MATH 320 Fall 2010, Homework 3, Audio filtering

Due October 18

1 Motivation

Music and audio signals are especially malleable for the aspiring Matlab programmer. Not only are they fantastically easy to obtain (high-quality, free audio software abounds), they are very conventient to manipulate using Matlab audio file tools. We address two specific problems in this assignment:

1. How to extract a (somewhat) hidden harmony from a piece of music, and
2. How to compress audio by using a lower sampling rate.

The first is sometimes of practical interest: there are improvisational recordings to which it would be useful to obtain sheet music for later reference. More broadly, this project will get you comfortable with doing processing tasks that you might want to do in the future. You are encouraged to collect and analyze your own sound clips using the programs you write – they'll actually be good-quality visualization tools. (Using essentially the same methodology with a different processing chain, you can do sonar imaging from your laptop!)

The second problem, that of compression, is always of practical import. The enabling technology in portable music players has been aggressive compression, via the MP3 standard. We aren't going to do MP3 here, but we'll make use of the famous Nyquist theorem to save on space after the harmony extraction is done.

2 Objectives

After this assignment, you should...

- Understand how to manipulate WAV files in Matlab
- Use FFT-based filters to process a signal
- Use approximation to change the sampling rate on a signal

This project is a bit harder and longer than the previous ones, because we are now interacting with measurements. Don't be discouraged!

3 Your tasks

What you are given: I'll give you the following files:

- lightly_row.wav. This is an audio file in WAV format, taken at a 44.1 kHz sampling rate. **Use this one for your processing, the others are for your reference only!** They can be run through your process, of course, but they're less interesting.
- lightly_row_violin.wav. This is the violin part, so you know what it sounds like.
- lightly_row_cello.wav. This is the cello part, for your reference.
- cello_music.png. The sheet music for the cello part.

The audio files were collected by me in my living room (the kids were asleep!). The piece is a traditional children's song, often used as an introductory lesson for stringed instruments. I'm playing the harmony on the cello, and my wife Donna Dietz (Penn CS department) has the melody on the violin. I used the free program Audacity (http://audacity.sourceforge.net) to record, crop, and clean the raw audio. In particular, the signal from my laptop microphone is pretty tinny (lots more high frequency content than there should be), so I equalized the energy across frequencies by ear. One can do this algorithmically, but I didn't want to mess with it.

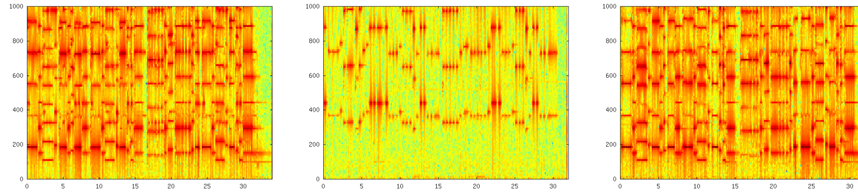
Problem 1:

Write a function with the following input/output specification that will ingest a WAV audio file, organize the samples, and then plot the spectrogram if requested.

```
function [y,yst_,xax,yax]=spectrogram(filename,ts,plot)
% Function to load, compute, and plot spectral data from a file
%
% function [y,yst_,xax,yax]=plotSpectrogram(filename,ts,plot)
%
% Input: filename = WAV file to read
%        ts       = transform size
%        plot     = nonzero if plots desired
% Output: y    = Samples loaded from WAV file
%         yst_ = frequency (rows) versus time (columns)
%         xax  = time (seconds) for each column of yst_
%         yax  = frequency (hertz) for each row of yst_
```

Part 1a: Reading the files.

If you run it on the three audio files I've supplied, you'll see something like each of the plots below.



The horizontal axis is time, over the entire recording. The vertical axis is frequency. The frequency axis in these plots is zoomed way in; there is much more high frequency content that is uninteresting. The one on the left is the duet (with both instruments). The one in the middle is the melody, played by the violin. The one on the right is the harmony, played by the cello.

First of all, you should look up the function *wavread*, which will let you read in a WAV file. It gives you several outputs, but the ones you care about are the samples and the sampling rate. As noted before, you should expect that the sample rate is 44100 (samples per second).

Part 1b: Reshaping the data.

Once you have the audio samples, we want to reorganize them. We could try to process the whole file at once, but this is bad for a number of reasons, not the least of which is that it gives poor estimates of note frequencies. So what we will do is break the file up into small-ish pieces (roughly 0.1 second each) and handle these independently. Just how many samples you use at a time is controlled by the ts parameter your function will take as input. (This should be a power of 2 for optimal speed, but it's not crucial.)

So I suggest that you make your function make an internal variable (not returned as output) yst which is a matrix with ts rows and enough columns to exhaust the data you read from the file. A quick way to do this is to use the command (if your samples are stored in y):

```
yst=reshape(y,ts,[]);
```

This **will not** work exactly. You'll need to add zeros to the end of y to make the length of y evenly divisible by ts.

Part 1c: Computing Fourier coefficients.

Once you've got that covered, the rest is easy. The function fft will give you the Fourier series coefficients (actually the discrete Fourier transform, but don't let that confuse you) of a signal. By reshaping as I suggest above, we want to do many little FFT operations, one along each column. You do this all at once by the command

```
yst_=fft(yst,[],1);
```

The rows of $yst_$ represent **frequency**, and the columns represent **time** (albeit, downsampled a bit from the original). So, you need to cook up scales for these axes if you want to plot them. So your function will compute what the time axis (the output xax) should be (in seconds), or in order words, make

a row vector whose entries indicate the time at which the first sample in each column occurs. Your function also will compute a column vector (the output yax) whose entries are the frequency corresponding to each row of $yst_$. (Read the documentation if you don't know what the frequency bounds are... You can probably guess!)

Part 1d: Plotting.

Use the function *imagesc* to plot the absolute value (or better, the logarithm of the absolute value) of your matrix $yst_$. The axis vectors you just computed can be the first two arguments of *imagesc* to put the correct scales on the plot. You'll notice that the y-axis is upside down! Here's a (somewhat obscure) fix:

```
set(gca,'ydir','normal');
```

Part 1e: Looking at the data.

Now that you have a nice plot of frequency versus time, what can you see in the data? Mark up a plot by identifying the following features:

1. The cello's notes
2. The violin's notes
3. Rests (periods of silence)
4. (important) The frequency of the first note of each instrument
5. The end of the piece
6. Other interesting features. For instance, my cello has a "wolf" which makes some of the notes have substantially more harmonic energy than other notes. Can you identify these?

Problem 2:

Write an approximator using Bernstein polynomials. I suggest that you write a function to compute values of a Bernstein polynomial using the formula for them from the book page 184.

```
function y=bern(n,j,x)
% Bernstein polynomial
%
% function y=bern(n,j,x)
%
% Input: n = degree (scalar)
%        j = index (scalar)
%        x = point to evaluate (vector)
% Output: y = output points (vector)
```

Once you have that, it's easy to write a function that implements the formula for the Bernstein approximant we discussed in class (or in the book, p.184). Important: this formula assumes that the reference samples are evenly sampled in the interval $[0, 1]$, so when you call this function to do approximation, you'll need to scale the requested points xi accordingly.

```
function yi=bernApprox(y,xi)
% Approximate using Bernstein polynomials
%
% function yi=bernApprox(y,xi)
%
% Input: y  = values of the function (at evenly-sampled points on 0..1)
%        xi = points to evaluate approximant at
% Output: yi = values of approximant at xi
```

Problem 3: Resample the audio

Write another function to use your *bernApprox* function to resample an audio signal. You could resample the whole thing at once to avoid discontinuities, but this may be too slow for your liking. So you might want to resample pieces of the signal at a time (probably using *yst*). As a reminder, the time scale that your *bernApprox* function will assume is that 0 is the beginning of the recording at 1 is the end.

```
function newAudio=resampleAudio(oldAudio,oldSampleRate,newSampleRate)
% Resample an audio signal at a different rate
%
% function newAudio=resampleAudio(oldAudio,oldSampleRate,newSampleRate)
%
% Input: oldAudio      = the signal to be resampled (vector, each element is a sample)
%        oldSampleRate = just what it says (samples/second of oldAudio)
%        newSampleRate = just what it says (samples/second of newAudio)
% Output: newAudio = resampled signal
```

Problem 4: Put it all together

Write a script that

1. uses your *plotSpectrogram* function to load and organize the data,
2. filters out the cello line (by zeroing out all of the Fourier coefficients higher than the cello's highest note),
3. resamples it to 8 kHz (down from 44.1 kHz), and finally
4. writes a new WAV file with your results.

A bunch of notes:

- When you use and FFT, there is up-down symmetry in the output. In particular, the lowest frequencies are at both the top *and bottom* rows of the matrix $yst_$. So when you filter, you'll be zeroing out a block in thy.e middle rows. This is a consequence of the Nyquist theorem, that the maximum unambiguous frequency you can resolve is *half* the sampling rate.
- To get back from Fourier coefficients to samples, use the inverse FFT function $ifft$

- After doing your filtering, you'll want to reshape the data into one long column. You can say something like *blah*(:) to get the matrix *blah* into a column.
- Use the function *wavwrite* to write the output file
- Listen to your output file! If it's too quiet, multiply the output by some constant (probably in the range of 10-1000 so that you can hear it! If you hear the violin (or don't hear the cello), adjust which frequencies you zero out. **When it works, it will not sound beautiful. We've gotten rid of the harmonic content that gives the cello it's characteristic timbre. All that is left is a record of the particular notes being played.**

Problem 5 (bonus):

If you're feeling bold, try to identify (automatically) which note is which. In short, can you reconstruct the sheet music I've given you? Using the naïve approach of detecting the peak frequency bin, I can get many (but not all) of the notes. I get faked out when the cello wolf strikes! This shouldn't be hard to fix, but I didn't try very hard.

Collect all of your functions into a directory ("Folder" for the Windows users!), zip it up, and submit using Blackboard. If you want, you can put scans or electronic documents in that zip file, or you can give them to me in class on paper, but the functions must be submitted electronically.