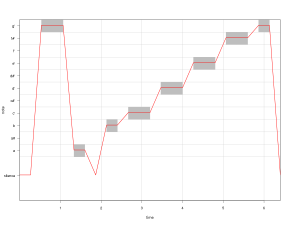
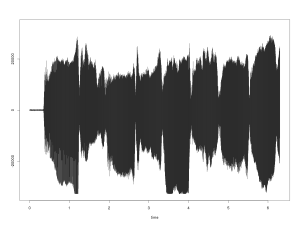
My goal was to extract the notes from an example sound file and then record myself playing it to see how close my notes are to the original ones and repeat this daily to see if I am getting better. The below script helps in *tRanscribing music from audio files*.

Code Chunks – *tRanscribing music from audio files*

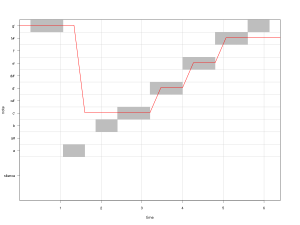
The resulting music annotation for G major scale was quite impressive (below, right). As you can see on the right figure below, transcribed notes almost exactly matched the notes originally played.

[](https://www.vesnam.com/Rblog/transcribing-music-from-audio-files-2/scale/)

G major scale  
Red lines – transcribed notes; Gray lines – notes played

[](https://www.vesnam.com/Rblog/transcribing-music-from-audio-files-2/scale_plot/)

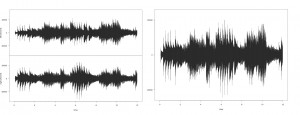
G major scale

[](https://www.vesnam.com/Rblog/transcribing-music-from-audio-files-2/scale_smooth/)

G major scale after applying the “smoother” function

Interestingly, when I applied the smoothing function (“*smoother*“) to estimated notes, the resulting annotation was not equally impressive (see left). My assumption is that the “median” function, currently the only option for smoothing, is  the main reason for such not-so-great performance.

As some of the tuneR functions required for a music transcription currently work only with the *mono*sounds, my second test included a stereo sound from an MP3 file. To transform a stereo sound into a mono sound, I had three options: to use left channel, to use right channel, or to use the average value of both channels.

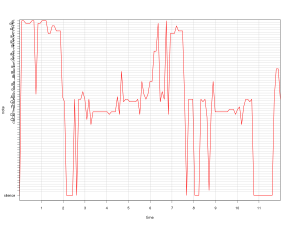
[](https://www.vesnam.com/Rblog/transcribing-music-from-audio-files-2/musicboth/)

Left (top left), right (bottom left), and averaged (average value of the left and right) channels

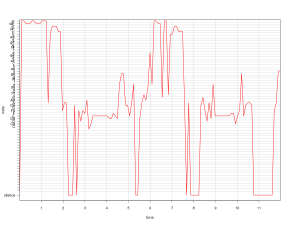
The average channel option sounded the most intuitive for me. However, it was hard to say whether the averaging would enhance the signal or whether it would introduce more noise. As it takes only a few seconds to get the annotation from a music file, I decided to test all three versions.

The resulting annotation for *For Elise* was not as impressive as for the G scale (this was expected), but the overall results were still quite good, especially if we take into consideration the difference in the tempo and complexity between these two.  You can Google for the *Für Elise* sheet music if you want to compare notes.

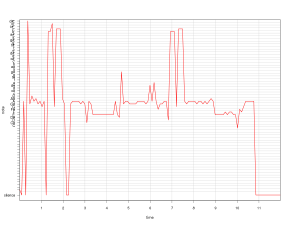
The averaged channel produced the best results. Notes derived from the right channel were not too much off either. On the other hand, notes derived from the left channel did not match the original notes especially well. I don’t know if there is a specific technical reason behind this or it is just the consequence of the audio compression in the MP3 file.

[](https://www.vesnam.com/Rblog/transcribing-music-from-audio-files-2/average_original/)

Annotation derived from the averaged channel

[](https://www.vesnam.com/Rblog/transcribing-music-from-audio-files-2/right/)

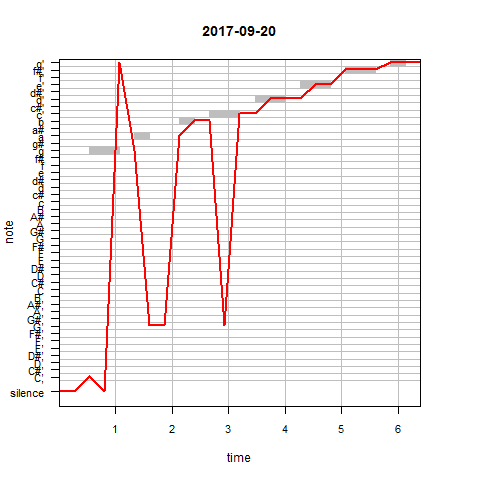
Annotation derived from the right channel

[](https://www.vesnam.com/Rblog/transcribing-music-from-audio-files-2/left/)

Annotation derived from the left channel

Here is the R code I used to transcribe the music:

|  |  |
| --- | --- |
|  | library("tuneR") |
|  |  |
|  | transcribeMusic <- function(wavFile, widthSample = 4096, expNotes = NULL) |
|  | { |
|  | #See details about the wavFile, plot it, and/or play it |
|  | #summary(wavFile) |
|  | plot(wavFile) |
|  | #play(wavFile, "/usr/bin/mplayer") |
|  |  |
|  | perioWav <- periodogram(wavFile, width = widthSample) |
|  | freqWav <- FF(perioWav) |
|  | noteWav <- noteFromFF(freqWav) |
|  |  |
|  | #Smooth frequecies |
|  | #noteWav <- smoother(noteWav) |
|  |  |
|  | melodyplot(perioWav, observed = noteWav, expected = expNotes, plotenergy = FALSE) |
|  |  |
|  | #Print out notes names |
|  | #noteWavNames <- noteWav[!is.na(noteWav)] |
|  | #notenames(noteWavNames) |
|  | } |
|  |  |
|  | #Test1 (mono) |
|  | testSound <- readWave("G-scale.wav") |
|  | scaleNotesFreqs <- c(NA, NA, NA, 392.0, 392.0, NA, 220.0, NA, NA, 246.9, NA, 261.6, 261.6, NA, 293.7, 293.7, NA, 329.6, 329.6, NA, 370.0, 370.0, NA, 392.0, NA) |
|  | scaleNotes <- noteFromFF(scaleNotesFreqs) |
|  | transcribeMusic(testSound, expNotes = scaleNotes) |
|  |  |
|  | #Now let see what happen if music is in stereo |
|  | #Test2 - 1 |
|  | songHlp <- readMP3("FE.mp3") |
|  | testSound\_stereo <- extractWave(songHlp, from = 0, to = 12, xunit = "time") |
|  | testSound <- mono(testSound\_stereo, "both") #average left and right channel |
|  | transcribeMusic(testSound) |
|  |  |
|  | #Test2 - 2 (use each channel separately) |
|  | testSound\_lc <- channel(testSound\_stereo, which = "left") |
|  | testSound\_rc <- channel(testSound\_stereo, which = "right") |
|  | transcribeMusic(testSound\_lc) |
|  | transcribeMusic(testSound\_rc) |

One of the examples used was the recording of a G scale played by a violin. Since the melody is easy to repeat and it is recorded with the right instrument – this [G-scale.wav](http://www.vesnam.com/Rblog/wp-content/uploads/2013/01/G-scale.wav) was actually perfect for my objective. Therefore, I decided to track my progress by playing the G-scale every day.  
The following R scripts can also be used with other .wav files as input (I also switched to other scales now) – however for this to work, the file needs to be in mono, with a bit resolution of 16 bit and a sampling rate of 16000 Hz. Not to worry though, converting a recording to those encoding settings is very easy with the [online-converter](https://audio.online-convert.com/convert-to-wav).  
[](https://datascienceplus.com/wp-content/uploads/2018/03/gscales_transcribed4.gif)

**Melody plots of daily G-scale recordings**

To see the full scripts and functions that I have been using, please see the ProgressAnalysis.R and the ProgressAnalysisFunctions.R files. In what follows, I will explain the process step by step.

ProgressiveAnalysis.R

|  |
| --- |
|  |
| setwd("SET WORKING DIRECTORY") |
|  |  |
|  | library(audio) |
|  | library(tuneR) |
|  |  |
|  | source("ProgressAnalysisFunctions.R") # The audiorec function in this script will look |
|  | # for a folder named "recordings" in the working directory and the transcribeMusic function |
|  | # looks for a folder named "plots" to save the plots in. |
|  |  |
|  | tuneR::play(originalSound) |
|  |  |
|  | audiorec(6.3, filename) |
|  |  |
|  | ### Wait running the script from here until after the restart! ### |
|  |  |
|  | testSound <- readWave(filename) |
|  | tuneR::play(testSound) |
|  |  |
|  | results <- transcribeMusic(testSound, expNotes = scaleNotes) |
|  |  |
|  | performance <- updatePerformance(results) |
|  |  |
|  | plotPerformance(performance) |
|  | plotProgress(performance, by = "session") |
|  | plotProgress(performance[performance$expected\_notenames != "g",], by = "session") |
|  | plotProgress(performance, by = "date") |
|  | plotProgress(performance[performance$expected\_notenames != "g",], by = "date") |
|  | plotProgress(performance, by = "expected\_notenames") |
|  | plotProgress(performance[performance$expected\_notenames != "g",], by = "expected\_notenames") |
|  |  |

ProgressiveAnalysisFunction.R

|  |
| --- |
| # Progression Analysis Functions |
|  |  |
|  | date = gsub(":","-",as.character(Sys.time())) |
|  |  |
|  | filename = paste0("recordings/",date,"\_recording.wav") |
|  |  |
|  | originalSound <- readWave("G-scale.wav") |
|  |  |
|  | scaleNotesFreqs<- c(NA, NA, NA, 196.00, 196.00, NA, 220.0, NA, NA, 246.9, NA, 261.6, 261.6, NA, 293.7, 293.7, NA, 329.6, 329.6, NA, 370.0, 370.0, NA, 392.0, NA) |
|  | scaleNotes <- noteFromFF(scaleNotesFreqs) |
|  |  |
|  | expected\_notes <- c(-14,-14,-14,-12,-12,-10,-10,-9,-9,-9,-7,-7,-7,-5,-5,-5,-3,-3,-3,-2,-2) |
|  | expected\_notenames <- notenames(expected\_notes) |
|  |  |
|  | # audio recording from https://stackoverflow.com/questions/22619561/audio-record-in-r |
|  | audiorec <- function(kk,f){ # kk: time length in seconds; f: filename |
|  | if(f %in% list.files()) |
|  | {file.remove(f); print('The former file has been replaced');} |
|  | require(audio) |
|  | s11 <- rep(NA\_real\_, 16000\*kk) # rate=16000 |
|  | message("5 seconds..") |
|  | for (i in c(5:1)){ |
|  | message(i) |
|  | Sys.sleep(1) |
|  | } |
|  | message("Recording starts now...") |
|  | record(s11, 16000, 1) # record in mono mode |
|  | wait(kk) |
|  | save.wave(s11,f) |
|  | .rs.restartR() |
|  | } |
|  |  |
|  | #http://www.vesnam.com/Rblog/transcribing-music-from-audio-files-2/ |
|  |  |
|  | widthSample = 4096 |
|  | expNotes = NULL |
|  |  |
|  | transcribeMusic <- function(wavFile, widthSample = 4096, expNotes = NULL) { |
|  | #See details about the wavFile, plot it, and/or play it |
|  | plot(wavFile) |
|  |  |
|  | perioWav <- periodogram(wavFile, width = widthSample) |
|  | freqWav <- FF(perioWav) |
|  | noteWav <- noteFromFF(freqWav) |
|  |  |
|  | melodyplot(perioWav, observed = noteWav, expected = expNotes, plotenergy = FALSE, main = Sys.Date()) |
|  |  |
|  | dev.copy(png, paste0("plots/plot\_",date,".png")) |
|  | dev.off() |
|  |  |
|  | #Print out notes names |
|  | noteWavNames <- noteWav[!is.na(noteWav)] |
|  | noteWavNames <- noteWavNames[1:21] |
|  | print(noteWavNames) |
|  | print(notenames(noteWavNames)) |
|  | return(noteWavNames) |
|  | } |
|  |  |
|  | updatePerformance <- function(results){ |
|  | files <- list.files() |
|  | if (("performance.csv" %in% files) == FALSE){ |
|  | message("No performance csv existing yet - creating it now...") |
|  | dat <- as.data.frame(results) |
|  | names(dat) <- "noteWavNames" |
|  | dat$notenames <- notenames(results) |
|  | dat$expected <- expected\_notes |
|  | dat$expected\_notenames <- expected\_notenames |
|  | dat$date <- as.character(Sys.Date()) |
|  | dat$rownum <- row.names(dat) |
|  | dat$session <- 1 |
|  | performance <- dat |
|  | write.csv2(performance, "performance.csv", row.names = FALSE) |
|  | print("Done!") |
|  | return(performance) |
|  | } else { |
|  | performance <- read.csv2("performance.csv", stringsAsFactors = FALSE) |
|  | dat <- as.data.frame(results) |
|  | names(dat) <- "noteWavNames" |
|  | dat$notenames <- notenames(results) |
|  | dat$expected <- expected\_notes |
|  | dat$expected\_notenames <- expected\_notenames |
|  | dat$date <- as.character(Sys.Date()) |
|  | dat$rownum <- row.names(dat) |
|  | session\_id <- performance[nrow(performance),"session"] + 1 |
|  | dat$session <- session\_id |
|  | performance <- rbind(performance, dat) |
|  | write.csv2(performance, "performance.csv", row.names = FALSE) |
|  | print("Done!") |
|  | right <- notenames(results)[notenames(results) %in% expected\_notenames] |
|  | print(paste0("Your score (% of correct notes): ",round(length(right)/length(results)\*100,1),"%")) |
|  | return(performance) |
|  | } |
|  | } |
|  |  |
|  | plotPerformance <- function(performance){ |
|  | plot(performance$noteWavNames, type = "l", col = "red", main = "Performance (green = expected & red = you)") |
|  | lines(performance$expected,col="green") |
|  | } |
|  |  |
|  | plotProgress <- function(performance, by){ |
|  | progress <- c() |
|  |  |
|  | for (i in unique(performance[,by])){ |
|  | print(i) |
|  | dat <- performance[performance[,by] == i,] |
|  | dat$res <- dat$expected-dat$noteWavNames |
|  | mse <- mean(dat$res^2) |
|  | print(mse) |
|  | progress <- c(progress,mse\*-1) |
|  | } |
|  |  |
|  | plot(progress, type = "l", yaxt="n", xaxt="n",ylim = c(min(progress),0), lwd = 2, col = "tomato", xlab = by, ylab = "accuracy", main = paste0("G-Scale Accuracy (",unique(performance$date[performance$session == min(performance$session)])," - ", unique(performance$date[performance$session == max(performance$session)]),")")) |
|  | axis(2, at = 0, labels="100%", las=2) |
|  | axis(1, at = c(1:length(unique(performance[,by]))),labels = unique(performance[,by])) |
|  | } |

**Read the sound file into R**

First of all, let's read the original G-scale.wav file into R. This turned out to be surprisingly easy with the tuneR package (I can confirm that this works on my PC – I have a 32 bit Windows 10 laptop):

#install.packages("tuneR")

library(tuneR)

originalSound <- readWave("G-scale.wav")

play(originalSound) # opens Windows Media Player to play the sound. Beware that the window does not automatically close when the audio file is finished playing. You need to close the window before you can continue in R.

Note how the last line of this code snippet is actually playing the sound file straight from R.

**Extract the frequencies and notes from the sound**

My next step is to extract the frequencies and the notes from this sound. With the expNotes parameter you can visualize the expected notes in the plot (for the *Singstar feeling*).

library(tuneR)

scaleNotesFreqs<- c(NA, NA, NA, 196.00, 196.00, NA, 220.0, NA, NA, 246.9, NA, 261.6, 261.6, NA, 293.7, 293.7, NA, 329.6, 329.6, NA, 370.0, 370.0, NA, 392.0, NA)

scaleNotes <- noteFromFF(scaleNotesFreqs)

transcribeMusic <- function(wavFile, widthSample = 4096, expNotes = NULL) {

#See details about the wavFile, plot it, and/or play it

#summary(wavFile)

plot(wavFile)

perioWav <- periodogram(wavFile, width = widthSample)

freqWav <- FF(perioWav)

noteWav <- noteFromFF(freqWav)

melodyplot(perioWav, observed = noteWav,

expected = expNotes, plotenergy = FALSE,

main = Sys.Date())

#Print out notes names

noteWavNames <- noteWav[!is.na(noteWav)]

noteWavNames <- noteWavNames[1:21] # I limited the number of notes to 21 here - because that is the number of notes extracted from the G-Scale.wav file and to make comparisons later I need the extractions to be of the same length.

print(noteWavNames)

print(notenames(noteWavNames))

return(noteWavNames)

}

transcribeMusic(originalSound, expNotes = scaleNotes)

*# [1] -2 -2 -2 -12 -12 -10 -10 -9 -9 -9 -7 -7 -7 -5 -5 -5 -3 -3 -3 -2 -2*

*# [1] "g'" "g'" "g'" "a" "a" "b" "b" "c'" "c'" "c'" "d'" "d'" "d'" "e'" "e'"*

*# [16] "e'" "f#'" "f#'" "f#'" "g'" "g'"*

**Record yourself in R**

Now we know how to read a sound file into R and how to extract the notes from it. What I want to do though, is to extract the notes from my own recordings right after I played them. With the audio package it is possible to record yourself in R. I could make the following function work to record myself:

#install.packages("audio")

library(audio)

audiorec <- function(kk,f){ # kk: time length in seconds; f: filename

if(f %in% list.files())

{file.remove(f); print('The former file has been replaced');}

require(audio)

s11 <- rep(NA\_real\_, 16000\*kk) # Samplingrate=16000

message("5 seconds..") # Counting down 5 seconds befor the recording starts

for (i in c(5:1)){

message(i)

Sys.sleep(1)

}

message("Recording starts now...")

record(s11, 16000, 1) # record in mono mode

wait(kk)

save.wave(s11,f)

.rs.restartR() # As mentioned in the above cited post: recording with the audio package works once, but for some reason it will not continue to work afterwards unless the R session is restarted. For this reason I included a restart in this function. I am hoping to find a more elegant solution one day soon.

}

]

I want the original recording and my version of it to be comparable – therefore, I need my recording to have exactly the same duration as the original sound file. Let's inspect the originalSound file:

summary(originalSound)

*#Wave Object*

*# Number of Samples: 100800*

*# Duration (seconds): 6.3*

*# Samplingrate (Hertz): 16000*

*# Channels (Mono/Stereo): Mono*

*# PCM (integer format): TRUE*

*# Bit (8/16/24/32/64): 16*

*#*

*#Summary statistics for channel(s):*

*#*

*# Min. 1st Qu. Median Mean 3rd Qu. Max.*

*#-32770.00 -6218.00 106.00 5.79 6830.00 29260.00*

We can see it has a duration of 6.3 seconds and a sampling rate of 16000 Hertz (corresponds to what is specified in our recording function). Quick sidenote: if you want to learn more about the digitalization of sound and what exactly the sampling rate is. Let's continue with the recording. Now that we know the duration that our recording should have, we can run:

# Here I create a unique filename with the current date and time (to avoid overwriting earlier recordings)

date = gsub(":","-",as.character(Sys.time()))

filename = paste0(date,"\_recording.wav")

# start the actual recording

audiorec(6.3, filename)

### Wait for R to restart! ###

We need a quick restart after using the record function from the audio package. Normally (at least in my experience) the objects should appear again in the local R environment (otherwise it can be quickly solved by rerunning some parts of the script and looking up the filename of the recording).

**Explore the results**

Now we can extract the notes from the recording and compare them to the notes from the originalSound:

testSound <- readWave(filename)

tuneR::play(testSound)

results <- transcribeMusic(testSound, expNotes = scaleNotes)

# the G-scale notes from the original sound in the same format:

expected\_notes <- c(-14,-14,-14,-12,-12,-10,-10,-9,-9,-9,-7,-7,-7,-5,-5,-5,-3,-3,-3,-2,-2)

expected\_notenames <- notenames(expected\_notes)

Since I want to keep track of my recordings over a longer period of time, I collect the results of each recording session in a csv file. To automate this I used the following updatePerfomance() function:

updatePerformance <- function(results){

files <- list.files()

if (("performance.csv" %in% files) == FALSE){

message("No performance csv existing yet - creating it now...")

dat <- as.data.frame(results)

names(dat) <- "noteWavNames"

dat$notenames <- notenames(results)

dat$expected <- expected\_notes

dat$expected\_notenames <- expected\_notenames

dat$date <- as.character(Sys.Date())

dat$rownum <- row.names(dat)

dat$session <- 1 # every recording gets a unique session ID

performance <- dat

write.csv2(performance, "performance.csv", row.names = FALSE)

print("Done!")

return(performance)

} else {

performance <- read.csv2("performance.csv", stringsAsFactors = FALSE)

dat <- as.data.frame(results)

names(dat) <- "noteWavNames"

dat$notenames <- notenames(results)

dat$expected <- expected\_notes

dat$expected\_notenames <- expected\_notenames

dat$date <- as.character(Sys.Date())

dat$rownum <- row.names(dat)

session\_id <- performance[nrow(performance),"session"] + 1

dat$session <- session\_id

performance <- rbind(performance, dat)

write.csv2(performance, "performance.csv", row.names = FALSE)

print("Done!")

return(performance)

}

}

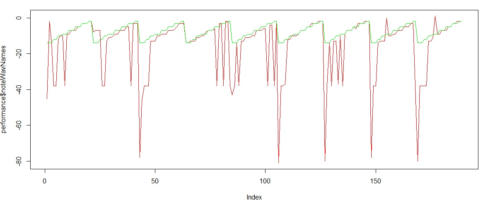
# Now we can add the results of the recording to our performance csv:

performance <- updatePerformance(results)

As a simple visualisation, we can plot the expected notes with our notes in the same graph to see how far off the notes of the recordings were:

plot(performance$noteWavNames, type = "l", col = "red") # the red line shows what I played

lines(performance$expected,col="green") # the green line shows what was expected

[](https://datascienceplus.com/wp-content/uploads/2018/03/performance_plot.png)

To measure the accuracy of the recorded G-scale, we can calculate the difference between numeric extractions of our notes and the expected notes. This means that we can treat the recorded notes as an estimation ($ŷ$) of the expected notes ($y$) and calculate the residuals from them. With the residuals, the MSE (Mean Squared Error) can be calculated per session, per day and also per note. The closer to zero the MSE, the more accurate our recreation of the G-scale is.

plotProgress <- function(performance, by){ # we can pass in the performance df and define the variable by which we want to calculate the accuracy (MSE)

progress <- c()

for (i in unique(performance[,by])){

print(i)

dat <- performance[performance[,by] == i,]

dat$res <- dat$expected-dat$noteWavNames

mse <- mean(dat$res^2)

print(mse)

progress <- c(progress,mse\*-1) # To make the visualizations of the MSE a bit more intuitive to read - I am converting them from positive to negative numbers (So when a value is closer to 0, the line goes up instead of down.)

}

plot(progress, type = "l", yaxt="n", xaxt="n",ylim = c(min(progress),0), lwd = 2, col = "tomato", xlab = by, ylab = "accuracy", main = paste0("G-Scale Accuracy (",unique(performance$date[performance$session == min(performance$session)])," - ", unique(performance$date[performance$session == max(performance$session)]),")"))

axis(2, at = 0, labels="100%", las=2)

axis(1, at = c(1:length(unique(performance[,by]))),labels = unique(performance[,by]))

return(progress)

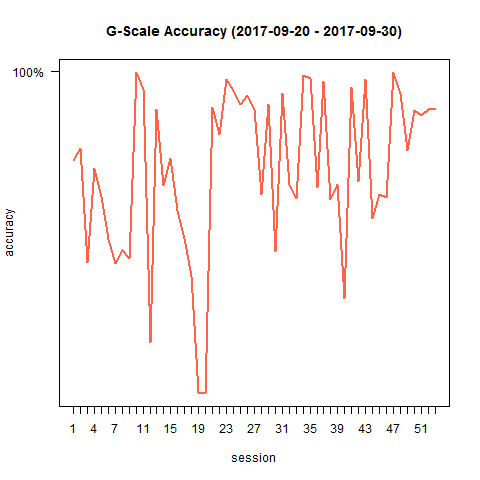
}

As soon as we have multitude of recording sessions in our performance dataframe, we can compare the accuracy of different sessions by plotting them:

plotProgress(performance, by = "session")

plotProgress(performance, by = "date")

plotProgress(performance[performance$expected\_notenames != "g",], by = "expected\_notenames") # The G is often the most "off" note as it is the first and the last one and suffers the most from noise/irregularities. I decided to filter it out so the other notes can be inspected more properly. I can see the E is my most accurate note.

[](https://datascienceplus.com/wp-content/uploads/2018/03/accuracy_session_day_notes.gif)

I can see a lot of ups and downs in the scores of my different sessions, but I am happy to see an overall increase in my accuracy of playing the G-scale. To take away the ups and downs and visualize the progress with a smooth line instead, we can use geom\_smooth() from ggplot2:

progress <- plotProgress(performance, by = "session")

library(ggplot2)

progress <- as.data.frame(progress)

progress$id <- unique(performance$session)

ggplot(progress, aes(id,progress)) +

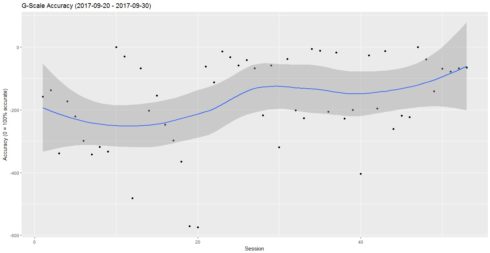
geom\_point() +

geom\_smooth() +

ggtitle(paste0("G-Scale Accuracy (",unique(performance$date[performance$session == min(performance$session)])," - ", unique(performance$date[performance$session == max(performance$session)]),")"))+

ylab("Accuracy (0 = 100% accurate)") +

xlab("Session")

[](https://datascienceplus.com/wp-content/uploads/2018/03/g-scale_accuracy_smooth.jpg)

As you can see, tracking your musical progress with R is not only possible but also fun. Besides, it's a great learning experience, and a good chance to explore topics like the digitalization of sound, the translation of frequencies into notes and the meaning of sampling rates.