A tutorial on how to use the EMU-SDMS [1]

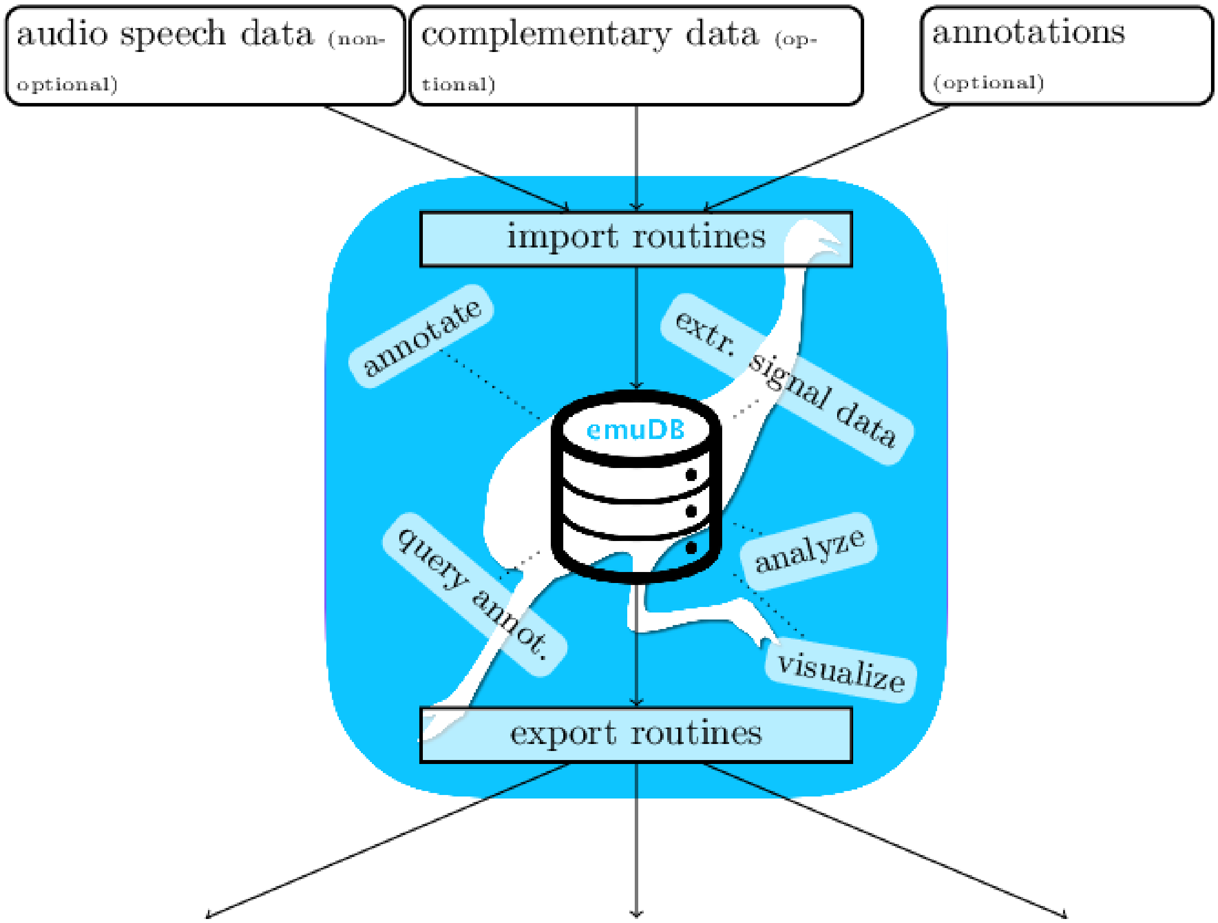

Figure 1: Dies ist die Bildunterschrift

Using the tools provided by the EMU-SDMS, this tutorial chapter gives a practical step-by-step guide to answering the question: *Given an annotated speech database, is vowel height (measured by its correlate, the first formant frequency) influenced by whether it appears in a strong or weak syllable?* The tutorial only skims over many of the concepts and functions provided by the EMU-SDMS. In-depth explanations of the various functionalities are given in later chapters of this documentation.

As the EMU-SDMS is not concerned with the raw data acquisition, other tools such as SpeechRecorder by @draxler:2004a are first used to record speech. However, once audio speech recordings are available, the system provides multiple conversion routines for converting existing collections of files to the new `emuDB` format described in Chapter @ref(chap:emuDB) and importing them into the new EMU system. The current import routines provided by the `emuDB` package are:

- `convert_TextGridCollection()` - Convert TextGrid collections (`.wav` and `.TextGrid` files) to the `emuDB` format,
- `convert_BPFCollection()` - Convert BPF collections (`.wav` and `.par`} files) to the `emuDB` format,
- `convert_txtCollection()` - Convert plain text file collections format (`.wav` and `.txt` files) to the `emuDB` format,
- `convert_legacyEmuDB()` - Convert the legacy EMU database format to the `emuDB` format and

[1]Some examples of this chapter are adapted versions of examples of the `emuR_intro` vignette.

- `create_emuDB()` followed by `add_link/levelDefinition` and `import_mediaFiles()` - Creating `emuDB`s from scratch with only audio files present.

The `emuDB` package comes with a set of example files and small databases that are used throughout the `emuDB` documentation, including the functions help pages. These can be accessed by typing `help(functionName)` or the short form `?functionName`. R Example @ref(rexample:tutorial_create_emuRdemoData) illustrates how to create this demo data in a user-specified directory. Throughout the examples of this documentation the directory that is provided by the base R function `tempdir()` will be used, as this is available on every platform supported by R (see `?tempdir` for further details). As can be inferred from the `list.dirs()` output in R Example @ref(rexample:tutorial_create_emuRdemoData), the `emuR_demoData` directory contains a separate directory containing example data for each of the import routines. Additionally, it contains a directory containing an `emuDB` called *ae* (the directories name is `ae_emuDB`, where `_emuDB` is the default suffix given to directories containing a `emuDB`; see Chapter @ref(chap:emuDB)).

```
# load the package
library(emuR)

# create demo data in directory provided by the tempdir() function
# (of course other directory paths may be chosen)
create_emuRdemoData(dir = tempdir())

# create path to demo data directory, which is
# called "emuR_demoData"
demoDataDir = file.path(tempdir(), "emuR_demoData")

# show demo data directories
list.dirs(demoDataDir, recursive = F, full.names = F)
```

```
## [1] "ae_emuDB"             "BPF_collection"      "legacy_ae"
## [4] "TextGrid_collection" "txt_collection"
```

This tutorial will start by converting a TextGrid collection containing seven annotated single-sentence utterances of a single male speaker to the `emuDB` format[2]. In the EMU-SDMS, a file collection such as a TextGrid collection refers to a set of file pairs where two types of files with different file extentions are present (e.g., `.ext1` and `.ext2`). It is vital that file pairs have the same basenames (e.g., `A.ext1` and `A.ext2` where `A` represents the basename) in order for the conversion functions to be able to pair up files that belong together. As other speech software tools also encourage such file pairs [e.g., @kisler:2015a] this is a common collection format in the speech sciences. R Example @ref(rexample:showTGcolContent) shows such a file collection that is part of `emuDB`'s demo data. Figure @ref(fig:msajc003_praatTG) shows the content of an annotation as displayed by Praat's `"Draw visible sound and Textgrid..."` procedure.

```
# create path to TextGrid collection
tgColDir = file.path(demoDataDir, "TextGrid_collection")

# show content of TextGrid_collection directory
list.files(tgColDir)
```

```
##  [1] "msajc003.TextGrid" "msajc003.wav"      "msajc010.TextGrid"
##  [4] "msajc010.wav"      "msajc012.TextGrid" "msajc012.wav"
##  [7] "msajc015.TextGrid" "msajc015.wav"      "msajc022.TextGrid"
## [10] "msajc022.wav"      "msajc023.TextGrid" "msajc023.wav"
## [13] "msajc057.TextGrid" "msajc057.wav"
```

[2]The other input routines are covered in the Section @ref(sec:emuRpackageDetails_importRoutines).

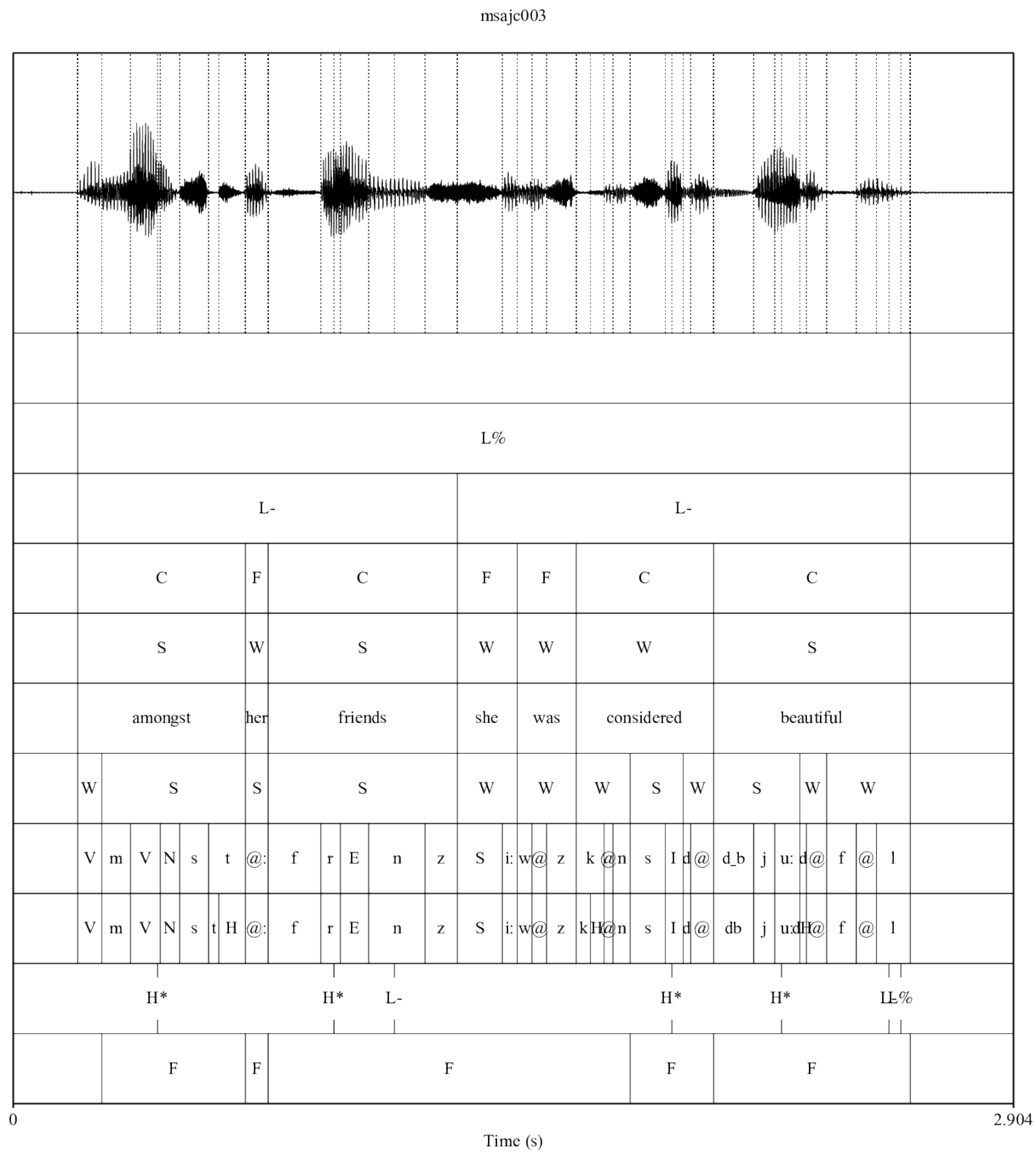

Figure 2: TextGrid annotation of the `emuR_demoData/TextGrid_collection/\allowbreak msajc003.wav` / `.TextGrid` file pair containing the tiers (from top to bottom): *Utterance*, *Intonational*, *Intermediate*, *Word*, *Accent*, *Text*, *Syllable*, *Phoneme*, *Phonetic*, *Tone*, *Foot*.(#fig:msajc003_praatTG)

Converting the TextGrid collection

The `convert_TextGridCollection()` function converts a TextGrid collection to the `emuDB` format. A precondition that all `.TextGrid` files have to fulfill is that they must all contain the same tiers. If this is not the case, yet there is an equal tier subset that is contained in all the TextGrid files, this equal subset may be chosen. For example, if all `.TextGrid` files contain only the tier `Phonetic: IntervalTier` the conversion will work. However, if a single `.TextGrid` of the collection has the additional tier `Tone: TextTier` the conversion will fail. In this case the conversion could be made to work by specifying the equal subset (e.g., `equalSubset = c("Phonetic")`) and passing it on to the `tierNames` function argument `convert_TextGridCollection(..., tierNames = equalSubset, ...)`. As can be seen in Figure @ref(fig:msajc003_praatTG), the TextGrid files provided by the demo data contain eleven tiers. To reduce the complexity of the annotations for this tutorial we will only convert the tiers *Text* (orthographic word annotations), *Syllable* (strong: *S* vs. weak: *W* syllable annotations), *Phoneme* (phoneme level annotations) and *Phonetic* (phonetic annotations) utilizing the `tierNames` parameter. This conversion can be seen in R Example @ref(rexample:tutorial_tgconv).

```
# convert TextGrid collection to the emuDB format
convert_TextGridCollection(dir = tgColDir,
                           dbName = "myFirst",
                           targetDir = tempdir(),
                           tierNames = c("Text", "Syllable","Phoneme", "Phonetic"))
```

The above call to `convert_TextGridCollection` creates a new `emuDB` directory in the `tempdir()` directory called `myFirst_emuDB`. This `emuDB` contains annotation files that contain the same *Text*, *Syllable*, *Phoneme* and *Phonetic* segment tiers as the original `.TextGrid` files as well as copies of the original (`.wav`) audio files. For further details about the structure of an `emuDB`, see Chapter @ref(chap:emuDB) of this document.

Loading and inspecting the database

As mentioned in Section @ref(sec:overview_sysArch), the first step when working with an `emuDB` is to load it into the current R session. R Example @ref(rexample:tutorial_loadEmuDB) shows how to load the converted TextGrid collection into R using the `load_emuDB()` function.

```
# get path to emuDB called "myFirst"
# that was created by convert_TextGridCollection()
path2directory = file.path(tempdir(), "myFirst_emuDB")

# load emuDB into current R session
dbHandle = load_emuDB(path2directory, verbose = FALSE)
```

Overview

Now the *myFirst* `emuDB` is loaded into R, an overview of the current status and configuration of the database can be displayed using the `summary()` function as shown in R Example @ref(rexample:tutorial_summary).

```
# show summary
summary(dbHandle)
```

```
## Name:     myFirst
## UUID:     35c49489-cce1-49da-9171-90e8a02f7384
## Directory:    /private/var/folders/vh/j2k1_0395x5_sgzpbl4bzzl00000gn/T/Rtmp94k1Po/myFirst_emuDB
## Session count: 1
## Bundle count: 7
## Annotation item count:  664
## Label count:  664
```

```
## Link count:  0
##
## Database configuration:
##
## SSFF track definitions:
## NULL
##
## Level definitions:
##       name    type nrOfAttrDefs attrDefNames
## 1     Text SEGMENT            1        Text;
## 2 Syllable SEGMENT            1    Syllable;
## 3  Phoneme SEGMENT            1     Phoneme;
## 4 Phonetic SEGMENT            1    Phonetic;
##
## Link definitions:
## NULL
```

The extensive output of `summary()` is split into a top and bottom half, where the top half focuses on general information about the database (name, directory, annotation item count, etc.) and the bottom half displays information about the various Simple Signal File Format (SSFF) track, level and link definitions of the `emuDB`. The summary information about the level definitions shows, for instance, that the *myFirst* database has a *Text* level of type `SEGMENT` and therefore contains annotation items that have a start time and a segment duration. It is worth noting that information about the SSFF track, level and link definitions corresponds to the output of the `list_ssffTrackDefinitions()`, `list_levelDefinitions()` and `list_linkDefinitions()` functions.

Database annotation and visual inspection

The EMU-SDMS has a unique approach to annotating and visually inspecting data-bases, as it utilizes a web application called the `EMU-webApp` to act as its GUI. To be able to communicate with the web application the `emuDB` package provides a `serve()` function which is used in R Example @ref(rexample:tutorial_serve).

```
# serve myFirst emuDB to the EMU-webApp
serve(dbHandle)
```

Executing this command will block the R console, automatically open up the system's default browser and display the following message in the R console:

```
## Navigate your browser to the EMU-webApp URL:
##  http://ips-lmu.github.io/EMU-webApp/ (should happen automatically)

## Server connection URL:
##  ws://localhost:17890

## To stop the server press the 'clear' button in the
## EMU-webApp or close/reload the webApp in your browser.
```

The `EMU-webApp`, which is now connected to the database via the `serve()` function, can be used to visually inspect and annotate the `emuDB`. Figure @ref(fig:tutorial_emuWebAppMyFirst) displays a screenshot of what the `EMU-webApp` looks like after automatically connecting to the server. As the `EMU-webApp` is a very feature-rich software annotation tool, this documentation has a whole chapter (see Chapter @ref(chap:emu-webApp)) on how to use it, what it is capable of and how to configure it. Further, the web application provides its own documentation which can be accessed by clicking the EMU icon in the top right hand corner of the application's top menu bar. To close the connection and free up the blocked R console, simply click the `clear` button in the top menu bar of the `EMU-webApp`.

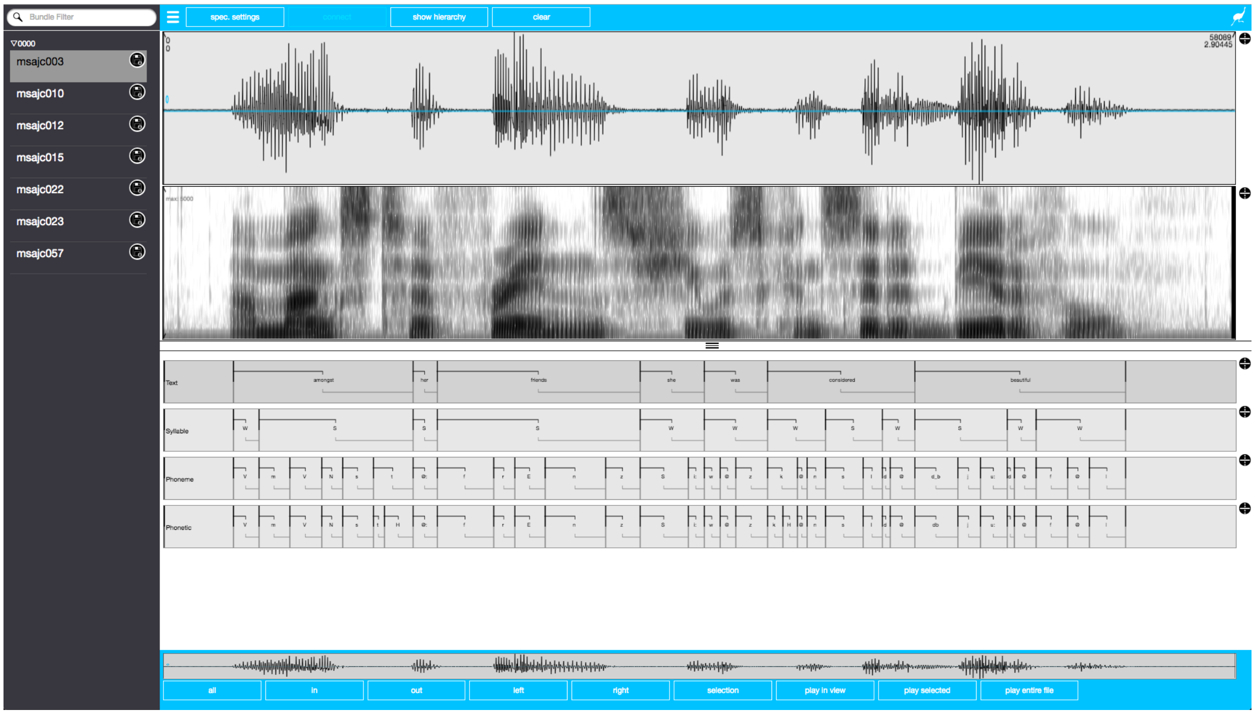

Figure 3: Screenshot of `EMU-webApp` displaying `msajc003` bundle of *myFirst* `emuDB`.(#fig:tutorial_emuWebAppMyFirst)

Querying and autobuilding the annotation structure

An integral step in the default workflow of the EMU-SDMS is querying the annotations of a database. The `emuDB` package implements a `query()` function to accomplish this task. This function evaluates an EMU Query Language (EQL) expression and extracts the annotation items from the database that match a query expression. As Chapter @ref(chap:querysys) gives a detailed description of the query mechanics provided by `emuDB`, this tutorial will only use a very small, hopefully easy to understand subset of the EQL.

The output of the `summary()` command in R Example @ref(rexample:tutorial_summary) and the screenshot in Figure @ref(fig:tutorial_emuWebAppMyFirst) show that the *myFirst* `emuDB` contains four levels of annotations. R Example @ref(rexample:tutorial_simpleQuery) shows four separate queries that query various segments on each of the available levels. The query expressions all use the matching operator `==` which returns annotation items whose labels match those specified to the right of the operator and that belong to the level specified to the left of the operator (i.e., `LEVEL == LABEL`; see Chapter @ref(chap:querysys) for a detailed description).

```
# query all segments containing the label
# "was" of the "Phonetic" level
sl_text = query(emuDBhandle = dbHandle,
                query = "Text == was")

# query all segments containing the label
# "S" (==strong syllable) of the "Syllable" level
sl_syl = query(emuDBhandle = dbHandle,
               query = "Syllable == S")

# query all segments containing the label
# "n" on the "Phoneme" level
```

```
sl_phoneme = query(dbHandle,
                   query = "Phoneme == f")

# query all segments containing the label
# "n" of the "Phonetic" level
sl_phonetic = query(dbHandle,
                    query = "Phonetic == E")

# show class vector of query result
class(sl_phonetic)
```

```
## [1] "emuRsegs"   "emusegs"    "data.frame"
```

```
# show first entry of sl
head(sl_phonetic, n = 1)
```

```
## segment  list from database:  myFirst
## query was:  Phonetic == E
##   labels   start      end session   bundle    level    type
## 1      E 949.925 1031.925    0000 msajc003 Phonetic SEGMENT
```

```
# show summary of sl
summary(sl_phonetic)
```

```
## segment  list from database:  myFirst
## query was:  Phonetic == E
##  with 8 segments
##
## Segment distribution:
##
## E
## 8
```

As demonstrated in R Example @ref(rexample:tutorial_simpleQuery), the result of a query is an `emuRsegs` object, which is a super-class of the common `data.frame`. This object is often referred to as a segment list, or "seglist". A segment list carries information about the extracted annotation items such as the extracted labels, the start and end times of the segments, the sessions and bundles the items are from and the levels they belong to. An in-depth description of the information contained in a segment list is given in Section @ref(sec:query_emuRsegs). R Example @ref(rexample:tutorial_simpleQuery) shows that the `summary()` function can also be applied to a segment list object to get an overview of what is contained within it. This can be especially useful when dealing with larger segment lists.

Autobuilding

The simple queries illustrated above query segments from a single level that match a certain label. However, the EMU-SDMS offers a mechanism for performing inter-level queries such as: *Query all Phonetic items that contain the label "n" and are part of a strong syllable*. For such queries to be possible, the EMU-SDMS offers very sophisticated annotation structure modeling capabilities, which are described in Chapter @ref(chap:annot_struct_mod). For the sake of this tutorial we will focus on converting the flat segment level annotation structure displayed in Figure @ref(fig:tutorial_emuWebAppMyFirst) to a hierarchical form as displayed in Figure @ref(fig:tutorial_violentlyHier), where only the *Phonetic* level carries time information and the annotation items on the other levels are explicitly linked to each other to form a hierarchical annotation structure.

As it is a very laborious task to manually link annotation items together using the `EMU-webApp` and the hierarchical information is already implicitly contained in the time information of the segments and events of each

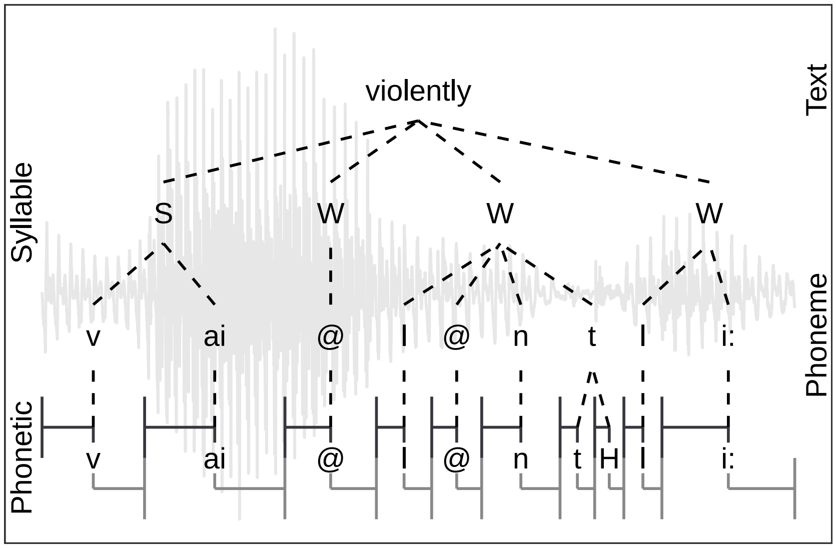

Figure 4: Example of a hierarchical annotation of the word *violently* belonging to the *msajc012* bundle of the *myFirst* demo `emuDB`.

level, we will now use a simple function provided by the `emuDB` package to build these hierarchical structures using this information called `autobuild_linkFromTimes()`. R Example @ref(rexample:tutorial_autobuild) shows the calls to this function which autobuild the hierarchical annotations in the *myFirst* database. As a general rule for autobuilding hierarchical annotation structures, a good strategy is to start the autobuilding process beginning with coarser grained annotation levels (i.e., the *Text*/*Syllable* level pair in our example) and work down to finer grained annotations (i.e., the *Syllable*/*Phoneme* and *Phoneme*/*Phonetic* level pairs in our example). To build hierachical annotation structures we need link definitions, which together with the level definitions define the annotation structure for the entire database (see Chapter @ref(chap:annot_struct_mod) for further details). The `autobuild_linkFromTimes()` calls in R Example @ref(rexample:tutorial_autobuild) use the `newLinkDefType` parameter, which if defined automatically adds a link definition to the database.

```
# invoke autobuild function
# for "Text" and "Syllable" levels
autobuild_linkFromTimes(dbHandle,
                        superlevelName = "Text",
                        sublevelName = "Syllable",
                        convertSuperlevel = TRUE,
                        newLinkDefType = "ONE_TO_MANY")

# invoke autobuild function
# for "Syllable" and "Phoneme" levels
autobuild_linkFromTimes(dbHandle,
                        superlevelName = "Syllable",
                        sublevelName = "Phoneme",
                        convertSuperlevel = TRUE,
                        newLinkDefType = "ONE_TO_MANY")

# invoke autobuild function
# for "Phoneme" and "Phonetic" levels
autobuild_linkFromTimes(dbHandle,
                        superlevelName = "Phoneme",
                        sublevelName = "Phonetic",
                        convertSuperlevel = TRUE,
```

```r
                         newLinkDefType = "MANY_TO_MANY")
```

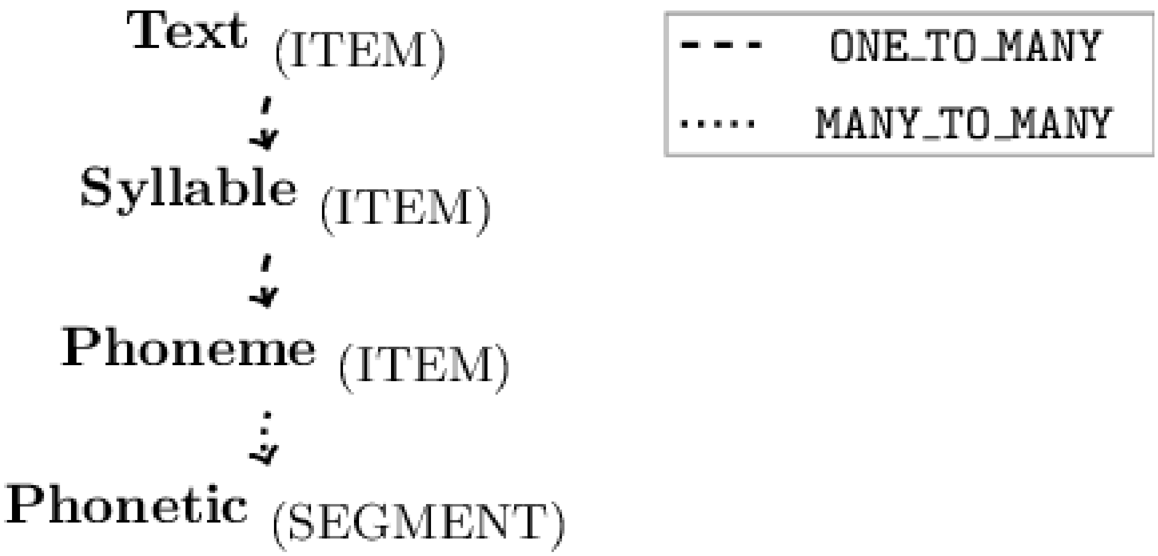

Figure 5: Schematic annotation structure of the `emuDB` after calling the autobuild function in R Example @ref(rexample:tutorial_autobuild).(#fig:tutorial_simpleAnnotStruct)

As the `autobuild_linkFromTimes()` function automatically creates backup levels to avoid the accidental loss of boundary or event time information, R Example @ref(rexample:tutorial_delBackupLevels) shows how these backup levels can be removed to clean up the database. However, using the `remove_levelDefinition()` function with its `force` parameter set to `TRUE` is a very invasive action. Usually this would not be recommended, but for this tutorial we are keeping everything as clean as possible.

```r
# list level definitions
# as this reveals the "-autobuildBackup" levels
# added by the autobuild_linkFromTimes() calls
list_levelDefinitions(dbHandle)
```

```
##                        name    type nrOfAttrDefs              attrDefNames
## 1                      Text    ITEM            1                     Text;
## 2                  Syllable    ITEM            1                 Syllable;
## 3                   Phoneme    ITEM            1                  Phoneme;
## 4                  Phonetic SEGMENT            1                 Phonetic;
## 5      Text-autobuildBackup SEGMENT            1     Text-autobuildBackup;
## 6  Syllable-autobuildBackup SEGMENT            1 Syllable-autobuildBackup;
## 7   Phoneme-autobuildBackup SEGMENT            1  Phoneme-autobuildBackup;
```

```r
# remove the levels containing the "-autobuildBackup"
# suffix
remove_levelDefinition(dbHandle,
                       name = "Text-autobuildBackup",
                       force = TRUE,
                       verbose = FALSE)

remove_levelDefinition(dbHandle,
                       name = "Syllable-autobuildBackup",
                       force = TRUE,
                       verbose = FALSE)
```

```r
remove_levelDefinition(dbHandle,
                       name = "Phoneme-autobuildBackup",
                       force = TRUE,
                       verbose = FALSE)

# list level definitions
list_levelDefinitions(dbHandle)
```

```
##       name    type nrOfAttrDefs attrDefNames
## 1     Text    ITEM            1        Text;
## 2 Syllable    ITEM            1    Syllable;
## 3  Phoneme    ITEM            1     Phoneme;
## 4 Phonetic SEGMENT            1    Phonetic;
```

```r
# list level definitions
# which were added by the autobuild functions
list_linkDefinitions(dbHandle)
```

```
##           type superlevelName sublevelName
## 1  ONE_TO_MANY           Text     Syllable
## 2  ONE_TO_MANY       Syllable      Phoneme
## 3 MANY_TO_MANY        Phoneme     Phonetic
```

As can be seen by the output of `list_levelDefinitions()` and \ `list_linkDefinitions()` in R Example @ref(rexample:tutorial_autobuild), the annotation structure of the *myFirst* `emuDB` now matches that displayed in Figure @ref(fig:tutorial_simpleAnnotStruct). Using the `serve()` function to open the `emuDB` in the `EMU-webApp` followed by clicking on the `show hierarchy` button in the top menu (and rotating the hierarchy by 90 degrees by clicking the `rotate by 90 degrees` button) will result in a view similar to the screenshot of Figure @ref(fig:tutorial_EMU-webAppScreenshotTutorialPostAutobHier).

Querying the hierarchical annotations

Having this hierarchical annotation structure now allows us to formulate a query that helps answer the originally stated question: *Given an annotated speech database, is vowel height (measured by its correlate, the first formant frequency) influenced by whether it appears in a strong or weak syllable?*. To keep things simple, here we will focus only on the vowels *i:*, *o:* and *V* (SAMPA annotation TODO: cite + mention SAMPA annotation above). R Example @ref(rexample:tutorial_labelGroupQuery) shows how all the vowels in the *myFirst* database are queried.

```r
# query the label group on the Phonetic level
sl_vowels = query(dbHandle, "Phonetic == i: | o: | V ")

# show first entry of sl
head(sl_vowels, n = 1)
```

```
## segment  list from database:  myFirst
## query was:  Phonetic == i: | o: | V
##   labels   start     end session   bundle    level    type
## 1      V 187.425 256.925    0000 msajc003 Phonetic SEGMENT
```

As the type of syllable (strong vs. week) for each vowel that was just extracted is also needed, we can use the requery functionality of the EMU-SDMS (see Chapter @ref(chap:querysys)) to retrieve the syllable type for each vowel. A requery essentially moves through a hierarchical annotation (vertically or horizontally) starting from the segments that are passed into the requery function. R Example @ref(rexample:tutorial_requery)

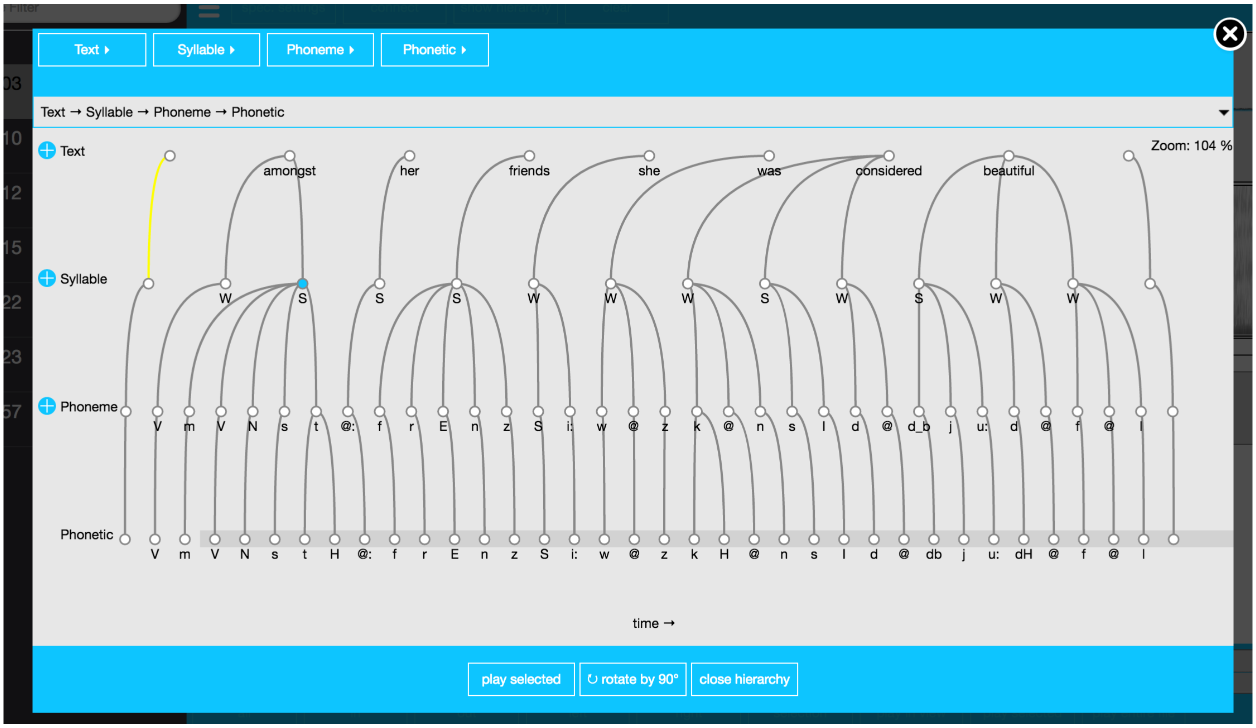

Figure 6: Screenshot of `EMU-webApp` displaying the autobuilt hierarchy of the *myFirst* `emuDB`.(#fig:tutorial_EMU-webAppScreenshotTutorialPostAutobHier)

illustrates the usage of the hierarchical requery function, `requery_hier()`, to retrieve the appropriate annotation items from the *Syllable* level.

```
# hierarchical requery starting from the items in sl_vowels
# and moving up to the "Syllable" level
sl_sylType = requery_hier(dbHandle,
                          seglist = sl_vowels,
                          level = "Syllable")

# show first entry of sl
head(sl_sylType, n = 1)
```

```
## segment  list from database:  myFirst
## query was:  FROM REQUERY
##    labels    start     end session   bundle    level type
## 1       W 187.425 256.925    0000 msajc003 Syllable ITEM
```

```
# show that sl_vowel and sl_sylType have the
# same number of row entries
nrow(sl_vowels) == nrow(sl_sylType)
```

```
## [1] TRUE
```

As can be seen by the `nrow()` comparison in R Example @ref(rexample:tutorial_requery), the segment list returned by the `requery_hier()` function has the same number of rows as the original `sl_vowels` segment list. This is important, as each row of both segment lists line up and allow us to infer which segment belongs to which syllable (e.g., vowel `sl_vowels[5,]` belongs to syllable `sl_sylType[5,]`).

Signal extraction and exploration

Now that the vowel and syllable type information including the vowel start and end time information has been extracted from the database, this information can be used to extract signal data that matches these segments. Using the `emuDB` function `get_trackdata()` we can calculate the formant values in real time using the formant estimation function, `forest()`, provided by the `wrassp` package (see Chapter @ref(chap:wrassp) for details). R Example @ref(rexample:tutorial_getTrackdata) shows the usage of this function.

```
# get formant values
td_vowels = get_trackdata(dbHandle,
                          seglist = sl_vowels,
                          onTheFlyFunctionName = "forest",
                          resultType = "emuRtrackdata",
                          verbose = F)

# show class vector
class(td_vowels)
```

```
## [1] "emuRtrackdata" "data.table"    "data.frame"
```

```
# show number of rows
nrow(td_vowels)
```

```
## [1] 220
```

```
# show vector indicating which row belongs to which
# segment list entry
td_vowels$sl_rowIdx
```

```
##   [1]  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2
##  [24]  2  2  2  2  2  2  2  2  3  3  3  3  3  3  3  3  3  4  4  4  4  4  4
##  [47]  4  4  4  4  4  4  4  4  4  5  5  5  5  5  5  5  5  5  5  5  5  5  5
##  [70]  5  5  5  5  5  5  5  5  5  5  5  5  5  6  6  6  6  6  6  6  6  6  6
##  [93]  6  6  6  6  6  6  6  6  6  6  6  6  6  6  7  7  7  7  7  7  7  7  7
## [116]  7  7  7  7  7  7  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8
## [139]  8  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9 10 10 10 10 10
## [162] 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 11 11 11 11 11
## [185] 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 12 12 12 12 12
## [208] 12 12 12 12 12 12 12 12 12 12 12 12 12
```

```
# show head of data belonging to sl_vowels[5,]
head(td_vowels[td_vowels$sl_rowIdx == 5,], n = 1)
```

```
##    sl_rowIdx labels    start      end          utts
## 1:         5     o: 1183.975 1322.425 0000:msajc012
##                                db_uuid session   bundle start_item_id
## 1: 35c49489-cce1-49da-9171-90e8a02f7384    0000 msajc012           103
##    end_item_id    level start_item_seq_idx end_item_seq_idx    type
## 1:         103 Phonetic                 15               15 SEGMENT
##    sample_start sample_end sample_rate times_rel times_orig  T1  T2   T3
## 1:        23680      26448       20000         0     1187.5 244 928 1827
##      T4
## 1: 3417
```

As can be seen by the call to the `class()` function, the resulting object is of the type `emuRtrackdata` and has 220 rows. The `td_vowels$sl_rowIdx` column of the `td_vowels` object is vital as it indicates which row of `td_vowels` belongs to which row in `sl_vowels`. As the columns `T1`, `T2`, `T3`, `T4` of the printed output of `head(sl_vowels[td_vowels$sl_rowIdx == 5,], n = 1)` suggest, the `forest` function estimates four formant values. We will only be concerned with the first (column `T1`) and second (column `T2`). R Example @ref(rexample:tutorial_dplot) shows a call to `ggplot()` which produces the plot displayed in Figure @ref(fig:tutorial_dplot). The call to the `ggplot()` function plots all 12 first formant trajectories (achieved by `group = sl_rowIdx`). The 12 trajectories are color coded by vowel label (`col = labels`).

```
ggplot(td_vowels) +
  aes(x=times_rel, y=T1, col=labels, group=sl_rowIdx) +
  geom_line() +
  labs(x = "vowel duration", y = "F1 (Hz)")
```

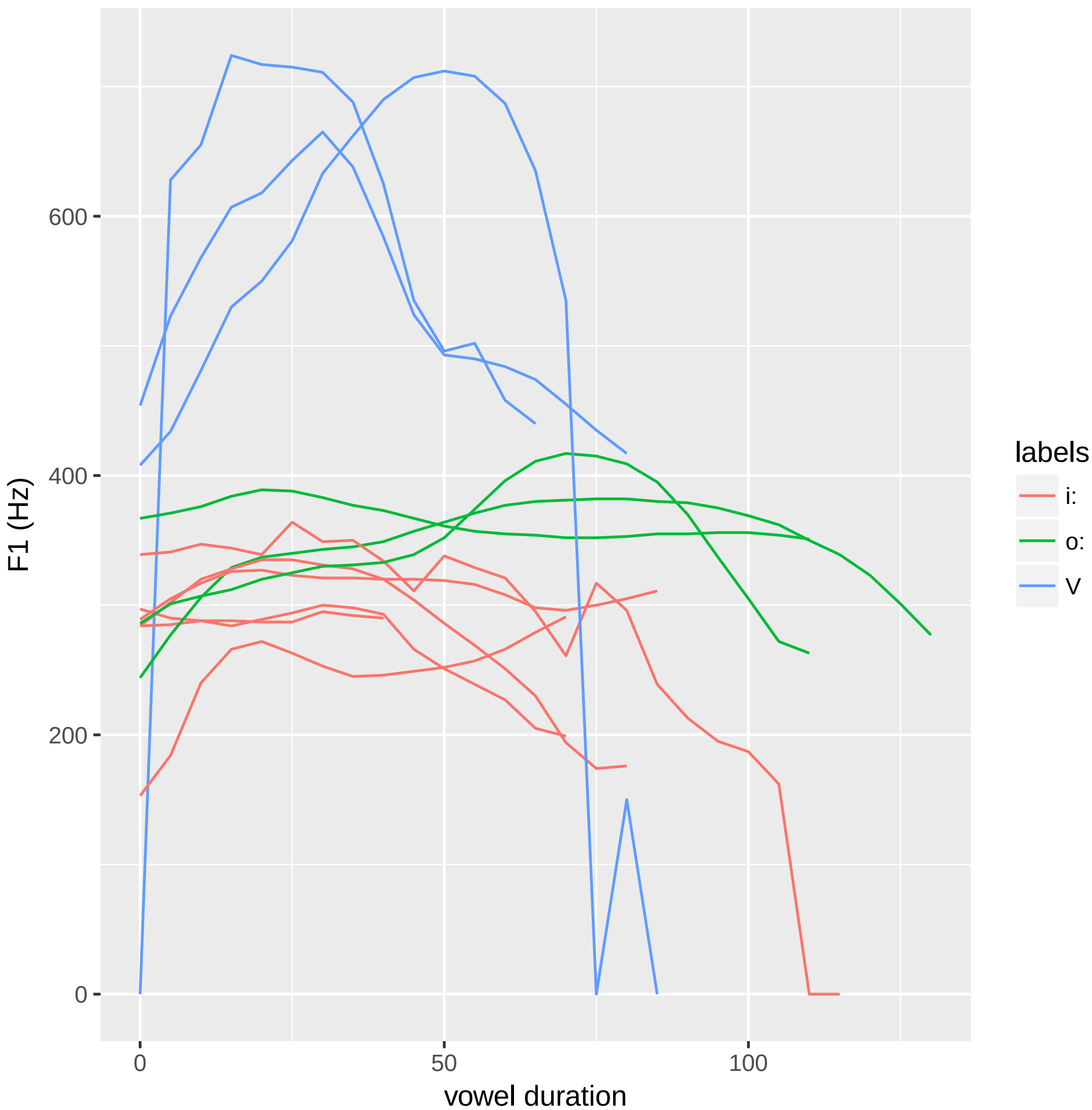

Figure 7: `dplot()` plots of F1 trajectories. The left plot displays 81 trajectories while the right plot displays ensemble averages of each vowel.

Figure @ref(fig:tutorial_dplot) gives an overview of the first formant trajectories by vowel class. For the purpose of data exploration and to get an idea of where the individual vowel classes lie on the F2 x F1 plane, which indirectly provides information about vowel height and tongue position, R Example @ref(rexample:tutorial_eplot) once again makes use of the `ggplot()` function. This produces Figure @ref(fig:tutorial_eplot). To be able to plot two dimensional data, the `td_vowels` object first has to be modified/re-extracted, as it contains entire formant trajectories but two dimensional data is needed to be able to display it on the F2 x F1 plain. This can, for example, be achieved by only re-extracting temporal mid-point formant values for each vowel using the `get_trackdata()` function utilizing its `cut` parameter. R Example @ref(rexample:tutorial_eplot) shows this approach by setting the `cut` parameter to `0.5`.

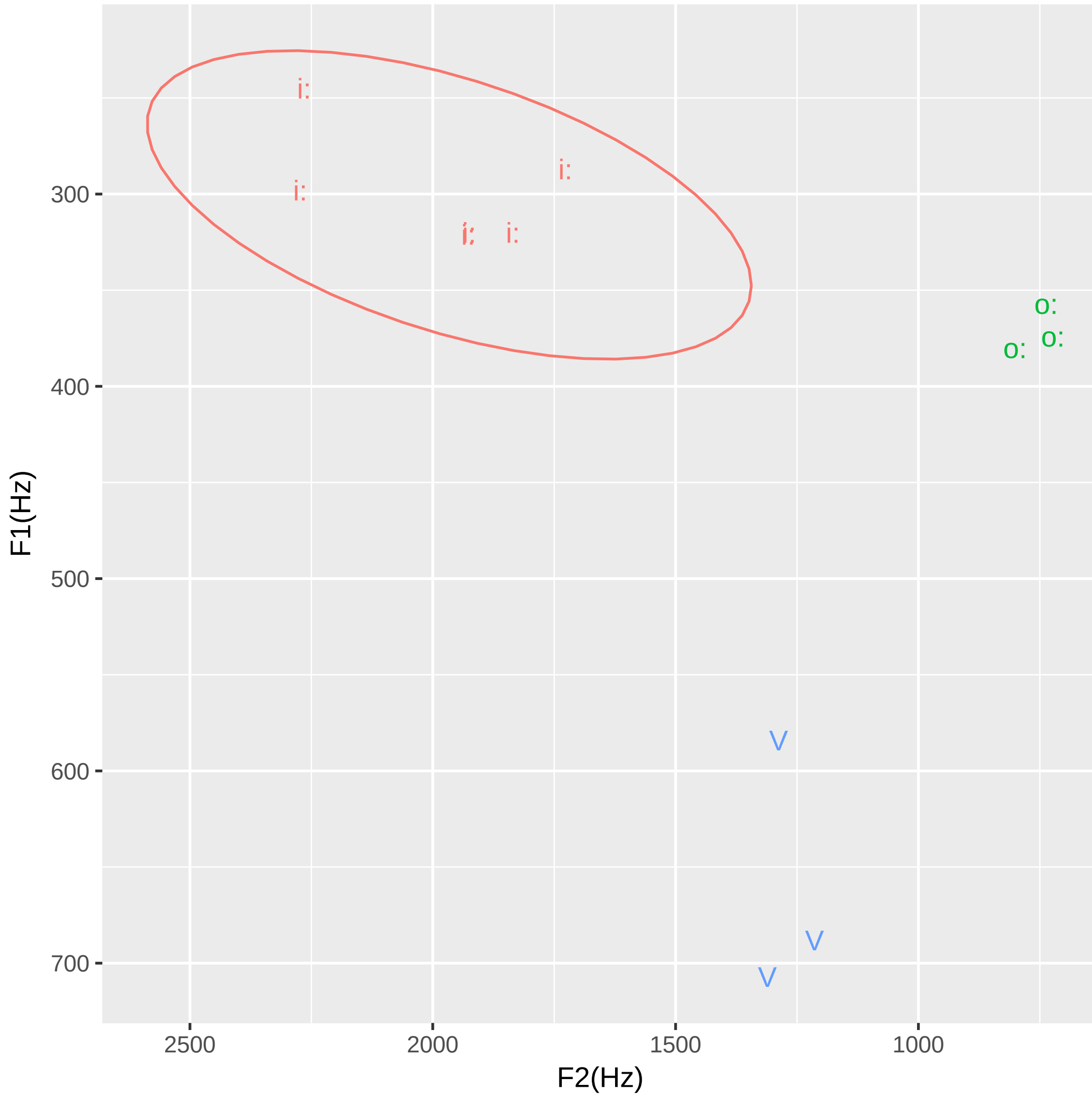

Figure 8: 95% ellipses for F2 x F1 data extracted from the temporal midpoint separated by vowel.

Figure @ref(fig:tutorial_eplot) displays the first two formants extracted at the temporal midpoint of every vowel in `sl_vowels`. These formants are plotted on the F2 x F1 plane, and their 95% ellipsis distributions are also shown (note that a minimum of four points have to be present for an ellipsis to be drawn). Although not necessarily applicable to the question posed at the beginning of this tutorial, the data exploration using the `ggplot()` function can be a very helpful tool for providing an overview of the data at hand.

Vowel height as a function of syllable types (strong vs. weak): evaluation and statistical analysis

The above data exploration only dealt with the actual vowels and disregarded the syllable type they occurred in. However, the question in the introduction of this chapter does not distinguish between vowel classes but focuses on whether a vowel occurs in a strong or weak syllable. For the sake of this tutorial we will disregard the fact that the vowel class influences the syllable type and only focus on whether a vowel occurred in a syllable labeled *S* (strong) or *W* (weak). For data inspection purposes, R Example @ref(rexample:tutorial_dplotSylTyp) initially replaces the labels of `sl_vowels` with those of `sl_sylType` re-extracts the formant trajectories and displays them using `geom_line()`.

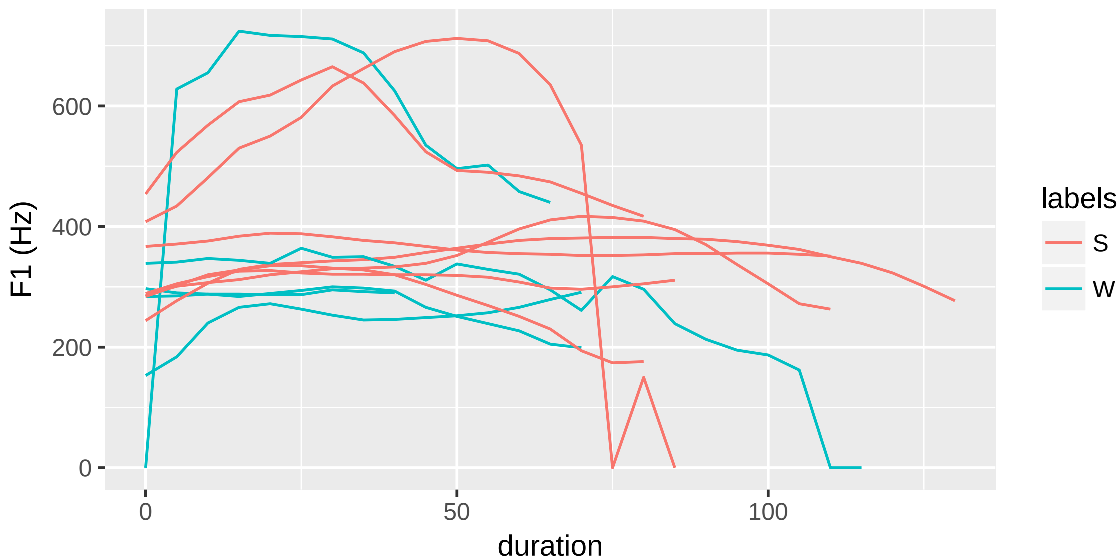

Figure 9: Ensemble averages of F1 contours of all tokens of the central 60% of vowels grouped by syllable type (strong (S) vs. weak (W)).

As can be seen in Figure @ref(fig:tutorial_dplotSylTyp), there seems to be a distinction in F1 trajectory height between vowels in strong syllables and weak syllables. R Example @ref(rexample:tutorial_boxplot) shows the code to produce a boxplot using the `dplyr` and `ggplot2` packages to further visually inspect the data (see Figure @ref(fig:tutorial_boxplot) for the plot produced by R Example @ref(rexample:tutorial_boxplot)).

TODO: Simple statistical analysis

Conclusion

The tutorial given in this chapter gave an overview of what it is like working with the EMU-SDMS to try to solve a research question. As many of the concepts were only briefly explained, it is worth noting that explicit explanations of the various components and integral concepts are given in following chapters. Further, additional use cases that have been taken from the `emuR_intro` vignette can be found in Appendix @ref(app_chap:useCases). These use cases act as templates for various types of research questions and will hopefully aid the user in finding a solution similar to what she or he wishes to achieve.

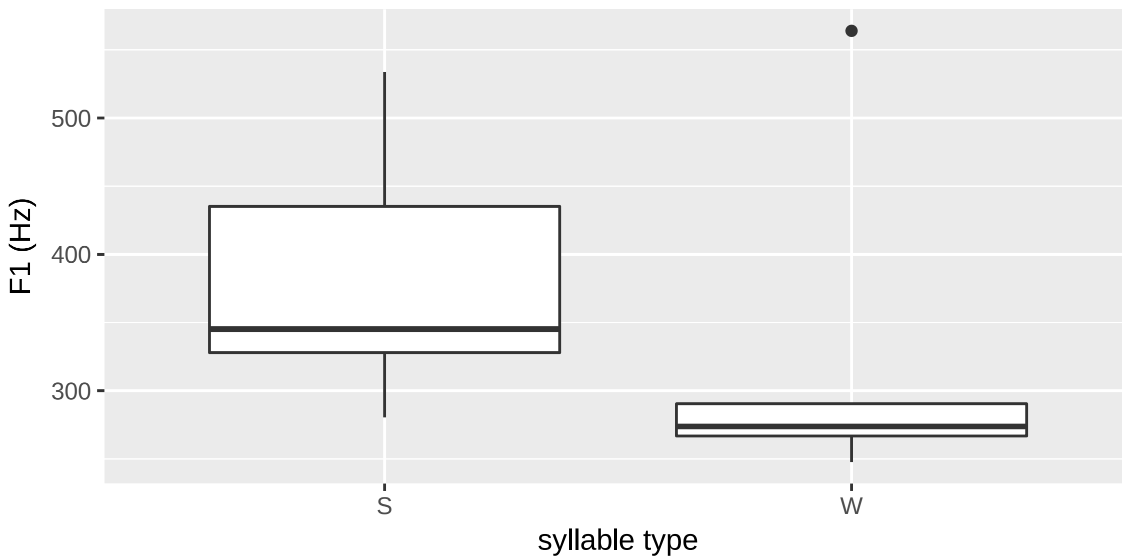

Figure 10: Boxplot produced using `ggplot2` to visualize the difference in F1 depending on whether the vowel occurs in strong (S) or weak (W) syllables.