Mini-Project #1: Music Synthesis

Prof. Brian L. Evans

Version 3.0 November 25, 2023

1.0 Introduction (*3 points*)

This mini-project uses a sum of sinusoids to synthesize a recorded musical note played on a violin. A musical note consists of a principal frequency of the note plus its harmonics. Our goal will be to use a sum of sinusoids to analyze and synthesize a recorded note played by a musical instrument. In the sum of sinusoids, we will use frequencies that are harmonics of a fundamental frequency, and some of those will be close in value to the principal frequency of the note being played and its harmonics.

2.0 Overview (*5 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:

$$x(t) = \sum_{k=-\infty}^{\infty} a_k e^{j 2\pi k f_0 t} \quad \text{where} \quad a_0 = \frac{1}{T_0}\int_0^{T_0} x(t)\, dt \quad \text{and} \quad a_k = \frac{1}{T_0}\int_0^{T_0} x(t) e^{-j 2\pi k f_0 t} dt$$

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.

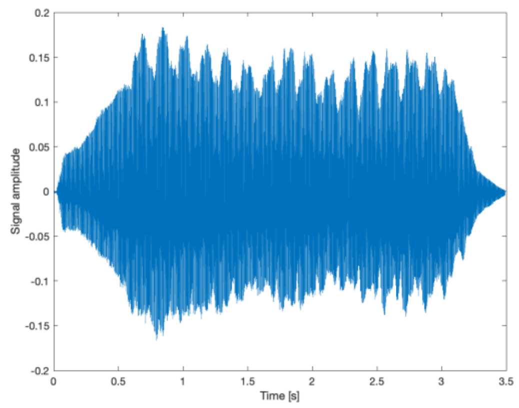
3.0Analyzing a Music Recording

There are many ways to create a digital audio signal, including recording from a microphone, reading from a file, and generating sound from a formula. In this section, we'll download a recording of an acoustical instrument, play the recording, and analyze the recording in the time and frequency domains.

3.1 Download the Recording

We analyzed a recording of a violin playing 'C' in the fourth octave ('C4') on the Western scale which has a principal frequency of 261.63 Hz. The recording is from a collection of single notes of different principal frequencies played by different acoustic instruments from [Prof. Dan Ellis](#) at Columbia University and sampled at 11025 Hz. We will analyze the recording in 'violin-C4.wav'.

3.2 Time-Domain Analysis



We can now analyze the musical note in the time domain.

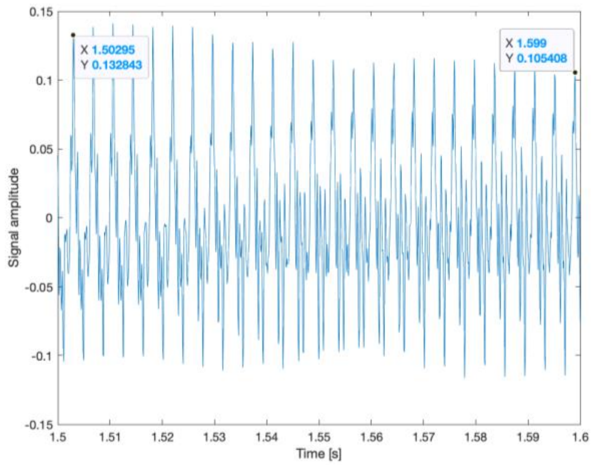
(a) *6 points.* A time-domain plot of the musical note is generated in Matlab and shown on the right. The note lasts for 3.491s.

```
% Read the contents of the audio file
waveFilename = 'violin-C4.wav';
[instrumentSound, fs] = audioread(waveFilename);

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

% Plot the waveform in the time domain
N = length(instrumentSound);
Ts = 1/fs;
Tmax = (N-1)*Ts;
t = 0 : Ts : Tmax;
figure;
plot(t, instrumentSound);
xlabel('Time [s]');
ylabel('Signal amplitude');
```

(b) *6 points.* We computed the average value of the signal using MATLAB:

```
mean(instrumentSound)
  -0.0016
```

The average value (DC component) is close to 0 relative to the amplitudes of the rest of the audio signal. It is common to remove the average value in speech and audio signals because the average value would occupy bits in the sampled data but cannot be perceived by the human auditory system. Moreover, playback systems typically pass frequencies within the range of human hearing of 20 Hz to 20 kHz.



(c) *6 points.* We estimated the principal frequency of the note being played from the time-domain plot. First, we zoomed into the time-domain plot from 1.5s to 1.6s using the MATLAB command

```
xlim( [1.5 1.6] );
```

To estimate the principal frequency, we counted the number of periods, estimated the fundamental period by dividing the duration of time by the number of periods, and inverted the estimate of the fundamental period.

i. See the zoomed plot from 1.5s to 1.6s on the right. We used the Data Tips Tool in MATLAB to click on the start of the first period at 1.50295s and end of the last period 1.599s to estimate the fundamental period. 25 periods are in the above plot.
ii. We compared the estimated and actual principal frequency of a 'C4' note using the following

MATLAB code

```
fundPeriod = (1.599 - 1.50295) / 25
fundFrequency = 1 / fundPeriod
C4fundFrequency = 261.63
fundFrequencyRelErr = (C4fundFrequency - fundFrequency) / C4fundFrequency
```

which yielded the following answers to five significant digits:

```
fundPeriod = 0.0038420
fundFrequency = 260.28
C4fundFrequency = 261.63
fundFrequencyRelErr = 0.0051557
```

The estimated fundamental frequency was 260.28 Hz and the actual fundamental frequency for a 'C4' note is 261.63 Hz, which is a relative error of 0.52%.

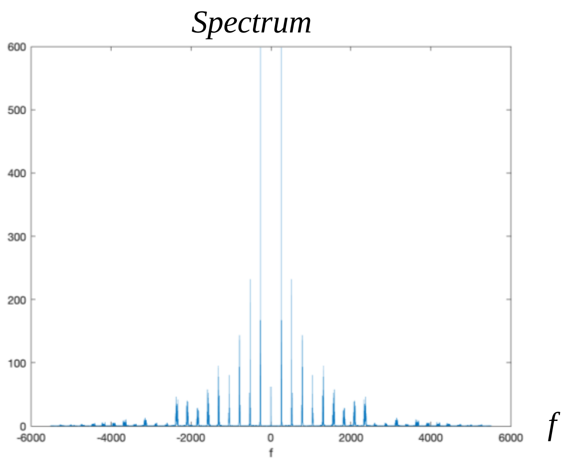
The fundamental frequency in the audio signal corresponds to the fundamental frequency of the string of the violin being played. On a violin, the string is attached on both ends, and the violinist places a finger on the string to shorten its length to play a certain note. When the note is played, the string produces standing waves at the frequency of the note and its harmonics. The vibration in the string causes the instrument to vibrate which produces the acoustic waves that we hear.

(d) *6 points*. The sampling rate of the original recording in the The McGill University Master Samples collection was 44100 Hz and the sampling rate of recording in the file 'violin-C4.wav' is 11025 Hz. The ratio between these sampling rates is 4:1.

3.3 Frequency-Domain Analysis

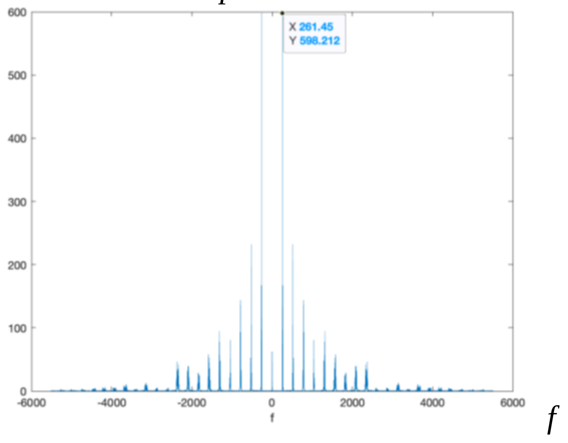
We can now analyze the recorded instrument in the frequency and time-frequency domains. For the frequency domain analysis, we'll use the Fast Fourier Transform (FFT) algorithm to compute the Fourier series. See Appendix A for the connection between the FFT and the Fourier series.

(a) *6 points*. We plot 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 11025 Hz, we capture frequencies in (-5512.5 Hz, 5512.5 Hz) because of the sampling theorem $f_s > 2\, f_{\max}$ where $f_{\max}$ is the highest frequency of interest and hence $f_{\max} < \frac{1}{2}\, f_s$. The computation uses the discrete-time Fourier series— see Appendix A for the connection to the continuous-time Fourier series.



```
% Read the contents of the audio file
waveFilename = 'violin-C4.wav';
[instrumentSound, fs] = audioread(waveFilename);

% Plot the magnitude of the frequency content
% using the discrete-time Fourier series
fourierSeriesCoeffs = fft(instrumentSound);
N = length(instrumentSound);
freqResolution = fs / N;
ff = (-fs/2) : freqResolution : (fs/2)-freqResolution;
figure;
plot(ff, abs(fftshift(fourierSeriesCoeffs)));
xlabel('f');
```

Spectrum



(b) *6 points*. Using the frequency-domain representation in part (a), we found the gain, frequency, and phase for 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 261.45 Hz, determine the corresponding index into the Fourier series coefficients vector, and compute the gain and phase from the Fourier series coefficient value.

Keep in mind that the first element in a vector in MATLAB is at index 1 and `fourierSeriesCoeffs` has N = 38500 discrete-time Fourier series coefficients X_k. We are going to ignore normalization between discrete-time and continuous-time Fourier series coefficients (see Appendix A) and handle the difference at audio playback using the `soundsc` command. For the `fourierSeriesCoeffs` vector,

- 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 = 0.286363… Hz.
- At index 3, it contains $X_2 = a_2$ for frequency 2 f_s / N = 0.572727… Hz.
- At index 19250, it contains $X_{19249} = a_{19249}$ for frequency 19249 f_s / N = 5512.214 Hz.
- At index 19251, it contains $X_{-19250} = a_{-19250}$ for frequency -19250 f_s / N = -5512.5 Hz.
- At index 38500, it contains $X_{-1} = a_{-1}$ for frequency -1 f_s / N = -0.286363… Hz.

We can convert a non-negative frequency to an index into an N-length FFT vector by

```
N = length(instrumentSound);
nonNegativeFrequency = 261.45;
f0 = fs / N;
fftIndex = round(nonNegativeFrequency / f0) + 1;   %%% Value is 914
```

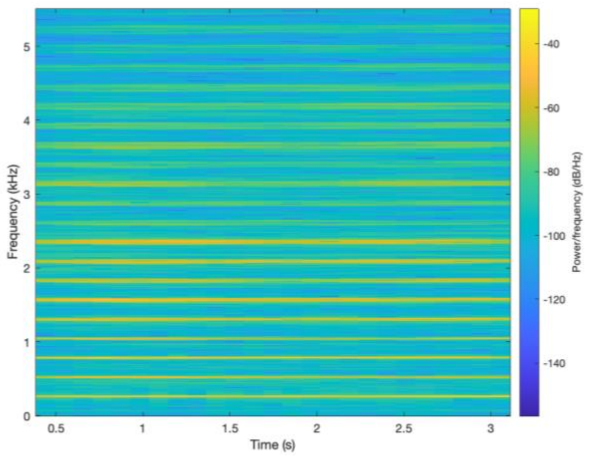
We extract the exponential Fourier series coefficient for 261.45 Hz by accessing the 914th element of the vector `fourierSeriesCoeffs`, which is a complex number. The absolute value will give the magnitude and the angle will give us the phase in radians over [-π, π]:

```
abs(fourierSeriesCoeffs(fftIndex))
598.21

angle(fourierSeriesCoeffs(fftIndex))
0.55884
```

Alternately, we automate finding gain, frequency, and phase for the highest peak in positive frequencies using

```
[maxval, maxind] = max(abs(fourierSeriesCoeffs));
gainValue = abs(fourierSeriesCoeffs(maxind))
phaseValue = angle(fourierSeriesCoeffs(maxind))
freqValue = (maxind - 1)*fs/N
```

Spectrogram



(c) 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 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 frequency values for the yellow lines match the frequency values where the peaks occur in the spectrum plot in part (a).

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

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 value of frequency and delay in time. Using the code below, we find that the maximum value is 164.4 in linear units. The spectrogram plots power values.

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

(d) 6 *points*. From the spectrogram, we see that the principal frequency of 261.45 Hz is present for the duration of the signal. This is the horizontal yellow line closest to zero frequency.

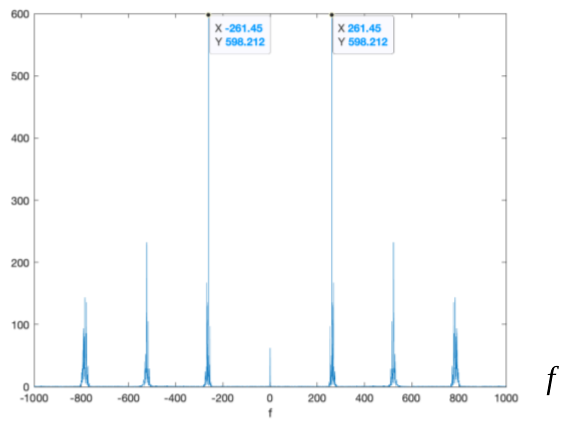
4.0 Synthesizing the Sound

The previous section analyzed in the time and frequency domain a recording of a ‘C4’ note on the Western scale played by violin. In this section, we use the Fourier series analysis to synthesize the sound using a small number of sinusoids.

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

Spectrum

(a) *12 points*. In our first algorithm, we will manually pick peaks in the spectrum using the Data Tips tool. On the right, the spectrum is zoomed to -1000 Hz < f < 1000 Hz and the Data Tips tool highlights the pair of the largest peaks as a starting point. We would need to find where the frequencies of 261.45 Hz and -261.45 Hz fall in `fourierSeriesCoeffs` vector.



Per Section 3(b), we convert a positive frequency f_k = 261.45 Hz via $f_k = k\frac{f_s}{N}$ which means $k = N\frac{f_k}{f_s} = 38500\ \frac{261.45\text{ Hz}}{11025\text{ Hz}} =$ 913. Since MATLAB vectors start at index 1, k = 913 means an index of 914 into the `fourierSeriesCoeffs` vector.

We convert a negative frequency f_k = -261.45 Hz via $f_k = f_s\left(\frac{k}{N} - 1\right)$ which means $k = N\left(\frac{f_k}{f_s} + 1\right) = 38500\ \left(\frac{-261.45\text{ Hz}}{11025\text{ Hz}} + 1\right) = 37587$, which is index 37588.

i. Manually synthesize the audio signal by using the two largest peaks and then the four largest peaks.

The two largest peaks are at 261.45 and -261.45 Hz which correspond to indices 914 and 37588. The next two largest peaks are at 522.9 and -522.9 Hz which correspond to indices 1827 and 36675.

```
synthFourierSeriesCoeffs = zeros(N, 1);

synthFourierSeriesCoeffs(914) = fourierSeriesCoeffs(914);
synthFourierSeriesCoeffs(37588) = fourierSeriesCoeffs(37588);
synthTwoPeaks = ifft(synthFourierSeriesCoeffs);
soundsc(synthTwoPeaks, fs);
pause(4);

synthFourierSeriesCoeffs(1827) = fourierSeriesCoeffs(1827);
synthFourierSeriesCoeffs(36675) = fourierSeriesCoeffs(36675);
synthFourPeaks = ifft(synthFourierSeriesCoeffs);
soundsc(synthFourPeaks, fs);
pause(4);
```

ii. Describe what each synthesized signal sounds like.

Synthesizing the sound with two peak frequencies sounds like a pure tone. In the synthesized sound with four peak frequencies, I hear beat frequencies.

(b) *12 points*. We can automate how many of the largest peaks to keep by using the code below which keeps the `Nkeep` strongest frequency components:

```
% Needs fourierSeriesCoeffs vector computed in Section 2.3 above
N = length(fourierSeriesCoeffs);
fourierSeriesCoeffsAbs = abs(fourierSeriesCoeffs);

% Nkeep must be even to have an equal number of negative and positive freq.
Nkeep = 10;
frequenciesKept = zeros(Nkeep, 1);

% Find the Nkeep strongest positive and negative frequency components
synthSoundCoeffs = zeros(N, 1);
```

```
for n = 1:Nkeep
    [ai, i] = max(fourierSeriesCoeffsAbs);
    synthSoundCoeffs(i) = fourierSeriesCoeffs(i);
    fourierSeriesCoeffsAbs(i) = 0;

    k = i - 1;
    if ( k < N/2 )
       frequenciesKept(n) = fs*k/N;
    else
        frequenciesKept(n) = fs*((k/N) - 1);
    end
end

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

i. Keep 2, 4, 6, and 8 of the largest peaks and describe what you hear.

2 peaks – sounds like a single tone at the same volume
4 peaks – when compared to using two peaks, sounds like a single tone at a higher frequency and with an increase in volume at the very beginning and a decrease in volume at the very end.
6 peaks – sounds like a single tone, but with an increase in volume at the very beginning and a decrease in volume at the very end.
8 peaks – sounds the same as when using six peaks.

ii. Give the peak frequencies of the peaks for each case of keeping 2, 4, 6, and 8 of largest peaks.

2 peaks – frequencies 261.45 and -261.45 Hz
4 peaks – frequencies 261.74 and -261.74 Hz plus above
6 peaks – frequencies 262.02 and -262.02 Hz plus above
8 peaks – frequencies 261.16 and -261.16 Hz plus above

(c) *12 points*. The algorithm in part (b) will keep the peaks at -261.45 Hz and 261.45 Hz, then keep the second highest peaks at -261.74 and 261.74 Hz, and so forth. In the human auditory system, the components at -261.74 and 261.74 Hz will be “masked” (i.e. not audible) by the stronger nearby components at -261.45 and 261.45 Hz, respectively. [2] To prevent masked frequencies from being kept, we have changed

```
fourierSeriesCoeffsAbs(k) = 0;
```

in the above code to

```
fourierSeriesCoeffsAbs(kmin:kmax) = 0;
```

to zero out all frequencies from the peak frequency minus 7.5 Hz to the peak frequency plus 7.5 Hz. The frequency masking range is chosen arbitrarily to match the bandwidth of a C4 note.

i. Give the code to compute `kmin` and `kmax`. In finding `kmin` and `kmax`, please note that the frequency resolution for the Fast Fourier Transform is f_s/N.

```
% Needs fourierSeriesCoeffs vector computed in Section 2.3 above
N = length(fourierSeriesCoeffs);
```

```
fourierSeriesCoeffsAbs = abs(fourierSeriesCoeffs);

% Nkeep must be even to have an equal number of negative and positive freq.
Nkeep = 10;
frequenciesKept = zeros(Nkeep, 1);

% Frequency masking +/- 7.5 Hz around a center frequency
freqOffset = 7.5;
indexOffset = round(freqOffset*N/fs);

% Find the Nkeep strongest positive and negative frequency components
synthSoundCoeffs = zeros(N, 1);
for n = 1:Nkeep
    [ai, i] = max(fourierSeriesCoeffsAbs);
    synthSoundCoeffs(i) = fourierSeriesCoeffs(i);

    % Frequency masking +/- 7.5 Hz around a center frequency
    % Keep indexing in fourierSeriesCoeffsAbs in range of [1, N]
    imin = max(i - indexOffset, 1);
    imax = min(i + indexOffset, N);
    fourierSeriesCoeffsAbs(imin:imax) = 0;

    k = i - 1;
    if ( k < N/2 )
       frequenciesKept(n) = fs*k/N;
    else
        frequenciesKept(n) = fs*((k/N) - 1);
    end
end

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

ii. Keep 2, 4, 6, and 8 of the largest peaks and describe what you hear.

2 peaks – synthesized sound sounds like a single tone at the same volume
4 peaks – when compared to using two peaks, synthesized sound sounds to be at higher frequency and with a little bit of vibrato
6 peaks – when compared to using four peaks sounds, sound is similar
8 peaks – when compared to using six peaks, synthesized sound sounds like beat frequencies

iii. Give the peak frequencies of the peaks for each case of keeping 2, 4, 6, and 8 of largest peaks.

2 peaks – frequencies	261.45	and	-261.45 Hz	
4 peaks – frequencies	522.90	and	-522.90 Hz	plus above
6 peaks – frequencies	784.35	and	-784.35 Hz	plus above
8 peaks – frequencies	1314.12	and	-1314.12 Hz	plus above

5.0 Conclusion (*8 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] James H. McClellan, Ronald W. Schafer & Mark A. Yoder, *Signal Processing First*, Prentice-Hall, ISBN 978-0130909992, 2003. Errata. On-line Companion.

[2] "Auditory Masking", Wikipedia, Accessed Sep. 12, 2023

[3] 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

```
% Section 3.2(a)
% Read the contents of the audio file
waveFilename = 'violin-C4.wav';
[instrumentSound, fs] = audioread(waveFilename);

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

% Plot the waveform in the time domain
N = length(instrumentSound);
Ts = 1/fs;
Tmax = (N-1)*Ts;
t = 0 : Ts : Tmax;
figure;
plot(t, instrumentSound);
xlabel('Time [s]');
ylabel('Signal amplitude');

% Section 3.2(b)
mean(instrumentSound)

% Section 3.2(c)
xlim( [1.5 1.6] );

fundPeriod = (1.599 - 1.50295) / 25
fundFrequency = 1 / fundPeriod
C4fundFrequency = 261.63
fundFrequencyRelErr = (C4fundFrequency - fundFrequency) / C4fundFrequency

% Section 3.3(a)
% Plot the magnitude of the frequency content
% using the discrete-time Fourier series
fourierSeriesCoeffs = fft(instrumentSound);
N = length(instrumentSound);
freqResolution = fs / N;
ff = (-fs/2) : freqResolution : (fs/2)-freqResolution;
figure;
plot(ff, abs(fftshift(fourierSeriesCoeffs)));
xlabel('f');

% Section 3.3(b)
N = length(instrumentSound);
nonNegativeFrequency = 261.45;
f0 = fs / N;
fftIndex = round(nonNegativeFrequency / f0) + 1;   %%% Value is 914

abs(fourierSeriesCoeffs(fftIndex))
angle(fourierSeriesCoeffs(fftIndex))

[maxval, maxind] = max(abs(fourierSeriesCoeffs));
gainValue = abs(fourierSeriesCoeffs(maxind))
phaseValue = angle(fourierSeriesCoeffs(maxind))
freqValue = (maxind - 1)*fs/N
```

```
% Section 3.3(c)
% Plot the spectrogram
figure;
blockSize = round(N/4);
overlap = round(0.875 * blockSize);
spectrogram(instrumentSound, blockSize, overlap, blockSize, fs, 'yaxis');

powVal = spectrogram(instrumentSound, blockSize, overlap, blockSize, fs, 'yaxis');
[maxValue, maxIndex] = max(abs(powVal), [], 'all', 'linear');
[row, col] = ind2sub(size(powVal), maxIndex);
maxPowVal = abs(powVal(row, col));

% Section 4.0(a)

synthFourierSeriesCoeffs = zeros(N, 1);

synthFourierSeriesCoeffs(914) = fourierSeriesCoeffs(914);
synthFourierSeriesCoeffs(37588) = fourierSeriesCoeffs(37588);
synthTwoPeaks = ifft(synthFourierSeriesCoeffs);
soundsc(synthTwoPeaks, fs);
pause(4);

synthFourierSeriesCoeffs(1827) = fourierSeriesCoeffs(1827);
synthFourierSeriesCoeffs(36675) = fourierSeriesCoeffs(36675);
synthFourPeaks = ifft(synthFourierSeriesCoeffs);
soundsc(synthFourPeaks, fs);
pause(4);

% Section 4.0(b)
% Needs fourierSeriesCoeffs vector computed in Section 2.3 above
N = length(fourierSeriesCoeffs);
fourierSeriesCoeffsAbs = abs(fourierSeriesCoeffs);

% Nkeep must be even to have an equal number of negative and positive freq.
Nkeep = 10;
frequenciesKept = zeros(Nkeep, 1);

% Find the Nkeep strongest positive and negative frequency components
synthSoundCoeffs = zeros(N, 1);
for n = 1:Nkeep
    [ai, i] = max(fourierSeriesCoeffsAbs);
    synthSoundCoeffs(i) = fourierSeriesCoeffs(i);
    fourierSeriesCoeffsAbs(i) = 0;

    k = i - 1;
    if ( k < N/2 )
       frequenciesKept(n) = fs*k/N;
    else
        frequenciesKept(n) = fs*((k/N) - 1);
    end
end

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

```
soundsc(synthSound, fs);
frequenciesKept

% Section 4.0(c)
% Needs fourierSeriesCoeffs vector computed in Section 2.3 above
N = length(fourierSeriesCoeffs);
fourierSeriesCoeffsAbs = abs(fourierSeriesCoeffs);

% Nkeep must be even to have an equal number of negative and positive freq.
Nkeep = 10;
frequenciesKept = zeros(Nkeep, 1);

% Frequency masking +/- 7.5 Hz around a center frequency
freqOffset = 7.5;
indexOffset = round(freqOffset*N/fs);

% Find the Nkeep strongest positive and negative frequency components
synthSoundCoeffs = zeros(N, 1);
for n = 1:Nkeep
    [ai, i] = max(fourierSeriesCoeffsAbs);
    synthSoundCoeffs(i) = fourierSeriesCoeffs(i);

    % Frequency masking +/- 7.5 Hz around a center frequency
    % Keep indexing in fourierSeriesCoeffsAbs in range of [1, N]
    imin = max(i - indexOffset, 1);
    imax = min(i + indexOffset, N);
    fourierSeriesCoeffsAbs(imin:imax) = 0;

    k = i - 1;
    if ( k < N/2 )
       frequenciesKept(n) = fs*k/N;
    else
        frequenciesKept(n) = fs*((k/N) - 1);
    end
end

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