Mini Project #1: Audio↔Image Signal Synthesis using a Time-Frequency Representation

Mr. Dan Jacobellis and Prof. Brian Evans

Solution Version 0.1

1.0 Introduction

In this project, we design and implement a system, $\mathcal{T}$, and its inverse, $\mathcal{T}^{-1}$, that converts a discrete-time audio signal $A[n]$ to a discrete space image signal $I[m,k]$ and vice versa using a time-frequency transformation.

$$\underbrace{I[m,k]}_{\substack{\text{visual}\\\text{representation}}} = \mathcal{T}\,\underbrace{\{A[n]\}}_{\substack{\text{audio}\\\text{signal}}}$$

$\mathcal{T}^{-1}$ is the approximate inverse. [1]

$$\mathcal{T}^{-1}\{I[m,k]\} = \mathcal{T}^{-1}\big\{\mathcal{T}\{A[n]\}\big\} \approx A[n]$$

$\mathcal{T}$ consists of three components:

$$\mathcal{T}\{A[n]\} = \underbrace{\mathcal{C}}_{\substack{\text{color}\\\text{mapping}}} \circ \underbrace{F}_{\text{companding}} \circ \underbrace{\textbf{STFT}\{A[n]\}}_{\substack{\text{time-frequency}\\\text{transformation}}}$$

1. The discrete short-time Fourier transform (STFT), which is a variant of the Fourier series

$$S[m,k] = \textbf{STFT}\{x[n]\}[m,k] = \sum_{n=0}^{N-1} x[n]w[n-m]e^{-j2\pi\frac{k}{N}n}$$

   Unlike the standard Fourier series, the STFT includes a window function $w[n-m]$ that isolates the part of the signal centered around m. For this project, we will use a rectangular window with $N = 448$ samples.

$$w[n] = \text{rect}_{N=448}[n] = \begin{cases} 1 & 0 \le n < 448 \\ 0 & \text{otherwise} \end{cases}$$

2. A companding operation F that reduces the STFT range so that it can be displayed as an image:

$$F(x) = \text{sign}(x)\left((|x| + \epsilon)^p - \epsilon^p\right)$$

3. A color mapping $\mathcal{C}$ that allows a complex-valued number to be represented by a color.

$$(\text{red intensity}, \text{green intensity}, \text{blue intensity}) = \mathcal{C}(\text{Re}\{x\} + j\ \text{Im}\{x\})$$

2.0 Loading an audio signal

```
%% load the audio signal and convert from stereo 44.1 kHz to mono 16 kHz
audio_vector = mean(audioread('sm_cello.mp3'),2);
audio_vector = resample(audio_vector,160,441);
audio_vector = audio_vector(1:448000);
```

[1] Finite precision and dynamic range of the image signal will result in some information loss, so the recovered audio signal will not be exactly the same as the original signal.

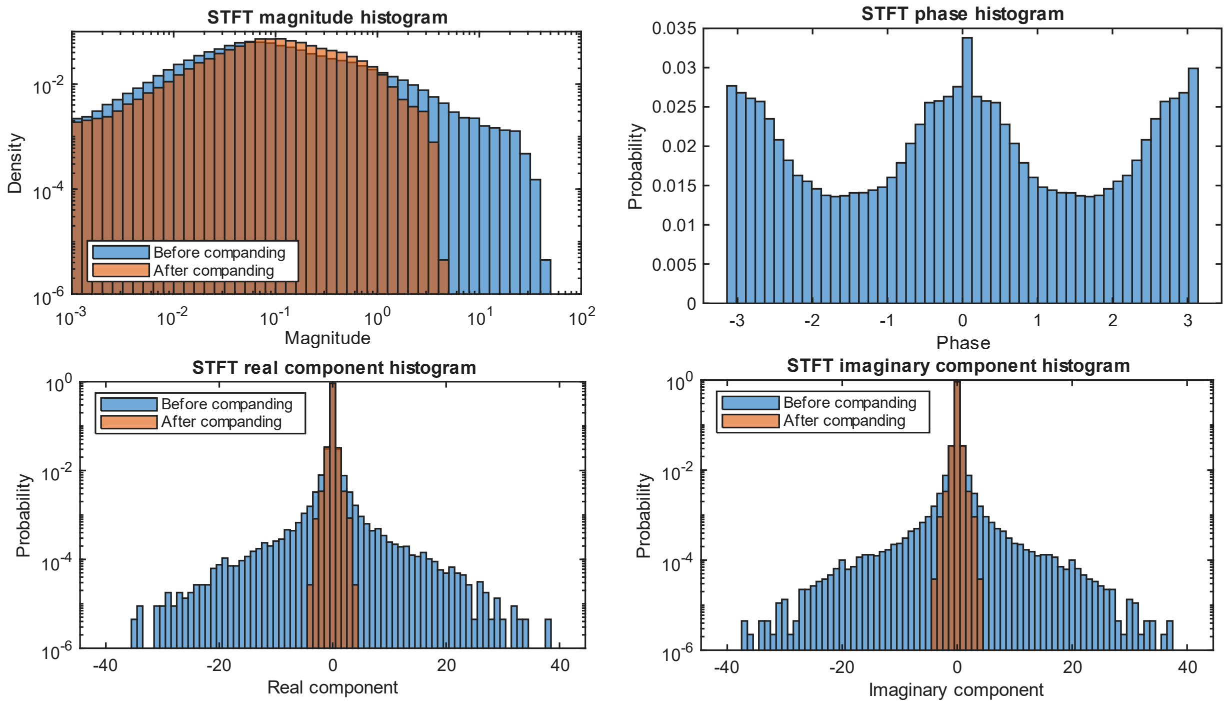
2.1 Computing the STFT and inverse STFT

```
%% Fourier series of each 448-sample window becomes a matrix column (STFT)
audio_matrix = reshape(audio_vector, 448, 1000);
STFT_matrix = fftshift(fft(audio_matrix));
```

2.3 Companding

The histogram of S_{mag} shows a one-sided (positive only) long-tail, similar to an exponential distribution. The values of S_{mag} are highly concentrated between zero and one, but the tail extends to much larger values (nearly 100). S_{real}, and S_{imag} are similar, except they are two sided, similar to a Laplacian distribution. The distribution of S_{phase} is roughly uniform between $-\pi$ and π.

After companding, S_{mag} becomes more concentrated near zero, with the tail extending only to about four. The range of the real and imaginary STFT components are also reduced after companding.



2.4 Mapping complex values to a color image

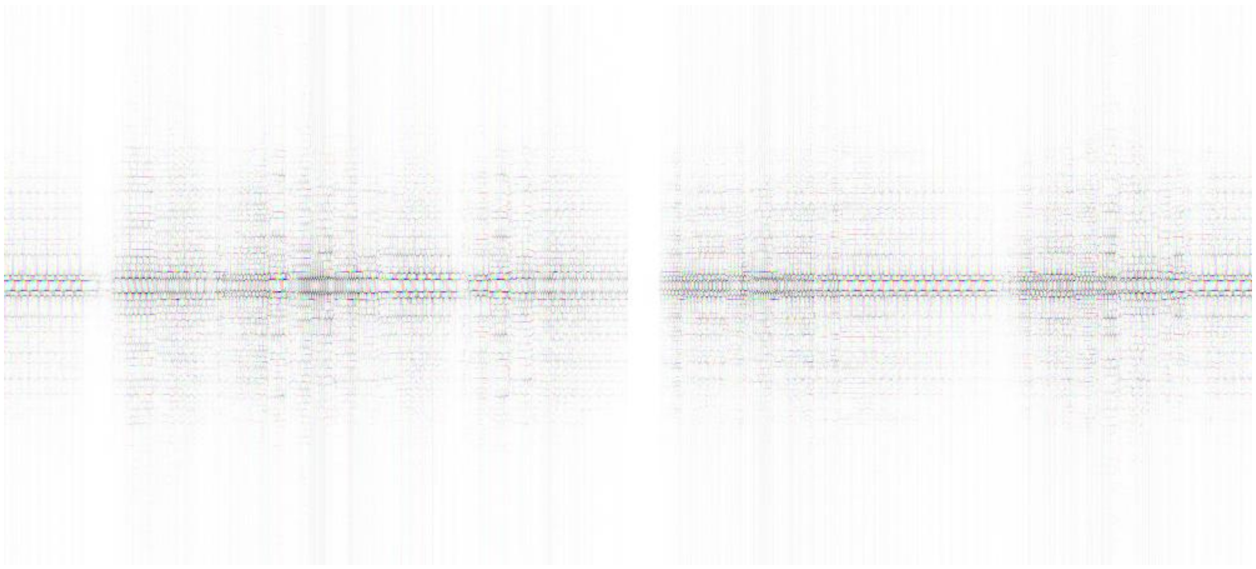
In a magnitude spectrogram, the brightness is used to represent the STFT magnitude values, and the phase is unused. By incorporating variations in hue, a similar effect can be achieved while keeping the phase information. We will use hue-saturation-value (HSV) as an intermediate representation in the mapping from complex values to RGB colors. Based on the ranges determined from the above histogram, we scale and shift the companded magnitude and phase components so that the HSV values lie in [0,1].

$$\text{Hue}[m,k] = \frac{1}{4}F(|S[m,k]|), \qquad \text{Saturation}[m,k] = \frac{1}{2} + \angle\frac{S[m,k]}{2\pi}, \qquad \text{Value}[m,k] = 1$$

```
%% Compand, scale, then map magnitude->value and phase->hue in HSV colorspace
hsv_hue = 0.5 + angle(STFT_matrix)/(2*pi);
hsv_saturation = 0.25*F(abs(STFT_matrix));
hsv_value = ones(size(hsv_hue));
```

Finally, we can convert from HSV to red, green, blue (RGB), which is the format expected by MATLAB's `imwrite` function. We use a lossless format (PNG) to store the image.

```
%% Convert from HSV to RGB and save using a lossless image format (PNG)
RGB = hsv2rgb(cat(3, hsv_hue, hsv_saturation, hsv_value));
imwrite(RGB,'mag_phase.png');
```



3.0 Recovering audio from the time-frequency image.

To recover the audio from the image, we must apply the inverse of each component of $\mathcal{T}$:

$$I[m,k] = \mathcal{T}\{A[n]\} = \underbrace{\mathcal{C}}_{\substack{\text{color}\\\text{mapping}}} \circ \underbrace{F}_{\text{companding}} \circ \underbrace{\mathbf{STFT}\{A[n]\}}_{\substack{\text{time-frequency}\\\text{transformation}}}$$

$$\hat{A}[n] = \mathcal{T}^{-1}\{I[m,k]\} = \underbrace{\mathbf{STFT}^{-1}}_{\substack{\text{inverse time-frequency}\\\text{transformation}}} \circ \underbrace{F^{-1}}_{\substack{\text{inverse}\\\text{companding}}} \circ \underbrace{\mathcal{C}^{-1}}_{\substack{\text{inverse color}\\\text{mapping}}}$$

The inverse color mapping consists of conversion from RGB to HSV, scaling, and shifting. After decompanding, the complex values can be reconstituted from their magnitude and phase components.

```
%% Load the image and convert from RGB to HSV
RGB = im2double(imread('mag_phase.png'));
HSV = rgb2hsv(RGB); hsv_hue = HSV(:,:,1); hsv_saturation = HSV(:,:,2);
```

```
%% Inverse  decompanding and reconstituting magnitude and phase components
STFT_magnitude = Finv(4*hsv_saturation);
STFT_phase = 2*pi*(hsv_hue - 0.5);
STFT_matrix = STFT_magnitude .* exp(1j*STFT_phase);
```

The full process of recovering the audio results in information loss. Although the original audio signal $A[n]$ was real-valued, the recovered audio $\hat{A}[n] = \{\mathcal{T}^{-1}\{\mathcal{T}\{A[n]\}\}\}$ may be complex valued. In order to play back the recovered audio and compute the PSNR, we will discard the imaginary part.

```
%% Inverse STFT and discarding imaginary part of recovered audio
recovered_audio_matrix = real(ifft(ifftshift(STFT_matrix)));
recovered_audio = recovered_audio_matrix(:);
soundsc(recovered_audio,16000);
```

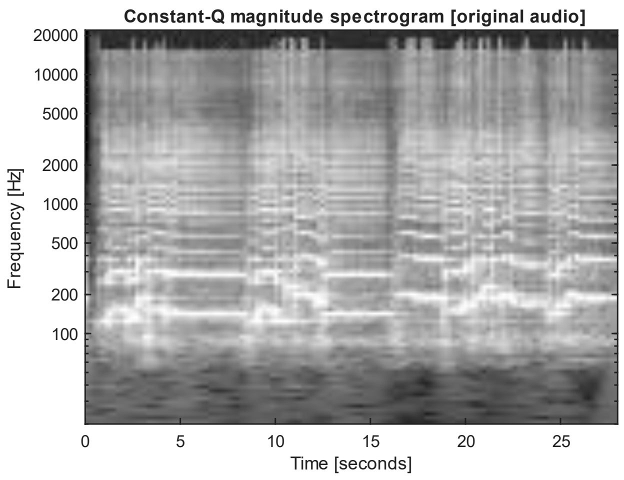
We can compute the PSNR at the original sampling rate (44.1 kHz) or the new sampling rate (16 kHz). Computing the PSNR at the original sampling rate will allow us to measure the impact of resampling along with the other sources of distortion. Before computing the PSNR, we scale both signals so that they use the expected range of [-1,1]. For the cello clip, the original range of values is [-0.354, 0.345], so we scale both the original audio and the reconstructed audio by $1/0.354 = 2.83$. Excluding this scaling step would increase the PSNR by $10\log_{10}(2.83^2) = 9.01$ dB.

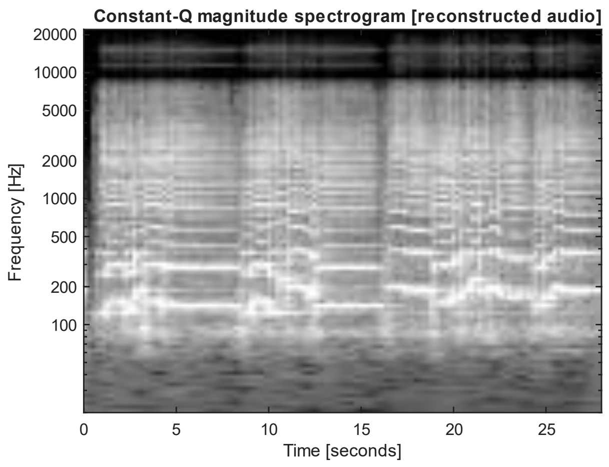
```
%% Compute the PSNR compared to the original audio signal at 44.1 kHz
original_audio_44kHz = mean(audioread('sm_cello.mp3'),2);
original_audio_44kHz = original_audio_44kHz(1:1234800);
recovered_audio_44kHz = resample(recovered_audio,441,160);
scale = 1/max(-min(original_audio_44kHz(:)),max(original_audio_44kHz(:)));
original_audio_44kHz = original_audio_44kHz*scale;
recovered_audio_44kHz = recovered_audio_44kHz*scale;
PSNR = @(x1,x2) -10*log10(mean(abs(x1 - x2).^2)) + 6.02;
PSNR(original_audio_44kHz,recovered_audio_44kHz)

ans =

   56.6122
```

The observed PSNR value of 56.6 dB indicates minimal distortion caused by the transformation process. However, the magnitude spectrograms of the original and recovered audio show some differences that account for the perceived distortion. In particular, the frequency band between 8 kHz and 22.05 kHz is lost during the resampling process.





```
%% display constant-q magnitude spectrograms of original and reconstructed
[cfs, f] = cqt(original_audio_44kHz, 'SamplingFrequency',44100, ...
    'BinsPerOctave',36,'FrequencyLimits',[20,22050],'TransformType','full');
cfs_mag = movmean(abs(cfs.c),11,2);
cfs_mag = 20*log10(cfs_mag(2:cfs.NyquistBin,1:11:end));
x = linspace(0,28,size(cfs_mag,2)); y = f(2:cfs.NyquistBin);
figure; imagesc(x,y,cfs_mag); axis xy; colormap('gray'); ylim([20,22050]);
clim([-130,-30]); set(gca,'YScale','log');
set(gca,'ytick',[100,200,500,1000,2000,5000,10000,20000]);
ylabel('Frequency [Hz]'); xlabel('Time [seconds]');
title('Constant-Q magnitude spectrogram [original audio]')

[cfs, f] = cqt(recovered_audio_44kHz, 'SamplingFrequency',44100, ...
    'BinsPerOctave',36,'FrequencyLimits',[20,22050],'TransformType','full');
cfs_mag = movmean(abs(cfs.c),11,2);
cfs_mag = 20*log10(cfs_mag(2:cfs.NyquistBin,1:11:end));
x = linspace(0,28,size(cfs_mag,2)); y = f(2:cfs.NyquistBin);
figure; imagesc(x,y,cfs_mag); axis xy; colormap('gray');
ylim([20,22050]); clim([-130,-30]); set(gca,'YScale','log')
set(gca,'ytick',[100,200,500,1000,2000,5000,10000,20000]);
ylabel('Frequency [Hz]'); xlabel('Time [seconds]')
title('Constant-Q magnitude spectrogram [reconstructed audio]')
```

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 $k = 1, \ldots, N-1$.

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

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

In the Fourier series coefficients $X[k]$, index k corresponds to continuous-time frequency $f_k = k\frac{f_s}{N}$ in Hz for $k = 0, 1, \ldots, \frac{N}{2} - 1$ and $f_k = f_s\left(\frac{k}{N} - 1\right)$ for $k = \frac{N}{2}, \frac{N}{2} + 1, \ldots, N - 1$, assuming N is even.

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 $k = 0, 1, \ldots, N-1$ in one dimension and n = 0, 1, …, N-1 in the other, form a column vector of $x[0]$, $x[1]$, …, $x[N\text{-}1]$, and multiply the matrix and vector to find the column vector of $X[0]$, $X[1]$, …, $X[N\text{-}1]$.