American University - DATA 312

Instructions for Homework 3-C

Unit 3 – Music analysis
Part C – Organizing music

Pre-requisite work:
Please complete Homework 3-B before starting this assignment.

Make sure that the following R libraries are installed using install.packages("<library>",dependencies=TRUE):

- tidyverse
- tabr
- (**NEW**!) audio (for loading WAV files)
- (**NEW**!) GENEAread (for time/frequency analysis)

This assignment uses the same four MIDI files that were included with Homework 3-B, so make sure you still have them available. In addition, you will also need the following file from Canvas:

- `lightly_row_violin.wav`: a rendition of the childrens' song "Lightly Row" on violin, played by Prof. Dietz

Objectives/overview:

This assignment explores the temporal (time) and tonal (frequency) structure of music, by doing some basic statistical analysis and visualizations. It starts out where Homework 3-B left off, with MIDI files, but ends with an analysis of an audio recording in a WAV file.

A good reference for the details of MIDI can be found here:
https://www.cs.cmu.edu/~music/cmsip/readings/MIDI%20tutorial%20for%20programmers.html

Work Process:

1. As usual, start by clearing the workspace and loading libraries:

```
rm(list=ls())
gc()
library(tidyverse)
library(tabr)
```

2. Like in Homework 3-B, load all of the MIDI files

```
bach<-read_midi('bach_846_format0.mid')
edwin_improv<-read_midi('improv.mid')
edwin_improv2<-read_midi('improv2.mid')
duodecatonic<-read_midi('justin_rubin_lyric.mid')
```

3. Also like in Homework 3-B, let's tidy the data and put all the pieces together into one tibble:

```
songs<-bind_rows(mutate(bach,name='bach'),
  mutate(edwin_improv,name='edwin_improv'),
  mutate(edwin_improv2,name='edwin_improv2'),
  mutate(duodecatonic,name='duodecatonic'))
```

It's convenient to use only the "Note On" MIDI events, so lets gather just those, again as in the previous assignments:

```
songs_notes <- songs %>% filter(!is.na(pitch)) %>%
  mutate(as_music_df(pitch),  end_time=time+length)
```

4. How long is each song? We can figure this out by simply looking for the last note time. That basically means that we want to compute max(end_time) for each song. The max() function pays attention to groups, so if we group_by(name), that'll do just what we want.

```
song_lengths <- songs_notes %>%
  group_by(name) %>%
  summarize(total_time=max(end_time))
```

The only tricky thing here is that since max() produces a single value per group, we need to tell R that this is a summary for each group... hence the summarize().

5. Look at the song_lengths table to see how it is organized. You can do other kinds of summaries as well. Perhaps you want to know the highest note in each song. That's easy:

```
songs_notes %>% group_by(name) %>% summarize(max(freq))
```

You can also figure out the lowest note, too. Figure out how!

6. Let's add the length of each song to our data, so that we can use it for later processing:

```
songs_notes <- songs_notes %>% inner_join(song_lengths)
```

7. If you listen to the different MIDI files (as you ought to, if you can!), you'll notice that the Bach piece has two main parts: a calm "prelude", and a more polyphonic "fugue." The other pieces have basically one part. Let's try to pin these parts down in time: find whether there are separate "movements" and when they start.

In order to do this, we are really interested in notes "per unit time". It's easiest (for programming purposes) to split each piece of music up into the same number of "time blocks" and count notes.

```
num_time_blocks <- 10
```

We can (and should) experiment with the number of time blocks to see how that changes things. But for the moment, let's add the time block number to each row of our songs_notes table. Here's how this works: suppose you have twelve evenly space notes at times 1 through 12, and you want four time blocks. We ultimately want a table like

time time_block

1 0
2 0
3 0
4 1
5 1
6 1
7 2
8 2
9 2
10 3
11 3
12 3

So how do we get that? Easy: divide the time by 4 and then round down! The rounding down part is done by the function floor(). So putting the pieces together, we get

```
songs_notes_blocked <- songs_notes %>%
  inner_join(song_lengths) %>%
  mutate(time_block=floor(num_time_blocks*time/total_time))
```

8. Let's see how many notes there are per time block. All we really need to do is count the occurrences of each time_block within each piece. This works each note is a row, so all the notes within a time block all have the same time block number.

```
songs_notes_blocked %>% group_by(name) %>% count(time_block)
```

That's a little hard to read, though, so let's pivot the table out by song name

```
songs_notes_blocked %>%
  group_by(name) %>%
  count(time_block) %>%
  pivot_wider(names_from=c(name),values_from=n)
```

OK, you can definitely see the onset of the Bach fugue starts in time block 6.

9. A little different visualization of the same data can be done by plotting instead of printing:

```
songs_notes_blocked %>%
  group_by(name) %>%
  count(time_block) %>%
  ggplot(aes(time_block,n,color=name)) +
  geom_line()
```

The fugue is definitely visible. The other songs clearly don't have much temporal (time) structure that we can see at this level.

10. Redo Steps 7-9 with some other num_time_blocks values. I suggest at least trying 5, 50 and 100. What features become visible? What features disappear?

11. Let's see what is going on with Bach. There are many more notes in the fugue (second part)... how does that work? Well, first let's check this by plotting

```
songs_notes %>%
  filter(name=='bach') %>%
  ggplot(aes(xmin=time,xmax=end_time,y=freq,color=channel)) +
  geom_linerange()
```

Notice that the fugue part uses more channels, and that the channels are sorted by frequency with channel 5 occupying the lowest notes.

12. If you split the channels into separate facet subplots (Hint: add "+ facet_wrap(~channel)" to the above), you'll see that the different channels are clustered in frequency. Basically, they're notes dedicated to separate "hands".

Let's see if there is any variability in terms of note usage versus time in the different pieces. This is a bit more subtle (and a bit less conclusive in these particular pieces), but let's try anyway!

13. First of all, let's just plot note usage versus time in each piece. There are obviously a lot of notes possible, and the `pitch` column or `frequency` column differentiates all of them. If you try to plot them all, it's a mess. You can do a little better using the `notes` column, since this collapses all the octaves together.

```
songs_notes_blocked %>%
  count(name,note,time_block) %>%
  ggplot(aes(time_block,n,group=note,color=name)) + # count() made the `n` column
  geom_line() +
  facet_wrap(~note)
```

This is a bit busy of a plot, but there are a few things that should jump out at you. First of all, the fugue movement in Bach is definitely visible. (Note: if you don't see the fugue in the plot, try setting num_time_blocks back to 10...)

14. The duodecatonic piece has some interesting structure in Step 13, so let's look specifically at it...

```
songs_notes_blocked %>%
  filter(name=='duodecatonic') %>%
  count(note,time_block) %>%
  ggplot(aes(time_block,n,group=note)) +
  geom_line() +
  facet_wrap(~note)
```

All the notes are used about the same amount, because that was actually the intention of the composer.

Compare this to the "edwin_improv" piece, where the pianist was specifically aiming to be in a certain key. (Keys don't use all the notes!)

Moreover, in the "bach", the fugue uses 'a', 'b', and 'e' much more frequently than the prelude. (Again, if you don't see that, try setting num_time_blocks back to 10.)

15. Our analysis in Step 14 is a bit misleading in the Bach fugue, because we found that there are simply more notes in the fugue than in the prelude. Let's normalize for that.

To normalize, we need to have the note counts per time block (that we computed in Step 8) available to divide the note counts per time block (that we computed in Step 14). This is a bit of a problem because count() defaults to calling both of these `n`. We can fix that by telling count what name to use for a new column. For instance, we can call the notes per time block `notes_per_block` via something like

```
songs_notes_blocked %>% count(name,time_block,name="notes_per_block") # R is smart enough to tell the difference between the name column and the name option...
```

However, as you'll see, count() deleted all the other columns. We needed those! Fortunately, there's a version of count() that doesn't remove all the other columns, called add_count():

```
songs_notes_blocked %>% add_count(name,time_block,name="notes_per_block")
```

OK, with that in hand, we can make a normalized plot

```
songs_notes_blocked %>% filter(name=='bach') %>%
  add_count(name,time_block,name='notes_per_block') %>%
  add_count(note,time_block) %>%
  ggplot(aes(time_block,n/notes_per_block,group=note)) +
  geom_line() +
  facet_wrap(~note)
```

Except for a few spikes, these are pretty flat. So this means that the proportion of each note is kept roughly constant throughout both movements.

16. In these four pieces, the playing speed is pretty constant. You can check this by plotting the note lengths over the time blocks and seeing that they don't vary too much:

```
songs_notes_blocked %>%
  count(name,length,time_block) %>%
  ggplot(aes(time_block,n,color=name)) +
  geom_point()
```

Recall that note lengths measure the time between "Note On" and "Note Off" events.

---- TOPIC SHIFT! ----

You may be wondering what you can do if instead of having a MIDI, you have an audio recording of a piece of music. There are lots of analyses you can run on recordings, though they work quite a bit differently from MIDI. In the second half of this worksheet, we'll be studying music stored in an uncompressed WAV file.

17. You need two libraries for the WAV analysis we'll be doing:

```
library(audio)    # To load the WAV files
library(GENEAread)   # For the analysis in Step 25 onwards
```

BTW, it looks like the GENEAread library is really intended for general timeseries analysis, and also for doing bioinformatics. No matter, the tools work just fine for music, too.

18. The first step is to load the WAV file we'll be using for the rest of this worksheet:

```
wavfile <- load.wave('lightly_row_violin.wav')
```

19. If you look at the wavfile variable, you'll see that it's not a data frame. The View command doesn't work on it. However, it has other structure to it. If you simply type

```
wavfile
```

you'll get a listing of a large pile of numbers! These numbers form a time series; they are measurements of the sound pressure level taken every few moments.

Note: if you use a different WAV file from what's included on Canvas, you may have multiple channels in the file. These correspond to different speakers or microphones. In this case, you need to pick out one channel for the rest of your analysis. You can do this using the syntax

```
wavfile[1,]
```

for the first channel. Replace `wavfile` with that in the following... (Unfortunately, if you only have one channel, then `wavfile[1,]` will give you an error. That's why we don't just use `wavfile[1,]` below...)

What's the time interval between measurements? Well, there are a few standard sample rates used by audio hardware, for instance 8000 Hz and 44100 Hz. The wavfile object knows its sample rate, and can tell you:

```
wavfile$rate
```

You should see that it says 44100, which means that the same rate is 44100 Hz. In other words, the time interval between samples is 1/44100 seconds. Pretty fast! (The 44100 Hz rate corresponds to CD-quality audio. Recording studios often use higher rates in their "master" recordings...)

20. Let's convert the wavfile into a table, so that we can analyze it a little easier. We want our table to have two columns: a `time` column that specifies the time at which a sample was taken, and a `sample` column that records the sound pressure level at that time. If you say

```
sample_table<-tibble(sample=wavfile)  # Be patient!  This might take a while
```

it'll get you most of the way there, but it doesn't have the times. Recall the row_number() function, which will tell you the row number of each column? If we multiply that by the sample interval (= divide it by the sample rate), we will have the time of each sample.

```
wav<-sample_table%>%mutate(time=row_number()/wavfile$rate)
```

21. You could (but shouldn't) try to plot wav directly. Your computer will be unhappy about doing this, and will take a long time. This is because there are 1406976 samples to draw. Plotting every hundredth sample is OK, though. The way to do this is with a slice() command. You tell slice to get every hundredth sample by telling it which rows you want, specifically. **That** is done using the seq() command:

```
wav %>% slice(seq(1,n(),by=100)) %>%
```

```
ggplot(aes(time,sample)) + geom_line()
```

This produces a harmless warning about scaling the y axis. If this bothers you, you can add

```
+ scale_y_continuous()
```

to the end.

Regardless, each spike in the plot you see corresponds to a note being played. If you play the WAV file on your computer, and compare the time within the plot with the time in your player

22. Music involves a mixture of time and frequency together. What step 21 showed you was the time part. Let's now examine the frequency part! The way to do this is with the Fast Fourier Transform or FFT. We won't get into the details of how the FFT works, but here's the general idea. It takes a timeseries as input (a list of measurements, evenly spaced in time) and produces a frequency series (a list of measurements, evenly spaced in frequency) as output. The two series are the same length, and the maximum frequency listed (the last entry in the frequency series) is the sample rate of the time series. Due to a fact called the Shannon-Nyquist sampling theorem, only half of the frequency series turns out to be useful in our analysis. (It's useful in other analysis; it's just that our analysis is a bit simplistic.) OK. Here's how to get an fft:

```
fft(wav$sample) # View() this!
```

That's the frequency series! (The fft() function is part of the stats library, which is loaded by default in R.) It's convenient to also have the frequencies listed alongside the frequency series, so we can reformat the data into a table like we did in Step 20. The columns will be called `sample_fft` for the measurements in the frequency series and `freq` for the frequencies.

```
wav_fft<-tibble(sample_fft=abs(fft(wav$sample))) %>%
  mutate(freq=row_number()/n()*wavfile$rate)
```

One slight wrinkle is that the output of fft() is usually a series of complex numbers. We don't need them to be complex numbers here, so abs() squashes them back into real, positive numbers... that's all we need. (Ask your instructor if you're curious!)

23. Let's plot the frequency series. That's easy, and just what you'd expect, I think:

```
wav_fft %>% slice(seq(1,n(),by=100)) %>%
  ggplot(aes(freq,sample_fft)) + geom_line()
```

You can see right away that there is duplication: the left and right half of the plots are mirror images of each other. Most people can't hear above 15000 Hz anyway, so the duplication (which happens above 22050 Hz) is basically harmless. Furthermore, most music happens below 1000 Hz. Let's zoom in on that region. We don't need to skip every hundred samples any more, too, because there aren't too many samples to plot:

```
wav_fft %>% filter(freq<1000) %>%
  ggplot(aes(freq,sample_fft)) + geom_line()
```

There, you can see several spikes that correspond to individual notes being played. The height (and width to some extent) corresponds both to the number of times a given note was played and to how loud it was.

24. The note frequencies in Step 23 are nice, but it's helpful to show which note is which. Remember that tabr can help us here! Let's make a table of the frequencies for all the notes that should show up on the horizontal axis of the plot in Step 23. The notes involved are

```
notes<-
c("a","b","c","d","e","f","g","a'","b'","c'","d'","e'","f'","g'","a''","b''","c''","d''","e''","f''","g''")
```

Note: each note string is double-quoted, so for instance, "a''" has an **a** followed by **two** single quotes. Gross, but it works.

It's easy to have tabr look the the note frequencies up:

```
freqs<-as_music_df(notes)$freq
```

Notice that I've disposed of the music_df once it's made; we just need the frequency...

The scale_x_continuous function takes both the locations ('breaks') and the labels for the axis:

```
wav_fft %>% filter(freq<1000) %>%
  ggplot(aes(freq,sample_fft)) + geom_line() +
  scale_x_continuous(name='Note', breaks=freqs, labels=notes)
```

You can see which notes were used in the piece, now!

25. If you watched the introductory video for the class, you saw a demonstration of a "spectrogram" of the sound of a cello. Let's make a spectrogram of the WAV file, though it's won't be "in real time" like the video. Spectrograms are made using the Short Time Fourier Transform (STFT), which is a fancier version of the FFT we used in the previous steps. This requires the GENEAread library you loaded in Step 17, as there is an stft() function that it provides. Like the FFT, the STFT takes a timeseries as input. But what it produces is a bit different. The output of the STFT is a timeseries **of frequency series.** Kind of crazy! This is represented by a table, in which each row is the time and each column is for each frequency. (This isn't compatible with tidyverse, so we'll clean it up in Step 26.)

The way the STFT works is that it takes a consecutive block of samples from the original timeseries, and runs the FFT on that to get a frequency series. Then it slides the block forward by a bit and repeats the process. This is similar to the num_time_blocks that we used in the first half of this worksheet. So, you need to tell the STFT two main parameters: the block size (called `win`, in seconds) and how far to advance the block each time (called `inc`, also in seconds). Also, the STFT needs to know the sample rate, which in our case is 44100 Hz.

Given all that setup, the command is easy:

```
wav_stft <- stft(wav$sample,  # The timeseries of sound pressures
                 freq=wavfile$rate,  # The same rate
                 win=0.5,   # Window size is 0.5 seconds; you can change this if you like
                 inc=0.1)    # The advance is 0.1 seconds; you can change this if you like
```

This takes a few moments to run on my computer. The runtime mostly depends on the `inc` parameter. Smaller values take longer to run. You can experiment with the `win` and `inc` parameters to see their effects later on, but try the ones mentioned above to start.

26. The output from the STFT is an rather elaborate R object, and is not very tidy. The data we will need are in wav_stft$values. You should

```
View(wav_stft$values)
```

to see how it's laid out. It is a grid, whose

- rows are indexed by time (no labels at all, but correspond to wav_stft$time)
- columns are indexed by frequency (the labels are in the form of strings, "Vnnn" where the nnn is an index into wav_stft$frequency)

So, let's tidy it up. We'll need a list of frequencies to start with. These are already computed for us, so we could start with

```
tibble(frequency=wav_stft$frequency)  # Don't run this yet!
```

But that is not the end of the story; we need to get these aligned with the column names (which are strings of the form "Vnnn") so that we can (later) do a join with them. We need to add a column to our table with the "Vnnn" names matched up with the frequencies. The way to do this is to attach the "V" character to the row number in the above table using the paste() function.

```
freqtable<-tibble(frequency=wav_stft$frequency)%>%
  mutate(index=paste('V',row_number(),   # This is "V" "some-row-number".
                     sep=''))            # NOT a double quote, but rather two single quotes!
```

Have a look at this table with View to see what we did.

27. Now we're ready to use this frequency table to tidy the STFT output. This is a bit of a multi-step operation, ultimately centered around a "pivot_longer", which rearranges the data from a wide matrix to a narrow table. Rather than explaining it all first, let's look at the whole thing and take it apart bit-by-bit:

```
wav_stft_tidy<-as_tibble(wav_stft$values,name_repair='minimal') %>%  # Render it as a tibble
  mutate(time=wav_stft$times) %>%  # Add the times
  pivot_longer(cols=starts_with('V')) %>%  # Pivot... so that each measurement is its own row
      # The "starts_with" bit selects everything but the new `time` column we just added!
  left_join(freqtable,by=c('name'='index'))%>% # Join in the frequencies
  mutate(name=NULL) # Get rid of the useless "Vnnn" names
```

(Ignore the warning about "name_repair" that happens from the first line! This is because the column names are kind of foolish.... "Vnnn" is not very informative!)

Have a look at the output; it now has three columns: `time`, `frequency`, and `value`.

28. Whew. Now we're ready to plot! It's time to introduce a new geom_, namely geom_raster() which is for making image plots. The STFT output (as a matrix) is actually well-suited for rendering as an

image. We already know that we don't want anything with too high a frequency, so we can start off with a filter to cut off all the high frequency data. The basic command is

```
wav_stft_tidy %>% filter(frequency<1000) %>%
 ggplot(aes(time,frequency)) +
 geom_raster(aes(fill=value))
```

But this probably doesn't look like much to you... that's because the colors are rather poorly chosen. You can fix that by scaling them like this:

```
wav_stft_tidy %>% filter(frequency<1000) %>%
 ggplot(aes(time,frequency)) +
 geom_raster(aes(fill=value),show.legend = FALSE) +
 scale_fill_gradient(low='white',high='black')
```

If you look at the help for scale_fill_gradient() you'll find other color scale options, too. You might prefer them over my boring grayscale...

One of the most interesting features is that the plot appears to be duplicated vertically. If you increase the frequency limit from 1000 Hz to 2000 Hz (try it!) you will see more copies of the notes. These copies are called "harmonics", and are characteristic of the timbre (or unique sound quality) of the instrument. Stringed instruments have lots of harmonics, and larger stringed instruments have a richer collection of them. We don't need to consider them here, but they are an interesting topic in their own right!

29. Let's fancy the frequency scale up, using the note scale rather than the frequency in Hz, like we did in Step 24. Assuming you've already run Step 24, you should have all the variables you need in your workspace. It's easy then! Just add

```
+ scale_y_continuous(name='Note', breaks=freqs, labels=notes)
```

to the end of the command you used in Step 28.

30. Finally, let's make a very basic attempt at detecting the notes. Automatically detecting and transcribing notes is not easy to do, so this will "barely" work, but it's not a bad first guess. We can assume that the note that's being played at any given time corresponds to the frequency with the largest value. Saying that in words isn't to hard, and it translates nicely into commands. We want to filter out the maximum value for each time, so we want to group_by(time), and we want to find where value == max(value), since max() pays attention to the groups. Given that this is a violin playing a rather easy piece, all the note frequencies lie between 256 Hz and 500 Hz.

```
wav_stft_tidy %>% filter(frequency>256,frequency<500) %>%
 group_by(time) %>%
 filter(value==max(value))
```

OK, that should give you a list of times and single frequencies for each. There were some rests in the music; in these places the maximum value of `value` isn't very big. It seems to be around 10, actually, so adding

```
filter(value>10)
```

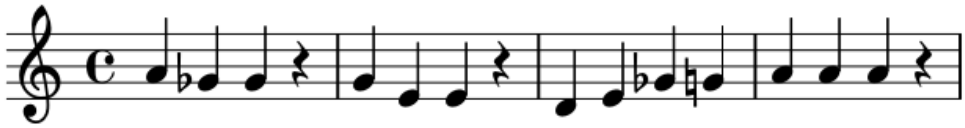
to the end will get rid of all the rests. This is then easy to plot:

```
wav_stft_tidy %>% filter(frequency>256,frequency<500) %>%
  group_by(time) %>%
  filter(value==max(value)) %>%
  filter(value>10) %>%
  ggplot(aes(time,frequency)) +
  geom_point()
```

Ultimately, this is merely the start of trying to automatically transcribe the music. At least what I can read (left to right) is

a' g_' g_' g' e' e' d' e' g_' g' a' a' a' ....

which is actually what was played.



If you wanted to -- say -- produce a MIDI file from this, you'd need to convert frequencies back to notes (not too hard, but notice that the notes jitter a bit in frequency), and determine where to place the "Note Off" events (much harder)... That is not something we'll do here, but ask if you're interested!

Assignment:

For submission, include your .R file as well as any data files (MIDI or WAV files).

This assignment has two options: MIDI or WAV. Select one to submit. You may do both if you like, but only one is required.

MIDI Option:

1. Like in HW3B, please legally obtain two MIDI files for your analysis, and be sure to include them in your upload to Canvas. The files must contain the following properties: (a) have more than 300 notes (b) contain at least two definite "movements", "verses", or other structural changes within them. Multiple channels or tracks are not necessary.
2. Show that your files satisfy property (a) by counting the notes in each file, and determine the length of each file in seconds.
3. Using the num_time_blocks idea discussed in Steps 7 (and following), produce a plot showing the number of notes played over time. Show that your files satisfy property (b) by using a plot or a table.
4. Rerun your analysis using several other num_time_blocks values. Explain what changes you see in your results.

5. Show how the usage of notes changes over time in your files, like Step 15. You may have to make several pages of plots; please make sure to explain (in comments) what the key features are.
6. Does the tempo vary in your piece? Justify your answer using an analysis like Step 16.

WAV Option:

1. Legally obtain two WAV files of music for your analysis. and be sure to include them in your upload to Canvas. (Hint: Wikipedia has many musical WAV files that are legal to use.) The files must contain at least 10 seconds and not more than 2 minutes of music. The upper limit is for your sanity and mine! **It will be much easier if there is only one instrument heard in the file.** (This is not a requirement, though.)
2. Load the file into R, and show a time series plot of the file. Can you see individual notes?
3. Produce a frequency series for the file, zooming in on the most interesting part of the file. Sometimes you will see "harmonics", which are spikes occurring at many multiples of the actual note frequency. Do you see any harmonics?
4. Using the frequency series for the file, what do you think the most common (or possibly important) notes are in the piece? If there are harmonics, you may have difficulty choosing between multiple copies of a note in different octaves. Make a note of this if this is happening.
5. Produce an STFT spectrogram plot for the file. Narrow the frequency range to cover the notes actually being played, or if you can't, please explain why! (Hint: you might have trouble if there are many harmonics)
6. Attempt to read off the first few notes of the song, much like was done in Step 30.