Fall 2021 EE 313 Linear Systems and Signals Prof. Evans

Mini-Project #1: Sinusoidal Speech Synthesis

Solution Set 2.0a by Prof. Brian L. Evans

September 17, 2021

1.0 Introduction (*3 points*)

This mini-project use of a sum of sinusoids to synthesize a recorded vowel sound in English [1] or another language. Vowel sounds have a nearly harmonic structure. Our goal will be to record a vowel sound and use enough sinusoidal terms to synthesize the vowel sound so that it is intelligible. Our first effort will to be use frequencies that are not harmonically related and then try to use ones that are.

2.0 Overview (*6 points*)

A continuous-time periodic signal with period T_0 in seconds is composed of a constant term plus frequency components at integer multiples (harmonic) of a fundamental frequency f_0 where $f_0 = 1 / T_0$. Fourier series analysis computes the constant term plus the magnitude and phase of each frequency term:

$$x(t) = \sum_{k=-\infty}^{\infty} a_k e^{j2\pi k f_0 t} \quad \text{where} \quad a_0 = \frac{1}{T_0}\int_0^{T_0} x(t)\,dt \quad \text{and} \quad a_k = \frac{1}{T_0}\int_0^{T_0} x(t) e^{-j2\pi k f_0 t}\,dt$$

For any real-valued signal $x(t)$, $a_{-k} = a_k^*$ where $a_k = \frac{1}{2} A_k e^{j\phi_k}$, as shown in Section 3.1 of [2],

$$x(t) = A_0 + \sum_{k=1}^{\infty} A_k \cos\left(2\pi k f_0 t + \phi_k\right)$$

We can synthesize $x(t)$ by keeping a finite number of terms. Sometimes, the synthesis is exact.

When propagating in air, audio signals consist of propagating acoustic pressure waves. A microphone can convert the intensity of the impinging acoustic pressure wave into an analog continuous-time voltage signal, which an analog-to-digital converter can convert into a digital discrete-time audio signal. For playback, a digital-to-analog converter will convert the digital discrete-time audio signal into an analog continuous-time voltage signal, which can be converted into acoustic pressure waves by an audio speaker. Common sampling rates for voice include 8000 Hz, 16000 Hz, and 32000 Hz as well as 11025 Hz and 22050 Hz and for audio include 44100 Hz (audio CD) and its multiple 88200 Hz as well as 48000 Hz for digital audio tape and its multiples 96000 Hz and 192000 Hz.

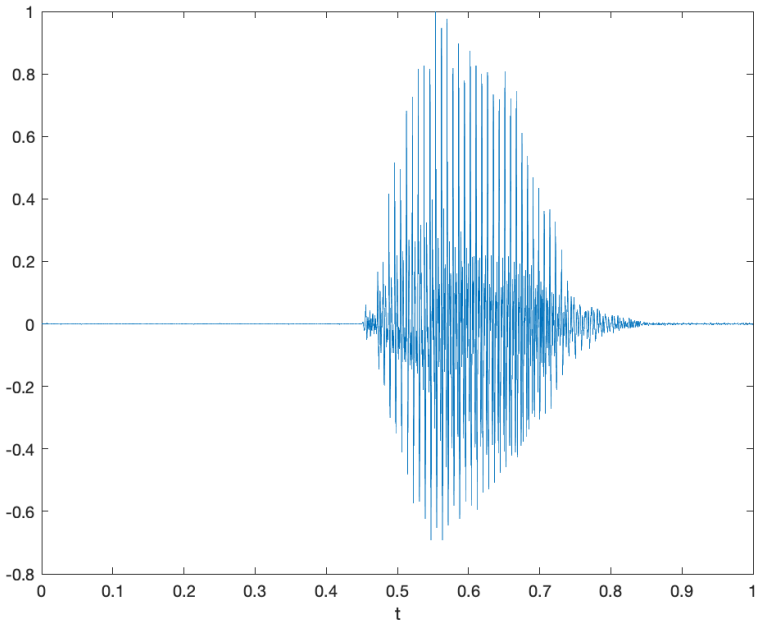
3.0 Analyzing a Vowel Sound

3.1 Time-Domain Analysis

We can now analyze the recorded speech in the time domain.

(a) *6 points*. We record a spoken vowel sound of a *long e* using a standard speech processing sampling rate of 8000 Hz (i.e. 8000 samples/s) for 1s using the MATLAB code UTAudioRecordAndPlayback.m (see Appendix B). That will give a vector 8000 speech amplitude values.

(b) *6 points.* A time-domain plot of the speech utterance using UTAudioTimeDomainAnalysis.m (see Appendix B) is give below on the left for the recording of a *long e* sound on the right:



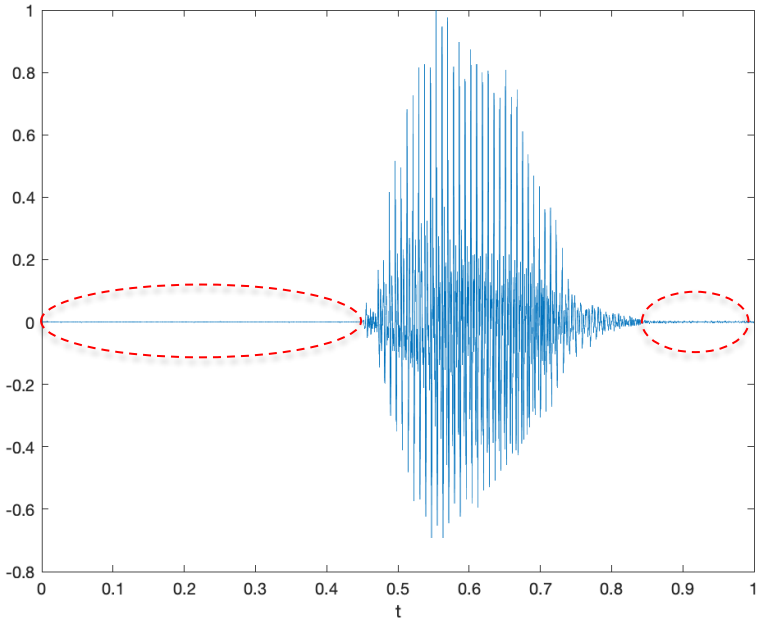
Sound clip
Long e sound

(c) *6 points.* We compute the average value of the signal using MATLAB:

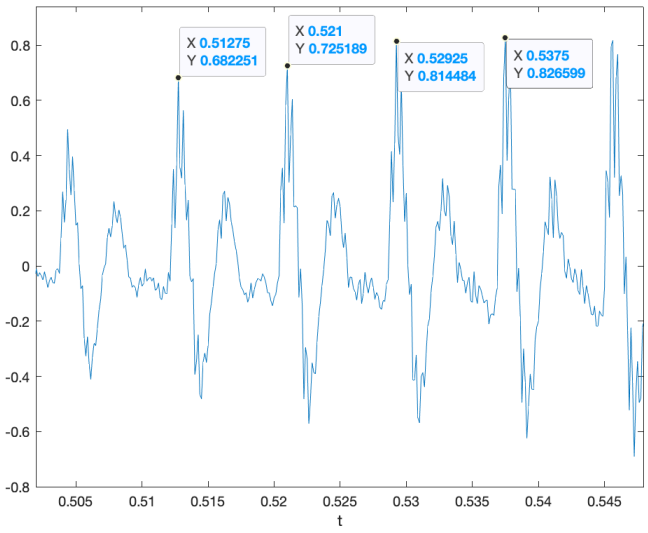
```
mean(myRecording)
  -1.4893e-05
```

Speech and audio signals typically have an average value of 0 or close to zero. The average value, or DC offset, cannot be perceived by the human auditory system. The average value would occupy bits in the sampled data without conveying meaningful information to the human auditory system. Moreover, playback systems typically pass frequencies within the range of normal human hearing of 20 Hz to 20 kHz. Some laptop playback systems, for example, cannot play frequencies below 200 Hz. The highest frequency perceived by the human auditory system depends on age.

(d) *6 points.* The quiet regions before and after the speech utterance are circled below.

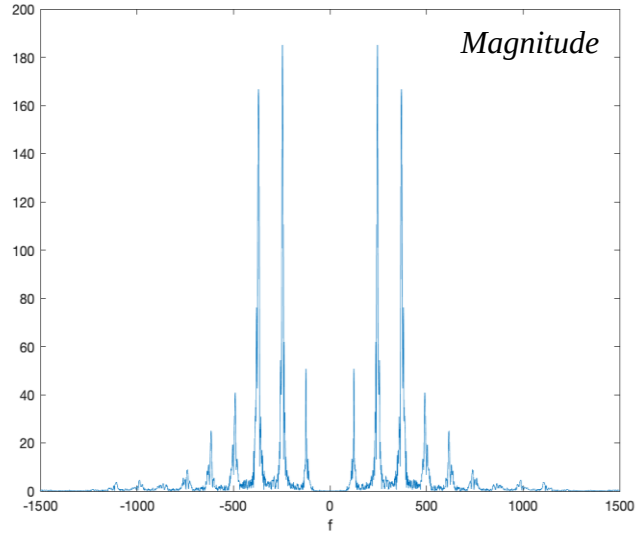


(e) *6 points.* The pitch period is an important quality in a human voice. It corresponds to the period between the opening, closing, and reopening of the glottis in the vocal tract. This opening and closing creates an impulsive flow of air from the lungs through the mouth and nose. The mouth and nose have a damped sinusoidal response, like the ringing of a bell. [3][4] For a vowel sound, the glottis opens and closes in a quasi-periodic fashion.



In the time-domain plot window, we will zoom in to show a few pitch periods using the "Zoom In" tool and then use the "Data Tips" tool to find the specific times for the initial peak response each time the glottis opens. [3] The time from the first highlighted peak to the fourth highlighted peak corresponds to three pitch periods. The average pitch period is (0.5375s - 0.51275s)/3 which is 8.25ms. A pitch period of 8.25ms corresponds to a pitch frequency of 121.2 Hz (i.e. 1/8.25ms is 121.2 Hz).

The pitch frequency in a human voice is typically in the 100-300 Hz range. An individual's normal voice under non-stressful conditions would have a pitch frequency that does not vary too much from utterance to utterance in English. This might be different for tonal languages.

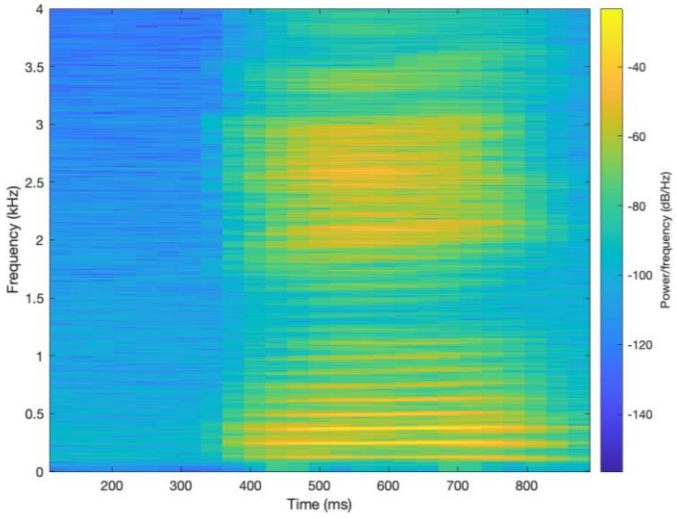


3.2 Frequency-Domain Analysis

Next, we analyze the speech in the frequency and time-frequency domains. From Appendix B, we'll use the UTAudioFreqDomainAnalysis.m script.

(a) *6 points.* The frequency content of the recorded speech is shown to the right in terms of the magnitude of each frequency component. Due to sampling at 8000 Hz, we capture frequencies in (-4000 Hz, 4000 Hz) because of the sampling theorem $f_s > 2\ f_{max}$ where f_{max} is the highest frequency of interest and hence $f_{max} < ½\ f_s$. The plot zooms into frequencies [-1500 Hz, 1500 Hz] because the vowel utterance has very low magnitudes outside that range. The computation uses the discrete-time Fourier series— see Appendix A for the connection to the continuous-time Fourier series.

(b) *6 points.* Shown on the right is the spectrogram for the recorded audio signal. The spectrogram analyzes the frequency content of a block of samples at a time. Because the spectrogram only gets a small glimpse of the signal at any one time, the spectrogram would compute different strengths of the frequency



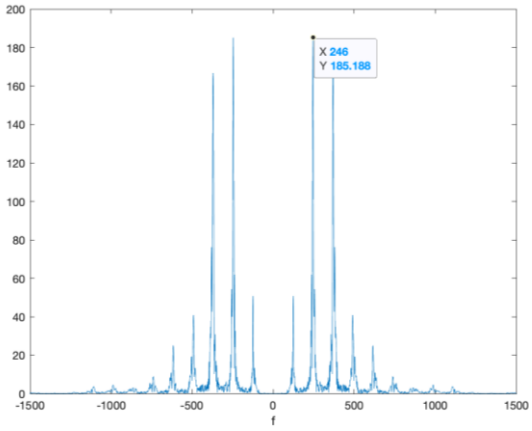
Spectrogram

components than analyzing the entire signal in one short as in part (a). By default, the spectrogram uses a blue-yellow color map where bright yellow corresponds to the frequency component with the highest power, and dark blue corresponds to the frequency component with the lowest power, on a decibel scale. The yellow portions in the time domain match the appearance of the vowel sound utterance in part (d).

(c) *6 points.* From the spectrogram, we can extract the frequency corresponding to the strongest frequency component either manually or automatically. Manually, we can place a "Data Tip" on the spectrogram and move it around horizontally and vertically with arrows on the keyboard. Or we can write MATLAB code to compute it automatically by realizing that the spectrogram is a visualization of a matrix of magnitude values at a given frequency sample and time sample. Using the code below, we find that the maximum magnitude value is 122.5 in linear units. The spectrogram plots power values.

```
spectValues = spectrogram(myRecording, blockSize, overlap, blockSize, fs, 'yaxis');
[maxValue, maxIndex] = max(abs(spectValues), [], 'all', 'linear');
[row,col] = ind2sub(size(spectValues), maxIndex);
```

(d) *6 points.* From the frequency-domain representation in part (a), we can find the gain, frequency, and phase for to the highest peak in positive frequencies. We can use a "Data Tip" in MATLAB to find frequency at the peak as shown on the right, which is 246 Hz.



In UTAudioFreqDomainAnalysis.m script, line 11 computes the exponential Fourier series coefficients using the fast Fourier transform (FFT). Please see Appendix A for the connection between continuous-time and discrete-time Fourier series and the FFT. Here's line 11:

```
fourierSeriesCoeffs = fft(myRecording);
```

The format of the vector follows, keeping in mind that the first element in a vector in MATLAB is at index 1 and that `fourierSeriesCoeffs` has $N = 8000$ discrete-time Fourier series coefficients X_k. We are going to ignore the normalization between discrete-time and continuous-time Fourier series coefficients (see Appendix A) and handle the difference at audio playback using `soundsc` command:

- At index 1, it contains $X_0 = a_0$. Should be real-valued and relatively small in magnitude.
- At index 2, it contains $X_1 = a_1$ for frequency $f_s / N = 1$ Hz.
- At index 3, it contains $X_2 = a_2$ for frequency $2\ f_s / N = 2$ Hz.
- At index 4000, it contains $X_{3999} = a_{3999}$ for frequency $3999\ f_s / N = 3999$ Hz.
- At index 4001, it contains $X_{-4000} = a_{-4000}$ for frequency $-4000\ f_s / N = -4000$ Hz.
- At index 8000, it contains $X_{-1} = a_{-1}$ for frequency $-1\ f_s / N = -1$ Hz.

We extract the exponential Fourier series coefficient for 246 Hz by accessing the 247th element of the vector `fourierSeriesCoeffs`, which is a complex number. The absolute value will give the magnitude and the angle will give us the phase in radians over $[-\pi, \pi]$:

```
abs(fourierSeriesCoeffs(247))
     185.187

angle(fourierSeriesCoeffs(247))
     -0.9878
```

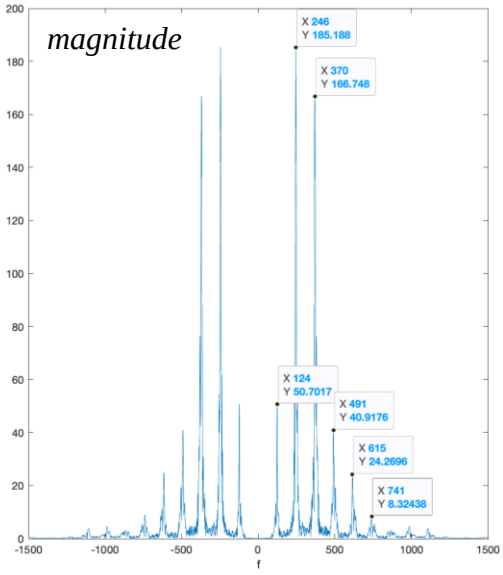
In the UTAudioFreqDomainAnalysis.m script, the `fftshift` command is used to swap the left half and right half of the vector `fourierSeriesCoeffs` for plotting.

4.0 Synthesizing a Vowel Sound

The previous section analyzed a recorded vowel sound (a long e) in the time and frequency domains. In the frequency domain, the plot of the magnitude of the Fourier series coefficients consisted of a several peaks at nearly harmonic frequencies. We will use Fourier synthesis to synthesize the vowel from a finite number of sinusoids:

$$x(t) = A_0 + \sum_{k=1}^{N} A_k \cos\left(2\pi f_k t + \phi_k\right)$$

On the right is a closer look at the magnitude of the frequency domain representation for the *long e* sound recorded in the previous section. The frequencies of the first six peaks are displayed. Next, we try three different ways to synthesize the vowel sound.



(a) *9 points*. In the first synthesis approach, the frequencies f_1, f_2, …, f_N will not be harmonically related. We will use the positive frequencies at the peaks to look up the corresponding Fourier series coefficient value. Per the discussion in Section 3.2(d), the first peak is at 124 Hz, which corresponds to index 125 in the vector `fourierSeriesCoeffs`. Here are the frequencies of the 10 strongest components in the array `abs(fourierSeriesCoeffs)` in descending order: 246, 247, 245, 370, 371, 369, 244, 372, 368, and 248 Hz. We synthesize the vowel sound using the following MATLAB code:

```
peakFreq = [ 246, 247, 245, 370, 371, 369, 244, 372, 368, 248 ];
freqIndices = peakFreq + 1;
numFreq = length(peakFreq);
numSamples = length(myRecording);

%%% Define constant term
A0 = fourierSeriesCoeffs(1);
synthSound = A0*ones(1, numSamples);
Ts = 1/fs;
t = 0 : Ts : 1-Ts;

for k = 1:numFreq
    fk = peakFreq(k);
    freqIndex = freqIndices(k);
    ak = fourierSeriesCoeffs(freqIndex);
    %%% Not including normalization in magnitude between discrete-time and
    %%% continuous-time Fourier series coefficients; use soundsc instead
    Ak = 2*abs(ak);
    Phik = angle(ak);
    synthSound = synthSound + Ak*cos(2*pi*fk*t + Phik);
end
soundsc(synthSound, fs);
```

When using the 10 strongest frequency components, the synthesized sound has a higher volume over the times the vowel sound was uttered, but the synthesized sound is not intelligible as a vowel sound.

(b) *9 points*. The second synthesis approach will be to use harmonically related frequencies. We'll

start with the positive frequencies marked in the plot above and alter them to make them harmonic. The frequencies are 124, 246, 370, 491, 615, 741 Hz, and correspond to f_0 and its first five multiples.

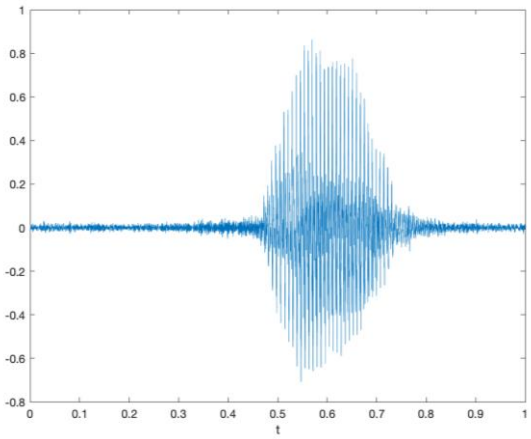
```
freqs = [ 124, 246, 370, 491, 615, 741];
f0est = mean( freqs ./ [1 2 3 4 5 6] );
```

Here, `f0est` is 123.26 Hz. Harmonic frequencies could be 123, 246, 369, 492, 615, and 738 Hz. We could reuse the code in part (a) with the first line changed:

```
peakFreq = [ 123, 246, 369, 492, 615, 738 ];
```

The synthesis sounds like a musical note being played. On the Western scale, these frequencies correspond to harmonics of 'B' in the second octave, which is at 123.47 Hz. [6]

(c) *9 points*. In the third synthesis approach, we used the UTAudioSynthUsingInvFFT.m script in Appendix B to determine the number of positive and negative frequencies needed to synthesize the vowel sound by trial-and-error by changing the value of `Nkeep`. The script keeps the `Nkeep` strongest frequency components. We found that the synthesized vowel sound was intelligible when `Nkeep` = 800, i.e. 400 negative and 400 positive frequencies. Here's the resulting synthesis in the time domain.



Synthesized sound clip
Long e sound

5.0 Conclusion (*10 points*)

The Fourier series is a powerful tool for analyzing the frequency content in one period of a periodic signal in both continuous-time and discrete-time domains. The Fourier series is also a powerful tool for synthesizing a signal from its Fourier series components. In practice, the Fourier synthesis might be incomplete in the continuous-time domain because only a finite number of terms can be used; however, the Fourier series for a discrete-time signal is fully characterized by a finite number of terms.

Fourier series can also be applied to a finite-time interval of a non-periodic signal by assuming the interval represents a fundamental period of a periodic signal. When applying the Fourier series, the Fourier series coefficients would indicate the average magnitude and phase of each harmonic frequency during the finite-time interval. The analysis would not be able to indicate when a frequency occurred. To determine the time a frequency occurs, a spectrogram can be used to break the finite-time interval into smaller, overlapping time intervals on which Fourier series analysis is performed.

References

[1] "Introduction to English Vowel Sounds", accessed Sept. 14, 2021.
[2] J. H. McClellan, R. W. Schafer and M. A. Yoder, *Signal Processing First*, 2003. Errata. Companion Site.
[3] M. Hasegawa-Johnson, "Lab 1", *ECE 498H Signal & Image Analysis*, University of Illinois Urbana-Champaign, Fall 2014.
[4] "Human Voice", *Wikipedia*, accessed Sept. 14, 2021.
[5] R. J. McAulay and T. F. Quatieri, "Speech Analysis/Synthesis Based on a Sinusoidal Representation", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-34, no. 4, August 1986, pp. 744-754.
[6] https://en.wikipedia.org/wiki/Piano_key_frequencies

Appendix A: Connections Between Continuous-Time and Discrete-Time Fourier Series

Continuous-Time Fourier Series. A continuous-time periodic signal $x(t)$ with period T_0 sec is composed of a constant term plus frequency components at integer multiples (harmonic) of a fundamental frequency f_0 where $f_0 = 1 / T_0$. Fourier series analysis computes the constant term a_0 plus the magnitude and phase of each frequency term a_k where k is any integer:

$$x(t) = \sum_{k=-\infty}^{\infty} a_k e^{j2\pi k f_0 t} \quad \text{where} \quad a_0 = \frac{1}{T_0}\int_0^{T_0} x(t)\,dt \quad \text{and} \quad a_k = \frac{1}{T_0}\int_0^{T_0} x(t) e^{-j2\pi k f_0 t} dt$$

An *infinite* number of coefficients could be needed to exactly represent $x(t)$.

Discrete-Time Fourier Series. A discrete-time periodic signal $x[n]$ with period N samples is composed of a constant term plus frequency components at integer multiples (harmonic) of a fundamental frequency of $2\pi/N$ rad/sample which is equivalent to f_s / N in Hz. Fourier series analysis computes the constant term $X[0]$ plus the magnitude and phase of each frequency term $X[k]$ for $k = 0, 1, \ldots, N\text{-}1$.

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]\, e^{j\left(k\frac{2\pi}{N}\right)n} \quad \text{where} \quad X[0] = \sum_{n=0}^{N-1} x[n] \quad \text{and} \quad X[k] = \sum_{n=0}^{N-1} x[n]\, e^{-j\left(k\frac{2\pi}{N}\right)n}$$

A *finite* number of Fourier series coefficients would always be needed to exactly represent $x[n]$.

Normalization. The scaling of the Fourier series coefficients is different in continuous-time vs. discrete-time. When using the discrete-time Fourier series to compute continuous-time Fourier series coefficients, we would have to divide the discrete-time Fourier series coefficient by N due the $(1/N)$ term in the equation for $x[n]$.

Fast Fourier Transform (FFT) is a fast algorithm to compute the N discrete-time Fourier series coefficients $X[k]$ from the N samples of a discrete-time signal $x[n]$. As with the continuous-time Fourier series, the discrete-time Fourier series assumes that the N samples of $x[n]$ represents the fundamental period of an infinitely long signal in the time domain. Unlike the continuous-time Fourier series, the discrete-time Fourier series always has a finite number of terms, N. One of the reasons for this is due to the Sampling Theorem, which says that the sampling rate $f_s > 2\, f_{max}$ where f_{max} is the highest frequency of interest and hence $f_{max} < ½\, f_s$. Sampling only captures continuous-time frequencies ($-½\, f_s$, $½\, f_s$) whereas continuous-time signals have frequencies. You can think of the discrete-time Fourier series as computing the Fourier series coefficients for frequency components from $-½\, f_s$ to $-½\, f_s$, which is a finite number because $f_s > 0$.

The Fast Fourier Transform (FFT) is requires $2M \log_2 M$ real-valued multiplications and additions and $4M$ words of memory instead of the $4\, M^2$ and $M^2 + 4M$, respectively, for the direct matrix-vector implementation of the discrete-time Fourier series. The direct matrix-vector implementation to compute $X[k]$ would create a complex-valued N x N matrix of the term $\exp(-j\,(2\pi/N)\,kn)$ for $k = 0, 1, \ldots, N\text{-}1$ in one dimension and $n = 0, 1, \ldots, N\text{-}1$ in the other, form a column vector of $x[0], x[1], \ldots, x[N\text{-}1]$, and multiply the matrix and vector to find the column vector of $X[0], X[1], \ldots, X[N\text{-}1]$.

Appendix B: MATLAB Scripts

```
% UTAudioRecordAndPlayback.m

% Record from the microphone
fs = 8000;
numBits = 16;
numChannels = 1;
recordingTime = 1;
recObj = audiorecorder(fs, numBits, numChannels);
disp('Start recording...');
recordblocking(recObj, recordingTime);
disp('End recording.');

% Store data in double-precision floating-point array
myRecording = getaudiodata(recObj);

% Play back the recording with automatic scaling
soundsc(myRecording, fs);

% Remove DC value and normalize amplitude to [-1, 1]
myRecording = myRecording - mean(myRecording);
myRecording = myRecording / max(abs(myRecording));

% Save the data to a file
waveFilename = 'shortAudioClip.wav';
audiowrite(waveFilename, myRecording, fs );
```

```
% UTAudioTimeDomainAnalysis.m

% Read the contents of the audio file
waveFilename = 'shortAudioClip.wav';
[myRecording, fs] = audioread(waveFilename);

% Play back the recording with automatic scaling
soundsc(myRecording, fs);

% Plot the waveform in the time domain
N = length(myRecording);
Ts = 1/fs;
t = 0 : Ts : (N-1)*Ts;
figure;
plot(t, myRecording);
xlabel('t');
```

```
% UTAudioFreqDomainAnalysis.m

% Read the contents of the audio file
waveFilename = 'shortAudioClip.wav';
[myRecording, fs] = audioread(waveFilename);

% Plot the magnitude of the frequency content
% using a discrete-time version of the Fourier series
% Zoom the frequency axis to [-1500 Hz, 1500 Hz]
% to focus on the strongest frequency components
fourierSeriesCoeffs = fft(myRecording);
N = length(myRecording);
freqResolution = fs / N;
ff = (-fs/2) : freqResolution : (fs/2)-freqResolution;
figure;
plot(ff, abs(fftshift(fourierSeriesCoeffs)));
xlabel('f');
xlim( [-1500, 1500] );

% Plot the spectrogram
figure;
blockSize = round(N/4);
overlap = round(0.875 * blockSize);
spectrogram(myRecording, blockSize, overlap, blockSize, fs, 'yaxis');
```

```
% UTAudioSynthUsingInvFFT.m
% Needs fourierSeriesCoeffs vector from UTAudioFreqDomainAnalysis.m

Nseries = length(fourierSeriesCoeffs);
fourierSeriesCoeffsAbs = abs(fourierSeriesCoeffs);

Nkeep = 10;
synthSoundCoeffs = zeros(Nseries, 1);

% Find the Nkeep strongest positive and negative frequency components
for n = 1:Nkeep
    [ak, k] = max(fourierSeriesCoeffsAbs);
    synthSoundCoeffs(k) = fourierSeriesCoeffs(k);
    fourierSeriesCoeffsAbs(k) = 0;
end

% Convert Fourier series coefficients to time domain using inverse FFT
synthSound = ifft(synthSoundCoeffs);
soundsc(synthSound, fs);
```