Homework 2: Gábor Transforms

Karl Marrett (kdmarrett@gmail.com)

Abstract

In this report, we use methods for analyzing changes in frequency content across time. To analyze frequency content in any window we will be implementing the Fast Fourier Transform (FFT), an algorithm that uses the insights of the similarly named mathematical technique that decomposes a time signal into a linear sum of periodic functions. The potential drawback of the FFT when specifically analyzing *changes* across time is the implicit assumption of the algorithm that the signal is continuous. Therefore, in order to produce information on fluctuations in frequency content across time (formally known as a spectogram) we must break any data sample into discrete windows. By implementing this technique known as the Gábor Transform we seek to explore the effect of:

1. The sampling window width
2. The number of sampling windows
3. The window function

This exploration will be conducted on a famous sample recording of the composer Handel. Applying the techniques learned from the classic sample to the performance of a more modern virtuoso, we will find:

1. The music score for the recording of 'Mary had a little lamb'
2. The difference of *timbre*, the characteristic quality of sound sources, between a piano and recorder recording.

1 Introduction and Overview

1.1 Gábor Transform

We learned from our analysis of frequency content over intervals that time information is lost using the FFT. This means that when confronted with the problem of analyzing frequency content over time for a given interval we must choose a new approach. The first idea that might come to mind to avoid this problem is simply splitting the data into discrete time windows and analyzing those windows separately. This approach is known as the Gábor transform, one of the methods of Short Time Fourier Transform (STFT). Designing Gábor

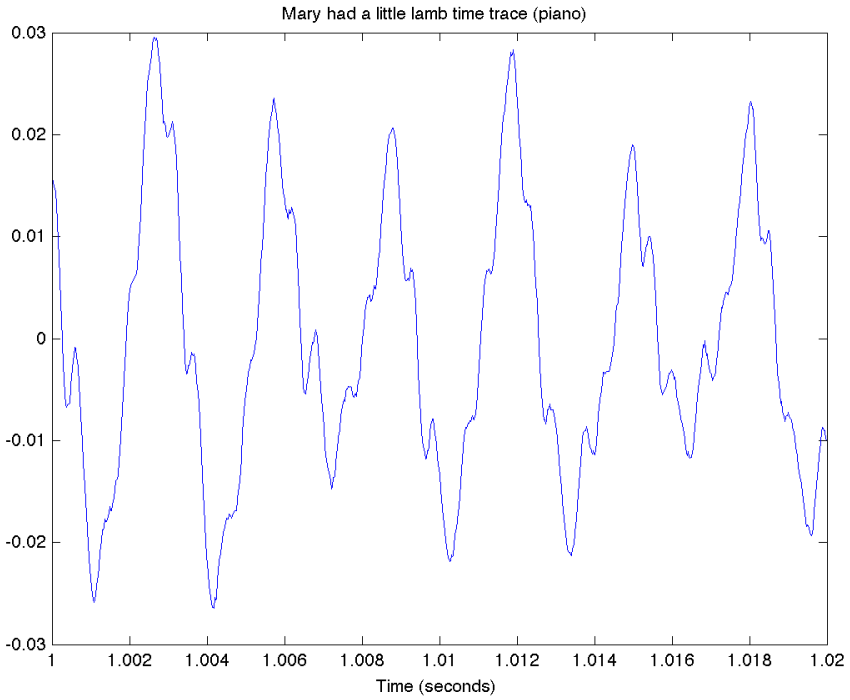

Figure 1: Time trace of a piano note zoomed in on a small window. We can see that although a note on the piano has a characteristic frequency, it is composed of many more frequencies than just a pure sine wave. We can see that this particular note is largely a lower frequency component combined with higher frequencies. The shape of the overall wave repeats in time. In order for the overall shape to appear periodic, all of higher frequencies must be integer multiples of the *fundamental* lower frequency.

transforms involve the choice of window size, window function (windows can be cut simply from data i.e. step filter or they can be created by applying a specific function shape), and the total number of windows across the data. Developing heuristics to get the most information out of our sounds with these choices will be discussed further in the next two sections.

1.2 Notes and Timbre

Decomposing a sound signal into its component frequencies over time allows one to create a musical score for the signal. We can verify that this is possible by recognizing that the notes that we identify in music are simply distinct frequencies of sound waves. However, when dealing with a real world voice or instrument, the single notes we hear are not pure sine waves at one frequency. In the realm of perception this is already obvious, for example you can distinguish the sound of a recorder playing middle A vs. that of a piano even if both instruments play with the exact same intensity and are perfectly in tune with the same frequency. The distinction that you naturally perceive is known as *timbre*, a term which is oddly defined in terms of what it is not – the character or quality of a musical sound that is

not due to its frequency (pitch) or intensity.[4]

Timbre can be visualized both in the time and frequency domains of signals. When we look very closely at notes of the piano for example (see Figure 1) we see a repeating shape through time. We know from Fourier analysis that any periodic signal must be decomposable into a set of periodic signals. Since we notice from this example that the shape of the waves is periodic we know that it is composed of some frequency ω and integer multiples of that frequency (i.e. $2\omega, 3\omega, 4\omega...$). If we look at the frequency components of the note, we can confirm this realization that a characteristic note in the real world has a set of integer multiples frequencies which are known as *overtones*. If a note is composed of numerous integer multiples of itself then what is the difference in a piano note of A at 440 Hz and its octave at 880 Hz? The answer is that notes have a frequency range with the most energy known as a *formant*. These formants can be detected by analyzing the amplitude or power of the sounds. In fact, by filtering around these formants we can get cleaner detections of our original music score.

2 Theoretical Background

Since this analysis is done with a computer, we will be implementing the fast Fourier transform (FFT), a discrete form of Fourier analysis. The FFT algorithm has a broad set of applications and implications but, for sake of brevity, we will only consider those that are directly relevant to the analysis of this report namely:

- It runs on a finite interval $x \in$ some interval $[-L, L]$
- The range $[-L, L]$ is discretized into 2^n points
- Assumes the function is periodic on an interval of 2π.

These implications require that we compute frequency content as discrete intervals across time. Let's first consider the trivial case where we take frequency content (via fft) across the whole time sample. This would return the frequency content of the whole sample but since these frequencies are independent of the time axis, we have no information about when different frequencies occurred, and instead only get a general average across the whole interval. If we instead split the sample into two we would have a similar case where now we only have an average of the first and second halves except one difference. Now since the window is half as large, the fundamental frequency we can compute is twice as big.[2] In other words, we are forced to have a smaller frequency range the smaller our time interval becomes. In addition to losing range we also lose resolution across our spectrum when we lower the number of samples as discussed further in the next section.[2] In the signal processing literature the constraints discussed above are called Nyquist frequencies. In addition to making the criteria of keeping frequencies between the Nyquist range we will also use information about the frequency ranges of instruments and music to further constrain our spectogram content. This fundamental trade-off between the temporal and frequency resolution of our Gábor transform will guide how we proceed in this report.

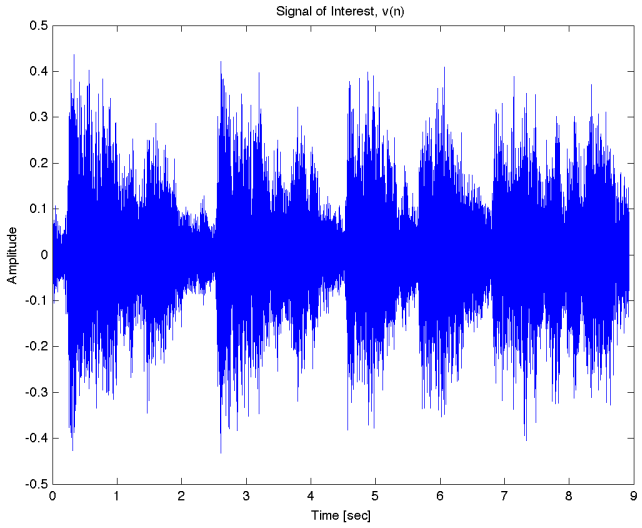

Figure 2: Shown is the simple time trace of the Handel sample in time. In order to compute frequency components across the time sample we will segregate this time sample into distinct windows.

3 Algorithm Implementation and Development

Although our sounds are loaded and manipulated as vectors, it is natural to think of them in terms of frequency (Hertz) across time (seconds), thus the first part of our script is devoted to converting our existing data in ways that make it intuitive to visualize. For example, we can initially visualize the sample by plotting the vector *handel* across the total number of samples divided by our sampling rate Fs which will give us an x axis in seconds. The simple visualization of the raw sound is shown in Figure 2.

3.1 Gábor Transform

To discretize our signal in time we will begin with building the Gaussian filter defined by equation 1:

$$g(t) = e^{a(t-b)^2} \tag{1}$$

where a represents the width of the Gaussian and b represents how it is shifted across the time axis. We will take the approach of reasoning about the various strategies we can take, visualizing the results to check our intuitions, and finally choosing our strategy and parameters intelligently from our observations.

The choice of window width plays a major role in the final resolution of the spectogram. Our spectral resolution or minimum frequency step δf equals the sampling frequency divided by the time points n.[1] Therefore, the smaller our number of time points in the window, the wider the frequency increments will become and the more blurred the frequency content will be.[2] This is shown clearly in Figure 3.

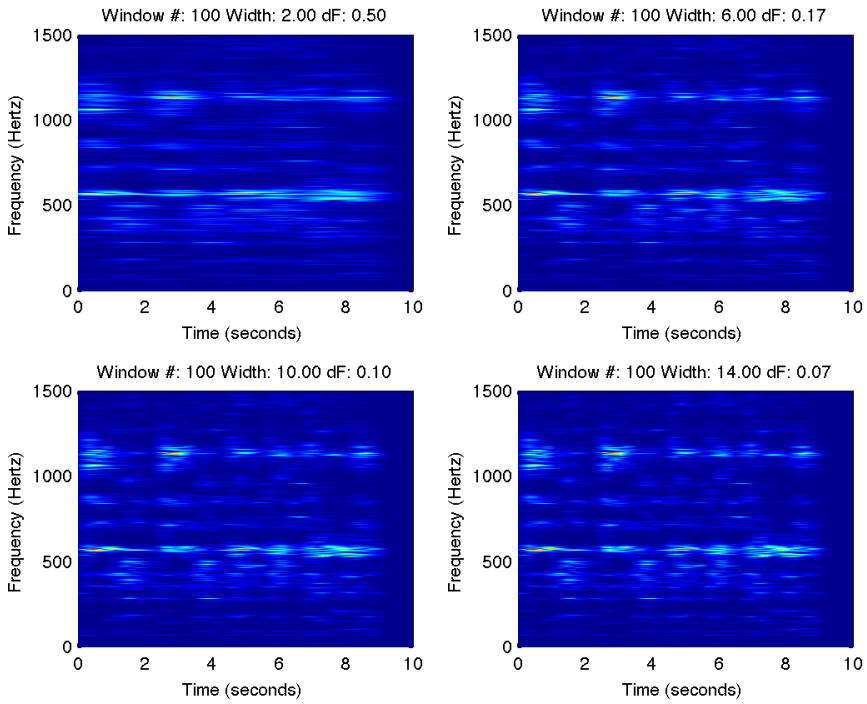

Figure 3: Shown are the Handel spectograms labeled by the window width, number of windows, and the frequency resolution *df* afforded by the choice in window size. As we can see with the *df* value and relative resolution along the frequency axis, the smaller the window width the less resolution we have to discriminate frequencies.

From a carefully chosen window size, we must also choose enough windows to accurately cross the window. If the translations of the filter are too coarse, we effectively loose the spectral information between the points of translations. We can see the blurring effects of exaggerated under-sampling in Figure **??**. Using a large window with too many samples can also blur an image by including too much of the data in each window, this is one major effect of oversampling that is especially accentuated by the step filter.

3.2 Filters

Another major focus in STFT or Gábor Transforms is the choice of windowing functions. The common function that we use is the Gaussian function as defined by Equation 1. Of the filters analyzed, the Gaussian had the highest temporal resolution (each note in the sample is very distinct in Figure 5) but had the lowest spectral resolution.

Another common choice of filter is the Mexican Hat function which is proportional to the double derivative of the Gaussian, and as we might intuit from it's shape, it exaggerates the contrast of the frequency information giving more precise measurements of pitch. However, we can also see that some temporal resolution is lost relative to the gaussian function. To build the Mexican Hat filter, the equation was taken from the *mexihat* function and scaled

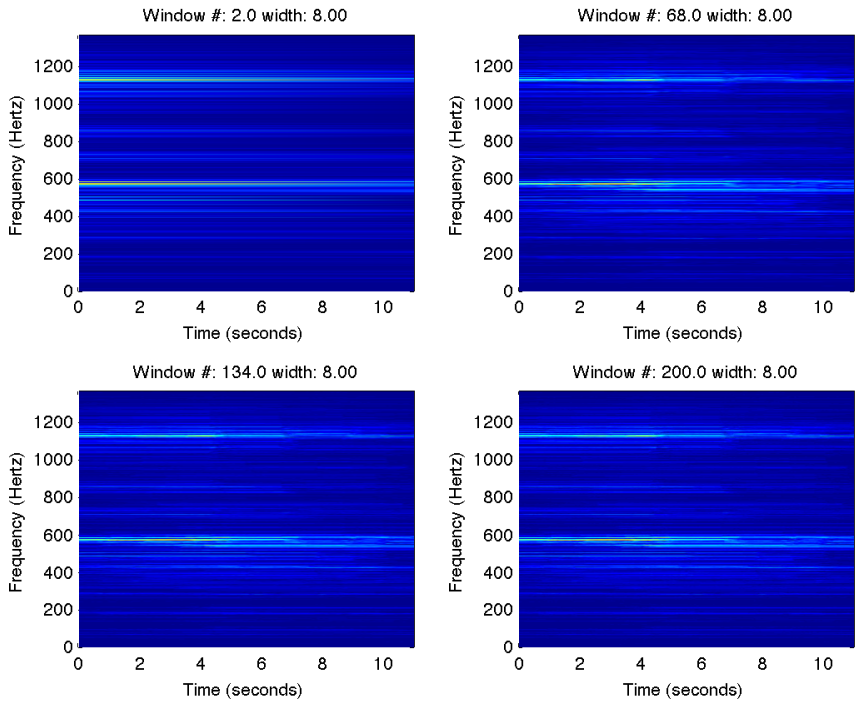

Figure 4: The spectogram of the Handel sample with different number of windows. In the top left corner, using only two windows clearly blurs the data, as we increase the samples however, the resolution in the temporal domain increases.

to the width and translation of choice.

Perhaps the simplest filter is the step function which simply cuts the sample into discrete windows. As we can see in Figure 5 it gives the highest relative spectral resolution but blurs much of the temporal resolution. This smearing in the temporal domain with the step filter is especially exaggerated with oversampling. We can intuit why this is by considering that while the other filters we have considered have a graded attenuation of time samples around the translation, the step function simply has a flat top, making the distinction in spectral between local translations less fine. Width and samples constant, the higher resolution in the frequency domain of the step filter implemented may just be because it keeps a larger window of data compared to the other filters.

3.3 Score

As previously discussed, the notes of real world instruments have multiple frequency components known as overtones. However, in order to create a score we want to find the formant frequency of each note that is played in the piano and recorder sample. In order to find the formant, we need to choose the frequency range with the highest energy. Either by listening to the music, observing the spectogram, or looking up the frequency ranges of these two instruments we would find that we are looking for frequencies in the range of about 200 -

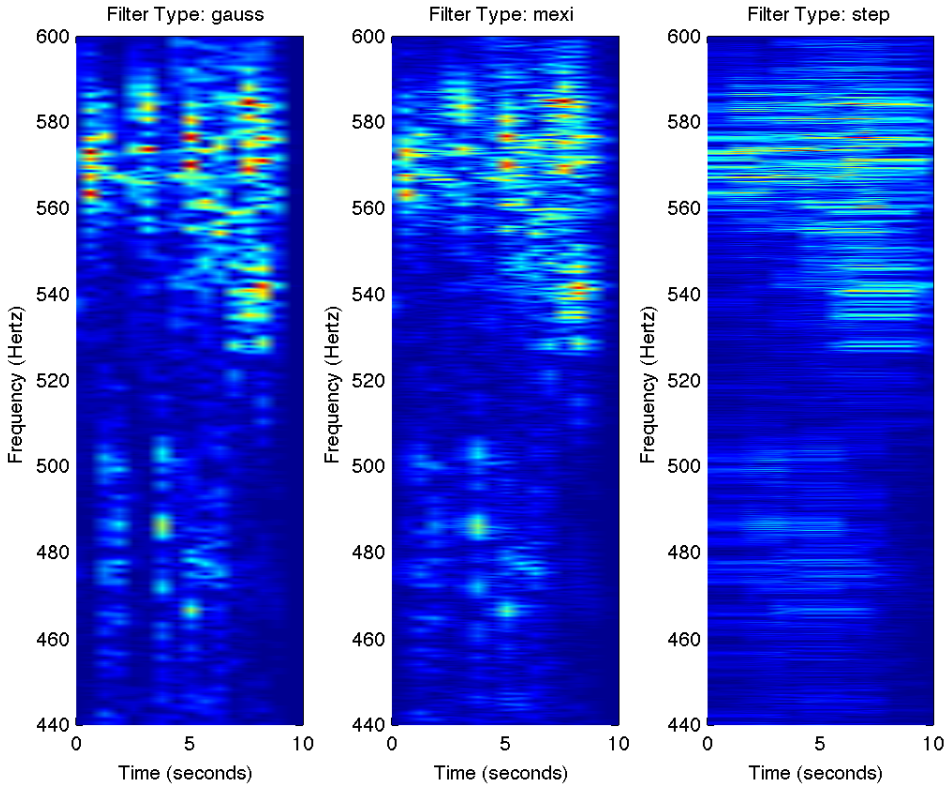

Figure 5: From left to right: the Gaussian, Mexican Hat, and Step filters applied to generate the Handel sample spectogram. As we can see each has unique effects on the temporal and spectral resolution of our final spectogram. Moving from left to right, the filter types gain finer and finer spectral resolution while losing temporal resolution.

600 Hz. In order to attenuate the higher frequencies we can simply apply a bandpass filter as detailed in the previous homework.

Having removed the overtones we can then take the maximum amplitudes across the sample. To automate the creation of a score, I generated a list of possible frequencies of notes and their respective names ('A1' 'B1' etc.) in the function *getNotes*(). I Used both vectors of the frequencies and names of notes as a rudimentary map for finding the nearest note of any particular slice. I found the function *roundtowardvec* (discussed and cited in Appendix A) for rounding the actual maximum frequencies to their ideal equivalents. This is done in the function *getScore*. By indexing into the note names I then had an estimate note name for each time slice of the piano and recorder spectograms as shown in Table 1 and Table 2 respectively.

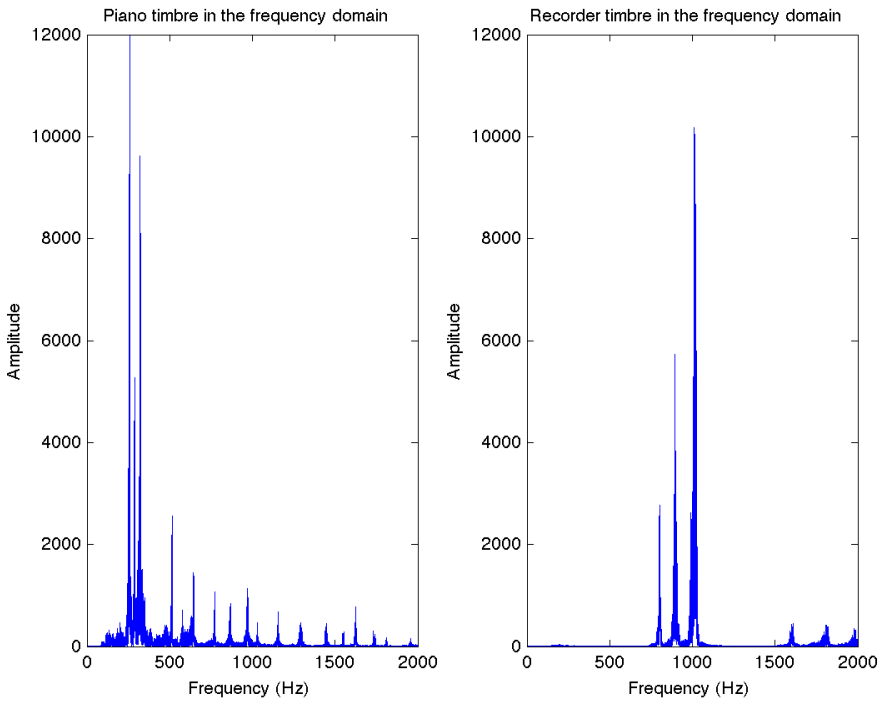

Figure 6: The FFT of a piano note and a recorder note. The distinct characteristic frequencies of each instrument produce the unique periodic waveform shown in Figure 7.

4 Computational Results

The timbre of the piano and recorder are distinct. This distinction is visualized most clearly in the spectral domain as shown in Figure 6. Even if the piano and recorder were playing the same note with the same intensity the energy of their overtones would appear distinct. The score generated is visualized in the table below. When band-pass filtering our spectograms (Figure 1) the results resemble the structure of our table.

5 Summary and Conclusions

The Gábor Transform is a powerful method for gaining information on spectral information across time. However, after having implemented it, we can see that there are more intelligent ways of getting around the inherent constraints of the FFT. For example, instead of choosing to maximize on either spectral or temporal resolution for a sample, we could choose to first take the low frequency information for the entire sample, then split it up into smaller time samples to gain higher temporal resolution for the higher frequencies in a technique known as spectral analysis.[3] Despite these drawbacks, we were still able to regenerate a musical score from the transform which attests to the performance of this rather old approach.

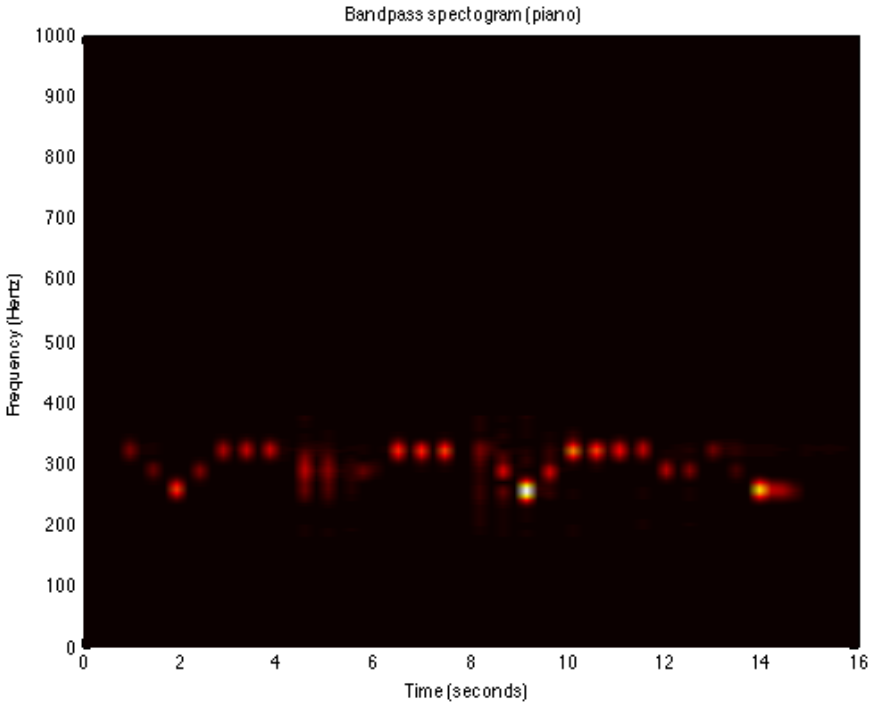

Figure 7: The FFT of the piano spectogram band-passed around 400 Hz. The notes of the song clearly match the frequencies we compute.

$E4$	$D4$	$C4$
✓		
	✓	
		✓
	✓	
✓		
✓		
✓		
	✓	
	✓	
	✓	
✓		
✓		
✓		
✓		
	✓	
		✓
	✓	
✓		
✓		
✓		
✓		
	✓	
	✓	
✓		
	✓	
		✓

Table 1: Piano score centered on three repeated notes.

$A5$	$G5$	$F5$
✓		
	✓	
		✓
	✓	
✓		
✓		
✓		
	✓	
	✓	
	✓	
✓		
✓		
✓		
✓		
	✓	
		✓
	✓	
✓		
✓		
✓		
✓		
	✓	
	✓	
✓		
	✓	
		✓

Table 2: Recorder score centered on three repeated notes.

A

getNotes() trivial function written for this assignment for generating a list of possible frequencies and their names (i.e. "A4", "Ab4", etc...)

resample(A,B,C) Take B of every C elements of A

mexihat(A,B,C) create a mexican hat filter from the lowerbound A to upperbound B with C points. Guaranteed effective for bounds -5, 5

pcolor(A,B,C) Plot C with respect to A and B axis

nextpow2(A) Get the next power of two value above integer A. Useful for being explicit with length of fft

[sound, Fs] = wavread('file') read in sound file scaled from -1 to 1 into a sound vector with it's recorded frequency Fs.

roundtowardvecA,B, 'round' round the elements of vector A to match the closest element of vector B. Submitted by Tom R on Matlab File Exchange.

B

```
% Author: Karl Marrett
% HW2 AMATH 482
% DUE: Feb. 5th
% solution.m

close all; clear all;

% set default behaviors
amp = 1;  % toggle for amplitude(1) vs. power spectrums
bandpass = 0; % toggle bandpass fitering of gabor transforms
bpWidth = 0; bpCenter = 0; %null values
% bpWidth = 1000; bpCenter = 440; bandpass = 1;
filterType = 'step';  %default

%%%%%%% PART 1 %%%%%%%%%%%

%%% handel sample %%%%%%
load handel
handel = y' / 2;
sampleFactor = 1;
handel = resample(handel, 1, sampleFactor);
Fs = Fs / sampleFactor;

% Raw time trace Handel
figure(1);
plot((1:length(handel))/Fs,handel);
xlabel('Time [sec]');
ylabel('Amplitude');
title('Signal of Interest, v(n)');
set(gcf, 'visible', 'off');
saveas(1, 'handelTimeTrace', 'png');

% Compare Windowing Width (Gaussian Gabor filter)
bpWidth = 1000; bpCenter = 440; bandpass = 1;
iterations = 4;  % trial number
width = linspace(2, 14, iterations); % width of filter (s)
samples = 100; % split window samples evenly spaced

figure(2);
for tnum = 1:iterations
[handelSpec, f, t_0, Nyquist, windowedDF] ...
```

```
= gabor(handel, width(tnum), samples, amp, ...
 bandpass, bpWidth, bpCenter, filterType, Fs);
subplot(2,2, tnum);
pcolor(t_0, f, handelSpec), shading interp,
title(sprintf('Window #: %d Width: %0.2f dF: %.2f', ...
samples, width(tnum), windowedDF));
xlabel('Time (seconds)');
ylabel('Frequency (Hertz)');
ylim([0 Nyquist]);
    ylim([0 1500]);
xlim([0 10]);
end
set(gcf, 'visible', 'off');
saveas(2,'compareWidth','png');

% Compare Sample number of windows (Gaussian Gabor filter)
bpWidth = 1000; bpCenter = 440; bandpass = 1;
iterations = 4;  % trial number
width = 8; % width of filter (s)
% exaggerated undersampling to oversampling
samples = round(linspace(2, 200, iterations)); % split window samples evenly spaced

figure(3);
for tnum = 1:iterations
[handelSpec, f, t_0, Nyquist, windowedDF] ...
=gabor(handel, width, samples(tnum), amp, ...
 bandpass, bpWidth, bpCenter, filterType, Fs);
subplot(2,2, tnum);
pcolor(t_0, f, handelSpec), shading interp
title(sprintf('Window #: %0.1f width: %0.2f', ...
 samples(tnum), width));
xlabel('Time (seconds)');
ylabel('Frequency (Hertz)');
ylim([0 Nyquist]);
xlim([0 11]);
end
set(gcf, 'visible', 'off');
saveas(3,'compareSample','png');

% Compare Filter Type
bpWidth = 0; bpCenter = 0; %null values
width = 4; % width of filter (s)
```

```
iterations = 3;  % trial number
samples = 20; % split window samples evenly spaced
filters = {'gauss', 'mexi', 'step'};

figure(4);
for tnum = 1:iterations
[handelSpec, f, t_0, Nyquist, windowedDF]=gabor(handel, width, samples, ...
 amp, bandpass, bpWidth, bpCenter, filters{tnum}, Fs);
subplot(1,3, tnum);
pcolor(t_0, f, handelSpec), shading interp,
title(sprintf('Filter Type: %s', filters{tnum}));
xlabel('Time (seconds)');
ylabel('Frequency (Hertz)');
ylim([440 600]);
xlim([0 10]);
end
saveas(4,'compareFilters','png');

%%%%%%%% PART 2 %%%%%%%%%%%%%%%%%%

[notes, names] = getNotes();
sampleFactor = 1;

%%% PIANO %%%%%

[piano, Fs] = wavread('music1');
recorder = resample(piano, 1, sampleFactor);
Fs = Fs / sampleFactor;
nPiano=length(piano);
filterType = 'none';
figure(5);
plot((0:(nPiano -1))/Fs, piano);
title('Mary had a little lamb time trace (piano)');
xlabel('Time (seconds)');
xlim([1.00 1.02]);
saveas(5, 'pianoTimbre', 'png');

width = .22; % width of filter (s)
samples = 10; % split window samples evenly spaced
bandpass = 0; % toggle bandpass on
bpWidth = 200; % width of filter
bpCenter = 280;  % center frequency to bandpass around
```

```
[pianoSpec, fPiano, t_0, Nyquist, windowedDF]=gabor(piano, width, ...
samples, amp, bandpass, bpWidth, bpCenter, filterType, Fs);

figure(6);
pcolor(t_0, fPiano, pianoSpec), shading interp, colormap(hot)
xlabel('Time (seconds)');
ylabel('Frequency (Hertz)');
title('Bandpass spectogram (piano)');
xlim([0 16]);
% ylim([220 370]);
ylim([0 1000]);
saveas(6,'pianoGaborBandpass','png');
[pianoScore, pianoPitches] = getScore(pianoSpec, f, notes, names);

% break

%%% RECORDER %%%%%

[recorder, Fs] = wavread('music2');
recorder = resample(recorder, 1, sampleFactor);
Fs = Fs / sampleFactor;
nRecorder=length(recorder);

% Raw trace
figure(7);
plot((0:(nRecorder -1))/Fs, recorder);
title('Mary had a little lamb time trace (recorder)');
xlabel('Time (seconds)');

% filterType = 'gauss';
width = .22; % width of filter (s)
samples = 10; % split window samples evenly spaced
bandpass = 1; % toggle bandpass on
bpWidth = 400; % width of Gaussian filter in frequencies
bpCenter = 440;  % center frequency to bandpass around
[recorderSpec, fRecorder, t_0, Nyquist, windowedDF]=gabor(recorder, width, ...
samples, amp, bandpass, bpWidth, bpCenter, filterType, Fs);

figure(8);
pcolor(t_0, fRecorder, recorderSpec), shading interp, colormap(hot)
xlabel('Time (seconds)');
ylabel('Frequency (Hertz)');
```

```
title('Bandpass Spectogram (recorder)');
% ylim([0 Nyquist]);
xlim([0 16]);
ylim([230 630]);
saveas(8,'recorderGabor','png');
[recorderScore, recorderPitches] = getScore(recorderSpec, f, notes, names);

% compare timbre
figure(9)
subplot(1,2,1)
plot(fPiano, pianoSpec);
title('Piano timbre in the frequency domain');
ylabel('Amplitude')
xlabel('Frequency (Hz)');
xlim([0 2000]);

subplot(1,2,2)
plot(fRecorder, recorderSpec);
title('Recorder timbre in the frequency domain');
ylabel('Amplitude')
xlabel('Frequency (Hz)');
xlim([0 2000]);
saveas(9, 'compareTimbre', 'png');

% gabor.m

function [spec, f, t_0, Nyquist, windowedDF]=gabor(soundVector, width, ...
samples, amp, bandpass, bpWidth, bpCenter, filterType, Fs);
% width is scaled to seconds that the time trace will be windowed with
% bandpass is a toggle to create a bandpass filter of the FFT at each window
% bpwidth and bpCenter are scaled to the set of frequencies
% bpCenter must be within the range of our single sided frequencies 'f'

% round to next highest power of two for fft
n = 2^nextpow2(length(soundVector));  % time points
L = n / Fs; % length in seconds
t_0 = linspace(0, L, samples);
if strcmp(filterType, 'none')
t_0 = 1;
end
t = (0:(n - 1))/Fs; % time (s)
widthSamples = width * Fs;
```

```
windowedDF = Fs / widthSamples;  % frequency resolution after filter
df = (Fs/n);
Nyquist = Fs/2 - df; % max sampling frequency
f_full = (0:(n -1))*df;
f = f_full(1:(n/2)); % single sided
specAmp = [];
specPwr = [];

% ensure soundVector length is n long
temp = zeros(1,n);
temp(1:length(soundVector)) = soundVector;
soundVector = temp;

% gaussian filter scaled to frequencies
if bandpass
bpFilter = stepFilter(floor(bpWidth / df), f_full, floor(bpCenter / df), 'fft', Fs);
end

for jj = 1:length(t_0)
if strcmp(filterType, 'gauss')
% gaussian filter scaled to time
fil = gaussFilter(width, t, t_0(jj), Fs);
elseif strcmp(filterType, 'mexi')
fil = mexiFilter(width, t, t_0(jj), Fs);
elseif strcmp(filterType, 'step');
fil = stepFilter(width, t, t_0(jj), 'time', Fs);
elseif strcmp(filterType, 'none')
fil = ones(1, n);
end
FFT = fft(fil.*soundVector, n);
if bandpass
FFT = FFT.*bpFilter;
end
doublePwr = abs(FFT(1:(n/2+1)).^2)/n;
doubleAmp = abs(FFT(1:(n/2+1)));
Pwr = 2 * doublePwr(1:(n/2)); % take single side double the energy to compensate
Amp = 2 * doubleAmp(1:(n/2));
specPwr = [specPwr; Pwr];
specAmp = [specAmp; Amp];
end

if amp
```

```
spec = specAmp.';
else
spec = specPwr.';
end
end

% mexiFilter.m

function [fil]=mexiFilter(width, t, t_0, Fs);
% width given in seconds
% t time vector of n length
% t_0 the center of the current window
% Fs sampling frequency

fil = zeros(1, length(t));
% scale our time (second) variables to index into fil
scaledWidth = Fs * width;
startInd = floor(t_0*Fs - (scaledWidth / 2));
endInd = floor(t_0*Fs + (scaledWidth / 2) - 1);
% range is arbitrary
temp = linspace(-5, 5, scaledWidth).^2;
% one mexihat function in scaledWidth samples
m = (2/(sqrt(3)*pi^0.25))*exp(-temp/2).*(1-temp);

% assumes either the start or end index is over limit but not both
% add mexihat on top of zeros filter
if (startInd < 1)
trimFrom = 1 + abs(startInd);
startInd = 1;
m = m(trimFrom:length(m));
fil(startInd:(length(m) )) = m;
elseif (endInd > length(t))
% trim excess samples that fall over the index of fil
excess = endInd - length(t);
m = m(1:(length(m) - excess));
fil(startInd:(startInd + length(m) - 1)) = m;
else
fil(startInd:(startInd + length(m) - 1)) = m;
end
assert(length(fil) == length(t), ...
    sprintf('improper fil length: %d t is length: %d', length(fil), length(t)));
end
```

```
% gaussFilter.m

function [fil] = gaussFilter(width, t, t_0, Fs);
% width given in seconds
% t time vector of n length
% t_0 the center of the current window
% Fs sampling frequency
% final

fil = zeros(1, length(t));
% scale our time (second) variables to index into fil
scaledWidth = Fs * width;
startInd = floor(t_0*Fs - (scaledWidth / 2));
endInd = floor(t_0*Fs + (scaledWidth / 2) - 1);
% range is arbitrary
temp = linspace(-5, 5, scaledWidth);
m = exp(-(temp.^2));

% assumes either the start or end index is over limit but not both
% add on top of zeros filter
if (startInd < 1)
trimFrom = 1 + abs(startInd);
startInd = 1;
m = m(trimFrom:length(m));
fil(startInd:(length(m) )) = m;
elseif (endInd > length(t))
% trim excess samples that fall over the index of fil
excess = endInd - length(t);
m = m(1:(length(m) - excess));
fil(startInd:(startInd + length(m) - 1)) = m;
else
fil(startInd:(startInd + length(m) - 1)) = m;
end
assert(length(fil) == length(t), ...
    sprintf('improper fil length: %d t is length: %d', length(fil), length(t)));
end

%stepFilter.m

function [fil] = stepFilter(width, t, t_0, sType, Fs);
% width given in seconds
```

```
% t time vector of n length
% t_0 the center of the current window
% Fs sampling frequency
%  final
fil = zeros(1, length(t));
if strcmp(sType, 'time')
scaledWidth = Fs*width;
startInd = floor(t_0*Fs - (scaledWidth / 2));
endInd = floor(t_0*Fs + (scaledWidth / 2) - 1);
elseif strcmp(sType, 'fft')
startInd = floor(t_0 - (width / 2));
endInd = floor(t_0 + (width / 2) - 1);
else
sprintf('Error in type passed to stepFilter');
end

if (startInd < 1)
startInd = 1;
end
if (endInd > length(t))
endInd = length(t);
end
fil(startInd:endInd) = 1;
assert(length(fil) == length(t), ...
    sprintf('improper fil length: %d t is length: %d', length(fil), length(t)));
end

% getNotes.m

function [notes, names] = getNotes()
% returns a vector of frequencies of all pitches
% across specified octaves and a corresponding
% list of names for each of those notes
% Together they are used as a dictionary
% for finding the score of a sound sample

oct = [55 110 220 440 880];
octName = {'A' 'Bb' 'B' 'C' 'Db' 'D' 'Eb' 'E' 'F' 'Gb'}; %encode by note name

for j = 1:length(oct)
for k = 1:length(octName)
    notes(k + (j - 1)*length(octName)) = oct(j).*2^((k - 1)/12);
```

```
    if (k > 3)
    names{k + (j - 1)*length(octName)} = strcat(octName{k}, int2str(j + 1));
else
    names{k + (j - 1)*length(octName)} = strcat(octName{k}, int2str(j));
end
end
end

% getScore.m

function [score, pitches] = getScore(spec, f, notes, names)

[~, maxIndex] = max(spec);  % get the freq index of the max at each sample
maxFreq = f(maxIndex);  % find frequency each index corresponds to
% round each maximum frequency to the frequency of a known note
pitches = roundtowardvec(maxFreq, notes, 'round');

% find the note name from the frequency
for j = 1:length(pitches)
[ind] = find(notes == pitches(j));
score{j} = names{ind};
end
```

References

[1] The Fft and One Dimension. Fast Fourier Transform (FFT). pages 1–15, 2015.

[2] Labview Help. The Fundamentals of FFT-Based Signal Analysis and Measurement in LabVIEW and LabWindows/CVI. pages 1–11, 2009.

[3] J N Kutz. *Data-Driven Modeling & Scientific Computation: Methods for Complex Systems & Big Data*. OUP Oxford, 2013.

[4] B Parker. *Good Vibrations: The Physics of Music*. Good Vibrations. Johns Hopkins University Press, 2010.