

C# Frequency Sampling-Based FIR Filter Design

Binh Quoc Tran
Electrical and Computer Engineering
Virginia Military Institute
Lexington, United States
tranbq23@mail.vmi.edu

Cole Bowyer
Electrical and Computer Engineering
Virginia Military Institute
Lexington, United States
bowyerccs25@mail.vmi.edu

James C. Squire
Electrical and Computer Engineering
Virginia Military Institute
Lexington, United States
squirejc@vmi.edu

Abstract—Finite Impulse Response (FIR) filters can be designed to be linear phase, causal, and they are guaranteed to be stable. These advantages result in their wide adoption in audio processing, communications, image processing, and pattern recognition, among other applications. Some common design methods of FIR filters include windowing, multiband with transition bands, constrained least squares, frequency-sampling, arbitrary response, and raised cosine. Yet, despite the ubiquity of FIR filters, no open-source implementation of the frequency-sampling method of FIR design in the popular C# language is available.

This paper presents open-source FIR filter design code that implements the frequency-sampling method in C#, and verifies its operation. This well-known filter design method takes a set of frequencies and the desired filter's amplitude at each, and then interpolates these points to create the same number of frequency/amplitude pairs as the desired FIR filter order, using equally-spaced frequencies spanning $\omega=0$ to π rad/s. The inverse Discrete Fourier Transform is applied to this data to create a time-domain response, and then this is windowed to create the impulse response of the system that implements the desired filter. Performance testing compared paired filters in MATLAB and C# that were each designed to mimic several audiograms. Each audiogram specified desired attenuations from -80 dB to 6 dB at eight logarithmically spaced frequencies from 250 Hz to 8 kHz, and these were realized with the design of a 1000 tap FIR filter.

In all cases, the C#-computed filter's frequency domain performance matched the one designed by MATLAB essentially perfectly, to within two orders of magnitude of the precision of the double data type, suggesting that the open-source FIR filter design method we describe is successful.

Keywords—FIR, filter, frequency sampling

I. INTRODUCTION

Signals exist everywhere and in forms such as speech, music, picture, and video [1]. They carry information in their dependent variable(s) such as pressure, voltage, illumination, stock price, or temperature as a function of their independent variable(s), which are often time or distance. Signals may be functions of one or multiple independent variables and encode information in one or multiple dependent variables. Their independent and dependent variables may be continuous or discretized into predefined quantized levels. If only the independent variable system is discretized we refer to the system

as a discrete time system, and if both the independent and dependent variables are quantized we refer to the system as digital [1].

This paper considers one type of signal, audio (see Fig. 1), to motivate the development of an open-source filter that modifies it. The audio data considered is recorded in discrete time at a given sampling frequency, and it is encoded at a sufficiently high bit depth that artifacts from amplitude quantization of the dependent variable can be ignored. Only one independent variable (time) and one dependent output variable (air pressure of the audio signal) are considered, although the extension to multiple channel audio is trivial.

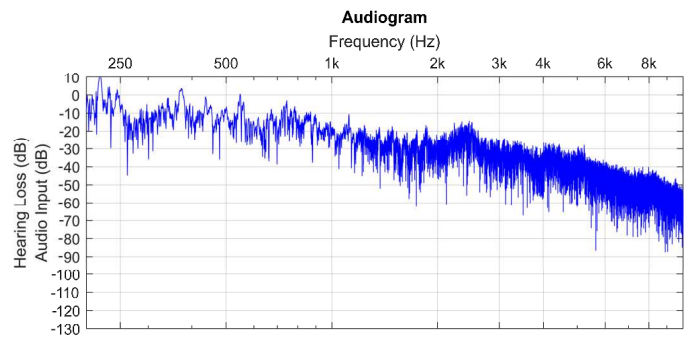


Fig.1 The energy content by frequency of the spoken phrase “Hello there.” In a speech, consonants form broadband signals that hold more speech information than the vowels which form more narrowband signals [2]. Vowel energy predominantly ranges from 250 Hz to 1 kHz, and consonant energy ranges from 250 Hz to 8 kHz. Speech may still be intelligible without vowels, so people with high-frequency hearing deficits may retain the ability to orally communicate. [3]

If signals are nouns, filters are the verbs that act on input signals and map them to corresponding output signals. Filters have many different purposes, may be linear or nonlinear, and may be time-variant or time-invariant [4]. Filters may be specified in the time-domain, for instance when creating matched filters or time-delay filters, or they may be specified in the frequency-domain, for instance, when amplifying or attenuating signals based upon their frequency components [1]. Frequency-based filters may have a simple specification, such as lowpass filters that attempt to pass signals below a given frequency and stop components with frequencies above it, or the specification may be more complex, such as specifying the desired attenuation in several different passbands.

This paper describes the development of the latter type of filter, one using the frequency sampling method, that permits the

specification of attenuation of the filter at any number of given frequencies. Fig. 2 shows an example audiogram, which is representation of the filtering that occurs within the human hearing system at eight predefined frequencies. A person with normotypic hearing has an audiogram of 0 dB across all tested frequencies. A hearing-impaired person will experience attenuated hearing at certain frequencies. Audiologists consider less than 20 dB attenuation as normal; 20-40 dB attenuation is considered mild hearing loss, 40-70 dB is moderate, and greater than 70 dB attenuation relative to normotypic hearing is considered profound hearing loss.

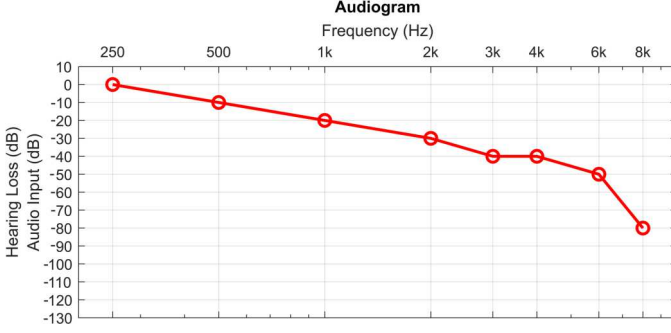


Fig. 2 This human audiogram is an example of a filter specified in the frequency domain. This is taken from a person with profound high frequency hearing loss, but relatively normal low frequency hearing. The figure depicts the filtering operation of the person's hearing system relative to a typical person's 0dB hearing across the range of frequencies from 250 Hz to 8 kHz.

The phase response of a filter describes the output phase relative to the input phase as a function of frequency. A nonlinear relationship between the frequency and phase of a filter results in the change of the time-domain shape of a signal in the passband. A linear relation between frequency and phase is preferred for applications that are sensitive to the morphology of a signal, including many audio applications, since these filters maintain a constant group delay.

Discrete-time filters may be broadly categorized by the length of their impulse response, into infinite impulse response (IIR) and finite impulse response (FIR) filters. IIR filters are typically chosen for applications where the linear phase is less important, since they cannot be made precisely linear phase if finite order, although they have an advantage over FIR filters by having generally a steeper frequency-domain response for a given filter complexity. FIR filters, in contrast, are guaranteed stable for any input, and can easily be made to have linear phase characteristics by designing a symmetric, or anti-symmetric, impulse response around their group delay. This paper describes the design of a linear phase, FIR filter.

Once the $M+1$ length impulse response $h[n]$ of a FIR filter is determined, the output $y[n]$ to a given input $x[n]$ can be calculated using the convolution sum shown below.

$$y[n] = \sum_{k=0}^M x[n-k] h[k] \quad (1)$$

This convolutional sum has several implementations. The Direct Form II Transposed implementation is shown in block diagram form in Fig. 3 and is implemented in code in Fig. 6.

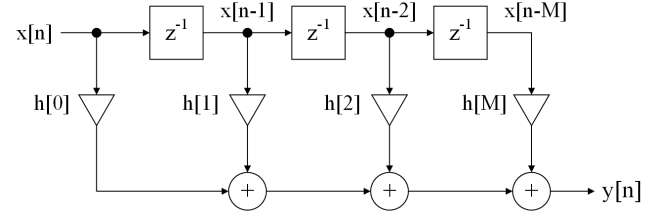


Fig. 3 The direct form implementation of a FIR filter of order M sums scaled values of $M+1$ taps of the input delay line. The $M+1$ scaling coefficients of $h[n]$ specify the FIR filter.

There are many well-known FIR filter design methods, including the window design method, the raised cosine method, the frequency sampling method, constrained least squares, and the Parks-McClellan method for equiripple filters [5]. The windowing method samples the central section of an ideal IIR impulse response to create a finite-duration impulse response. The raised cosine method produces low pass, high pass, band pass, and band stop filters with smooth frequency band transitions. The frequency sampling, Parks-McClellan, and least square minimization techniques all permit the design of filters with arbitrary frequency-domain specified behavior. This paper describes an implementation of the frequency sampling FIR, linear phase filter in C#.

II. FILTER DESIGN METHOD

The frequency sampling method permits the design of an arbitrary-magnitude filter in the frequency domain as shown in Fig. 4. The user specifies two vectors holding the desired magnitude response at a corresponding set of frequencies. The user also selects the filter order M corresponding to the desired FIR filter order. The larger the order, the more closely the designed filter will approximate the specified response, although it may develop larger artifacts at non-specified frequencies. The filter coefficients are extracted and form the scaling factors $h[n]$ in Fig. 3 to implement the filter.

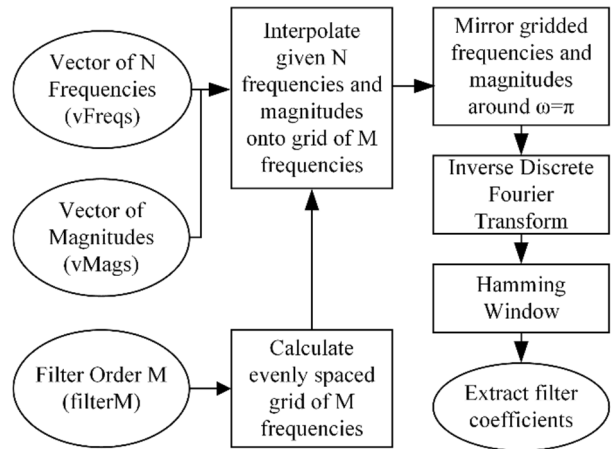


Fig. 4 The frequency sampling method of FIR filter design.

III. CODE

The C# code that constructs the filter, an example of its use, and a readme document is available through the link below in GitHub.

<http://bit.ly/3gBNJDI>

```
double fs = 44100;
double[] fHz = {0,250,500,1e3,2e3, 3e3, 4e3, 6e3, 8e3, fs/2};
double[] vaudiogram = {6, 0, 0, -20, -30, -40, -50, -60};

// Filter Order M
double filterM = 1000;

// Calculate the Vector of N frequencies
double[] vFreqs = new double[fHz.Length];
for (int i = 0; i < fHz.Length; i++)
    vFreqs[i] = fHz[i] / (fs / 2.0);

// Calculate the Vector of Magnitudes
double[] vA = new double[vaudiogram.Length];
for (int i = 0; i < vaudiogram.Length; i++)
    vA[i] = Math.Pow(10, (double)vaudiogram[i] / 20.0);

double[] vMags = new double[vA.Length + 2];
vMags[0] = vA[0];
vMags[vMags.Length - 1] = vA[vA.Length - 1];

for (int i = 1; i < vMags.Length - 1; i++)
    vMags[i] = vA[i - 1];

// Calculate the Filter Coefficients
double[] h = new double[(int)filterM + 1];
Filter(filterM, vFreqs, vMags, out h);

// Print the Filter Coefficients
Console.WriteLine("Filter Coefficients: ");
for (int i = 0; i <= filterM; i++)
    Console.WriteLine(h[i]);
```

Fig. 5 This code provides an example of generating the filter specified by the upper panel of Fig. 9. It takes the fHz vector of specified frequencies in Hz and the corresponding desired filter magnitude response vaudiogram vector. The FIR filter coefficients are returned in vector h in the same order as listed in eqn. 1 and Fig. 3.

The core C# filter design program is named Filter, and its arguments are listed in Table I.

TABLE I. FILTER ARGUMENTS

| Input | Data Type | Description | Notes |
|---------|-------------------|-------------------------|-----------------------|
| filterM | double | filter order | must be even |
| vFreqs | vector of doubles | vector of N frequencies | ranges from 0 to 1 |
| vMags | vector of doubles | vector of N magnitudes | same length as vFreqs |

The FIR filter coefficients generated by the Fig 5 code can be used to filter an input signal as shown in Fig 6. This example first generates an input signal using eight segments of one-second sinusoids, each segment corresponding to the eight frequencies in Fig. 5. It then completes the filtering operation with FIR filter h using the direct-form II transposed implementation [6]. The output, shown in Fig 7, demonstrates the specified filter attenuations at each of these eight frequencies. A precise analysis of the filter performance is conducted in the following Results section.

```
// Define variables
int fs = 44100;
double[] x = new double[fs*8]; // input vector
double[] fHz = {250, 500, 1e3, 2e3, 3e3, 4e3, 6e3, 8e3};

// Create the input signal x of 1s of each frequency in fHz
double t = 0; // current time
for (int iFreq = 0; iFreq < 8; iFreq++, t += 1.0/fs)
    for (int i = 0; i < fs; i++)
        x[i+iFreq*fs] = Math.Cos(2*Math.PI*fHz[iFreq]*t);

// Filter x using the filter h created in Fig. 5
double[] y = new double[x.Length]; // output vector
for (int i = 0; i < x.Length; i++)
{
    for (int j = 0; j < h.Length; j++)
    {
        if (i - j < 0) continue;
        y[i] += h[j] * x[i-j];
    }
}

Console.WriteLine("Filtered output Data: ");
for (int i = 0; i <= filterM; i++)
    Console.WriteLine(y[i]);
```

Fig. 6 This example code creates a test signal composed of eight one-second signals from 250Hz to 8kHz to validate the accuracy of the filter designed in Fig. 5.

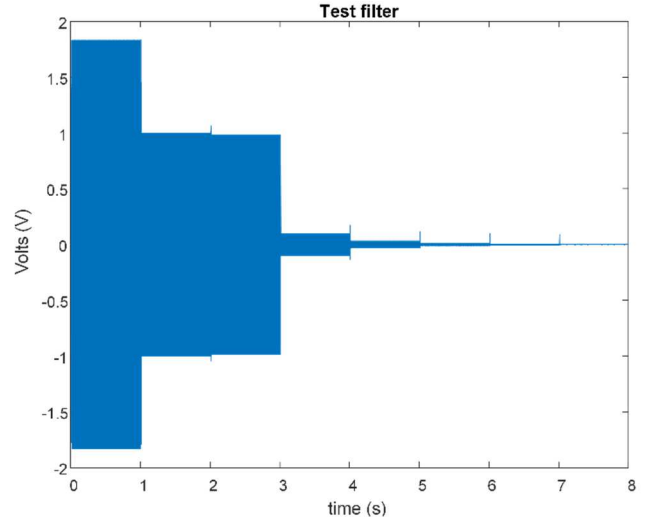


Fig. 7 The time-domain output of the filter designed in Fig. 5 to a set of one second sinusoids of 1V magnitude corresponding to the frequencies of the designed filter, specifically 250, 500, 1k, 2k, 3k, 5k, 6k, and 8kHz. The resulting signal's gain approximately mirrors that of the design specifications.

IV. RESULTS

Filters designed using the C# code reported in this paper were compared with both the desired filter's specification and with filters designed using MATLAB's frequency sampling-based FIR filter design (fir2) procedure [7]. Three representative audiograms were chosen for testing, and for each audiogram two graphs were generated. The upper panel compares the specified filter magnitude/frequency pairs (shown as a scatterplot using boxes) with the C#-generated filter (shown as a dotted line) and the MATLAB-generated filter (shown as a dashed line). Since the MATLAB and C# data are very similar and thus difficult to distinguish, the difference between them are plotted in the lower panel.

The first example in Fig. 8 is a simple reference case of a constant -10 dB hearing loss across all frequencies. The upper panel indicates there is no apparent difference between this scaling filter designed by the C# program and MATLAB, and both filters perform the specified 10 dB attenuation. The lower panel shows the maximum deviations between the desired specifications and the C# filter, as well as the maximum difference between the filters designed in C# and MATLAB, computed as (C# attenuation in dB – MATLAB attenuation in dB). The largest of these differences is 1.42×10^{-14} , on the order of machine epsilon given the double precision IEEE numbers used.

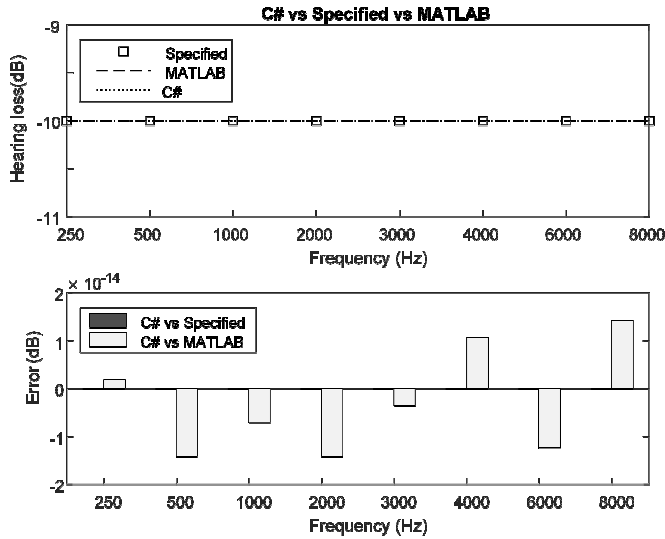


Fig. 8 A reference filter design that cuts all frequencies by -10 dB performs as expected, with matched C# and MATLAB reference filter design outputs indistinguishable and identical to the filter specification to within an order of magnitude of machine precision. There are no visible black bars in the lower panel since there was no difference between the specified and C# computed filter attenuations.

The second example in Fig. 9 illustrates a typical audiogram with a complex relationship between test frequencies and hearing loss. This is also the same specification as represented in Fig. 5 and illustrated in the time domain by Fig. 7. This example shows excellent agreement between the C# filter and the specified audiogram, with the largest absolute error at 250 Hz. At this frequency, the C# filter's attenuation is about 5.3 dB, about 0.7 dB less than the specified 6 dB. This can be seen in Fig. 7, in which the first second of the output sinusoid, corresponding to 250Hz, only has an amplitude of 1.84, which corresponds to 5.3 dB. This 0.7 dB maximum error compares favorably with the 66 dB spread of specified gains of the filter, and is a result of the limited frequency resolution of the filter, from its order, coupled with smoothing effects of the Hamming window.

There was no difference between the C# designed filter and the MATLAB designed filter at the scale of the lower panel, and thus only the black bars of the C# vs. specified gain bars are visible.

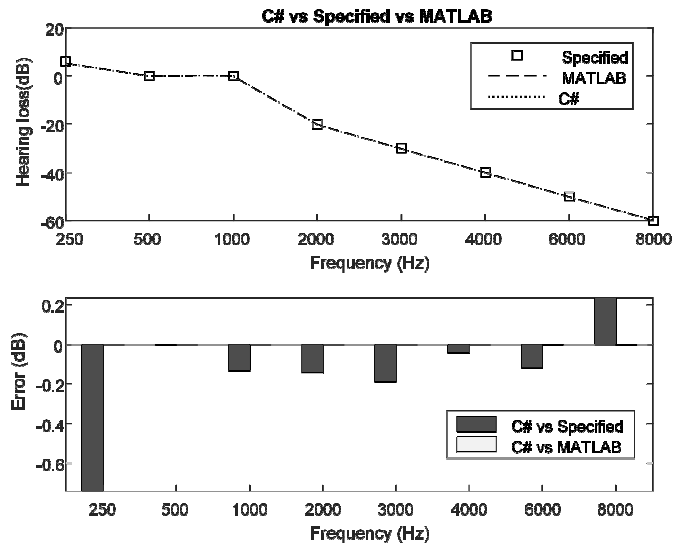


Fig. 9 This complex filter specification, typical of an actual audiogram for a person with severe high frequency hearing loss, shows excellent agreement among the filter specification, the C#-designed filter, and the MATLAB-designed filter, with a maximum deviation of about 0.7 dB at 250 Hz.

The final example demonstrates how large changes in the specified attenuation over short changes in frequency strain the ability of this frequency-specified filter to match accurately. Fig. 10 shows an audiogram in which over 50 dB of filter attenuation must change given a 250 Hz change in input. At these extremes, up to a 10.5 dB error between specified and actual attenuation occurs. As in previous examples, the difference between the C#-designed filter and the MATLAB-designed filters is not noticeable at this scale and in fact is within two orders of magnitude of machine epsilon.

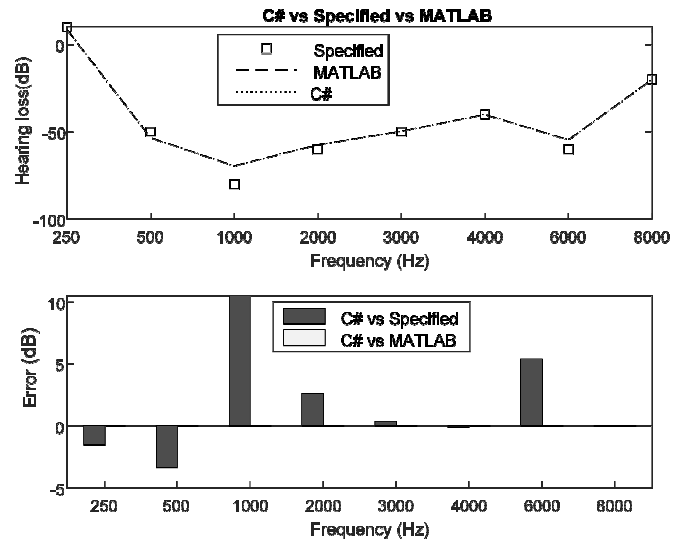


Fig. 10 Specifying large changes in filter attenuation over small changes in frequency can strain the ability of the frequency-specification filter design method. In this example, changes of roughly 50 dB of attenuation over 250 Hz cause errors of about 10 dB between the specified and designed filter.

Other than the filtering performance shown above, several other performance metrics of this C# filter were compared with the gold standard of discrete-time filter design, MATLAB.

Specifically, the speed, cost, and system requirements for the 1000 tap frequency sampling-based FIR C# filter was compared to MATLAB, as shown in Table II.

TABLE II. PERFORMANCE METRIC COMPARISON

| | MATLAB | C# |
|-------------------|-------------|-------------|
| Speed | 478 μ s | 200 μ s |
| Disk Space | 700 MB | 120 kB |
| RAM | 1.2 GB | 2.7 MB |

It must be noted that it is difficult to perform a direct comparison between C# and MATLAB since the former is compiled into a compact intermediate language, and the latter is typically executed interactively as an interpreted language. While MATLAB does support the MATLAB Coder toolbox to create C or C++ code, Mathworks does not support direct compilation of their Signal Processing Toolbox functions, including fir2.m [8]. Instead, one must use the MATLAB Compiler Runtime (MCR) to create a standalone application. This involves a large amount of disk space and RAM. A user who prefers the MATLAB development environment could avoid this overhead by recreating the fir2.m functionality by hand coding it in MATLAB without use of toolboxes, much as we have done in C#; this is outside the scope of this paper.

A. Speed

Iterated testing showed that the average speed to design a 1000th order FIR filter using the MATLAB frequency specification method is 478 μ s, while the average speed for the C# filter is 200 μ s, about 2.4 fold faster. The absolute speed is clearly a function of the particular computer used (Intel® Xeon® E-2224G CPU, 3.5GHz, 32 GB RAM), but the fact that the C# code is about 2.4x faster is a relatively invariant metric.

B. Disk Space

Mathworks advises that a minimum installation of MATLAB requires 4 GB of disk space. Since MATLAB is usually used interactively through its IDE to design filters, this is therefore the same space that it takes to design frequency sampled filters. Use of the MCC compiler to compile the code to run without MATLAB reduces the file size to about 700 MB with the MCR. In contrast, C# console code such as shown in Figs. 5 and 6 compile to about 120 kB, about 0.02% of the compiled MATLAB size. The C# code can be further reduced to 25 kB if the optional Windows icons are removed. Minor changes to port this to the C language would make this appropriate for use in embedded systems.

C. RAM

The memory requirement for MATLAB is listed by Mathworks as 4 GB, however Windows reports that it actually uses only approximately 1.2 GB when running the frequency specified filter design function. Windows reports that the C# compiled program uses 2.7 MB of RAM, about 0.2% of the memory that MATLAB requires. This also suggests its suitability for embedded systems use if ported to C. This is a relatively simple port since no object-oriented functionality or external dependencies are used.

V. CONCLUSION

This paper presents a frequency sampling-based FIR filter design program using the C# language. Example filter design and filter testing code is provided in a GitHub repository. Testing was performed using speech audiogram processing as an example application, and its filtering performance was found to be indistinguishable from MATLAB, within rounding errors caused by limited machine precision. Since this code is far more compact than MATLAB's, and is fully specified without requiring associated .dll's, it may also be appropriate for embedded system use with slight modifications into the C language.

REFERENCES

- [1] S. K. Mitra, "Digital Signal Processing: A Computer Based Approach," McGraw-Hill Higher Education, pp. 1-15, 2001.
- [2] J. W. Pitton, K. Wang, B. Juang, et al., "Time-Frequency Analysis and Auditory Monitoring for Automatic Recognition of Speech," Proc. of the IEEE, vol. 84, pp. 1199-2004, September 1994.
- [3] S. Furui, "Digital Speech Processing, Synthesis, and Recognition," Marcel Dekker, pp. 14-20, 2000.
- [4] W. M. Siebert, "Circuits, Signals, and Systems," The MIT Press, pp. 4-5, 1986.
- [5] L. R. Rabiner and B. Gold, "Theory and Application of Digital Signal Processing," Prentice-Hall, pp. 136-140, 1975.
- [6] A. V. Oppenheim, R. W. Schaffer, Discrete-Time Signal Processing. Prentice Hall, pp. 309-313, 1999.
- [7] "Frequency sampling-based FIR filter design - MATLAB fir2," [www.mathworks.com](https://www.mathworks.com/help/signal/ref/fir2.html). <https://www.mathworks.com/help/signal/ref/fir2.html> (accessed Dec. 10, 2022).
- [8] "Support Limitations for MATLAB for Code Generation - MATLAB & Simulink," [www.mathworks.com](https://www.mathworks.com/help/sldv/ug/support-limitations-for-matlab-for-code-generation-1.html). <https://www.mathworks.com/help/sldv/ug/support-limitations-for-matlab-for-code-generation-1.html> (accessed Dec. 10, 2022).