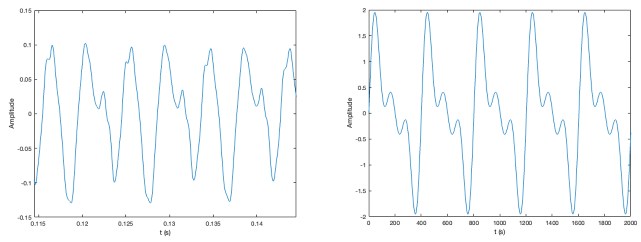
FOURIER ANALYSIS EXPLORATION

Sound and Music Signal Analysis Miniproject[*]

Rasmus Kjærbo
Sound and Music Computing
Aalborg University
Copenhagen, Denmark
rkjarb19@student.aau.dk

***Abstract*—In this report an overview of the Fourier transform is given and feature extraction and noise reduction approaches are explored on analytical functions and sampled audio data in Matlab. It is shown how the Fourier transform is derived from continuous functions and then applied to discrete audio signals. Lastly, a noise reduction algorithm is proposed with a resynthesis component. The algorithm can properly remove added noise from both analytical signals and sampled audio data, though the reconstruction of the frequency components without noise ended up 'sounding' better for frequency component feature extraction rather than denoising. Further work has to be done to tweak the algorithm towards creative music production and sound design or towards noise reduction of sampled audio signals.**

***Index Terms*—Sound and music signal analysis**



(a) Plucked guitar string, A note at approximately 110 Hz plus harmonics

(b) Three summed sinusoids ($F_1 = 110, F_2 = 220, F_3 = 440$)

Fig. 1: Amplitude waveform of two periodic signals, difference and similarities.

I. Introduction

The Fast Fourier Transform (FFT) is one of the original data algorithms being used everyday all around the world. It is used for solving PDEs, denoising data, for audio and image compression, for analysis of data and much, much more. [1]

A. Fourier Transform

In the early 1800s, Jean-Baptiste Joseph Fourier derived the transform and series to approximate solutions to partial differential equations (PDE). Fourier was especially interested in simplifying a solution to the heat equation $u(x, y, t) \rightarrow u_t = \alpha \nabla^2 u$, trying to decompose the problem (a periodic signal) intro smaller 'atoms' (its sine and cosine parts). [2]

The Fourier transform is a coordinate transformation into eigen functions (sines and cosines) and eigen vectors. In short, the Fourier transform seeks to approximate the function $f(x)$ as a sum of increasingly high frequencies of both sines and cosines in order to calculate the magnitude and phase of each decomposed frequency component of the original signal.

B. Sinusoids

Take a properly tuned guitar, pluck the second string from the top and you should hear the note A_2 with a fundamental frequency $f_0 = 110$ Hz (cf. fig. 1a). Apart from f_0 you also experience a plethora of overtones which constitute the overall sonic characteristic of 'the guitar sound' - these we call harmonics where f_0 is the fundamental frequency. To tell precisely how many harmonics (with respective amplitudes) are present in a sound is complicated. One could try a heuristic approach of manually adding up sines and cosines of various frequencies to arrive at an acceptable sonic representation. Not ideal - or analytical - but it gets us somewhat there. Consider a signal comprised of three harmonic components:

$$y(t) = b_1 \sin(2\pi f_1 t) + b_2 \sin(2\pi f_2 t) + b_3 \sin(2\pi f_3 t) \quad (1)$$

where b_i is the amplitude of the respective component and f_i is its corresponding frequency in Hz. The amplitude waveform is visible on fig. 1b. As seen on the figure, we have three distinct modes which somewhat look like the amplitude waveform of the guitar string playing A_2 but with a visible (and audible) error. In order to better approximate audio signals - and to do function analysis - we need more precise methods. Let us explore the Fourier sine and cosine series.

C. Fourier Sine and Cosine Series

To accurately reproduce a periodic real-world signal, often more than the sum of three frequencies are required. As the number of added frequencies increase, so does writing the function with increasingly many sine terms. Therefore, we rewrite the equation $y(t)$ from eq. (1) in a more elegant manner:

$$y(t) = \sum_{i=1}^{\infty} b_i \sin(2\pi f_i t) \quad (2)$$

To accurately reproduce periodic signals, a (theoretical) sum of infinitely many sinusoidal components with respective frequency f_i and and amplitude b_i are added up. In practice, this

[*]This project is an independent work of the author and not supported in any commercial means.

process is computed on a finite number of frequencies though - and at a specific sampling interval:

$$f(t) = \sum_{n=1}^{\infty} b_n \sin\left(\frac{n\pi}{L}t\right) \tag{3}$$

L determines the periodicity of the function f, which has period $2L$. Therefore f will have periodic copies every $2L$. Each term in the sum is known as a Fourier mode. For example, the fundamental mode (with $n = 1$) is

$$t = 1 \rightarrow f_1 = b_1 \sin\left(\frac{\pi}{L}\right) \tag{4}$$

The first coefficient b_1 determines how strongly this mode is represented in the overall sum.

D. Fourier Series

The Fourier sine series from eq. (5) is comprised of infinitely many increasing Fourier modes with respective frequencies. This collection of modes enables the series to (re)produce various waveforms. However, sine mode signals does not contain phase information. Typically, real-world signals contain phase information, which renders the Fourier sine series insufficient to properly represent them. The solution is to add a cosine series along to produce signals with any phase. Eq. (5) shows the sum of cosine and sine series, a standard definition of the Fourier series:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos\left(\frac{n\pi}{L}t\right) + b_n \sin\left(\frac{n\pi}{L}t\right) \tag{5}$$

where a_n is the cosine series coefficients, b_n the sine coefficients, and $\frac{a_0}{2}$ is a scaling factor. The sine terms phase are $0°$ while the cosine signals have a phase of $90°$. The combination of a cosine and sine signal with identical frequency produces a phase-shifted signal:

$$a\cos(\omega t) + b\sin(\omega t) = c\sin(\omega t - \phi) \tag{6}$$

where the ratio of the coefficients a and b determines the phase shift, ϕ.

E. Fourier Coefficient Formulas

The individual Fourier coefficient can be estimated using the following two formulas:

$$a_n = \frac{1}{L}\int_{-L}^{L} y(t) cos\left(\frac{n\pi}{L}t\right) dt \tag{7}$$

$$b_n = \frac{1}{L}\int_{-L}^{L} y(t) sin\left(\frac{n\pi}{L}t\right) dt \tag{8}$$

The integrals from eq. 7 and 8 return the weight of the n^{th} mode in order to match the signal at a given time for $y(t)$. As each mode has a unique frequency, the integrals determine how strongly each frequency is represented in the original signal $y(t)$. Note that the function $y(t)$ must be periodic with a period of $2L$ to be represented by the Fourier series. If it is not, then the derived Fourier series will not match $y(t)$ everywhere, but will create a periodic extension of the original function with period $2L$. [3] The magnitude of a frequency is a combination of both coefficients, specifically the magnitude of the n^{th} mode is

$$A_n = \sqrt{{a_n}^2 + {b_n}^2} \tag{9}$$

F. The Fourier Transform for audio signal analysis

In audio signal analysis for sound and music, we often work with complex phasors instead of sinusoids.

$$\hat{f}(w) = \mathcal{F}\big(f(x)\big) = \int_{-\infty}^{\infty} f(x)e^{-i\omega x}dx \tag{10}$$

G. Derivatives & Convolution

The Fourier transform is a great tool for working with derivatives and convolution.

$$\begin{aligned}\mathcal{F}\left(\frac{d}{dx}f(x)\right) &= \int_{-\infty}^{\infty} \frac{df}{dx} e^{-iwx}dx \\ &= iw\int_{-\infty}^{\infty} f(x)e^{-iwx}dx \\ &= iw\mathcal{F}\Big(f(x)\Big)\end{aligned} \tag{11}$$

where $\mathcal{F}$ is the Fourier transform. Researchers have compared the solutions of spectral derivatives compared to finite difference (central, forwards, backwards etc.) and found them to be superior [3]. The Fourier transform can also be sued to convert PDEs to Ordinary Differential Equations (ODE) which simplifies the solution.

$$\mathcal{F}(f \star g) = \mathcal{F}(f)\mathcal{F}(g) = \hat{f}\hat{g} \tag{12}$$

H. Discrete Fourier Transform - DFT

Moving from analytic functions to a data vector and deriving its sine and cosine components yields the Discrete Fourier Transform (DFT). This is the mathematical definition of the Fourier Series for a data vector: a unitary Vandermonde DFT matrix which is rarely computed due to the computational cost given by its scaling order (cf. section I-I)

$$\hat{f}_k = \sum_{j=0}^{n-1} f_j e^{-i2\pi jk/n} \tag{13}$$

$$f_k = \left(\sum_{j=0}^{n-1} \hat{f}_j e^{i2\pi jk/n}\right)\frac{1}{n} \tag{14}$$

$$\omega_n = e^{-2\pi i/n} \tag{15}$$

where $i = \sqrt{-1}$

I. Fast Fourier Transform - FFT

The FFT is the algorithm you use to compute the DFT. Need is for speed and efficiency. Scaling order for FFT is $\mathcal{O}\left(nlog(n)\right)$ whereas for the DFT is $\mathcal{O}\left(n^2\right)$. The FFT algorithm utilizes the symmetry in the DFT matrix to decrease the order of computation. This is especially efficient if the data has a number of data points, e.g. sample rate or bit depth, that is a power of two. [3]

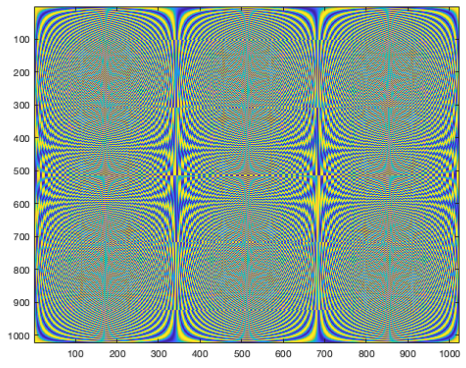


Fig. 2: Visual representation of the real part of the DFT matrix - cf. section VI for code

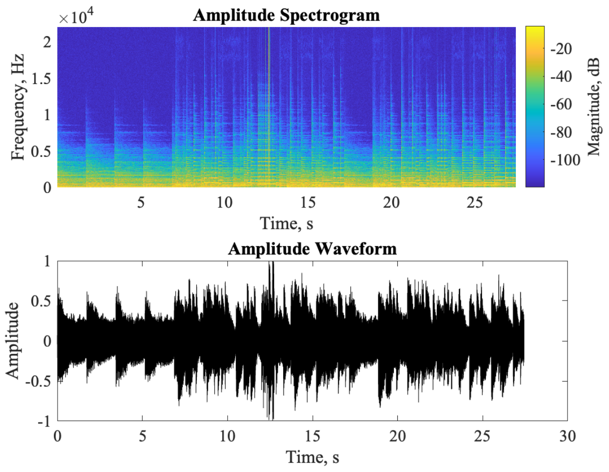


Fig. 3: Short-Time Fourier Transform and Waveform Plot - cf. section VI for code

J. Gabor Transform, STFT & The Spectrogram

Plotting the waveform of a discrete function of time yields high temporal accuracy but little information about the spectral contents. The Fourier transform of a signal yields high accuracy of frequency components throughout the whole signal, but little information about which frequencies are present with respective power at a given time. The Gabor transform, or the Short-Term Fourier Transform (STFT), is a mixture of the two, prioritising either temporal or spectral resolution. The weight or trade off is determined by your window size. The spectrogram is a balance between the two, optimized for as much temporal and spectral resolution possible.

The process is: generate a gaussian, fixed width window, and convolve Fourier transform with a sliding window. Compute the power spectrum of a short window of time. Repeat for the duration of your signal. The generated time-frequency plot $\rightarrow$ sliding power spectrum which shows the time-specific frequency components of the signal. [1] [3] [4]

$$\begin{aligned} G(f) &= \hat{f}_g(t,\omega) \\ &= \int_{-\infty}^{\infty} f(\tau) e^{-j\omega\tau} g(t-\tau) d\tau \end{aligned} \tag{16}$$

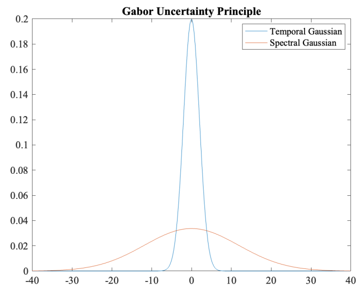


Fig. 4: Gabor Uncertainty Principle relationship - cf. section VI for code

where $g(t-\tau)$ is the sliding gaussian window.

This method is used in many signal processing concepts, i.e. music analysis software like Shazaam. [5]

1) Gabor Uncertainty Principle: How much is possible to resolve in either time or frequencies.

$$\left(\int_{-\infty}^{\infty} x^2 |f(x)|^2 dx\right) \left(\int_{-\infty}^{\infty} w^2 |\hat{f}(w)|^2 dw\right) \geq \frac{1}{16\pi^2} \tag{17}$$

The first term is a measure of the function $f(x)$ in space, which is multiplied by the same measure of its Fourier transform, this conjugate pair is depicted on fig. 4 and illustrates the uncertainty principle.

II. Results

Using the above theory, an audio sample analysis and resynthesis script was constructed in Matlab. Below are the results of loading an audio sample adding random noise to the signal (c.f. fig 5, computing the FFT, PSD, selecting only frequency components above a certain PSD value and then reconstructing the Power Spectrum from said selected frequency components.

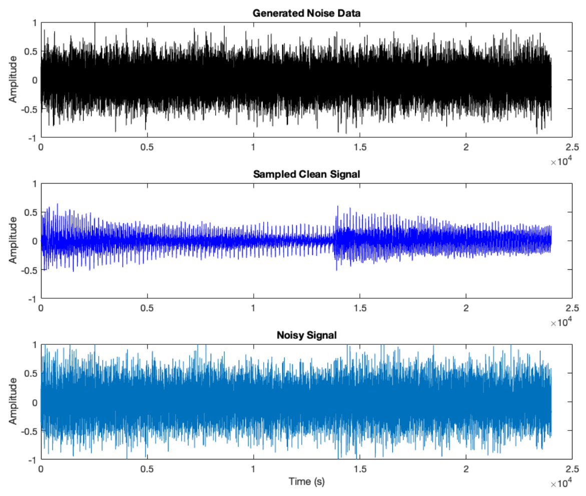


Fig. 5: Piano chords sample with added noise - cf. section VI for code

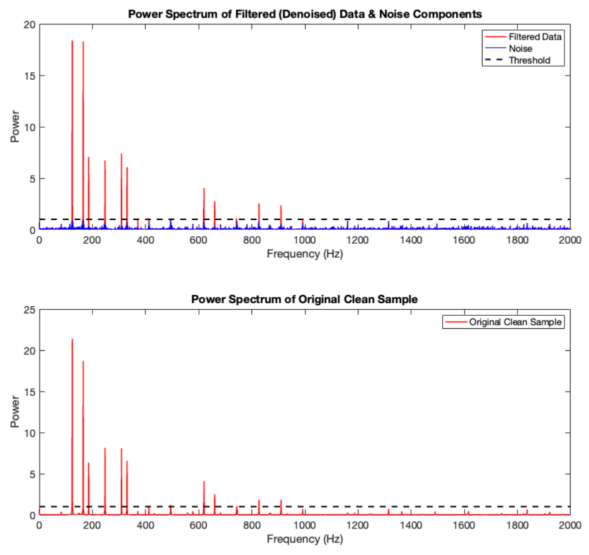


Fig. 6: Power spectrum comparison between original sample and filtered signal - cf. section VI for code

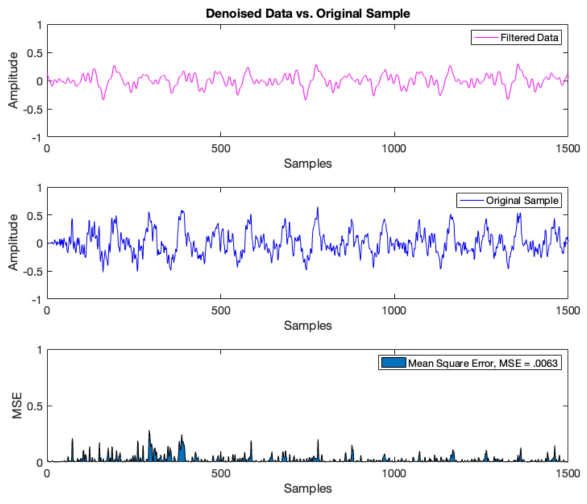


Fig. 7: Mean Square Error (MSE) and amplitude waveform comparison - cf. section VI for code

III. Discussion

Decomposing sampled audio data into its cosine and sine components give rise to many signal analysis and computational approaches. The power spectra plots of the original sample and the filtered data show a decent amount of coherence, c.f. fig. 6. The overall Mean Square Error (MSE) between a segment of the sampled audio and filtered denoised audio is 0,0063 and the MSE is plotted on fig. 7.

IV. Conclusion

The power of the Fourier transform us immense. This report explored just a bit of what is possible with decomposition,

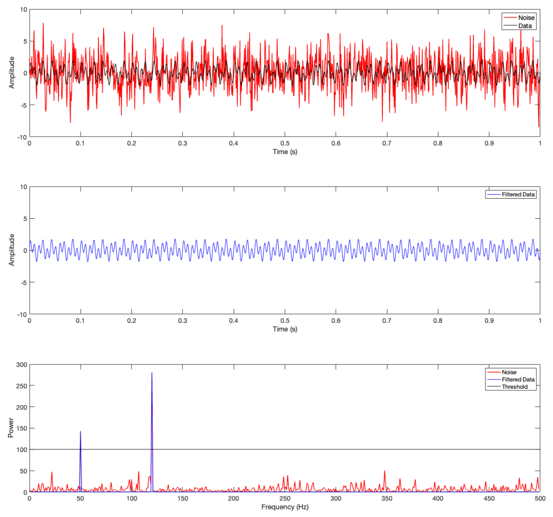


Fig. 8: Denoise algorithm with FFT and iFFT - cf. section VI for code

windowing, convolution, and reconstruction, leaving a good basis for further work and implementations.

V. Future work

Noise reduction and feature extraction with NMF would be great to implement. Exporting fundamental frequencies to MIDI with pitch, time, and tempo.

References

[1] M. M. Goodwin, "The STFT, Sinu 12. The STFT, Sinusoidal Models, and Speech Modification," *Part B*, p. 30, 2008.

[2] M. G. Christensen, *Introduction to Audio Processing*. Cham: Springer International Publishing, 2019.

[3] S. L. Brunton and J. N. Kutz, *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 1 ed., Jan. 2019.

[4] S. J. Orfanidis, *Introduction to signal processing*. Prentice Hall signal processing series, Englewood Cliffs, N.J: Prentice Hall, 2010.

[5] A. L.-C. Wang, "An Industrial-Strength Audio Search Algorithm," p. 7.

VI. APPENDIX A: CODE

Scripts and code can be found on https://github.com/rallevondalle/SMSAMiniProject

A. Compute and Visualize DFT Matrix as Colour Space

```
%% Computing the DFT matrix
n = 1024;                     % sampling points
w = exp(-i*2*pi/n);

[I,J] = meshgrid(1:n,1:n);
DFT   = w.^((I-1).*(J-1));

imagesc(real(DFT))            % display DFT matrix
                              % as an color image
```

B. Compute Fourier Coefficients MATLAB Function

```
function [freq,a,b] = computeFourierCoef(y,fs)
    % estimate coefficients for the Fourier series
    % using numerical trapezoidal integration

    Nmodes = 1000;
    yc     = y(1e4:2e4);

    tc     = 1/fs*(0:(length(yc)-1));
    L      = tc(end)/2;                   % the domain is assumed to be 2L
    freq = (1:Nmodes)/(2*L);              % frequencies

    % Estimate coefficients using trapezoid rule
    a      = zeros(1,Nmodes);
    b      = zeros(1,Nmodes);

    a0     = 1/L * trapz(tc,yc);

    for n = 1:Nmodes                      % trapezoidal numerical integration
        a(n)  = 1/L * trapz(tc,cos(n*pi/L*tc).*yc);
        b(n)  = 1/L * trapz(tc,sin(n*pi/L*tc).*yc);
    end
end
```

C. Short-Time Fourier Transform Implementation by Zhivomirov

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               Short-Time Fourier Transform           %
%                with MATLAB Implementation            %
%                                                      %
% Author: Ph.D. Eng. Hristo Zhivomirov        12/21/13 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [STFT, f, t] = stft(x, win, hop, nfft, fs)

% Input:
% x      - signal in the time domain
% win    - analysis window function
% hop    - hop size
% nfft   - number of FFT points
% fs     - sampling frequency, Hz

% Output:
% STFT   - STFT-matrix (only unique points, time
%         across columns, frequency across rows)
% f      - frequency vector, Hz
% t      - time vector, s

% representation of the signal as column-vector
x    = x(:);

% determination of the signal length
xlen = length(x);

% determination of the window length
wlen = length(win);

% stft matrix size estimation and preallocation
NUP  = ceil((1+nfft)/2);        % calculate the number of unique fft points
L    = 1+fix((xlen-wlen)/hop);  % calculate the number of signal frames
STFT = zeros(NUP, L);           % preallocate the stft matrix

% STFT (via time-localized FFT)
for l = 0:L-1
    % windowing
    xw = x(1+l*hop : wlen+l*hop).*win;

    % FFT
    X = fft(xw, nfft);

    % update of the stft matrix
    STFT(:, 1+l) = X(1:NUP);
end

% calculation of the time and frequency vectors
t = (wlen/2:hop:wlen/2+(L-1)*hop)/fs;
f = (0:NUP-1)*fs/nfft;
end
```

D. *STFT of Piano Phrase with Melodic & Harmonic Parts*

```matlab
% load audio file
[x, fs] = audioread('CTPiano.wav'); % load audio file and set sample rate
x       = x(:, 1);                  % get the first channel, mono signal
N       = length(x);                % N data points
tWave   = (0:N-1)/fs;               % time vector for waveform plot

% define analysis parameters
wlen = 1024;                        % window length         (power of 2 optimal)
hop  = wlen/4;                      % hop size              (power of 2 optimal)
nfft = 4096;                        % number of fft points  (power of 2 optimal)

% perform STFT
win = blackman(wlen, 'periodic');   % windowing function
[S, f, t] = stft(x, win, hop, nfft, fs);

% calculate the coherent amplification of the window
C = sum(win)/wlen;

% take the amplitude of fft(x) and scale it, so not to be a
% function of the length of the window and its coherent amplification
S = abs(S)/wlen/C;

% correction of the DC & Nyquist component
if rem(nfft, 2)                     % odd nfft excludes Nyquist point
    S(2:end, :) = S(2:end, :).*2;
else                                % even nfft includes Nyquist point
    S(2:end-1, :) = S(2:end-1, :).*2;
end

% convert amplitude spectrum to dB (min = -120 dB)
S = 20*log10(S + 1e-6);

% plot spectrogram
figure
subplot(2,1,1);
surf(t, f, S)
shading interp;;axis tight;view(0, 90)
xlabel('Time, s')
ylabel('Frequency, Hz')
title('Amplitude Spectrogram')
hcol = colorbar;
ylabel(hcol, 'Magnitude, dB')

subplot(2,1,2);
plot(tWave,x,'black');
xlabel('Time, s')
ylabel('Amplitude')
title('Amplitude Waveform')

print('STFTSpectrogramWaveform','-dpng')
```

E. Denoise algorithm with FFT and iFFT

```
%% Generate signal data and noise
dt     = .001;                             % time resolution, 'sample rate'
t      = 0:dt:1;                           % time vector
fclean = sin(2*pi*50*t) + sin(2*pi*120*t);  % sum of 2 frequencies
f      = fclean + 2.5*randn(size(t));      % add noise


% Plot signal and noise vectors
figure
subplot(3,1,1)
plot(t,f,'r','LineWidth',1.5), hold on
plot(t,fclean,'k','LineWidth',1.2)
l1 = legend('Noise','Data');set(l1,'FontSize',14)
ylim([-10 10]); set(gca,'FontSize',14)
xlabel('Time (s)')
ylabel('Amplitude')
hold off


% Compute FFT of noisy signal data
n       = length(t);
fhat    = fft(f,n);                   % compute the fast Fourier transform
PSD     = fhat.*conj(fhat)/n;         % calculate power spectrum
freq    = 1/(dt*n)*(0:n);             % create x-axis of frequencies in Hz
L       = 1:floor(n/2);               % vector with only half of the frequencies


% Use PSD to filter noise
denoiseThreshold = 100;               % denoise threshold
indices  = PSD>denoiseThreshold;      % select frequency components above threshold
PSDclean = PSD.*indices;              % zero out below threshold
fhat     = indices.*fhat;             % zero out small Fourier coeffs. in signal
ffilt    = ifft(fhat);                % Inverse FFT


% Plot denoised data
subplot(3,1,2)
plot(t,ffilt,'blue')
legend('Filtered Data')
ylim([-10 10]); set(gca,'FontSize',14)
xlabel('Time (s)')
ylabel('Amplitude')


% Generate threshold vector for plot
noiseThreshold = ones(1,1/dt+1);
noiseThreshold = noiseThreshold*denoiseThreshold;
xVector        = [0:1000];


% Plot power spectrum of noise, filtered data, and threshold
subplot(3,1,3)
plot(freq(L),PSD(L),'r','LineWidth',1.5), hold on
plot(freq(L),PSDclean(L),'-b','LineWidth',1.2)
plot(xVector,noiseThreshold,'black')
xlim([0 500]);set(gca,'FontSize',14)
legend('Noise','Filtered Data','Threshold')
xlabel('Frequency (Hz)')
ylabel('Power')
```

F. Sampled Piano Chords Denoise FFT and Resynthesis

```
%% PIANO DENOISE TESTER AND RESYNTHESIZER

[x, fs] = audioread('CTPiano8k.wav');   % load an audio file, fs = 8000
xmono = x(:, 1);                        % mono signal, select first channel

analysisLength = 3;                     % 3 seconds

xshort = xmono(1:analysisLength*fs,1);  % select 1 second of audio

n = length(xshort);                     % number of samples to calculate

length = length(xshort);                % length of FFT

h = length/n;                           % sampling interval
t = (0:h:length-h);                     % time vector
S = xshort;                             % shorten sampled audio

% Random noise
RN = 0.25*randn(n,1);
NS = RN + S;                            % add noise to signal
NSmono = NS(1:n,1);                     % reduce dimensions to 1 channel

% Plots
figure(1)
subplot(3,1,1)
plot(t,RN,'k');
title('Generated Noise Data');set(gca,'FontSize',14)
ylabel('Amplitude')
ylim([-1 1])

subplot(3,1,2)
plot(t,S,'blue');
title('Sampled Clean Signal');set(gca,'FontSize',14)
ylabel('Amplitude')
ylim([-1 1])

subplot(3,1,3)
plot(t,NSmono);
title('Noisy Signal');set(gca,'FontSize',14)
xlabel('Time (s)')
ylabel('Amplitude')
ylim([-1 1])


% Compute FFT and PSD of noisy signal data
fhat    = fft(NSmono,n);                % compute the fast Fourier transform
PSD     = fhat.*conj(fhat)/n;           % calculate power spectrum
freq    = (0:n)/2;                      % create x-axis of frequencies in Hz
L       = 1:floor(n/2);                 % vector with only half of the frequencies

figure(2)
plot(freq(L),PSD(L),'r','LineWidth',1.5), hold on
title('Power Spectrum, Noisy Signal'); set(gca,'FontSize',14)


% Use PSD to filter noise
PSDthreshold = 1;                       % denoise threshold

indices      = PSD>PSDthreshold ;       % select frequency components above threshold
PSDclean     = PSD.*indices;            % zero out below threshold
indicesNoise = PSD<PSDthreshold ;       % select noise components, below PSD threshold
PSDnoise     = PSD.*indicesNoise;       % zero out above threshold
fhat         = indices.*fhat;           % zero out small Fourier coeffs. in signal
ffilt        = ifft(fhat);              % Inverse FFT
```

```
% Compute FFT and PSD of clean
fhatCleanS    = fft(S,n);                % compute the fast Fourier transform
PSDcleanS     = fhatCleanS.*conj(fhatCleanS)/n; % calculate power spectrum


% Uncomment to listen
%soundsc(NSmono,fs)
%soundsc(ffilt,fs)


% Plot denoised data vs. original sample
figure(3)
subplot(2,1,1)
plot(t,ffilt,'m')
legend('Filtered Data')
ylim([-1 1]); set(gca,'FontSize',14)
xlim([0 1500])
xlabel('Samples')
ylabel('Amplitude')
title('Denoised data vs. Original Sample')

subplot(2,1,2)
plot(t,S,'blue')
legend('Original Sample')
ylim([-1 1]); set(gca,'FontSize',14)
xlim([0 1500])
xlabel('Samples')
ylabel('Amplitude')


% Generate threshold vector for plot
noiseThreshold = ones(1,n+1);
noiseThreshold = noiseThreshold*1;
xVector        = [0:n];


% Plot power spectrum of noise, filtered data, and threshold
figure(4)
subplot(2,1,1)
plot(freq(L),PSD(L),'r','LineWidth',1.5), hold on
plot(freq(L),PSDnoise(L),'-b','LineWidth',1.2)
plot(xVector,noiseThreshold,'k','LineWidth',2,'LineStyle','--')
xlim([0 2000]);set(gca,'FontSize',14)
legend('Filtered Data','Noise','Threshold')
xlabel('Frequency (Hz)')
ylabel('Power')
title('Power Spectrum of Filtered (Denoised) Data & Noise Components')

subplot(2,1,2)
plot(freq(L),PSDcleanS(L),'r','LineWidth',1.5), hold on
plot(xVector,noiseThreshold,'k','LineWidth',2,'LineStyle','--')
xlim([0 2000]);set(gca,'FontSize',14)
legend('Original Clean Sample')
xlabel('Frequency (Hz)')
ylabel('Power')
title('Power Spectrum of Original Clean Sample')


% Overall MSE between sampled audio and filtered signal
MSE = norm(ffilt-S,'fro')^2/numel(S);


% Compute Mean Square Error for each sample
mse = zeros(1,n);                        % initialize mse vector

for i = 1:n                              % mse per sample
```

```
    mse(i) = mean((ffilt(i)-S(i)).^2);
end

figure(5)


% Plot Power Spectra of PSDs, noise, and threshold
subplot(2,1,1)
plot(freq(L),PSD(L),'r','LineWidth',1.5), hold on
plot(freq(L),PSDnoise(L),'-b','LineWidth',1.2)
plot(xVector,noiseThreshold,'k','LineWidth',2,'LineStyle','--')
xlim([0 2000]);set(gca,'FontSize',14)
legend('Filtered Data','Noise','Threshold')
xlabel('Frequency (Hz)')
ylabel('Power')
title('Power Spectrum of denoised & reconstructed signal')

subplot(2,1,2)
plot(freq(L),PSDcleanS(L),'r','LineWidth',1.5), hold on
plot(xVector,noiseThreshold,'k','LineWidth',2,'LineStyle','--')
xlim([0 2000]);set(gca,'FontSize',14)
legend('Original Sample')
xlabel('Frequency (Hz)')
ylabel('Power')
title('Power Spectrum of original clean sample')


% Plot amplitude waveforms of sampled audio, resynthesized audio and MSE
figure(6)
subplot(3,1,1)
plot(t,ffilt,'m')
legend('Filtered Data')
ylim([-1 1]); set(gca,'FontSize',14)
xlim([0 1500])
xlabel('Samples')
ylabel('Amplitude')
title('Denoised Data vs. Original Sample')

subplot(3,1,2)
plot(t,S,'blue')
legend('Original Sample')
ylim([-1 1]); set(gca,'FontSize',14)
xlim([0 1500])
xlabel('Samples')owe
ylabel('Amplitude')

subplot(3,1,3)
area(abs(mse))
legend('Mean Square Error, MSE = .0063')
ylim([0 1]); set(gca,'FontSize',14)
xlim([0 1500])
xlabel('Samples')
ylabel('MSE')
ylim([0 1])
```