Guidelines to Run Montreal Forced Aligner for a PCIbex Experiment

Utku Turk

Table of contents

1 Introduction

We have our data from PCIbex and our zip files from the server. Let's assume we have unzipped them and converted them from `.webm` to `.wav` files. Now, it's time to align them using MFA. But before that, we need to prepare our files accordingly. Here are the steps we will follow in this document:

- Load the PCIbex results
- Filter out irrelevant sound files
- Move all of our `.wav` and `.TextGrid` files to the same directory
- Rename our files according to MFA guidelines
- Run MFA
- Create a dataframe

Before we start, let me load my favorite packages. The `library()` function loads the packages we need, assuming they are already installed. If not, use the `install.packages()` function to install them. While `library()` does not require quotes, you should use quotes with `install.packages()`, e.g., `install.packages("tidyverse")`. If it asks you to select a mirror from a list, choose a location geographically close to you, such as the UK.

```
library(tidyverse) # I have to have tidyverse
library(stringr) # to manipulate string
library(readtextgrid) # to read TextGrid files
library(dplyr)
```

2 PCIbex Results

2.1 Read the results

The main reason we are loading PCIbex results is because sometimes we use the `async()` function in our PCIbex code. The `async()` function allows us to send recordings to our server whenever we want without waiting for the end of the experiment. Even though it is extremely helpful in reducing some of the server-PCIbex connection load at the end of the experiment, it also creates some pesky situations. For example, if a participant decides not to complete their experiment, we will still end up with some of their recordings. We do not want participants who were window-shopping, mainly because we are not sure about the quality of their data. Luckily for us, PCIbex only saves the results of participants who complete the entire experiment.

To read the PCIbex results, we are going to use the function provided in the PCIbex documentation. Scroll down on that page, and you will see the words "Click for Base R Version." The function is provided there as well. Moreover, please be careful whenever you are copying and pasting functions from this file, or any file, as sometimes PDF or HTML files can include unwanted elements, like a page number.

```
# User-defined function to read in PCIbex Farm results files
read.pcibex <- function(
    filepath,
    auto.colnames=TRUE,
    fun.col=\(col,cols){cols[cols==col]<-paste(col,"Ibex",sep=".");return(cols)}
    ) {
  n.cols <- max(count.fields(filepath,sep=",",quote=NULL),na.rm=TRUE)
```

```
  if (auto.colnames){
    cols <- c()
    con <- file(filepath, "r")
    while ( TRUE ) {
      line <- readLines(con, n = 1, warn=FALSE)
      if ( length(line) == 0) {
        break
      }
      m <- regmatches(line,regexec("^# (\\d+)\\. (.+)\\.$",line))[[1]]
      if (length(m) == 3) {
        index <- as.numeric(m[2])
        value <- m[3]
        if (is.function(fun.col)){
         cols <- fun.col(value,cols)
        }
        cols[index] <- value
        if (index == n.cols){
          break
        }
      }
    }
    close(con)
    return(read.csv(filepath, comment.char="#", header=FALSE, col.names=cols))
  }
  else{
    return(read.csv(filepath, comment.char="#", header=FALSE, col.names=seq(1:n.cols)))
  }
}
```

So, now what we have to do is load our file. I also want to check my file using the `str()` function. Please run `?str` to see what this function does. For any function that you do not understand, you can run the `?` operator to see the help pages and some examples.

```
ibex <- read.pcibex("./octo-recall-ibex.csv")
str(ibex)
```

```
'data.frame':   15265 obs. of  23 variables:
 $ Results.reception.time          : int  1716599254 1716599254 1716599254 1716599254 1716599254 1716599254 1716599254 1716599
 $ MD5.hash.of.participant.s.IP.address: chr  "eb9865e3e6ad96f9ff9db3a12b5565ae" "eb9865e3e6ad96f9ff9db3a12b5565ae" "eb9865
 $ Controller.name                 : chr  "PennController" "PennController" "PennController" "PennController" ...
 $ Order.number.of.item                : int  1 1 1 1 1 1 1 1 1 1 ...
 $ Inner.element.number                : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Label                               : chr  "consent_form" "consent_form" "consent_form" "consent_form" ...
 $ Latin.Square.Group                  : chr  "NULL" "NULL" "NULL" "NULL" ...
 $ PennElementType                     : chr  "PennController" "PennController" "PennController" "Html" ...
 $ PennElementName                     : chr  "0" "0" "0" "consent" ...
 $ Parameter                           : chr  "_Trial_" "_Header_" "_Header_" "prolificID" ...
```

```
 $ Value                              : chr  "Start" "Start" "End" "66481a6d648519a547aebd68" ...
 $ EventTime                        : chr  "1716597822554" "1716597822554" "1716597822554" "1716597873803" ...
 $ w4                                 : chr  "undefined" "undefined" "undefined" "undefined" ...
 $ w3                                 : chr  "undefined" "undefined" "undefined" "undefined" ...
 $ w2                                 : chr  "undefined" "undefined" "undefined" "undefined" ...
 $ w1                                 : chr  "undefined" "undefined" "undefined" "undefined" ...
 $ nw                                 : chr  "undefined" "undefined" "undefined" "undefined" ...
 $ itemnum                            : chr  "undefined" "undefined" "undefined" "undefined" ...
 $ trigger_type                       : chr  "undefined" "undefined" "undefined" "undefined" ...
 $ trigger                            : chr  "undefined" "undefined" "undefined" "undefined" ...
 $ verb_type                          : chr  "undefined" "undefined" "undefined" "undefined" ...
 $ head                               : chr  "undefined" "undefined" "undefined" "undefined" ...
 $ Comments                           : chr  "NULL" "NULL" "NULL" "text input" ...
```

Now, what I want to do is to get to filenames that are recorded in the PCIbex results. Before doing that, I advise you to go to this documentation and read more about PCIbex under the Basic Concepts header.

Column	Information
1	Time results were received (seconds since Jan 1 1970)
2	MD5 hash identifying subject. This is based on the subject's IP address and various properties of their browser. Together with the value of the first column, this value should uniquely identify each subject.
3	Name of the controller for the entity (e.g. "DashedSentence")
4	Item number
5	Element number
6	Label. Label of the newTrial()
7	Latin.Square.Group. The group they are assigned to.
8	PennElementType. Name of the specific element, like "Html", "MediaRecorder"
9	PennElementName. Name we have given to the specific Penn Elements.
10	Parameter. This is about what type of element the script is running and saving as a parameter.
11	Value. Value saved for the parameters in column 10.
12	EventTime. Time that specific Element is screened or any action taken with that Element (seconds since Jan 1 1970)

2.2 Filter the results

Since we are dealing with media recordings, I will first filter the file using the `PennElementType` column and will only select rows with "MediaRecorder" values in that column.

```
ibex <- ibex |> filter(PennElementType == "MediaRecorder")
unique(ibex$Value)[1:3]
```

```
[1] "test-recorder.webm" "1_prac-1_jztb.webm" "2_prac-1_jztb.webm"
```

After checking my `Value` column where the file names for `MediaRecorder` are stored, I realize that this will not be enough given that we still have other unwanted elements like `test-recorder.webm` file or some practice files. There are multiple ways to get rid of these files, and you have to think about how to get rid of them for your own specific dataframe. For my own data, I will filter my data utilizing the labels I provided in my PCIbex code. They are stored in the `Labels` column. What I want is to only get the `MediaRecorders` that are within a trial whose label starts with the word `trial`.

You may have coded your data differently; you may have used a different word; you may not even have any practice or test-recorders, so maybe you do not even need this second filtering. Check your dataframe using the `View()` function. I am also using a function called `str_detect()`, which detects a regular expression pattern, in this case `^trial`, meaning starting with the word trial. Now, when I check my dataframe, I will only see experimental trials and recordings related to those trials. Just to make sure, I am also using the `unique()` function so that I do not have repetitions. And, I am assigning my filenames to a list called `ibex_files`. You can see that any random sample with the `sample()` function will give filenames related to experimental trials.

```
ibex <- ibex |> filter(str_detect(Label, "^trial"))
ibex_files <- ibex$Value |> unique()
sample(ibex_files, 3)
```

```
[1] "chef_Unaccusative_unrelated_zyyc.webm"
[2] "penguin_Unaccusative_related_qghz.webm"
[3] "ballerina_Unaccusative_related_fzsh.webm"
```

2.3 Operators used in this section

`?`

Opens the help page for any function.

example use: `?library()`

`==`

Test for equality. **Don't confuse with a single `=`, which is an assignment operator (and also always returns TRUE).**

example use: "

`|>`

(Forward) pipe: Use the expression on the left as a part of the expression on the right.

- Read `x |> fn()` as *'use `x` as the **only** argument of function `fn`'.*
- Read `x |> fn(1, 2)` as *'use `x` as the **first** argument of function `fn`'.*
- Read `x |> fn(1, ., 2)` as *'use `x` as the **second** argument of function `fn`'.*

example use: "

2.4 Our Task List

- ~~Load the PCIbex results~~
- Filter irrelevant sound files
- Move all of our `.wav` and `.TextGrid` files to the same directory
- Rename our files according to MFA guidelines
- Run MFA
- Create a dataframe

3 Filtering Files from the Server

Now that we have a *gold list*, we can go ahead and filter our files from the server according to our gold list, based on the results from PCIbex. To do this, first we will create a temporary folder called `gold`. We will strip every file name of its extension `.wav` or `.TextGrid` and check if that name exists in our *gold list*. If that is the case, we will move the file. To this end, we are going to use something called *for loops* and *if statements*. You can click on the hyperlinks to watch more on them. [1]

- First, we need to list all of our files. I have all my `.wav` and `.TextGrid` files in the same place, so I just need to list a single directory. You might have them in different places. Check the commented-out part. To only get relevant files, I am using a regular expression saying only choose the ones that end (`$`) with `.wav` or (`|`) `.TextGrid` using the pattern argument.
- Second, we create our *gold directory* with the `dir.create()` function.
- Third, we iterate over every file in the files list, get its name without its extension using the `tools::file_path_sans_ext()` function and check whether it exists in our *gold list* using the `%in%` operator.
- If it is also present in the gold list, we specify where the file exists and assign it to a variable called `old_file_path`. We also specify where it should be moved and assign it to a variable called `new_file_path`. We use the `file.path()` function and `dir`/`gold_dir` variables, along with the file `f` we are iterating over.
- Lastly, we move the file using the `file.rename()` function. Even though the name of the function is *rename*, it can be used to move files. Every file exists with its pointer, such as `~/data/important.R`. By changing this pointer, we can change its name, as well as its location to something like `~/gold_data/really_important.R`. We did not only change the file name from important to really_important. We also changed other parts of this pointer, that is the folder part. If the folder `gold_data` exists, it will be moved there.
- To make sure this for loop works, I also print a message after every file movement using the `cat()` function.

```
dir <- "~/data"
gold_dir <- "~/data/gold"
# Use these lines if you have sound and transcriptions in different places.
# wav_dir <- "~/wav_data"
# tg_dir <- "~/tg_data"

files <- list.files(data, pattern = "\\.wav$|\\.TextGrid")
# Use these lines if you have sound and transcriptions in different places.
# tg_files <- list.files(wav_dir, pattern = "\\.wav$|\\.TextGrid")
# wav_files <- list.files(tg_dir, pattern = "\\.wav$|\\.TextGrid")
```

[1] On principle, I am against for loops in R, but it is better to use here instead of confusing you more.

```
dir.create(gold_dir)

for (f in files) {
  if (tools::file_path_sans_ext(f) %in% tools::file_path_sans_ext(ibex_files)) {
    old_file_path <- file.path(dir, f)
    new_file_path <- file.path(gold_dir, f)
    file.rename(old_file_path, new_file_path)
    cat("Moved ", f, " to ", new_file_path, "\n")
  }
}
```

One important thing to note here is that if you have your `.wav` and `.TextGrid` files in different directories, you can either manually move them to a single folder using your file management application. Alternatively, you can run the commands above using `wav_dir` and `tg_dir` variables, along with `tg_files` and `wav_files` variables. It should look like the following. There are, of course, better ways to solve this problem, and I leave that to your creativity.

```
wav_dir <- "~/wav_data"
tg_dir <- "~/tg_data"
gold_dir <- "~/data/gold"


tg_files <- list.files(wav_dir, pattern = "\\.wav$|\\.TextGrid")
wav_files <- list.files(tg_dir, pattern = "\\.wav$|\\.TextGrid")

dir.create(gold_dir)

for (f in tg_files) {
  if (tools::file_path_sans_ext(f) %in% tools::file_path_sans_ext(ibex_files)) {
    old_file_path <- file.path(tg_dir, f)
    new_file_path <- file.path(gold_dir, f)
    file.rename(old_file_path, new_file_path)
    cat("Moved", f, "to", new_file_path, "\n")
  }
}

for (f in wav_files) {
  if (tools::file_path_sans_ext(f) %in% tools::file_path_sans_ext(ibex_files)) {
    old_file_path <- file.path(wav_dir, f)
    new_file_path <- file.path(gold_dir, f)
    file.rename(old_file_path, new_file_path)
    cat("Moved", f, "to", new_file_path, "\n")
  }
}
```

3.1 Operators used in this section

`%in%`

Test for membership

example use: “

3.2 Our Task List

- ~~Load the PCIbex results~~
- ~~Filter irrelevant sound files~~
- ~~Move all of our .wav and .TextGrid files to the same directory~~
- Rename our files according to MFA guidelines
- Run MFA
- Create a dataframe

4 Renaming Files

This section is going to be the section you have to be most careful about. If you mess anything up in this section, you will have to delete everything, go back, unzip your files, convert them to `.wav` from `.webm`, and do everything in this file again. So, before giving you the for-loop to rename all files, I want to make sure that we go over one of the files and make sure we do it correctly.

The main reason we rename our files is because we want Montreal Forced Aligner to understand that we have multiple speakers in our dataset. If we do not do that, it will treat all files as if they are from a single speaker and probably will be very confused due to inter-speaker variance in speech. Since we have erroneously specified in our PCIbex files to use the `subject_id` as a suffix rather than a prefix, we have to fix that. If in the future we fix that in our PCIbex script, we do not have to go over this part.

4.1 One-file example

My files, after unzipping, converting to `.wav`, and filtering according to the gold list, look like the following. There are some important things to keep in mind here. First of all, now that everything is in our *gold directory*, we have to use the `gold_dir` variable to list our files. Secondly, we again need to use the pattern argument to make sure we only select relevant files. The last thing to be aware of in the next code is that I am using indexing with square brackets to refer to the first elements in the list. I will use this element to first ensure that what I am doing is correct.

```
gold_dir <- "~/data/gold"
files <- list.files(gold_dir, pattern = "\\.wav$|\\.TextGrid$")
example_file <- files[1]
example_file
```

```
[1] "shota-rep-trial_ballerinaUnacc_sgsg_related_dltc.TextGrid"
```

4.1.1 Get the extension and the name

Now that we have the name of an example file, we can start by extracting its extension. We will use the function `file_ext` from the package called `tools`. Sometimes, we do not want to load an entire package, but we want to access a single function. In those cases, we use the operator `::`. Additionally, we will use `paste0` to prefix the extension with a dot, so that we can use it later when we rename our files.

```
extension <- tools::file_ext(example_file)
extension <- paste0(".", extension)
extension
```

```
[1] ".TextGrid"
```

As for the rest of the name, we will use the `file_path_sans_ext()` function that we used earlier.

```
rest <- tools::file_path_sans_ext(example_file)
rest
```

```
[1] "shota-rep-trial_ballerinaUnacc_sgsg_related_dltc"
```

4.1.2 Get the subject id

Now, the most important part is getting the subject name. If you look at what my `rest` variable returned, you can see that it consists of the last 4 characters, which are also the last set of characters after the last underscore. There are multiple ways to extract the subject id. I will show you both methods so that you can choose and adapt them for your own data. For the underscore version, we will use the function `str_split()`, and for the character counting, we will use `str_sub()`.

4.1.2.1 Underscore approach

`str_split()` takes a string and splits it according to the separator you provide. In our case, the separator is the underscore. We are also using an additional argument called `simplify` to make the resulting elements more user-friendly. Our function now returns a small table with 1 row and 5 columns. To select the values in the 5th column, we use square brackets again, this time with a comma. When you apply this approach to your own data, remember that you may end up with fewer or more than 5 columns depending on your naming convention. Be sure to adjust the column number accordingly. It might also be the case that your subject id is not stored last or that your separators are not underscores but simple "-". Modify the code according to your specific needs.

```
# Using the underscore information
subj <- str_split(rest, "_", simplify = TRUE)
subj
```

```
     [,1]             [,2]            [,3]   [,4]      [,5]
[1,] "shota-rep-trial" "ballerinaUnacc" "sgsg" "related" "dltc"
```

```
subj <- subj[,5]
subj
```

```
[1] "dltc"
```

Lastly, we have to modify the `rest` variable so that we do not include the subject id twice. I will use the same approach again. After obtaining the table, I will use the `paste()` function to concatenate the columns back together with the underscore separator. Adjust the number of columns used in this function and the separator according to your own data needs.

```
nosubj <- str_split(rest, "_", simplify = TRUE)
nosubj <- paste(nosubj[,1], nosubj[,2], nosubj[,3], nosubj[,4], sep = "_")
nosubj
```

```
[1] "shota-rep-trial_ballerinaUnacc_sgsg_related"
```

4.1.2.2 Character approach

`str_sub()` allows you to extract a substring using indices. In my case, the subject id is the last four characters. To refer to characters from the end, you can use the minus symbol `-`. I specify `-4` in the `start` argument, which means I want to extract the string starting from the fourth character counting back from the end.

```
subj <- str_sub(rest, start = -4)
subj
```

```
[1] "dltc"
```

To get the rest of the filename, I specify the starting point as `1` and the endpoint as `-6`. Using `-5` would include the underscore as well.

```
nosubj <- str_sub(rest, start = 1, end = -6)
nosubj
```

```
[1] "shota-rep-trial_ballerinaUnacc_sgsg_related"
```

4.1.3 Put the new name and the path together

At this point, we have everything we need: (i) the subject id prefix, (ii) the rest of our file name, and (iii) the extension. Now, we need to combine all of this together. We are going to use the `paste0()` function. Remember, this function is different from `paste()`. The main difference is that with `paste0()`, we cannot specify separators; we have to provide everything. This might seem like a disadvantage at first, but it is beneficial for non-pattern cases like this.

```
new_name <- paste0(subj, "_", nosubj, extension)
new_name
```

```
[1] "dltc_shota-rep-trial_ballerinaUnacc_sgsg_related.TextGrid"
```

We also need to create a new path to rename our file.

```
new_path <- file.path(gold_dir, new_name)
new_path
```

```
[1] "~/data/gold/dltc_shota-rep-trial_ballerinaUnacc_sgsg_related.TextGrid"
```

4.1.4 Rename the file

We will once again use the `file.rename()` function. This time, we are only changing the file name and not the path, so the file will remain in its current location. We also need to obtain the full path of our `example_file`. We can achieve this easily using the `file.path` function again.

```
example_file_path <- file.path(gold_dir, example_file)
example_file_path
```

```
[1] "~/data/gold/shota-rep-trial_ballerinaUnacc_sgsg_related_dltc.TextGrid"
```

```
file.rename(example_file_path, new_path)
```

After running this, make sure the naming convention is as we want. Check your folder by searching for the trial. It should look something like `subj_rest.wav` or `subj_rest.TextGrid`. In my case, it is `dltc_shota-rep-trial_ballerinaUnacc_sgsg_related.TextGrid`, where `dltc` is my subject id or subj.

4.2 Little treat for you

I know that some of your files look like the following: `squid_S_jtfr.wav`. Here, I will provide you with the code to rename this. Please check this code before using it. First, let's arbitrarily assign this name to a variable. Remember, in your case, you will obtain this from your `files` list.

```
example_file <- "squid_S_jtfr.wav"
```

Now, I am going to put all the code together in one chunk, except for moving. Also, be aware that I am using my own `gold_dir`; please specify yours according to your needs. Additionally, be mindful of your operating system (Windows or Mac). If you are using Windows, your `gold_dir` variable should look like the second line. I have commented out that part with a hashtag/pound symbol. Uncomment it by deleting the first pound symbol.

```
gold_dir <- "~/data/gold"
# gold_dir <- "C:/Users/utkuturk/data/gold" # for windows
extension <- tools::file_ext(example_file)
extension <- paste0(".", extension)
rest <- tools::file_path_sans_ext(example_file)
subj <- str_sub(rest, start = -4)
nosubj <- str_sub(rest, start = 1, end = -6)
new_name <- paste0(subj, "_", nosubj, extension)
new_path <- file.path(gold_dir, new_name)
new_path
```

```
[1] "~/data/gold/jtfr_squid_S.wav"
```

This would be your original example file path.

```
example_file_path <- file.path(gold_dir, example_file)
example_file_path
```

```
[1] "~/data/gold/squid_S_jtfr.wav"
```

And this line would handle the renaming from the old `example_file_path` to the `new_path`, thereby assigning the new name.

```
file.rename(example_file_path, new_path)
```

4.3 For loop

If you have ensured that the code above works correctly for you, you are now ready to implement the for loop. Within the loop, define a variable like `f` and use it instead of `example_file`. This way, you will iterate over every file in your list. To verify that it is functioning correctly, I also added a line to print a message each time a file is renamed.

```
gold_dir <- "~/data/gold"
# gold_dir <- "C:/Users/utkuturk/data/gold" # for windows
files <- list.files(gold_dir, pattern = "\\.wav$|\\.TextGrid$")

for (f in files) {
  extension <- tools::file_ext(f)
  extension <- paste0(".", extension)
  rest <- tools::file_path_sans_ext(f)
  subj <- str_sub(rest, start = -4)
  nosubj <- str_sub(rest, start = 1, end = -6)
  new_name <- paste0(subj, "_", nosubj, extension)
  new_path <- file.path(gold_dir, new_name)
  file_path <- file.path(gold_dir, f)
```

```
  file.rename(file_path, new_path)
  cat("Renamed", f, "to", new_name, "\n")
}
```

4.4 Operators used in this section

`df[selected_rows, indices_columns] or list[selected_element]`

[], Indexing operator: Accesses specific rows and/or columns of a data frame. If it is a list, it only takes a single argument to select an element. Remember in R indices start with 1, unlike python.

- `selected_rows` A vector of indices or names.
- `selected_columns` A vector of indices or names.
- `selected_element` A vector of indices or names.

example use: `files[1]`

`::`

Double colon operator: Accesses functions and other objects from packages. Read `x::y` as *'function y from package x.'*

example use: `tools::file_ext()`

4.5 Our Task List

- ~~Load the PCIbex results~~
- ~~Filter irrelevant sound files~~
- ~~Move all of our .wav and .TextGrid files to the same directory~~
- ~~Rename our files according to MFA guidelines~~
- Run MFA
- Create a dataset

5 Running the Montreal Forced Aligner

Now that we have our files in the format we want, we can place all of our files in the MFA folder and start running the aligner. We can either move our files using the Explorer app and usual copy-paste or we can use the `file.rename()` function again. Due to the aligner's limitations, the second option is far better. The main constraint is that MFA starts encountering problems when we feed it more than 2000 files at once. Since I have a lot of data, I will use the following function to divide my files and move them into smaller subfolders. But before that, I will show you how to move files without dividing them into subfolders.

5.1 Moving the Files to MFA Directory

5.1.1 Without Dividing

Again, we are going to use the `file.rename()` and `dir.create()` functions to create the directory we are moving files to, and of course, to move files.

```
# gold directory, where all of our files are
gold_dir <- "~/data/gold"
# MFA directory
mfa_dir <- "~/Documents/MFA/mycorpus"
dir.create(mfa_dir)

# Files
files <- list.files(gold_dir, pattern = "\\.wav$|\\.TextGrid$")
for (f in files) {
  old_file_path <- file.path(gold_dir, f)
  mfa_path <- file.path(mfa_dir, f)
  file.rename(old_file_path, mfa_path)
  cat("Moved", f, "to", mfa_dir, "\n")
}
```

5.1.2 With Dividing them into subfolders

I will introduce the following function that I use. Here, I will not go into details, but it basically performs the following steps:

- Creates a subfolder called `s1` and moves files into it.
- Counts up to 2000.
- When it surpasses 2000, it creates another subfolder by incrementing the number from `s1` to `s2`.
- Continues this process until there are no more files.

```
divide_and_move <- function(source, target, limit=2000) {
  files <- list.files(source, pattern = "\\.wav$|\\.TextGrid$", full.names = TRUE)
  base_names <- unique(tools::file_path_sans_ext(basename(files)))
  s_index <- 1
  f_index <- 0
  s_path <- file.path(target, paste0("s", s_index))
  dir.create(s_path)

  for (b in base_names) {
    rel_files <- files[grepl(paste0("^", b, "\\."), basename(files))]

    if (f_index + length(rel_files) > limit) {
      s_index <- s_index + 1
      s_path <- file.path(target, paste0("s", s_index))
      dir.create(s_path)
```

```
      f_index <- 0
    }

    for (f in rel_files) {
      file.rename(f, file.path(s_path, basename(f)))
    }
    f_index <- f_index + length(rel_files)
  }
}
```

You can use this function by simply providing the source and target folders.

```
# gold directory, where all of our files are
gold_dir <- "~/data/gold"
# MFA directory
mfa_main_dir <- "~/Documents/MFA"
dir.create(mfa_dir)
divide_and_move(gold_dir, mfa_main_dir)
```

5.2 Terminal Codes

After moving the files either with code or by hand to a specific MFA folder, `~/Documents/MFA`, we can start running the terminal commands. At this point, I assume you have gone through the MFA documentation for installation instructions. I am also assuming that you have used a `conda` environment. If you haven't, here are the 3 lines to install MFA.

Listing 1 `Conda Installation in Terminal`

```
conda activate base
conda install -c conda-forge mamba
mamba create -n aligner -c conda-forge montreal-forced-aligner
```

There are again two ways to do this. One way is to open your Terminal app or use the Terminal tab in the R console below. The other way, which I prefer more, is to execute commands using the R function `system()`. I will first go over the easier one, which is using the Terminal app or the Terminal tab in R. But the reason I prefer the `system()` function is that I can loop over multiple folders more easily that way, and I do not have to run my commands again and again.

5.2.1 Using Terminal

The first command we want to run is the conda environment code. Following the MFA documentation, I renamed my environment to `aligner`. So, I start by activating that environment.

```
conda activate aligner
```

After activating the environment, I need to download three models: (i) an acoustic model to recognize phonemes given previous and following acoustic features, (ii) a dictionary to access pretrained phone-word mappings, and (iii) a g2p model to generate sequences of phones based on orthography. For all of these models, we are going to use the `english_us_arpa` model. You can visit this website to explore various languages and models.

```
mfa model download acoustic english_us_arpa
mfa model download dictionary english_us_arpa
mfa model download g2p english_us_arpa
```

After downloading these models, we are going to validate our corpus. There are many customizable parameters for this step. You can check them here. I am going to use my favorite settings here. You can interpret the following command like this: *Dear Montreal Forced Aligner (`mfa`), can you please analyze my files located in `~/Documents/MFA/mycorpus` and `validate` them using the `english_us_arpa` acoustic model and `english_us_arpa` dictionary? Please also consider that I have multiple speakers, indicated by the first 4 characters (`-s 4`). It would be great to use multiprocessing (`--use_mp`) for faster execution. Lastly, please clean up previous and new temporary files (`--clean --final_clean`).*

```
mfa validate -s 4 --use_mp --clean --final_clean ~/Documents/MFA/mycorpus english_us_arpa english_us_arpa
```

This process will take some time. Afterward, you will have some *out of vocabulary* words found in your TextGrids. You can easily create new pronunciations for them and add them to your model.

The `mfa g2p` command can take many arguments; here I am using only three. First, the path to the text file that has *out of vocabulary* words. This file is automatically created in your folder where your files are located. The path may vary depending on your system and folder naming, but the name of the `.txt` file will be the same. In my case, it is `~/Documents/MFA/mycorpus/oovs_found_english_us_arpa.txt`. The second argument is the name of the g2p model. As you may recall, we downloaded it earlier, and its name is `english_us_arpa`. Finally, the third argument is the path to a target `.txt` file to store new pronunciations. I would like to store them in the same place, so I am using the following path: `~/Documents/MFA/mycorpus/g2pped_oovs.txt`.

```
mfa g2p ~/Documents/MFA/mycorpus/oovs_found_english_us_arpa.txt english_us_arpa ~/Documents/MFA/mycorpus/g2pped_oovs.txt
```

After creating the pronunciations, you can add them to your model with `mfa model add_words`. This command takes the name of the dictionary as an argument (`english_us_arpa`) and the output of the `mfa g2p` command, which was a `.txt` file storing pronunciations: `~/Documents/MFA/mycorpus/g2pped_oovs.txt`.

```
mfa model add_words english_us_arpa ~/Documents/MFA/mycorpus/g2pped_oovs.txt
```

The last step is the alignment process. It will align (`mfa align`) the words and the phones inside our TextGrids stored in `~/Documents/MFA/mycorpus` using our previously downloaded dictionary (`english_us_arpa`) and model (`english_us_arpa`), and store the newly aligned TextGrids in a new folder called `~/Documents/MFA/output`.

```
mfa align ~/Documents/MFA/mycorpus english_us_arpa english_us_arpa ~/Documents/MFA/output
```

5.2.2 Using R

We can also accomplish all of this in R. One advantage of this approach is that it allows us to iterate over multiple subfolders more easily, which can be useful if we have more than 2000 files. We will use four components:

(i) `system()` function to execute terminal commands,
(ii) `paste()` function to create multiline templates,
(iii) `%s` string placeholder to create template codes,
(iv) `sprintf()` function to format our templates.

5.2.2.1 Introduction to `sprintf()` and `%s`

Before going further with MFA codes, let me illustrate with an example. Suppose we have a list of folder names, and we want to create a `.txt` file in each of these folders. We can use the `system()` function to perform this action. Below, I define my `folder_list`, then create paths for my `.txt` files in each folder, such as `~/data1/mydocument.txt`. Afterwards, I generate a list of commands to create these files using `touch`, which is a command-line tool for creating files. Finally, I execute these commands using the `system()` function.

```
folder_list <- c("~/data1", "~/data2", "~/data3")



txt_list <- paste(folder_list, "mydocument.txt", sep="/")
txt_list
command_list <- paste("touch", txt_list, sep=" ")
command_list

for (command in command_list) {
  system(command)
}
```

Technically, we didn't need to use a for loop; instead, we could have concatenated all these commands with `;` and run a single system command. Bash can execute multiple commands in a single line when separated by `;`.

```
concatenated_commands <- paste(command_list[1],
                               command_list[2],
                               command_list[3],
                               sep=";")

system(concatenated_commands)
```

We could achieve the same without needing a folder list by utilizing the `%s` placeholder and the `sprintf()` function.

```
command_template <- "touch %s/mydocument.txt"
concatenated_commands <- paste(sprintf(command_template, "~/data1"),
                               sprintf(command_template, "~/data2"),
                               sprintf(command_template, "~/data3"),
```

```
                                sep=";")

system(concatenated_commands)
```

This approach becomes particularly useful when dealing with multiple placeholders within the same command. For instance, the command template will replace the first `%s` with the first argument, such as `~/data1`, and the second `%s` with the second argument, like `mydoc1`, when formatted using `sprintf()`.

```
command_template <- "touch %s/%s.txt"
concatenated_commands <- paste(sprintf(command_template, "~/data1", "mydoc1"),
                                sprintf(command_template, "~/data2", "mydoc2"),
                                sprintf(command_template, "~/data3", "mydoc3"),
                                sep=";")

system(concatenated_commands)
```

5.2.2.2 Running MFA in R

Next, we'll consolidate the previous code by concatenating it using `paste()` and separating commands with `;`. If needed, we'll incorporate placeholders. Each line will be assigned to a new variable, and then they'll be combined into a single command string using `paste()`. Finally, we'll execute the command string using `system()` with the argument `intern = TRUE` to capture the output into an R variable, which allows for later inspection.

```
conda_start <- "conda activate aligner"
get_ac <- "mfa model download acoustic english_us_arpa"
get_dic <- "mfa model download dictionary english_us_arpa"
get_g2p <- "mfa model download g2p english_us_arpa"

mfa_init <- paste(conda_start, get_ac, get_dic, get_g2p, sep = ";")

mfa_init_output <- system(mfa_init, intern = TRUE)
```

After initializing the model, the next step involves validation. Again, I'll use the same approach and concatenate the commands together. However, sometimes we may have too many files and need to use subfolders. To accommodate this, I'll use `%s` placeholders. The validation command has one placeholder for different subfolders. Similarly, our pronunciation creation for g2p has two placeholders, though they'll be filled with the same value. Lastly, the `add_words` command will use a single placeholder. Fortunately, all these folders are the same, so we can reuse the same variable repeatedly.

```
conda_start <- "conda activate aligner"

validate <- "mfa validate -s 4 --use_mp --clean --final_clean ~/Documents/MFA/%s english_us_arpa english_us_arpa"
g2p_words <- "mfa g2p ~/Documents/MFA/%s/oovs_found_english_us_arpa.txt english_us_arpa ~/Documents/MFA/%s/g2pped_oovs.tx
add_words <- "mfa model add_words english_us_arpa ~/Documents/MFA/%s/g2pped_oovs.txt"

mfa_val <- paste(conda_start, validate, g2p_words, add_words, sep = ";")
```

Since this step takes longer and there's more room for errors, I want to save all my outputs in a list. First, I need to identify which folders exist in my MFA directory. Because my `divide_and_move` function prefixes every subfolder with `s`, I'll use `^s` to filter for relevant folders.

```
output_val <- list()

# Define the base path where folders are located
base_path <- "~/Documents/MFA"
folders <- list.dirs(base_path, recursive = FALSE, full.names = FALSE)
folders <- folders[str_detect(folders, "^s")]

folders
```

Now, we can iterate over this list of folders using a for loop. First, we create a temporary script using `sprintf()` with four placeholders. Next, we execute the current script and save the output in a `temp_output` variable. Later, we assign this output to specific `output_name` variables for each folder using `paste0()` and `assign()` functions.

```
for (f in folders) {
  cur_mfa_val <- sprintf(mfa_val, f, f, f, f)

  temp_output <- system(cur_mfa_val, intern = TRUE)

  output_name <- paste0("output_val_", f)

  assign(output_name, temp_output, envir = .GlobalEnv)
}
```

Now you can check the outputs by calling specific variables like `output_val_s1` or `output_val_s2`. After this step, the only task remaining is to run the aligner. We will create a template again, iterate over folders, and assign outputs to their respective names for verification. Meanwhile, the bash code will execute in the background. This time, our placeholders will refer to different inputs and an output folder. Fortunately, we can use the same output folder for every subfolder, so instead of using two placeholders, we'll use a single `%s` placeholder.

```
conda_start <- "conda activate aligner"

align <- "mfa align ~/Documents/MFA/%s english_us_arpa english_us_arpa ~/Documents/MFA/output"

mfa_align <- paste(conda_start, align, sep = ";")

for (f in folders) {
  cur_mfa_align <- sprintf(mfa_align, f)
  temp_output <- system(cur_mfa_align, intern=TRUE)
  output_name <- paste0("output_align_", f)
  assign(output_name, temp_output, envir = .GlobalEnv)
}
```

This for loop completes the MFA alignment. There is one final task remaining: creating a dataframe for further data analysis.

5.3 Our Task List

- ~~Load the PCIbex results~~
- ~~Filter irrelevant sound files~~
- ~~Move all of our .wav and .TextGrid files to the same directory~~
- ~~Rename our files according to MFA guidelines~~
- ~~Run MFA~~
- Create a dataframe

6 Dataframe Creation

Now that we have all our `.TextGrid` files aligned, we can create a dataframe for subsequent statistical analyses using the `readtextgrid` package. First, I'll demonstrate the process for a single file. Later, I'll explain how to extend this to an entire directory. Let's begin by specifying our directory and listing the files. You can view the first few elements of a list or dataframe using the `head()` function.

```
# Define the directory and list files
tg_dir <- "~/Documents/MFA/output"
file_list <- list.files(path = tg_dir, pattern = "\\.TextGrid$")
head(file_list, n = 5)
```

```
[1] "bpyuohtj_ballerina_unacc_pl_pl.TextGrid"
[2] "bpyuohtj_ballerina_unacc_pl_sg.TextGrid"
[3] "bpyuohtj_ballerina_unacc_sg_pl.TextGrid"
[4] "bpyuohtj_ballerina_unacc_sg_sg.TextGrid"
[5] "bpyuohtj_ballerina_unerg_pl_pl.TextGrid"
```

6.1 One-file example

Again, let's work with an example file from our list, starting with the first file `[1]`. First, we'll retrieve its full file path. Then, we'll use the `read_textgrid()` function to create a dataframe for this single file. I'll print the structure of the dataframe to give you a clearer view of its contents.

```
example_file <- file_list[1]
file_path <- file.path(tg_dir, example_file)
example_df <- readtextgrid::read_textgrid(file_path)
str(example_df)
```

```
tibble [36 x 10] (S3: tbl_df/tbl/data.frame)
 $ file       : chr [1:36] "bpyuohtj_ballerina_unacc_pl_pl.TextGrid" "bpyuohtj_ballerina_unacc_pl_pl.TextGrid" "bpyuohtj_bal
 $ tier_num   : num [1:36] 1 1 1 1 1 1 1 1 1 1 ...
 $ tier_name  : chr [1:36] "words" "words" "words" "words" ...
 $ tier_type  : chr [1:36] "IntervalTier" "IntervalTier" "IntervalTier" "IntervalTier" ...
 $ tier_xmin  : num [1:36] 0 0 0 0 0 0 0 0 0 0 ...
```

```
 $ tier_xmax     : num [1:36] 2.52 2.52 2.52 2.52 2.52 2.52 2.52 2.52 2.52 2.52 ...
 $ xmin          : num [1:36] 0 0.57 0.61 1.11 1.37 ...
 $ xmax          : num [1:36] 0.57 0.61 1.11 1.37 1.43 ...
 $ text          : chr [1:36] "" "the" "ballerinas" "next" ...
 $ annotation_num: int [1:36] 1 2 3 4 5 6 7 8 9 10 ...
```

In this project, which involves aligning words, we are interested in only a couple of these columns. Specifically, we focus on the file identifier (`file`) to determine the trial from which the data originates, the tier name (`tier_name`) to differentiate between word and phone tiers, the start (`xmin`) and end (`xmax`) of each interval, and finally, the `text`. Additionally, I am not interested in retaining the file extension in the `file` identifier. Therefore, we will first filter to include only annotated words, then select the important columns using `select()`, remove the `.TextGrid` extension, and concatenate the words so that we can see the full response for each trial.

```
example_df <- example_df |>
  # Filter annotated "words" tier
  filter(tier_name == "words" & text != "") |>
  # Select relevant columns
  select(file, xmin, xmax, text, annotation_num) |>
  # Remove .TextGrid and put the response together
  mutate(file = str_remove(file, "\\.TextGrid$"),
         response = paste(text, collapse = " "))

example_df
```

file	xmin	xmax	text	annotation_num	response
bpyuohtj_ballerina_unacc_pl_p1	0.1695	0.6095	the	2	the ballerinas next to the axes are shrinking
bpyuohtj_ballerina_unacc_pl_p2	0.6095	1.1095	ballerinas	3	the ballerinas next to the axes are shrinking
bpyuohtj_ballerina_unacc_pl_p3	1.1095	1.3695	next	4	the ballerinas next to the axes are shrinking
bpyuohtj_ballerina_unacc_pl_p4	1.3695	1.4295	to	5	the ballerinas next to the axes are shrinking
bpyuohtj_ballerina_unacc_pl_p5	1.4295	1.5495	the	6	the ballerinas next to the axes are shrinking
bpyuohtj_ballerina_unacc_pl_p6	1.5495	1.8995	axes	7	the ballerinas next to the axes are shrinking
bpyuohtj_ballerina_unacc_pl_p7	1.8995	1.9595	are	8	the ballerinas next to the axes are shrinking
bpyuohtj_ballerina_unacc_pl_p8	1.9595	2.4895	shrinking	9	the ballerinas next to the axes are shrinking

We also need some information about the trial. Luckily, all of our information is provided in our file name. So, I am going to parse that name to create a dataframe with more information. I am using a set of function that all start with `separate_wider_`.

- The `delim` version uses a deliminator to split a row of a dataframe.
- The `regex` version uses regular expressions to split the data.
- Finally, the `position` version uses the number of characters to split the data.

I am doing all of this because of how I initially coded my experiment output in my PCIbex script. You may need to change this code to process your own data.

```
example_df <- example_df |>
  # split the `file` column into 5 different columns.
  separate_wider_delim(file, "_", names = c("subj", "exp", "headVerb", "NumNum", "sem_type"), cols_remove = F  |>
  # split the headVerb column from the "U" character
  separate_wider_regex(headVerb, c(head = ".*", "U", verb_type = ".*")) |>
  # Add the "U" character back to "Unacc" and "Unerg"s
  mutate(verb_type = paste0("U", verb_type)) |>
  # split the head and distractor numbers.
  separate_wider_position(NumNum, c(head_num = 2, dist_num = 2))

example_df
```

6.2 Another Treat for you

Let's see what you will need to do. You will find your file in the `MFA/output` folder as well, and your file will look like `jtfr_squid_S.TextGrid`. Let's arbitrarily put them here. Remember, you will have to use the `file.list()` function as well. You will not need to change anything in the first part where we work on the TextGrid. The necessary changes will need to be done in the parsing procedure. Instead of using the entire `regex` or `position` methods, you will just need to use the `delim` version of the function.

```
# Define the directory and list files
tg_dir <- "~/Documents/MFA/output"
your_file <- "jtfr_squid_S.TextGrid"

file_path <- file.path(tg_dir, your_file)



### LETS SAY YOU RAN read_textgrid function.
### I commented out this part, because I do not have your data.
### Final dataframe will be slightly different here, because I do not have the textgrid data here.
# your_df <- readtextgrid::read_textgrid(path = file_path) |>
#     filter(tier_name == "words" & text != "") |>
#     select(file, xmin, xmax, text, annotation_num) |>
#     mutate(file = str_remove(file, "\\.TextGrid$"),
#            response = paste(text, collapse = " "))
#


your_df <- your_df |>
  separate_wider_delim(file, "_", names = c("subj", "head", "condition"), cols_remove = F)
```

```
your_df
```

subj	head	condition	file	xmin	xmax	text	annotation_num	response
jtfr	squid	S	jtfr_squid_S			squid	1	squid jumped over the fence
jtfr	squid	S	jtfr_squid_S			jumped	2	squid jumped over the fence
jtfr	squid	S	jtfr_squid_S			over	3	squid jumped over the fence
jtfr	squid	S	jtfr_squid_S			the	4	squid jumped over the fence
jtfr	squid	S	jtfr_squid_S			fence	5	squid jumped over the fence

6.3 For-loop

Now, we need to apply this process to all files in our output directory. To simplify this, I'll start by creating a function for processing each file individually and then apply it to all files. The function takes a file name and its directory as inputs and returns a dataframe. Before creating each dataframe, it prints "Reading the file." to indicate progress.

```
process_textgrid <- function(file, directory) {
  cat("Reading", file, "\n")
  file_path <- file.path(directory, file)
  df <- readtextgrid::read_textgrid(path = file_path) |>
    filter(tier_name == "words" & text != "") |>
    select(file, xmin, xmax, text, annotation_num) |>
    mutate(file = str_remove(file, "\\.TextGrid$"),
           response = paste(text, collapse = " "))

  df <- df |>
    separate_wider_delim(file, "_", names = c("subj", "exp", "headVerb", "NumNum", "sem_type"), cols_remove = F) |>
    separate_wider_regex(headVerb, c(head = ".*", "U", verb_type = ".*")) |>
    mutate(verb_type = paste0("U", verb_type)) |>
    separate_wider_position(NumNum, c(head_num = 2, dist_num = 2))

  return(df)
}
```

This is essentially the same process as before, but encapsulated within a function for easier application. Here's an example of how I'm redefining my directory and file list:

```r
# Define the directory and list files
tg_dir <- "~/Documents/MFA/output"
file_list <- list.files(path = tg_dir, pattern = "\\.TextGrid$")
example_file <- file_list[1]
process_textgrid(example_file, tg_dir)
```

Your version will look like this.

```r
process_textgrid <- function(file, directory) {
  cat("Reading", file, "\n")
  file_path <- file.path(directory, file)
  df <- readtextgrid::read_textgrid(path = file_path) |>
    filter(tier_name == "words" & text != "") |>
    select(file, xmin, xmax, text, annotation_num) |>
    mutate(file = str_remove(file, "\\.TextGrid$"),
           response = paste(text, collapse = " "))

  df <- df |>
    separate_wider_delim(file, "_", names = c("subj", "head", "condition"), cols_remove = F)

  return(df)
}
```

```r
# Define the directory and list files
tg_dir <- "~/Documents/MFA/output"
file_list <- list.files(path = tg_dir, pattern = "\\.TextGrid$")
example_file <- file_list[1]
process_textgrid(example_file, tg_dir)
```

Now, we need to integrate this function into our for-loop. Instead of using a single file like `file_list[1]`, we will apply it to an entire directory. Unlike previous for loops, we will use the `map` function from the `purrr` package. It is faster and easier to use in cases like this. `map()` will return all of our dataframes embedded in a list. After using `map()`, we need to combine all these smaller dataframes into a larger one using `bind_rows`.

```r
# Define the directory and list files
tg_dir <- "~/Documents/MFA/output"
file_list <- list.files(path = tg_dir, pattern = "\\.TextGrid$")
# Run our function, I am using mine, you should use your own.
process_textgrid <- function(file, directory) {
  cat("Reading", file, "\n")
  file_path <- file.path(directory, file)
  df <- readtextgrid::read_textgrid(path = file_path) |>
    filter(tier_name == "words" & text != "") |>
    select(file, xmin, xmax, text, annotation_num) |>
    mutate(file = str_remove(file, "\\.TextGrid$"),
           response = paste(text, collapse = " "))
```

```
  df <- df |>
    separate_wider_delim(file, "_", names = c("subj", "exp", "headVerb", "NumNum", "sem_type"), cols_remove = F) |>
    separate_wider_regex(headVerb, c(head = ".*", "U", verb_type = ".*")) |>
    mutate(verb_type = paste0("U", verb_type)) |>
    separate_wider_position(NumNum, c(head_num = 2, dist_num = 2))

  return(df)
}

dfs <- map(file_list, process_textgrid, directory = tg_dir)

final_df <- bind_rows(dfs)
```

This completes our MFA Aligning work. We have successfully completed every task on our list, aligned our data, and created a dataframe for analysis. You can check the structure of our final dataframe, the number of rows, and the count of unique trials.

```
str(final_df)

nrow(final_df)

length(unique(final_df$file))
```

7 An Example Descriptive Summary

Let me also show a small example of descriptive statistics using basic functions. One thing we might want to do is check whether there is a difference between conditions in people's time to start uttering the sentence. Let's assume people are still thinking about the sentence when they utter the first determiner "*the*," since it is kind of automatic in English and all of our sentences start with this determiner anyway. I am going to filter my dataframe using the `text` column. Additionally, I want to ensure it is the first occurrence of "*the*" and not any other "*the*" in the sentences, so I will use the `annotation_num` column in my filtering.

```
first_thes <- final_df |> filter(text == "the" & annotation_num < 3)
```

Then, what I am going to do is summarize my dataframe. I will group my dataframe by columns `verb_type` and `sem_type` since my experiment had 2 different verb types (unergative, unaccusative) and 2 different types of semantic distractors (related, unrelated). Then, I am going to use the `summarize()` function to get the mean, standard error, and credible interval for each condition. Since we are assuming people are still planning sentences while they utter the first determiner, I am going to summarize my data using the offset of the *the* (`xmax`). Before doing this, I am going to convert `xmax` from seconds to milliseconds.

```
first_thes |>
  mutate(xmax = xmax*1000) |>
  group_by(verb_type, sem_type) |>
  summarise_each(funs(mean, se=sd(.)/sqrt(n()), ci=se*1.96), xmax)
```

From this point on, I leave the modeling and plotting of the data for another guide.