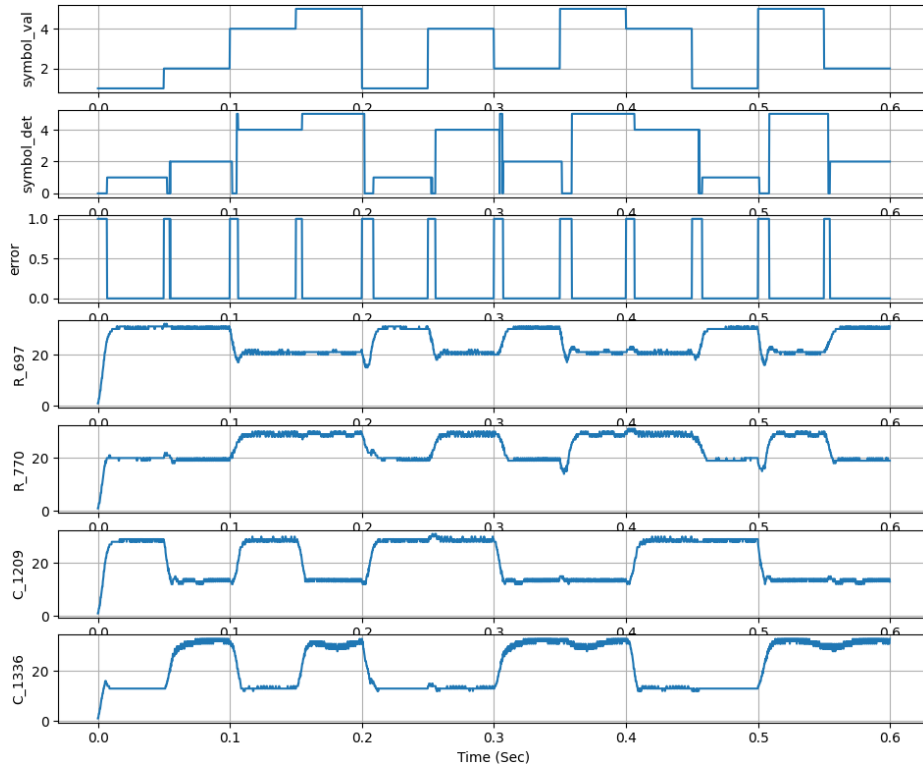


Figure 3. Symbol and detected symbol values. Shown here with an error of ~13%. Note that the error is confined to the transition between symbols. Hence, ISI in nature.

### ***Integer-Based Digital Filters***

Floating point arithmetic requires significantly more chip real estate, compared to integer-based implementations. And, floating point operations are typically slower than integer versions. As a result, the highest performance DSP processors are usually dominated by integer-based devices.

Typically, the coefficients for a digital filter will include fractional values. How can the fractional values be accommodated? Consider the following, starting with the original coefficients  $b_k$  and defining new ones  $B_k$ , for a length 2 filter. Defining a positive constant  $C$ ,

$$\begin{aligned}
 y[n] &= b_0 x[n] + b_1 x[n-1] \\
 y[n] &= C ( b_0 x[n] + b_1 x[n-1] ) / C \\
 y[n] &= ( C b_0 x[n] + C b_1 x[n-1] ) / C
 \end{aligned}$$

At this point nothing has changed. Now introduce rounding to create integer  $B_k$  coefficients

$$\begin{aligned}
 B_k &= \text{round}( C b_k ), \text{ so we have} \\
 y[n] &= ( B_0 x[n] + B_1 x[n-1] ) / C
 \end{aligned}$$

Clearly, larger C values are desirable. The larger values reduce inaccuracies introduced by rounding the filter coefficients, and any resulting change to the frequency response. Another issue remains. Division is the costliest of the four basic arithmetic operations (in terms of delay and chip area). However, division by certain values is simpler and faster to evaluate! ( $C=2^r$ )

### Design Standards and Constraints

The following standards and constraints must be respected when optimizing or experimenting with your design

- DTMF standard including tone frequencies
- Narrow bandpass filters (“high Q”) need to be IIR type for good frequency selectivity and for a computationally efficient design (only a few calculations are needed for each difference equation)
- Integer-based filter implementation, also for a computationally efficient design
- Use signals as given, including the sample rate of 4000 Hz
- Signal detection errors must be confined to inter-symbol transients, not mid symbol

### Filter Design by Pole/Zero Placement

Design your filters using the method of pole / zero placement. Poles should be located at  $p_i = r e^{\pm j\omega_0}$ . Consider using an initial pole radius of  $r=0.9$ , and feel free to adjust this value. You will need to enter a gain factor, G, to provide a passband gain that is approximately 1. A rough guess for  $G = 1 - r$ . You can experiment with different values for G when working in MatLab.

$$H(z) = \frac{Y(z)}{X(z)} = \frac{G z^2}{z^2 - 2r \cos(\omega_0)z + r^2} \quad \text{Eq 1}$$

Beginning with Eq 1, convert to negative powers of z and then “cross multiply.” Convert to the time domain, to reveal a difference equation.

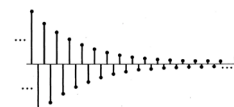
$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 z^2}{z^2 + a_1 z + a_2} = \frac{b_0}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

$$Y(z) + a_1 z^{-1} Y(z) + a_2 z^{-2} Y(z) = b_0 X(z)$$

$$y[n] + a_1 y[n-1] + a_2 y[n-2] = b_0 x[n]$$

$$y[n] = b_0 x[n] - a_1 y[n-1] - a_2 y[n-2]$$

A system with 2<sup>nd</sup> order complex conjugate poles such as this will have an impulse response in the form

$$h[n] = A (r)^n \cos (\omega_0 n + \theta)$$


The impulse response has the same form as the transient response produced by a filter for any new input. Note that the exponential decay envelope of  $h[n]$  (i.e.,  $r^n$ ) is established by the pole radius, and the oscillation frequency is set by the angular position of the poles ( $\omega_0$ ). In the context of this project, this means that the ISI will be impacted by the pole radius, as it is related to the duration of a transient (the larger the pole radius  $\rightarrow$  the slower the transient decay  $\rightarrow$  the longer it takes for the filters to respond to the arrival or end of a tone at its center frequency).

MatLab's `fvtool()` is helpful for analysis. It displays the frequency response, impulse response and a pole / zero plot, for a filter. See the "Analysis" menu.

```
fvtool([b0],[1, a1, a2],'Fs',4000)    or  
fvtool(bk,ak,'Fs',4000)
```

It can also help when approximating the impact of integer arithmetic

```
C = 1024;  
Ak = round(C .* ak) ./ C;  
Bk = round(C .* bk) ./ C;
```

### Procedure

As with most design projects, your process will likely be circuitous and iterative.

- 1) Create an initial "high level" block diagram to define your system. You may want to create different versions, or leave some aspects that are yet to be defined. Nevertheless, it can serve as a guide for you.
- 2) Use MatLab to create and analyze your filters. You will likely modify your filter specifications over the course of the project. Create a simple M-File that computes the rounded filter coefficients and calls `fvtool()`. This M-File will include the individual command-line steps that would otherwise be needed. Find appropriate values for  $G$  (a.k.a.  $b_0$ ) when working in MatLab.
- 3) Create an initial Python program to simulate the operation of your detector system. Use the main program provided as a starting point. It illustrates the steps of signal input and plotting. It also can compute the mean of a signal, such as the ISI error signal. Consider implementing the first stages of your processing system (*e.g.*, bandpass filters) and examine the results using the plotting routines. Compute the filter coefficients within your Python program to make them easy to modify. The example program shows how to add signals to a plot.
- 4) Repeat steps (1)-(3) as needed, and in any order.

### Development Process and Performance Goal

Begin working with the slow version of the signal and floating point computations in your filter implementations. Once they are working properly, convert your filters to integer computations with a large integer scale factor,  $C = 1024$  (for very little roundoff errors). Establish an ISI under 15%, or at most 20%. When you get this working, try switching to the faster version of the signals and also reduce the integer scale factor (by factors of two) while maintaining an ISI error of ~15%. Modify your system design as needed to achieve as good an ISI level as you can with integer computations (faster) and lower scale factors (requiring fewer bits for numeric calculations).

## Documentation

- Group members
- Summarize your results. What was your final ISI error? How many bits are needed for your filter coefficients?
- A “high level” block diagram describing your system. This should define functionality using subsystem blocks, such as a filter, absolute value, or similar.
  - Label each block and include relevant parameters
  - Include all interconnecting signals and all input/outputs
- A “low level” block hardware diagram of one of your bandpass filters. Include integer coefficients and all arithmetic operations
  - Use any of the standard block layouts (Direct Form I or II, or transpose). Or create your own ad hoc structure
  - Include the shifter associated with integer-based DSP
- Include both floating point and integer versions of your BPF coefficients. Report integer coefficients as scaled integers.
- Complete listing of your Python source code used to implement your system. The Python version of your system should use integer coefficients. (Even though Python will often promote intermediate results to float variables).
- MatLab filter analysis for your integer-based coefficients
  - Frequency response plots of each filter, with magnitude and phase
  - M-File used to compute filter coefficients and to call fvtool()
  - What are the true center frequencies of your bandpass filters (Hz)
  - Approximately how long (in Secs) does it take for the impulse response to drop by at least 50%? Try using the “Impulse Response” menu in the fvtool().

## Additional Questions

- Referring to your high-level block diagram provide a brief description (a paragraph) of how your system works.
- Describe your system design evolved over the course of the project. Were there significant changes to your approach?
- How would you modify your system design to provide a separate output signal indicating that no symbol is present?
- How might your system be improved, in terms of: ISI performance, computational requirements or numerical representation?
- What are benefits of using the dual tones of the DTMF standard?
- Could a square wave be used? What are pros and cons of using a square wave to generate the various DTMF signals?