The EMU-SDMS Manual

Raphael Winkelmann

Contents

Welcome

Welcome to the EMU-SDMS Manual!

Disclaimer: This manual is still in the making! It will eventually replace all the vignettes of emuR, wrassp as well as the EMU-webApp's own documentation. This manual is intended to consolidate all of this information in one easy to find location that can easily be updated as new features are added.

Chapter 1

Installing the EMU-SDMS

1. R
   - Download the R programming language from https://cran.r-project.org/
   - Install the R programming language by executing the downloaded file and following the on-screen instructions.
2. `emuR`
   - Start up R.
   - Enter `install.packages("emuR")` after the `>` prompt to install the package. (You will only need to repeat this if package updates become available.)
   - As the `wrassp` package is a dependency of the `emuR` package, it does not have to be installed separately.
3. `EMU-webApp` (prerequisite)
   - The only thing needed to use the `EMU-webApp` is a current HTML5 compatible browser (Chrome/Firefox/Safari/Opera/…). However, as most of the development and testing is done using Chrome we recommend using it, as it is by far the best tested browser.

1.1 Version disclaimer

This document describes the following versions of the software components:

- `wrassp`
  - Package version: 0.1.6
  - Git tag name: v0.1.6 (on master branch)
- `emuR`
  - Package version: 1.0.0
  - Git tag name: v1.0.0 (on master branch)
- `EMU-webApp`
  - Version: 0.1.12
  - Git SHA1: 7b044a9f9fe19f2eb6d03ec6ec3f20d5b1d25db2

As the development of the EMU Speech Database Management System is still ongoing, be sure you have the correct documentation to go with the version you are using.

1.2 For developers and people interested in the source code

The information on how to install and/or access the source code of the developer version including the possibility of accessing the versions described in this document (via the Git tag names mentioned above) is

given below.

- `wrassp`
  - Source code is available here: https://github.com/IPS-LMU/wrassp/
  - Install developer version in R: `install.packages("devtools"); library("devtools"); install_github("IPS-LMU/wrassp")`
  - Bug reports: https://github.com/IPS-LMU/wrassp/issues
- `emuR`
  - Source code is available here: https://github.com/IPS-LMU/emuR/
  - Install developer version in R: `install.packages("devtools"); library("devtools"); install_github("IPS-LMU/emuR")`
  - Bug reports: https://github.com/IPS-LMU/emuR/issues
- `EMU-webApp`
  - Source code is available here: https://github.com/IPS-LMU/EMU-webApp/
  - Bug reports: https://github.com/IPS-LMU/EMU-webApp/issues

Part I

Overview and tutorial

Chapter 2

An overview of the EMU-SDMS [1]

The EMU Speech Database Management System (EMU-SDMS) is a collection of software tools which aims to be as close to an all-in-one solution for generating, manipulating, querying, analyzing and managing speech databases as possible. It was developed to fill the void in the landscape of software tools for the speech sciences by providing an integrated system that is centered around the R language and environment for statistical computing and graphics (R Core Team (2016)). This manual contains the documentation for the three software components `wrassp`, `emuR` and the `EMU-webApp`. In addition, it provides an in-depth description of the `emuDB` database format which is also considered an integral part of the new system. These four components comprise the EMU-SDMS and benefit the speech sciences and spoken language research by providing an integrated system to answer research questions such as: *Given an annotated speech database, is the vowel height of the vowel @ (measured by its correlate, the first formant frequency) influenced by whether it appears in a strong or weak syllable?*

This manual is targeted at new EMU-SDMS users as well as users familiar with the legacy EMU system. In addition, it is aimed at people who are interested in the technical details such as data structures/formats and implementation strategies, be it for reimplementation purposes or simply for a better understanding of the inner workings of the new system. To accommodate these different target groups, after initially giving an overview of the system, this manual presents a usage tutorial that walks the user through the entire process of answering a research question. This tutorial will start with a set of `.wav` audio and Praat `.TextGrid` (Boersma and Weenink (2016)) annotation files and end with a statistical analysis to address the hypothesis posed by the research question. The following Part **??** of this documentation is separated into six chapters that give an in-depth explanation of the various components that comprise the EMU-SDMS and integral concepts of the new system. These chapters provide a tutorial-like overview by providing multiple examples.

[1] Sections of this chapter have been published in Winkelmann et al. (2017)

To give the reader a synopsis of the main functions and central objects that are provided by EMU-SDMS's main R package `emuR`, an overview of these functions is presented in Part **??**. Part **??** focuses on the actual implementation of the components and is geared towards people interested in the technical details. Further examples and file format descriptions are available in various appendices. This structure enables the novice EMU-SDMS user to simply skip the technical details and still get an in-depth overview of how to work with the new system and discover what it is capable of.

A prerequisite that is presumed throughout this document is the reader's familiarity with basic terminology in the speech sciences (e.g., familiarity with the international phonetic alphabet (IPA) and how speech is annotated at a coarse and fine grained level). Further, we assume the reader has a grasp of the basic concepts of the R language and environment for statistical computing and graphics. For readers new to R, there are multiple, freely available R tutorials online (e.g., https://en.wikibooks.org/wiki/Statistical_Analysis:_an_Introduction_using_R/R_basics). R also has a set of very detailed manuals and tutorials that come preinstalled with R. To be able to access R's own "An Introduction to R" introduction, simply type `help.start()` into the R console and click on the link to the tutorial.

2.1 The evolution of the EMU-SDMS

The EMU-SDMS has a number of predecessors that have been continuously developed over a number of years (e.g., Harrington et al. (1993), Cassidy and Harrington (1996), Cassidy and Harrington (2001), Bombien et al. (2006), Harrington (2010), John (2012)). The components presented here are the completely rewritten and newly designed, next incarnation of the EMU system, which we will refer to as the EMU Speech Database Management System (EMU-SDMS). The EMU-SDMS keeps most of the core concepts of the previous system, which we will refer to as the legacy system, in place while improving on things like usability, maintainability, scalability, stability, speed and more. We feel the redesign and reimplementation elevates the system into a modern set of speech and language tools that enables a workflow adapted to the challenges confronting speech scientists and the ever growing size of speech databases. The redesign has enabled us to implement several components of the new EMU-SDMS so that they can be used independently of the EMU-SDMS for tasks such as web-based collaborative annotation efforts and performing speech signal processing in a statistical programming environment. Nevertheless, the main goal of the redesign and reimplementation was to provide a modern set of tools that reduces the complexity of the tool chain needed to answer spoken language research questions down to a few interoperable tools. The tools the EMU-SDMS provides are designed to streamline the process of obtaining usable data, all from within an environment that can also be used to analyze, visualize and statistically evaluate the data.

Upon developing the new system, rather than starting completely from scratch it seemed more appropriate to partially reuse the concepts of the legacy system in order to achieve our goals. A major observation at the time was that the R language and environment for statistical computing and graphics (R Core Team (2016)) was gaining more and more traction for statistical and data visualization purposes in the speech and spoken language research community. However, R was mostly only used towards the end of the data analysis chain where data usually was pre-converted into a comma-separated values or equivalent file format by the user using other tools to calculate, extract and pre-process the data. While designing the new EMU-SDMS, we brought R to the front of the tool chain to the point just beyond data acquisition. This allows the entire data annotation, data extraction and analysis process to be completed in R, while keeping the key user requirements in mind. Due to personal experiences gained by using the legacy system for research puposes and in various undergraduate courses (course material usually based on Harrington (2010)), we learned that the key user requirements were data and database portability, a simple installation process, a simplified/streamlined user experience and cross-platform availability. Supplying all of EMU-SDMS's core functionality in the form of R packages that do not rely on external software at runtime seemed to meet all of these requirements.

As the early incarnations of the legacy EMU system and its predecessors were conceived either at a time that predated the R system or during the infancy of R's package ecosystem, the legacy system was implemented as a modular yet composite standalone program with a communication and data exchange interface to the

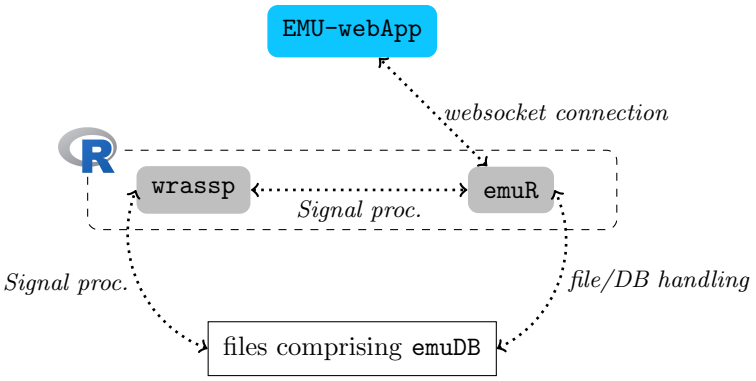


Figure 2.1: Schematic architecture of the EMU-SDMS

R/Splus systems (see Cassidy and Harrington (2001) Section 3 for details). Recent developments in the package ecosystem of R such as the availability of the `DBI` package (R Special Interest Group on Databases (R-SIG-DB) et al. (2016)) and the related packages `RSQLite` and `RPostgreSQL` (Wickham et al. (2014), Conway et al. (2016)), as well as the `jsonlite` package (Ooms (2014)) and the `httpuv` package (RStudio and Inc. (2015)), have made R an attractive sole target platform for the EMU-SDMS. These and other packages provide additional functional power that enabled the EMU-SDMS's core functionality to be implemented in the form of R packages. The availability of certain R packages had a large impact on the architectural design decisions that we made for the new system.

R Example **??** shows the simple installation process which we were able to achieve due to the R package infrastructure. Compared to the legacy EMU and other systems, the installation process of the entire system has been reduced to a single R command. Throughout this documentation we will try to highlight how the EMU-SDMS is also able to meet the rest of the above key user requirements.

rexample:overview-install

```
# install the entire EMU-SDMS
# by installing the emuR package
install.packages("emuR")
```

It is worth noting that throughout this manual R Example code snippets will be given in the form of R Example **??**. These examples represent working R code that allow the reader to follow along in a hands-on manor and give a feel for what it is like working with the new EMU-SDMS.

2.2 EMU-SDMS: System architecture and default workflow

As was previously mentioned, the new EMU-SDMS is made up of four main components. The components are the `emuDB` format; the R packages `wrassp` and `emuR`; and the web application, the `EMU-webApp`, which is EMU-SDMS's new GUI component. An overview of the EMU-SDMS's architecture and the components' relationships within the system is shown in Figure 2.1. In Figure 2.1, the `emuR` package plays a central role as it is the only component that interacts with all of the other components of the EMU-SDMS. It performs file and DB handling for the files that comprise an `emuDB` (see Chapter @ref(chap:annot_struct_mod)); it uses the `wrassp` package for signal processing purposes (see Chapter 8; and it can serve `emuDB`s to the `EMU-webApp` (see Chapter 9).

Although the system is made of four main components, the user largely only interacts directly with the `EMU-webApp` and the `emuR` package. A summary of the default workflow illustrating theses interactions can be seen below:

1. Load database into current R session (`load_emuDB()`).

2. Database annotation / visual inspection (`serve()`). This opens up the `EMU-webApp` in the system's default browser.
3. Query database (`query()`). This is optionally followed by `requery_hier()` or `requery_seq()` as necessary (see Chapter 6 for details).
4. Get trackdata (e.g. formant values) for the result of a query (`get_trackdata()`).
5. Prepare data.
6. Visually inspect data.
7. Carry out further analysis and statistical processing.

Initially the user creates a reference to an `emuDB` by loading it into their current R session using the `load_emuDB()` function (see step 1). This database reference can then be used to either serve (`serve()`) the database to the `EMU-webApp` or query (`query()`) the annotations of the `emuDB` (see steps 2 and 3). The result of a query can then be used to either perform one or more so-called requeries or extract signal values that correspond to the result of a `query()` or `requery()` (see step 4). Finally, the signal data can undergo further preparation (e.g., correction of outliers) and visual inspection before further analysis and statistical processing is carried out (see steps 5, 6 and 7). Although the R packages provided by the EMU-SDMS do provide functions for steps 4, 5 and 6, it is worth noting that the plethora of R packages that the R package ecosystem provides can and should be used to perform these duties. The resulting objects of most of the above functions are derived `matrix` or `data.frame` objects which can be used as inputs for hundreds if not thousands of other R functions.

2.3 EMU-SDMS: Is it something for you?

Besides providing a fully integrated system, the EMU-SDMS has several unique features that set it apart from other current, widely used systems (e.g., Boersma and Weenink (2016), Wittenburg et al. (2006), Fromont and Hay (2012), Rose et al. (2006), McAuliffe and Sonderegger (2016)). To our knowledge, the EMU-SDMS is the only system that allows the user to model their annotation structures based on a hybrid model of time-based annotations (such as those offered by Praat's tier-based annotation mechanics) and hierarchical timeless annotations. An example of such a hybrid annotation structure is displayed in Figure 2.2. These hybrid annotations benefit the user in multiple ways, as they reduce data redundancy and explicitly allow relationships to be expressed across annotation levels (see Chapter 4 for further information on hierarchical annotations and Chapter 6 on how to query these annotation structures).

Further, to our knowledge, the EMU-SDMS is the first system that makes use of a web application as its primary GUI for annotating speech. This unique approach enables the GUI component to be used in multiple ways. It can be used as a stand-alone annotation tool, connected to a loaded `emuDB` via `emuR`'s `serve()` function and used to communicate to other servers. This enables it to be used as a collaborative annotation tool. An in-depth explanation of how this component can be used in these three scenarios is given in Chapter 9.

As demonstrated in the default workflow of Section 2.2, an additional unique feature provided by EMU-SDMS is the ability to use the result of a query to extract derived (e.g., formants and RMS values) and complementary signals (e.g., electromagnetic articulography (EMA) data) that match the segments of a query. This, for example, aids the user in answering questions related to derived speech signals such as: *Is the vowel height of the vowel @ (measured by its correlate, the first formant frequency) influenced by whether it appears in a strong or weak syllable?*. Chapter 3 gives a complete walk-through of how to go about answering this question using the tools provided by the EMU-SDMS.

The features provided by the EMU-SDMS make it an all-in-one speech database management solution that is centered around R. It enriches the R platform by providing specialized speech signal processing, speech database management, data extraction and speech annotation capabilities. By achieving this without relying on any external software sources except the web browser, the EMU-SDMS significantly reduces the number of tools the speech and spoken language researcher has to deal with and helps to simplify answering research questions. As the only prerequisite for using the EMU-SDMS is a basic familiarity with the R platform, if

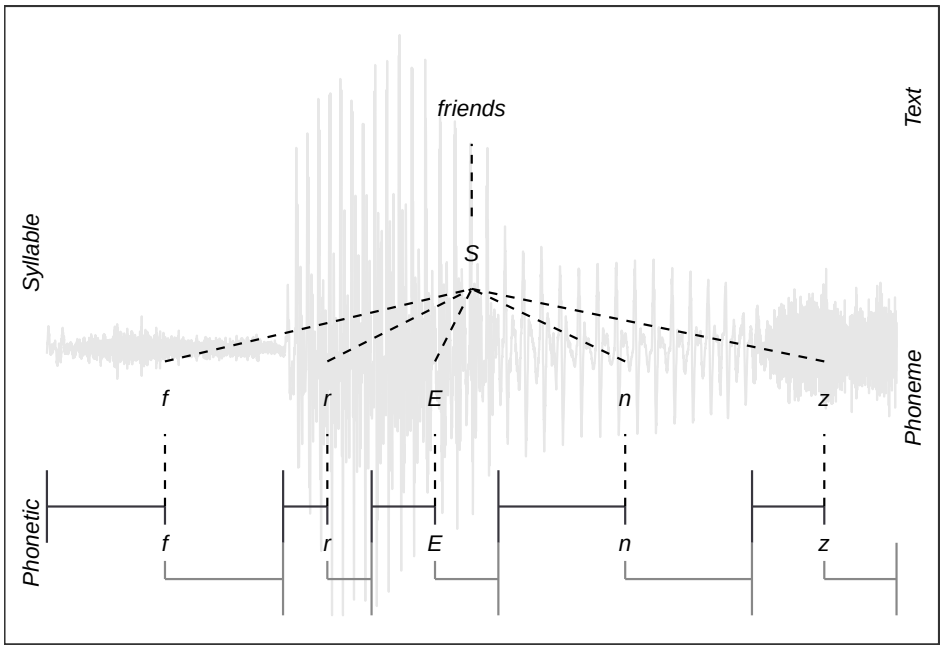
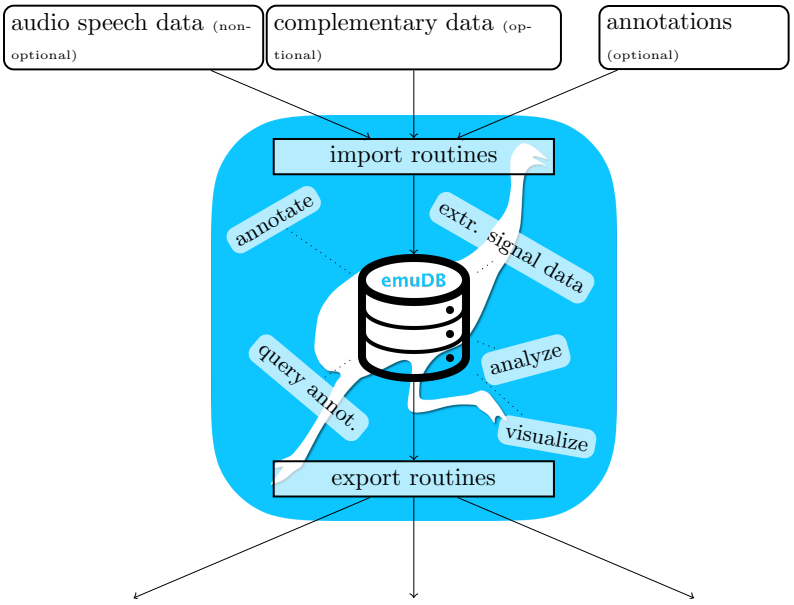


Figure 2.2: Example of a hybrid annotation combining time-based (*Phonetic* level) and hierarchical (*Phoneme*, *Syllable*, *Text* levels including the inter-level links) annotations.

the above features would improve your workflow, the EMU-SDMS is indeed for you.

Chapter 3

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



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 the vowel height of the vowel @ (measured by its correlate, the first formant frequency) influenced by whether it appears in a content or function word?* 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 and Jänsch (2004) 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 5 and importing them into the new EMU system. The current import routines provided by the `emuR` package are:

- `convert_TextGridCollection()` - Convert TextGrid collections (`.wav` and `.TextGrid` files) to the `emuDB` format,

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

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

The `emuR` package comes with a set of example files and small databases that are used throughout the `emuR` documentation, including the functions help pages. These can be accessed by typing `help(functionName)` or the short form `?functionName`. R Example **??** 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 **??**, 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 5).

rexample:tutorial-create-emuRdemoData

```
# 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 et al. (2015)) this is a common collection format in the speech sciences. R Example **??** shows such a file collection that is part of `emuR`'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.

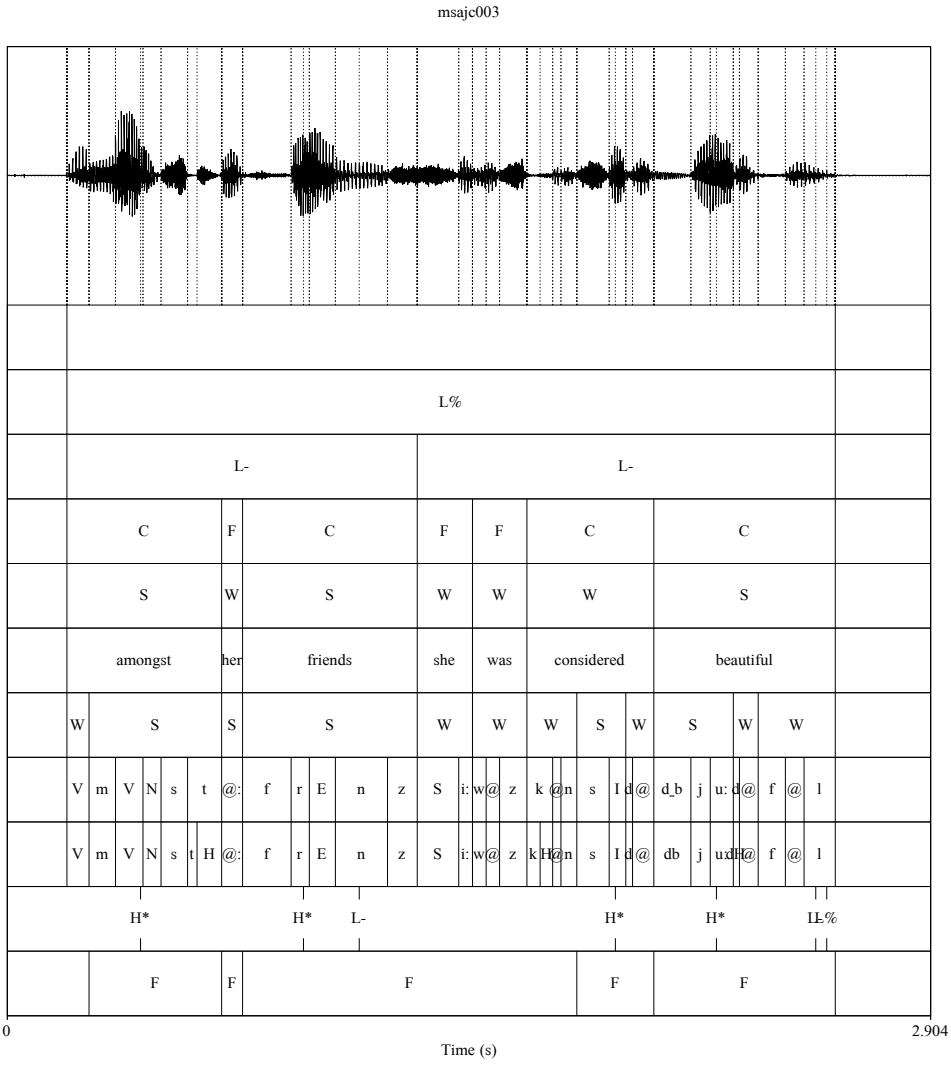
rexample:showTGcolContent

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

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

[2]Future versions of `emuR` may allow `_meta.json` files containing meta information in the form of key-value pairs to be placed in either the `_emuDB`, the `_ses` or the `_bndl` directories.

```
##  [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"
```

\begin{figure}



{

}

\caption{TextGrid annotation of the `emuR_demoData/TextGrid_collection/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) \end{figure}

3.1 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 **??**, 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 *Word* (content: *C* vs. function: *F* word annotations), *Syllable* (strong: *S* vs. weak: *W* syllable annotations), *Phoneme* (phoneme level annotations) and *Phonetic* (phonetic annotations using Speech Assessment Methods Phonetic Alphabet (SAMPA) symbols - Wells et al. (1997)) using the `tierNames` parameter. This conversion can be seen in R Example @ref(rexample:tutorial_tgconv).

rexample:tutorial_tgconv

```
# convert TextGrid collection to the emuDB format
convert_TextGridCollection(dir = tgColDir,
                           dbName = "myFirst",
                           targetDir = tempdir(),
                           tierNames = c("Word", "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 *Word*, *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 5 of this document.

3.2 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 **??** shows how to load the converted TextGrid collection into R using the `load_emuDB()` function.

rexample:tutorial-loadEmuDB

```
# 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)
```

3.2.1 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).

rexample:tutorial_summary

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

```
## Name:      myFirst
## UUID:      e24a42a8-dd2d-4358-8ca7-f9866bf005a9
## Directory:    /private/var/folders/yk/8z9tn7kx6hbcg_9n4c1sld980000gn/T/Rtmpfb765T/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     Word SEGMENT            1        Word;
## 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 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 *Word* 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.

3.2.2 Database annotation and visual inspection

The EMU-SDMS has a unique approach to annotating and visually inspecting databases, 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 `emuR` package provides the `serve()` function which is used in R Example @ref(rexample:tutorial_serve).

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 autom...
```

```
## 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 3.1 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 9) on how to use it, what it is capable of and how to

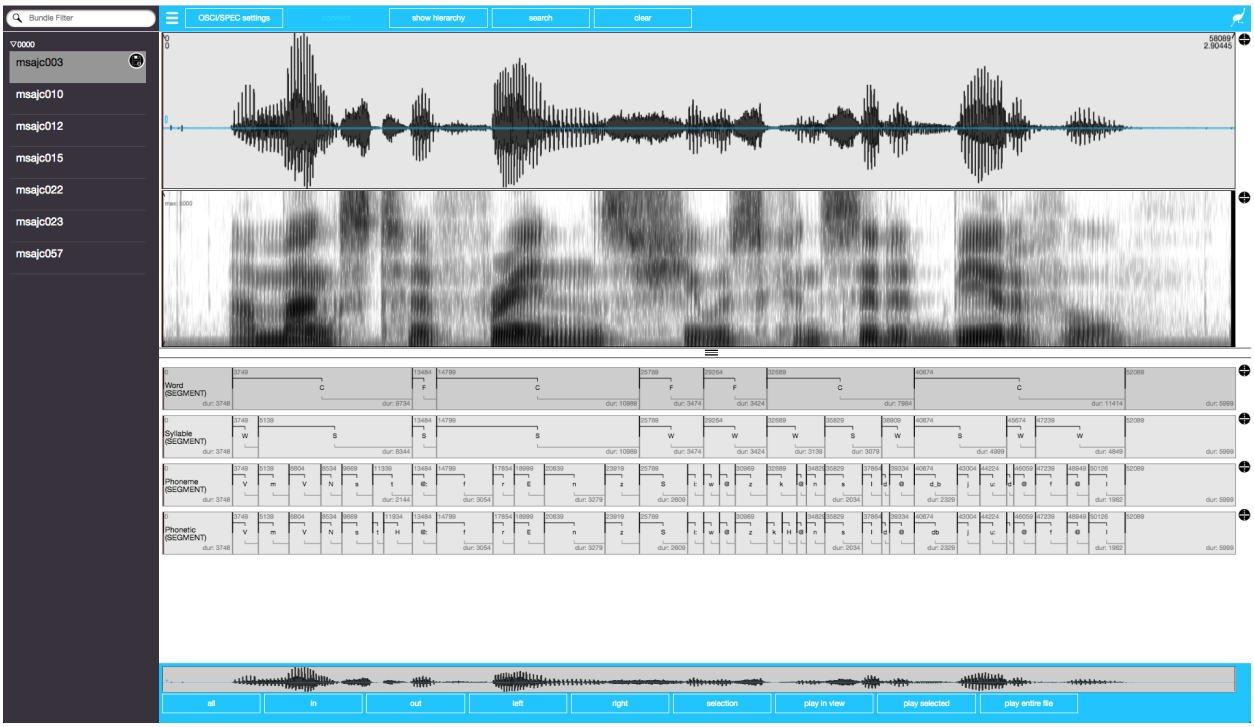


Figure 3.1: Screenshot of 'EMU-webApp' displaying 'msajc003' bundle of *myFirst* 'emuDB'.

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`.

3.3 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 `emuR` 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 6 gives a detailed description of the query mechanics provided by `emuR`, 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 3.1 show that the *myFirst* `emuDB` contains four levels of annotations. R Example **??** 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 6 for a detailed description).

rexample:tutorial-simpleQuery

```
# query all segments containing the label
# "C" (== content word) of the "Word" level
sl_text = query(emuDBhandle = dbHandle,
                query = "Word == C")

# 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
# "f" 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 == n")

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

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

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

```
## segment  list from database:  myFirst
## query was:  Phonetic == n
##   labels    start      end session   bundle    level    type
## 1      n 1031.925 1195.925    0000 msajc003 Phonetic SEGMENT
```

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

```
## segment  list from database:  myFirst
## query was:  Phonetic == n
##  with 12 segments
##
## Segment distribution:
##
##  n
## 12
```

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 **??**. 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.

3.4 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 content word*. For such queries to be possible, the EMU-SDMS offers very sophisticated annotation structure modeling capabilities, which are described in Chapter **??**. For the sake of this tutorial we will focus on converting the flat segment level annotation

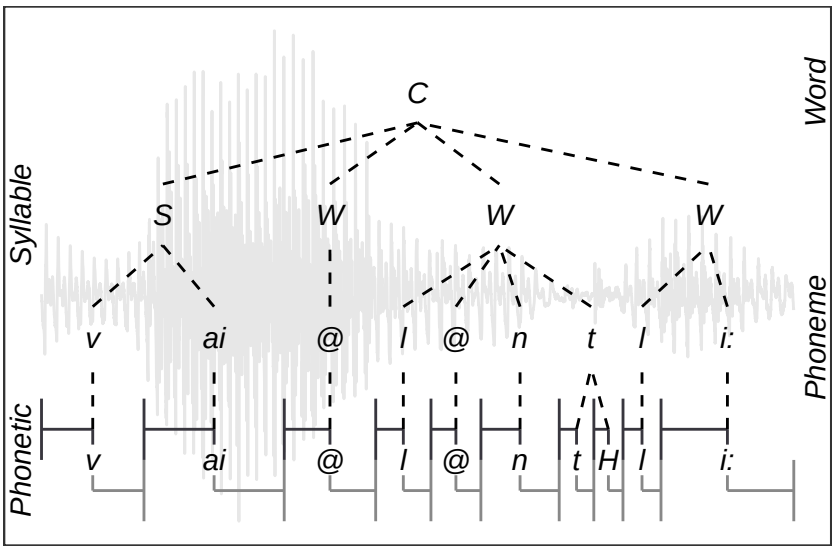


Figure 3.2: Example of a hierarchical annotation of the content (==*C*) word *violently* belonging to the *msajc012* bundle of the *myFirst* demo 'emuDB'.

structure displayed in Figure 3.1 to a hierarchical form as displayed in Figure 3.2, 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 level, we will now use a function provided by the `emuR` package to build these hierarchical structures using this information called `autobuild_linkFromTimes()`. R Example **??** 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 *Word*/*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 **??** for further details). The `autobuild_linkFromTimes()` calls in R Example **??** use the `newLinkDefType` parameter, which if defined automatically adds a link definition to the database.

rexample:tutorial-autobuild

```
# invoke autobuild function
# for "Word" and "Syllable" levels
autobuild_linkFromTimes(dbHandle,
                        superlevelName = "Word",
                        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")
```

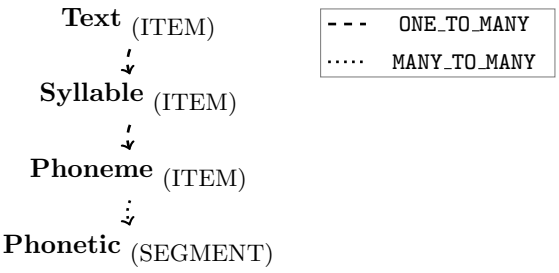


Figure 3.3: Schematic annotation structure of the 'emuDB' after calling the autobuild function in R Example ??.

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

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.

rexample:tutorial-delBackupLevels

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

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

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

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

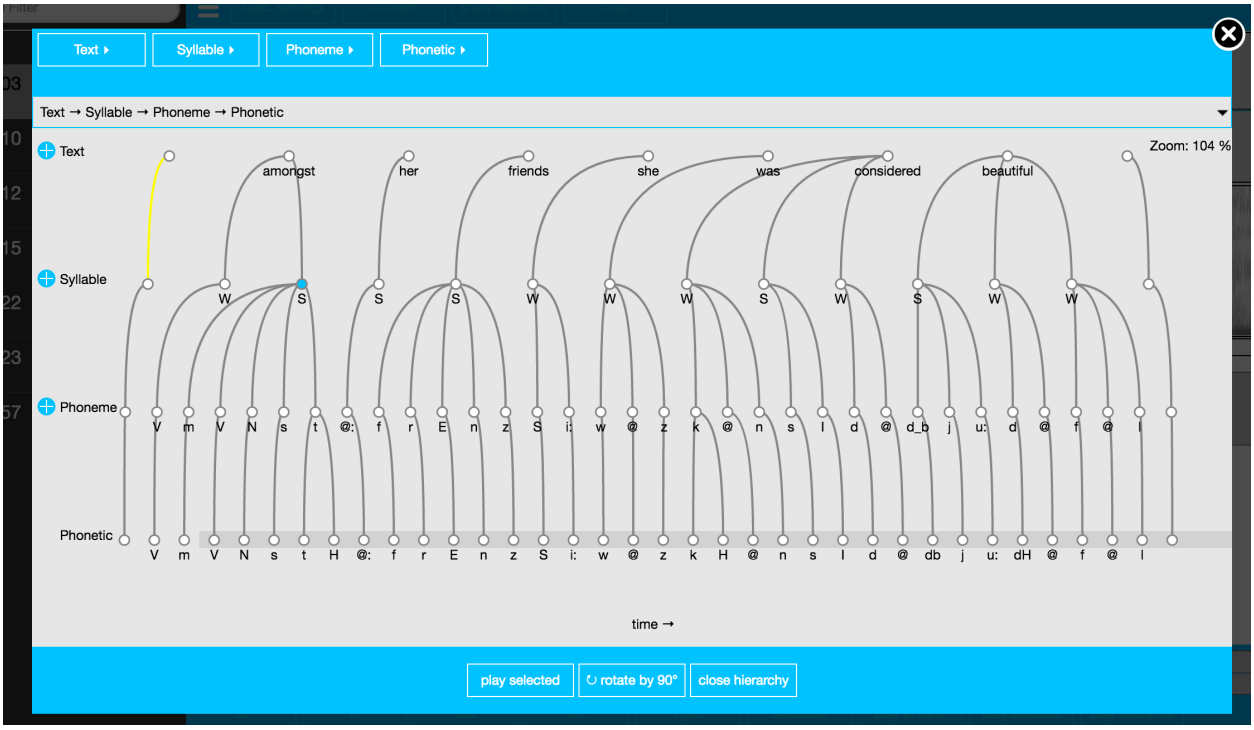


Figure 3.4: Screenshot of 'EMU-webApp' displaying the autobuilt hierarchy of the *myFirst* 'emuDB'.

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

# list level definitions
list_levelDefinitions(dbHandle)
```

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

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

```
##           type superlevelName sublevelName
## 1  ONE_TO_MANY           Word     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 ??, 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 ??.

3.4.1 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 the vowel height of the vowel @ (measured by its correlate, the first formant frequency) influenced by whether it appears in a content or function word?*. R Example **??** shows how all the @ vowels in the *myFirst* database are queried.

rexample:tutorial-labelGroupQuery

```
# query annotation items containing
# the labels @ on the Phonetic level
sl_vowels = query(dbHandle, "Phonetic == @")

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

```
## segment  list from database:  myFirst
## query was:  Phonetic == @
##   labels    start      end session   bundle    level    type
## 1      @ 1506.175 1548.425    0000 msajc003 Phonetic SEGMENT
```

As the type of word (content vs. function) for each @ vowel that was just extracted is also needed, we can use the requery functionality of the EMU-SDMS (see Chapter 6) to retrieve the word 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 **??** illustrates the usage of the hierarchical requery function, `requery_hier()`, to retrieve the appropriate annotation items from the *Word level.

rexample:tutorial-requery

```
# hierarchical requery starting from the items in sl_vowels
# and moving up to the "Word" level
sl_wordType = requery_hier(dbHandle,
                           seglist = sl_vowels,
                           level = "Word",
                           calcTimes = FALSE)

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

```
## segment  list from database:  myFirst
## query was:  FROM REQUERY
##   labels start end session   bundle level type
## 1      F    NA  NA    0000 msajc003  Word ITEM
```

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

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

As can be seen by the `nrow()` comparison in R Example **??**, 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 word type (e.g., vowel `sl_vowels[5,]` belongs to the word type `sl_wordType[5,]`).

3.5 Signal extraction and exploration

Now that the vowel and word 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 `emuR` 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 8 for details). R Example **??** shows the usage of this function.

rexample:tutorial-getTrackdata

```
# get formant values for the vowel segments
td_vowels = get_trackdata(dbHandle,
                          seglist = sl_vowels,
                          onTheFlyFunctionName = "forest",
                          verbose = F)

# show class vector
class(td_vowels)
```

```
## [1] "trackdata"
```

```
# show dimensions
dim(td_vowels)
```

```
## [1] 28  4
```

```
# display all values for fifth segment
td_vowels[5,]
```

```
## trackdata from track: fm
## index:
##  left right
##     1    12
## ftime:
##       start    end
## [1,] 2447.5 2502.5
## data:
##         T1   T2   T3   T4
## 2447.5 303 1031 2266 3366
## 2452.5 289  967 2250 3413
## 2457.5 296  905 2273 3503
## 2462.5 321  885 2357 3506
## 2467.5 316  889 2397 3475
## 2472.5 306  863 2348 3548
## 2477.5 314  832 2339 3611
## 2482.5 325  795 2342 3622
## 2487.5 339  760 2322 3681
## 2492.5 335  746 2316 3665
## 2497.5 341  734 2306 3688
## 2502.5 361  733 2304 3692
```

As can be seen by the call to the `class()` function, the resulting object is of the type `trackdata` and has 28 entries. This corresponds to the number of rows contained in the segment lists extracted above (i.e., `nrow(sl_vowels)`). This indicates that this object contains data for each of the segments that correspond to each of the row entries of the segment lists (i.e., `td_vowels[5,]` are the formant values belonging to `sl_vowels[5,]`). As the columns `T1`, `T2`, `T3`, `T4` of the printed output of `td_vowels[5,]` suggest, the `forest` function estimates four formant values. We will only be concerned with the first (column `T1`) and

second (column T2). R Example ?? shows a call to emuR's dplot() function which produces the plot displayed in Figure 3.5. The first call to the dplot() function plots all 28 first formant trajectories (achieved by indexing the first column i.e., T1: x = td_vowels[, 1]). To clean up the cluttered left plot, the second call to the dplot() function additionally uses the average parameter to plot only the ensemble averages of all @ vowels and time-normalizes the trajectories (normalise = TRUE) to an interval between 0 and 1.

rexample:tutorial-dplot

```
# two plots next to each other
formantNr = 1

par(mfrow = c(1,2))

dplot(x = td_vowels[, formantNr],
      labs = sl_vowels$labels,
      xlab = "Duration (ms)",
      ylab = paste0("F", formantNr, " (Hz)"))

dplot(x = td_vowels[, 1],
      labs = sl_vowels$labels,
      normalise = TRUE,
      average = TRUE,
      xlab = "Normalized time",
      ylab = paste0("F", formantNr, " (Hz)"))

# back to single plot
par(mfrow = c(1,1))
```

Figure 3.5 gives an overview of the first formant trajectories of the @ vowels. 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 ?? makes use of the eplot() function. This produces Figure 3.5. To be able to use the eplot() function, the td_vowels object first has to be modified, 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 extracting temporal mid-point formant values for each vowel using the get_trackdata() function utilizing its cut parameter. R Example ?? shows an alternative approach using the dcut() function to essentially cut the formant trajectories to a specified proportional segment. By using only the left.time = 0.5 (and not specifying right.time) only the formant values that are closest to the temporal mid-point are cut from the trajectories.

rexample:tutorial-eplot

```
# cut formant trajectories at temporal mid-point
td_vowels_midpoint = dcut(td_vowels,
                          left.time = 0.5,
                          prop = TRUE)

# show dimensions of td_vowels_midpoint
dim(td_vowels_midpoint)

# generate plot
eplot(x = td_vowels_midpoint[,1:2],
      labs = sl_vowels$labels,
      dopoints = TRUE,
      formant = TRUE,
```

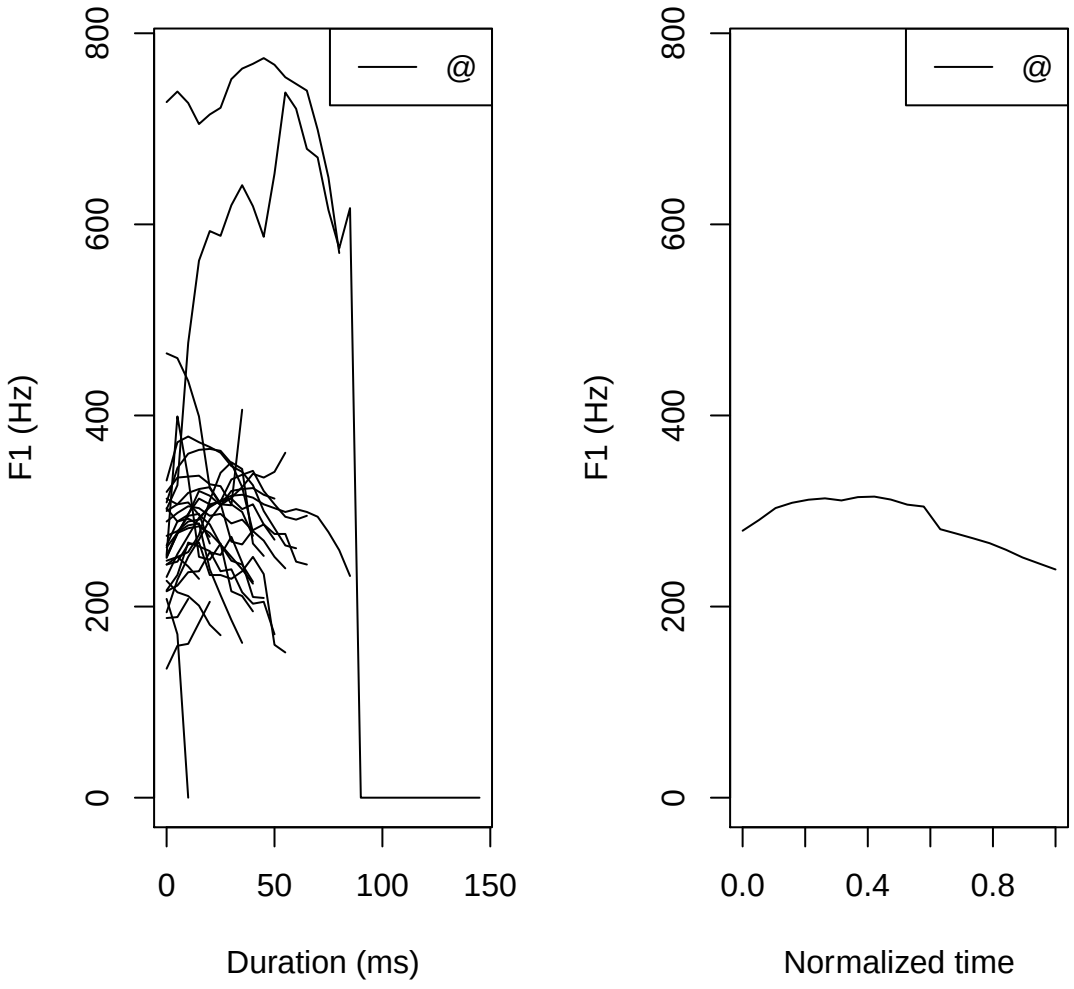
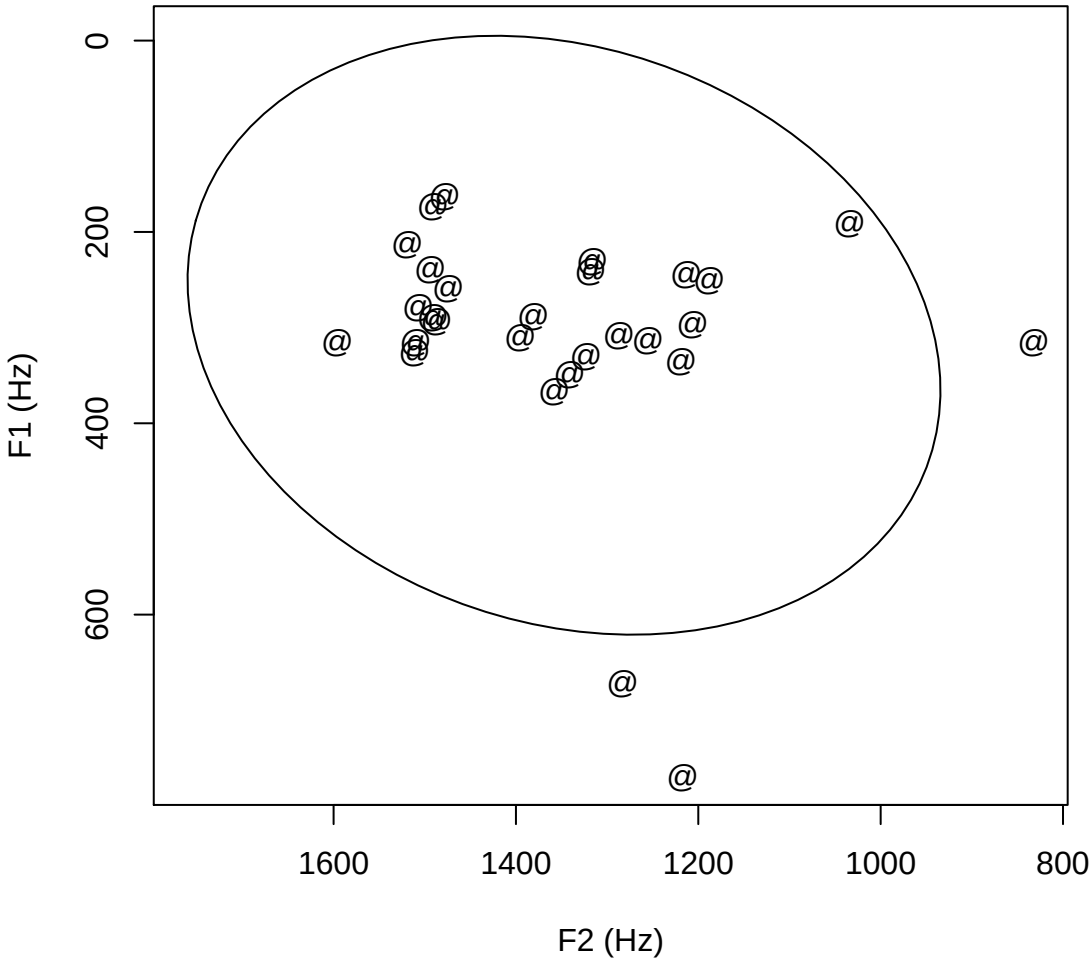


Figure 3.5: 'dplot()' plots of F1 trajectories. The left plot displays all trajectories while the right plot displays the ensemble average of all ** vowels.

```
      xlab="F2 (Hz)",
      ylab = "F1 (Hz)"
      )
```

\begin{figure}



{

}

\caption{95% ellipses for F2 x F1 data extracted from the temporal midpoint of the vowel segments.}
\end{figure}

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 distribution is also shown. Although not necessarily applicable to the question posed at the beginning of this tutorial, the data exploration using the `dplot()` and `eplot()` functions can be very helpful tools for providing an overview of the data at hand.

3.6 Vowel height as a function of word types (content vs. function): evaluation and statistical analysis

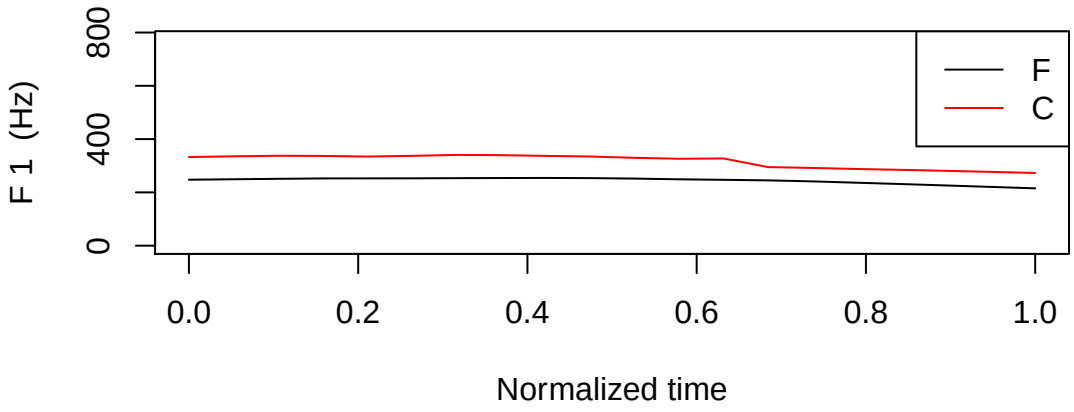
The above data exploration only dealt with the actual @ vowels and disregarded the syllable type they occurred in. However, the question in the introduction of this chapter focuses on whether the @ vowel

occurs in a content (labeled *C*) or function (labeled *F*) word. For data inspection purposes, R Example **??** initially extracts the central 60% (`left.time = 0.2` and `right.time = 0.8`) of the formant trajectories from `td_vowels` using `dcut()` and displays them using `dplot()`. It should be noted that the call to `dplot()` uses the labels of the `sl_wordType` object as opposed to those of `sl_vowels`. This causes the `dplot()` functions to group the trajectories by their word type as opposed to their vowel labels as displayed in Figure @ref(fig:tutorial_dplotSylTyp).

rexample:tutorial-dplotSylTyp

```
# extract central 60% from formant trajectories
td_vowelsMidSec = dcut(td_vowels,
                       left.time = 0.2,
                       right.time = 0.8,
                       prop = TRUE)

# plot first formant trajectories
formantNr = 1
dplot(x = td_vowelsMidSec[, formantNr],
      labs = sl_wordType$labels,
      normalise = TRUE,
      average = TRUE,
      xlab = "Normalized time",
      ylab = paste("F", formantNr, " (Hz)"))
```

\begin{figure}



{

}

\caption{Ensemble averages of F1 contours of all tokens of the central 60% of vowels grouped by word type (function (*F*) vs. content (*W*)).} \end{figure}

As can be seen in Figure 3.6, there seems to be a distinction in F1 trajectory height between vowels in content and function words. R Example **??** shows the code to produce a boxplot using the `ggplot2` package to further visually inspect the data (see Figure 3.6 for the plot produced by R Example @ref{rexample:tutorial-boxplot}).

rexample:tutorial-boxplot

```
formantNr = 1
# use trapply to calculate the means of the 60%
# formant trajectories
td_vowelsMidSec_mean = trapply(td_vowelsMidSec[, formantNr],
                               fun = mean,
                               simplify = T)
```

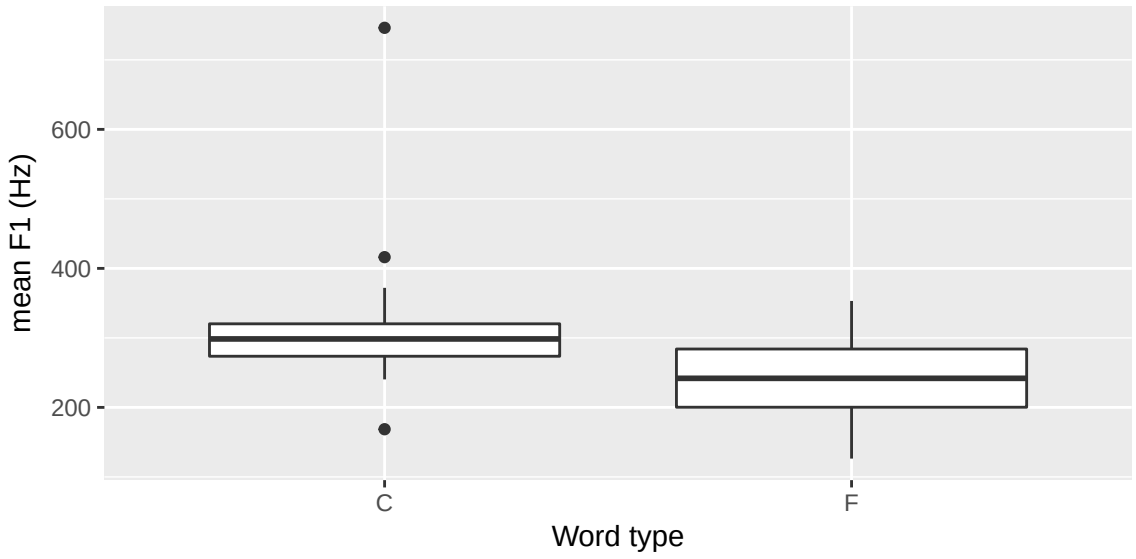


Figure 3.6: Boxplot produced using 'ggplot2' to visualize the difference in F1 depending on whether the vowel occurs in content (*C*) or function (*F*) word.

```
# create new data frame that contains the mean
# values and the corresponding labels
df = data.frame(wordType = sl_wordType$labels,
                meanF1 = td_vowelsMidSec_mean)

# load library
library(ggplot2)

# create boxplot using ggplot
ggplot(df, aes(wordType, meanF1)) +
  geom_boxplot() +
  labs(x = "Word type", y = paste0("mean F", formantNr, " (Hz)"))
```

To confirm or reject this, R Example **??** presents a very simple statistical analysis of the F1 mean values of the 60% mid-section formant trajectories [3]. First, a Shapiro-Wilk test for normality of the distributions of the F1 means for both word types is carried out. As only one type is normally distributed, a Wilcoxon rank sum test is performed. The density distributions (commented out `plot()` function calls in R Example **??**) are displayed in Figure @ref(fig:tutorial_stats1).

rexample:tutorial-stats1

```
# calculate density for vowels in function words
distrF = density(df[df$wordType == "F",]$meanF1)

# uncomment to visualize distribution
# plot(distrF)

# check that vowels in function
# words are normally distributed
shapiro.test(df[df$wordType == "F",]$meanF1)
```

[3] It is worth noting that the sample size in this toy example is quite small. This obviously influences the outcome of the simple statistical analysis that is performed here.

```
##
##  Shapiro-Wilk normality test
##
## data:  df[df$wordType == "F", ]$meanF1
## W = 0.98687, p-value = 0.9887
```

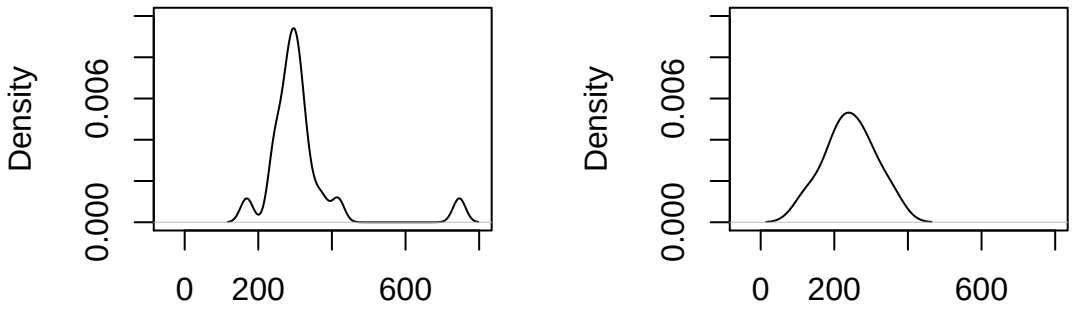
```
# p-value > 0.05 implying that the distribution
# of the data ARE NOT significantly different from
# normal distribution -> we CAN assume normality

# calculate density for vowels in content words
distrC = density(df[df$wordType == "C",]$meanF1)

# uncomment to visualize distribution
# plot(distrC)

# check that vowels in content
# words are normally distributed:
shapiro.test(df[df$wordType == "C",]$meanF1)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  df[df$wordType == "C", ]$meanF1
## W = 0.66506, p-value = 1.506e-05
```

```
# p-value < 0.05 implying that the distribution
# of the data ARE significantly different from
# normal distribution -> we CAN NOT assume normality
# (this somewhat unexpected result is probably
# due to the small sample size used in this toy example)
# -> use Wilcoxon rank sum test

# perform Wilcoxon rank sum test to establish
# whether vowel F1 depends on word type
wilcox.test(meanF1 ~ wordType, data = df)
```

```
##
##  Wilcoxon rank sum test
##
## data:  meanF1 by wordType
## W = 121, p-value = 0.03752
## alternative hypothesis: true location shift is not equal to 0
```

\begin{figure}



{

}

\caption{Plots of density distributions of vowels in content words (left plot) and vowels in function words (right plot) in R Example **?**(rexample:tutorial_stats1).} \end{figure}

As shown by the result of `wilcox.test()` in R Example @ref(rexample:tutorial_stats1), word type (*C* vs. *F*) has a significant influence on the vowel's F1 (W=121, p<0.05). Hence, the answer to the initially proposed question: *Given an annotated speech database, is vowel height of the vowel @ (measured by its correlate, the first formant frequency) influenced by whether it appears in a content or function word?* is yes!

3.7 Conclusion

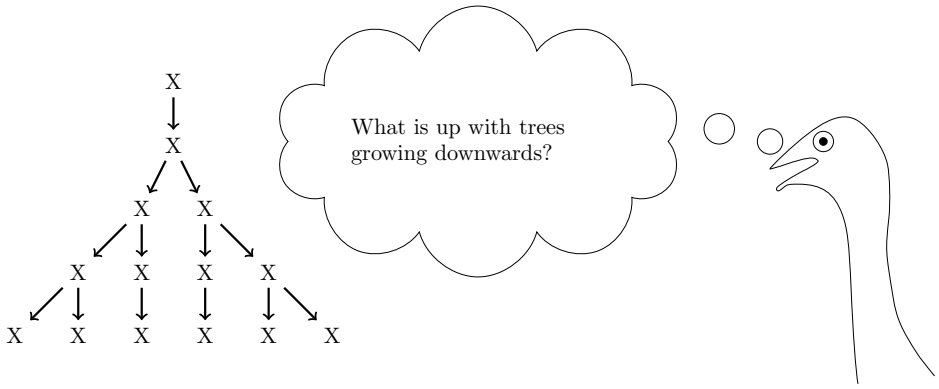
The tutorial given in this chapter gave an overview of what it is like working with the EMU-SDMS to try to solve a research question. As many of the concepts were only briefly explained, it is worth noting that explicit explanations of the various components and integral concepts are given in following 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.

Part II

Main components and concepts

Chapter 4

Annotation Structure Modeling [1]



The EMU-SDMS facilitates annotation structure modeling that surpasses that available in many other commonly used systems. This chapter provides an in-depth explanation of the annotation structure modeling capabilities the EMU-SDMS offers. One of the most common approaches for creating time-aligned annotations has been to differentiate between events that occur at a specific point in time but have no duration and segments that start at a point in time and have a duration. These annotation items are then grouped into time-ordered sets that are often referred to as tiers. As certain research questions benefit from different granularities of annotation, the timeline is often used to relate implicitly items from multiple tiers to each other as shown in Figure 4.1A. While sufficient for single or unrelated tier annotations, we feel this type of representation is not suitable for more complex annotation structures, as it results in unnecessary, redundant data and data sets that are often difficult to analyze. This is because there are no explicit relationships between annotation items, and it is often necessary to introduce error tolerance values to analyze slightly misaligned time values to find relationships iteratively over multiple levels. The main reason for the prevalence of this sub-optimal strategy is largely because the available software tools (e.g., Praat by Boersma and Weenink (2016)) do not permit any other forms of annotations. These widely used annotation tools often only permit the creation and manipulation of segment and event tiers which in turn has forced users to model their annotation structures on these building blocks alone.

Linguists who deal with speech and language on a purely symbolic level tend to be more familiar with a different type of annotation structure modeling. They often model their structures in the form of a vertically oriented, directed acyclic graph that, but for a few exceptions that are needed for things like elision modeling (e.g., the / / elision that may occur between the canonical representation of the word *family* /fæm li/ and its phonetic representation [fæmli]), loosely adheres to the formal definition of a tree in the graph-theoretical sense (Knuth (1968)) as depicted in Figure @ref(fig:annot_structhybridAnnot)B. While this form of modeling explicitly defines relationships between annotation items (represented by

[1] Sections of this chapter where previously published in Winkelmann et al. (2017)

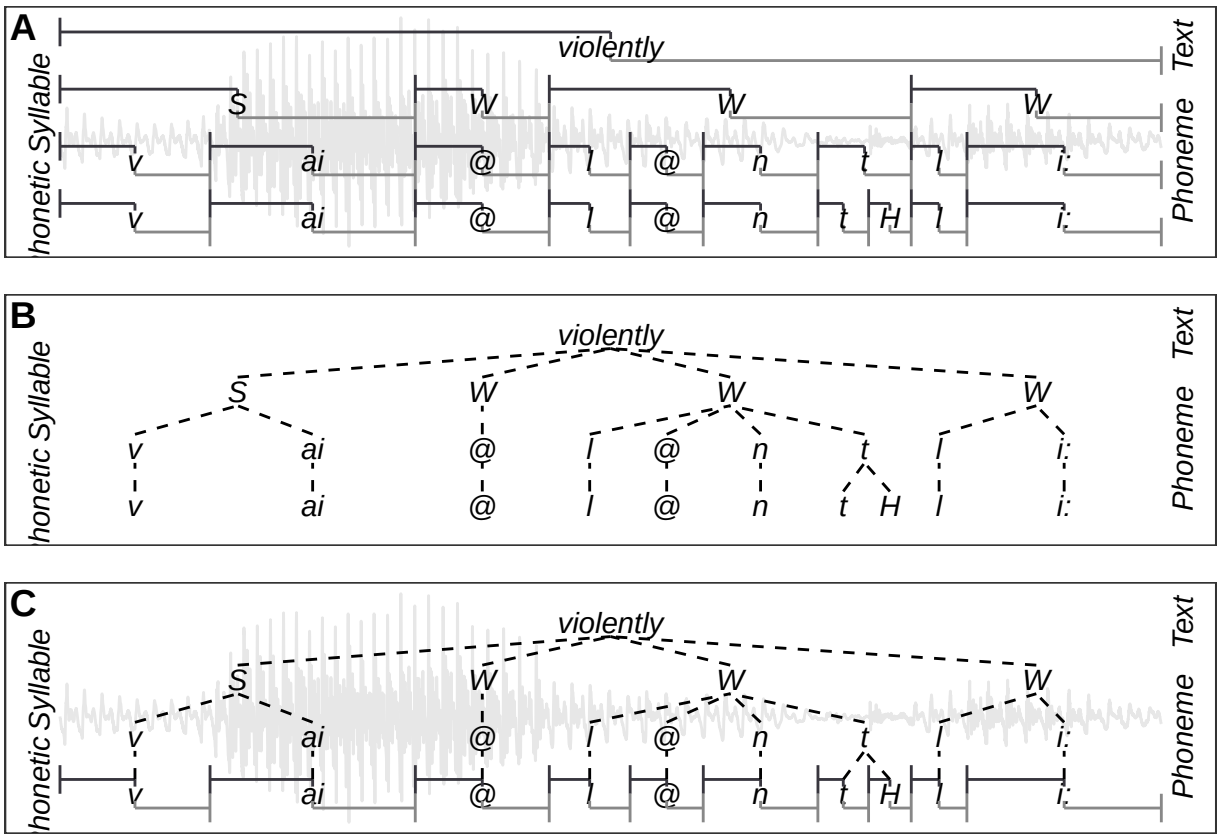


Figure 4.1: **A:** a purely time-aligned annotation; **B:** a purely timeless, symbolic annotation; **C:** a time-aligned hierarchical annotation.

dashed lines in Figure @ref(fig:annot_structhybridAnnot)B), it lacks the ability to map these items to the timeline and therefore the matching speech signal.

To our knowledge, the legacy EMU system (Cassidy and Harrington (2001)) and its predecessors (e.g., Harrington et al. (1993)) were the first to fuse pragmatically purely time-aligned and symbolic tree-like annotations. This was achieved by providing software tools that allowed for these types of annotation structures to be generated, queried and evaluated. In practice, each annotation item had its own unique identifier within the annotation. These unique IDs could then be used to reference each individual item and link them together using dominance relations to form the hierarchical annotation structure. On the one hand, this dominance relation implies the temporal inclusion of the linked sub-level items and was partially predicated on the *no-crossing constraint* as described in Coleman and Local (1991)). This constraint does not permit the crossing of dominance relationships with respect to their sequential ordering (see also Section 4.2 of Cassidy and Harrington (2001)). Since the dominance relations imply temporal inclusion, events can only be children in a parent-child relationship. To allow for timeless annotation items, a further timeless level type was used to complement the segment and event type levels used for time-aligned annotations. Each level of annotation items was stored as an ordered set to ensure the sequential integrity of both the time-aligned and timeless item levels. The legacy system also reduced data redundancy by allowing parallel annotations to be defined in the form of linearly linked levels for any given level (e.g., a segment level bearing SAMPA annotations as well as IPA UTF-8 annotations).

The new EMU-SDMS has adopted some concepts of the legacy system in that levels of type `SEGMENT` and `EVENT` contain annotation items with labels and time information, similar to the tiers known from other software tools such as Praat, while levels of type `ITEM` are timeless and contain annotation items with labels only. `SEGMENT` and `EVENT` levels differ in that units at the `SEGMENT`s level have a start time and a

duration, while units at the EVENT level contain a single time point only. Additionally, every annotation item is able to contain multiple labels and has a unique identifier which is used to link items across levels. These building blocks provide the user with a general purpose annotation modeling tool that allows complex annotation structures to be modeled that best represent the data. An example of a time-aligned hierarchical annotation is depicted in Figure @ref(fig:annot_structhybridAnnot)C, which essentially combines the annotation of Figure @ref(fig:annot_structhybridAnnot)B with the most granular time-bearing level (i.e. the "Phonetic" level) of Figure @ref(fig:annot_structhybridAnnot)A.

In accordance with other approaches (among others see Bird and Liberman (2001), Zipser and Romary (2010), Ide and Romary (2004)), the EMU-SDMS annotation structure can be viewed as a graph that consists of three types of nodes (EVENTs, SEGMENTs, ITEMs) and two types of relations (dominance and sequence) which are directed, transitive and indicate the dominance and sequential relationships between nodes of the graph. As was shown in a pseudo-code example that converted an Annotation Graph (Bird and Liberman (2001)) into the legacy EMU annotation format in Cassidy and Harrington (2001), these formats can be viewed as conceptually equivalent sub- or super-set representations of each other. This has also been shown by developments of meta models with independent data representation such as Salt (Zipser and Romary (2010)), which enable abstract internal representations to be derived that can be exported to equal-set or super-set formats without the loss of information. We therefore believe that the decision as to which data format serializations are used by a given application should be guided by the choice of technology and the target audience or research field. This is consistent with the views of the committee for the Linguistic Annotation Framework (LAF) who explicitly state in the ISO CD 24612 (LAF) document (ISO (2012));

> Although the LAF pivot format may be used in any context, it is assumed that users will represent annotations using their own formats, which can then be transduced to the LAF pivot format for the purposes of exchange, merging and comparison.

The transduction of an EMU annotation into a format such as the LAF pivot format is a simple process, as they share many of the same concepts and are well defined formats.

4.1 Per database annotation structure definition

Unlike other systems, the EMU-SDMS requires the user to define the annotation structure formally for all annotations within a database. Much as Document Type Definition (DTD) or XML Schema Definition (XSD) describe the syntactically valid elements in an Extensible Markup Language (XML) document, the database configuration file of an emuDB defines the valid annotation levels and therefore the type of items that are allowed to be present in a database. Unlike DTDs or XSDs, the configuration file can also define semantic relationships between annotation levels which fall outside the scope of traditional, syntactically oriented schema definitions and validation. This global definition of an annotation structure has numerous benefits for the data integrity of the database, as the EMU-SDMS can perform consistency checks and prevent malformed as well as semantically void annotation structures [2]. Because of these formal definitions, the EMU system generally distinguishes between the actual representations of a structural element which are contained within the database and their formal definitions. An example of an actual representation, that is a subset of the actual annotation, would be a level contained in an annotation file that contains SEGMENTs that annotate a recording. The corresponding formal definition would be a level definition entry in the database's configuration file, which specifies and validates the level's existence within the database.

As mentioned above, the actual annotation files of an emuDB contain the annotation items as well as their hierarchical linking information. To be able to check the validity of a connection between two items, the user specifies which links are permitted for the entire database just as for the level definitions. The permitted hierarchical relationships in an emuDB are expressed through link definitions between level

[2] Although the consistency is ensured by the EMU-SDMS while annotation editing operations are performed, currently no actual consistency checks are performed while loading or saving an annotation. However, we plan to add this functionality in future releases.

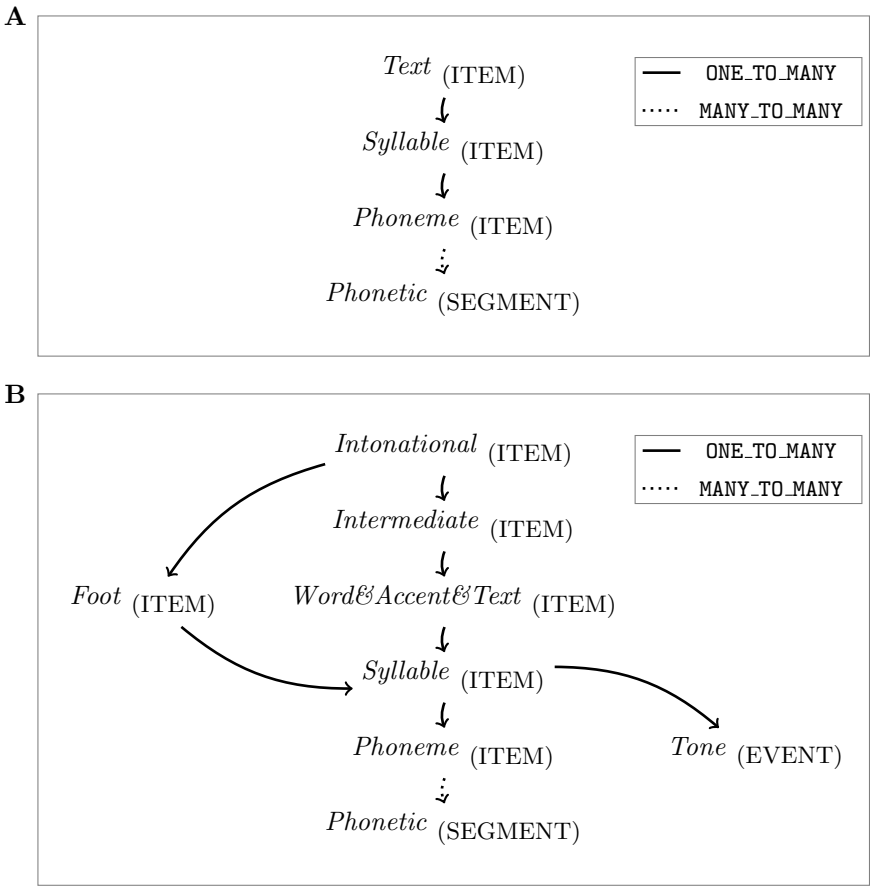


Figure 4.2: **A:** a schematic representation of the hierarchical structure of an 'emuDB' that corresponds to the annotation depicted in 4.1C; **B:** example of a more complex, intersecting hierarchical structure.

definitions as part of the database configuration. There are three types of valid hierarchical relationships between levels: `ONE\_TO\_MANY`, `MANY_TO_MANY` and `ONE_TO_ONE`. These link definitions specify the permitted relationships between instances of annotation items of one level and those of another. The structure of the hierarchy that corresponds to the annotation depicted in Figure @ref(fig:annot_structhybridAnnot)C can be seen in Figure 4.2A. The structure in Figure 4.2A is a typical example of an EMU hierarchy where only the *Phonetic* level of type `SEGMENT` contains time information and the others are timeless as they are of the type `ITEM`. The top three levels, *Text*, *Syllable* and *Phoneme*, have a `ONE_TO_MANY` relationship specifying that a single item in the parent level may have a dominance relationship with multiple items in the child level. In this example, the relationship between *Phoneme* and *Phonetic* is `MANY_TO_MANY`: this type of relationship can be used to represent schwa elision and subsequent sonorant syllabification, as when the final syllable of *sudden* is *d@n* at the *Phoneme* level but *dn* at the *Phonetic* level. Figure 4.2B displays an example of a more complex, intersecting hierarchical structure definition where Abercrombian feet (Abercombie (1967)) are incorporated into the tones and break indices (ToBI) (Beckman and Ayers (1997)) prosodic hierarchy by allowing an intonational phrase to be made up of one or more feet (for further details see Harrington (2010) page 98).

Based on our experience, the explicit definition of the annotation structure for every database which was also integral to the legacy system addresses the excessively expressive nature of annotational modeling systems mentioned in Bird and Liberman (2001). Although, in theory, infinitely large hierarchies can be defined for a database, users of the legacy system typically chose to use only moderately complex annotation structures. The largest hierarchy definitions we have encountered spanned up to fifteen levels while the average amount of levels was between three and five. This self-restriction is largely due to the

primary focus of speech and spoken language domain-specific annotations, as the number of annotation levels between chunks of speech above the word level (intonational phrases/sentences/turns/etc.) and the lower levels (phonetic segmentation/EMA gestural landmark annotation/tone annotation/etc.) is a finite set.

4.2 Parallel labels and multiple attributes

The legacy EMU system made a distinction between linearly and non-linearly linked inter-level links. Linearly linked levels were used to describe, enrich or supplement another level. For example, a level called *Category* might have been included as a separate level from *Word* for marking words' grammatical category memberships (thus each word might be marked as one of adjective, noun, verb, etc.), or information about whether or not a syllable is stressed might be included on a separate *Stress* tier (description taken from Harrington (2010) page 77). Using `ONE_TO_ONE` link definitions to define a relationship between two levels, it is still possible to model linearly linked levels in the new EMU-SDMS. However, an additional, cleaner concept that reduces the extra level overhead has been implemented that allows every annotation item to carry multiple attributes (i.e., labels). Further, using this construct reduces the number of levels, items and links and therefore the hierarchical complexity of an annotation. The generic term "attribute"" (vs. "label") was chosen to have the flexibility of adding attributes that are not of the type `STRING` (i.e., labels) to the annotation modeling capabilities of the EMU-SDMS in future versions. Figure @ref(fig:paraLabels} shows the annotation structure modeling difference between linearly linked levels (see Figure @ref(fig:paraLabels}A) and an annotation structure using multiple attributes (see Figure @ref(fig:paraLabels}B). Figure @ref(fig:paraLabels}A shows three separate levels (*Word*, *Accent* and *Text*) that have a `ONE_TO_ONE` relationship. Each of their annotation items is linked to exactly one annotation item in the child level (e.g., *A1-A3*). Figure @ref(fig:paraLabels}B shows a single level that has three attribute definitions (*Word*, *Accent* and *Text*) and each annotation item contains three attributes (e.g., *A1-A3*).

It is worth noting that every level definition must have an attribute definition which matches its level name. This primary attribute definition must also be present in every annotation item belonging to a level. As `emuR`'s database interaction functions, such as `add_levelDefinition()`, and the `EMU-webApp` automatically perform the necessary actions this should only be of interest to (semi-)advanced users wishing to automatically generate the `_annot.json` format.

4.3 Metadata strategy using single bundle root nodes

As the legacy EMU system and the new EMU-SDMS do not have an explicit method for storing metadata associated with bundles[3], over the years an annotation structure convention has been developed to combat this issue. The convention is to use a generic top level (often simply called `bundle`) that contains a single annotation item in every annotation file. Using the multiple attribute annotation structure modeling capability of the EMU-SDMS, this single annotation item can hold any meta data associated with the bundle. Additionally linking the item to all the annotation items of its child level effectively makes it a parent to every item of the hierarchy. This linking information can later be exploited to query only bundles with matching meta data (see Chapter @ref(chap:querysys} for details). Figure @ref(fig:singleBundleRootNode} displays a hierarchical annotation where the top level (`bundle`) contains information about the speaker's `gender`, the city of birth (`COB`) and `age`.

[3]Future versions of `emuR` may allow `_meta.json` files containing meta information in the form of key-value pairs to be placed in either the `_emuDB`, the `_ses` or the `_bndl` directories.

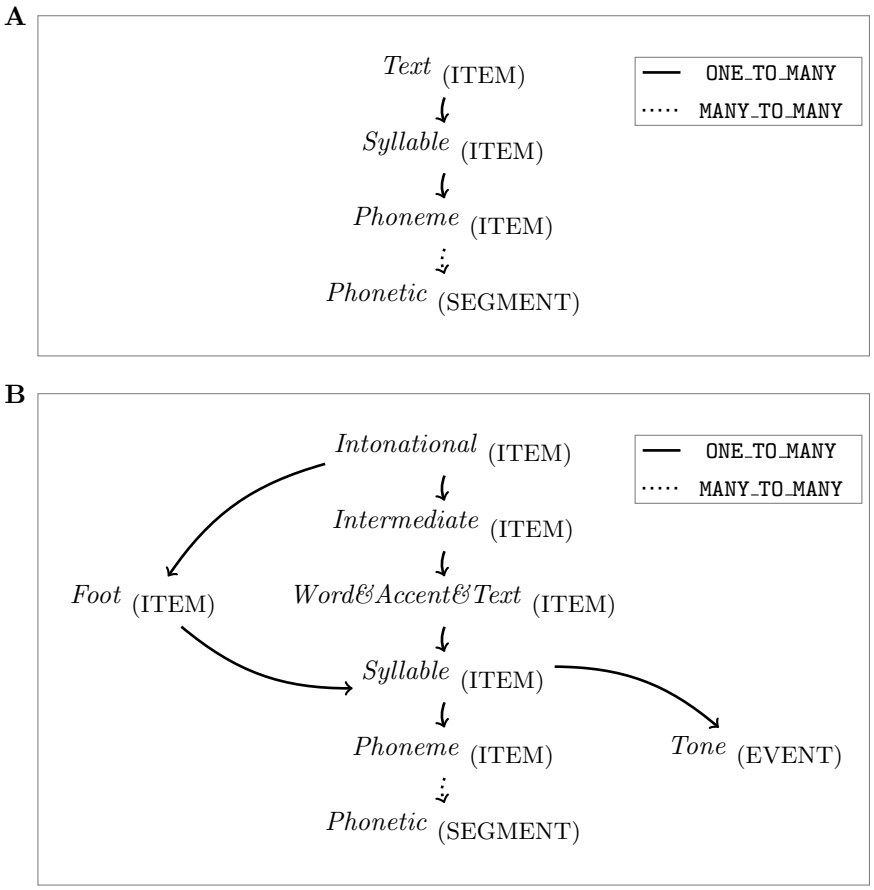


Figure 4.3: Schematic representation of annotation structure modeling difference between **A:** linearly linked levels and **B:** an annotation structure using multiple attributes.

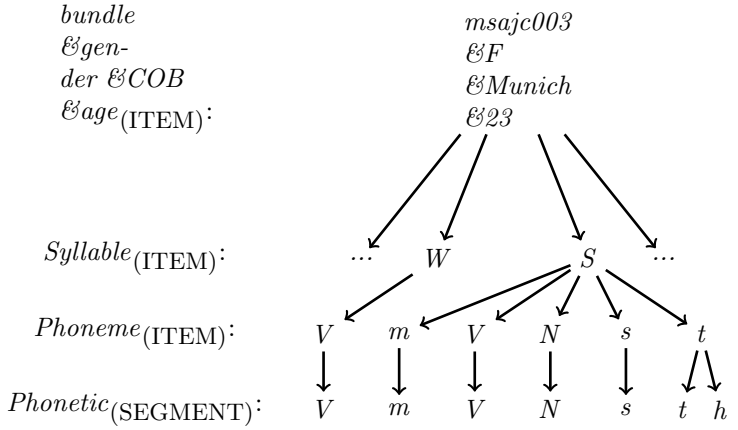


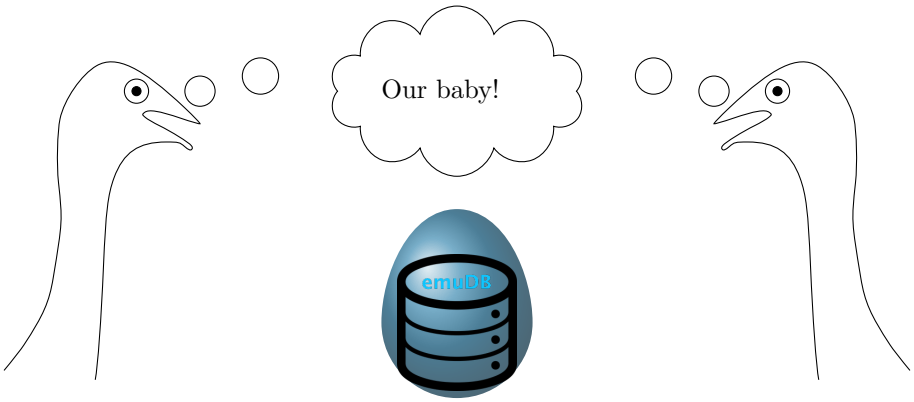
Figure 4.4: Hierarchical annotation displaying single bundle root node metadata strategy where the label of the primary attribute definition (*bundle*) is empty, *gender* encodes the speaker's gender, *COB* encodes the speakers city of birth and *age* encodes the speaker's age in the form of a string.

4.4 Conclusion

The annotation structure modeling capabilities of the EMU-SDMS surpass those of many other commonly used systems. They do so by not only allowing the use of levels containing time information (levels of type `SEGMENT` and `EVENT`) but also timeless levels (levels of type `ITEM`). Additionally, they allow users to define hierarchical annotation structures by allowing explicit links to be implemented from one level's items to those of another. Although it is not obligatory to use them in the EMU-SDMS, we feel the usage of hierarchical annotations allow for complex rich data modeling and are often cleaner representations of the annotations at hand.

Chapter 5

The emuDB Format [1]



This chapter describes the emuDB format, which is the new database format of the EMU-SDMS, and shows how to create and interact with this format. The emuDB format is meant as a simple, general purpose way of storing speech databases that may contain complex, rich, hierarchical annotations as well as derived and complementary speech data. These different components will be described throughout this chapter, and examples will show how to generate and manipulate them. On designing the new EMU system, considerable effort went into designing an appropriate database format. We needed a format that was standardized, well structured, easy to maintain, easy to produce, easy to manipulate and portable.

We decided on the JavaScript Object Notation (JSON) file format[2] as our primary data source for several reasons. It is simple, standardized, widely-used and text-based as well as machine and human readable. In addition, this portable text format allows expert users to (semi-) automatically process and/or generate annotations. Other tools such as the BAS Webservices (Kisler et al., 2012) and SpeechRecorder (Draxler and Jänsch, 2004) have already taken advantage of being able to produce such annotations. Using database back-end options such as relational or graph databases of either the SQL or NoSQL variety as the primary data source for annotations would not directly permit other tools to produce annotations because intermediary exchange file formats would have to be defined to permit this functionality with these back-ends. Our choice of the JSON format was also guided by the decision to incorporate web technologies as part of the EMU-SDMS for which the JSON format is the de facto standard (see Chapter 9). Further, as the default encoding of the JSON format is UTF-8 the EMU-SDMS fully supports the Unicode character set for any user-defined string within an emuDB (e.g. level names and labels)[3].

[1] Sections of this chapter where published in Winkelmann et al. (2017) and some examples taken from the emuDB vignette of the emuR package.

[2] JSON schema files available here https://github.com/IPS-LMU/EMU-webApp/tree/master/dist/schemaFiles

[3] According to the JSON specification (see https://json.org/) the only characters that have to be escaped within a JSON string are: " (as this marks the start/end of a string), \ (as this is the escape character) or control-characters (\ b = backspace, \f = form feed, \n = new line, \r = carriage return, \t = tab). Unicode characters in their hexadecimal form using the \u

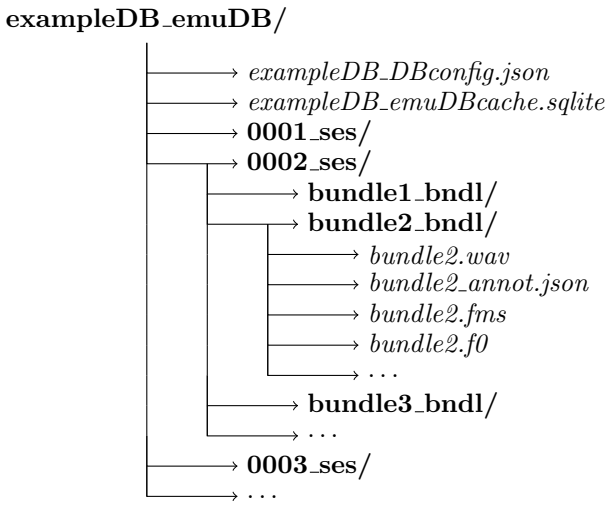


Figure 5.1: Schematic 'emuDB' file and directory structure.

We chose to use the widely adopted Waveform Audio File Format (`WAVE`, or more commonly known as `WAV` due to its filename extension) as our primary media/audio format. Although some components of the EMU-SDMS, notably the `wrassp` package, can handle various other media/audio formats (see `?wrassp::AsspFileFormats` for details) this is the only audio file format currently supported by every component of the EMU-SDMS. Nevertheless, the `wrassp` package can be utilized to convert files from one of it's other supported file formats to the `WAV` format [4]. Future releases of the EMU-SDMS might include the support of other media/audio formats.

In contrast to other systems, including the legacy EMU system, we chose to fully standardize the on-disk structure of speech databases with which the system is capable of working. This provides a standardized and structured way of storing speech databases while providing the necessary amount of freedom and separability to accommodate multiple types of data. Further, this standardization enables fast parsing and simplification of file-based error tracking and simplifies database subset and merging operations as well as database portability. An overview of all database interaction functions is given in Section 10.2.

5.1 Database design

An `emuDB` consists of a set of files and directories that adhere to a certain structure and naming convention (see Figure 5.1). The database root directory must include a single `_DBconfig.json` file that contains the configuration options of the database such as its level definitions, how these levels are linked in the database hierarchy and how the data is to be displayed by the graphical user interface. A detailed description of the `_DBconfig.json` file is given in Appendix **??**. The database root directory also contains arbitrarily named session directories (except for the obligatory `_ses` suffix). These session directories can be used to group the recordings of a database in a logical manner. Sessions can be used, for example, to group all recordings from speaker `AAA` into a session called `AAA_ses`.

Each session directory can contain any number of `_bndl` directories (e.g., `rec1_bndl` `rec2_bndl` … `rec9_bndl`). All files belonging to a recording (i.e., all files describing the same timeline) are stored in the same bundle directory. This includes the actual recording (`.wav`) and can contain optional derived or supplementary signal files in the simple signal file format (SSFF) (Cassidy, 2013) such as formants (`.fms`) or the fundamental frequency (`.f0`), both of which can be calculated using the `wrassp` package (see

followed by for-hex-digits may also be used.

[4] However, if things like resampeling are required we suggest using other tools such as the freely available Sound eXchange (SoX) command line tool (see http://sox.sourceforge.net/) to perform these operation

Chapter 8). Each bundle directory contains the annotation file (`_annot.json`) of that bundle (i.e., the annotations and the hierarchical linking information; see Appendix **??** for a detailed description of the file format). JSON schema files for all the JSON files types used have been developed to ensure the syntactic integrity of the database (see https://github.com/IPS-LMU/EMU-webApp/tree/master/dist/schemaFiles). All files within a bundle that are associated with that bundle must have the same basename as the `_bndl` directory prefix. For example, the signal file in bundle `rec1_bndl` must have the name `rec1.wav` to be recognized as belonging to the bundle. The optional `_emuDBcache.sqlite` file in the root directory (see Figure 5.1 contains the relational cache representation of the annotations of the `emuDB` (see Chapter **??** for further details). All files in an `_bndl` directory that do not follow the above naming conventions will simply be ignored by the database interaction functions of the `emuR` package.

5.2 Creating an emuDB

The two main strategies for creating `emuDB`s are either to convert existing databases or file collections to the new format or to create new databases from scratch where only `.wav` audio files are present. Chapter 3 gave an example of how to create an `emuDB` from an existing TextGrid file collection and other conversion routines are covered in Section 10.1. In this chapter we will focus on creating an `emuDB` from scratch with nothing more than a set of `.wav` audio files present.

5.2.1 Creating an emuDB from scratch

R Example **??** shows how an empty `emuDB` is created in the directory provided by R's `tempdir()` function. As can be seen by the output of the `list.files()` function, `create_emuDB()` creates a directory containing a `_DBconfig.json` file only.

```
# load package
library(emuR, warn.conflicts = F)

# create demo data in directory
# provided by tempdir()
create_emuRdemoData(dir = tempdir())

# create emuDB called "fromScratch"
create_emuDB(name = "fromScratch",
             targetDir = tempdir(),
             verbose = F)

# generate path to the empty fromScratch created above
dbPath = file.path(tempdir(), "fromScratch_emuDB")

# show content of empty fromScratch emuDB
list.files(dbPath)
```

```
## [1] "fromScratch_DBconfig.json"
```

5.2.2 Loading and editing an empty database

The initial step in manipulating and generally interacting with a database is to load the database into the current R session. R Example **??** shows how to load the *fromScratch* database and shows the empty configuration by displaying the output of the `summary()` function.

```
# load database
dbHandle = load_emuDB(dbPath, verbose = F)

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

```
## Name:     fromScratch
## UUID:     cc5ef3b4-79bd-4330-bb0b-b72c2037d5c6
## Directory:    /private/var/folders/yk/8z9tn7kx6hbcg_9n4c1sld980000gn/T/RtmpXeiHpe/fromScratch_emuDB
## Session count: 0
## Bundle count: 0
## Annotation item count:  0
## Label count:  0
## Link count:  0
##
## Database configuration:
##
## SSFF track definitions:
## NULL
##
## Level definitions:
## NULL
##
## Link definitions:
## NULL
```

```
# show class vector of dbHandle
class(dbHandle)
```

```
## [1] "emuDBhandle"
```

As can be seen in R Example **??**, the class of a loaded `emuDB` is `emuDBhandle`. A `emuDBhandle` object is used to reference a loaded `emuDB` in the database interaction functions of the `emuR` package. In this chapter we will show how to use this `emuDBhandle` object to perform database manipulation operations. Most of the `emuDB` manipulation functions follow the following function prefix naming convention:

- `add_XXX` add a new instance of `XXX` / `set_XXX` set the current instance of `XXX`,
- `list_XXX` list the current instances of `XXX` / `get_XXX` get the current instance of `XXX`,
- `remove_XXX` remove existing instances of `XXX`.

5.2.3 Level definitions

Unlike other systems, the EMU-SDMS requires the user to formally define the annotation structure for the entire database. An essential structural element of any `emuDB` are its levels. A level is a more general term for what is often referred to as a tier. It is more general in the sense that people usually expect tiers to contain time information. Levels can either contain time information if they are of the type `EVENT` or of the type `SEGMENT` but are timeless if they are of the type `ITEM` (see Chapter **??** for further details). It is also worth noting that an `emuDB` distinguishes between the definition of an annotation structure element and the actual annotations. The definition of an annotation structure element such as a level definition is merely an entry in the `_DBconfig.json` file which specifies that this level is allowed to be present in the `_annot.json` files. The levels that are present in an `_annot.json` file, on the other hand, have to adhere to the definitions in the `_DBconfig.json`.

As the *fromScratch* database (already loaded) does not contain any annotation structural element definitions, R Example **??** shows how a new level definition called *Phonetic* of type `SEGMENT` is added to the

emuDB.

```
# show no level definitions
# are present
list_levelDefinitions(dbHandle)
```

```
## NULL
```

```
# add level defintion
add_levelDefinition(dbHandle,
                    name = "Phonetic",
                    type = "SEGMENT")

# show newly added level definition
list_levelDefinitions(dbHandle)
```

```
##       name    type nrOfAttrDefs attrDefNames
## 1 Phonetic SEGMENT            1    Phonetic;
```

R Example @ref(rexample:emuDB-addLevelDefWord} shows how a further level definition is added that will contain the orthographic word transcriptions for the words uttered in our recordings. This level will be of the type `ITEM`, meaning that elements contained within the level are sequentially ordered but do not contain any time information.

```
# add level definition
add_levelDefinition(dbHandle,
                    name = "Word",
                    type = "ITEM")

# list newly added level definition
list_levelDefinitions(dbHandle)
```

```
##       name    type nrOfAttrDefs attrDefNames
## 1 Phonetic SEGMENT            1    Phonetic;
## 2     Word    ITEM            1        Word;
```

The function `remove_levelDefinition()` can also be used to remove unwanted level definitions. However, as we wish to further use the levels *Phonetic* and *Word*, we will not make use of this function here.

5.2.3.1 Attribute definitions

Each level definition can contain multiple attributes, the most common, and currently only supported attribute being a label (of type `STRING`). Thus it is possible to have multiple parallel labels (i.e., attribute definitions) in a single level. This means that a single annotation item instance can contain multiple labels while sharing other properties such as the start and duration information. This can be useful when modeling certain types of data. An example of this would be the *Phonetic* level created above. It is often the case that databases contain both the phonetic transcript using IPA UTF-8 symbols as well as a transcript using Speech Assessment Methods Phonetic Alphabet (SAMPA) symbols. To avoid redundant time information, both of these annotations can be stored on the same `Phonetic` level using multiple attribute definitions (i.e., parallel labels). R Example **??** shows the current attribute definitions of the `Phonetic` level.

```
# list attribute definitions of 'Phonetic' level
list_attributeDefinitions(dbHandle,
                          levelName = "Phonetic")
```

```
##       name    level   type hasLabelGroups hasLegalLabels
## 1 Phonetic Phonetic STRING          FALSE          FALSE
```

Even though no attribute definition has been added to the `Phonetic` level, it already contains an attribute definition that has the same name as its level. This attribute definition represents the obligatory primary attribute of that level. As every level must contain an attribute definition that has the same name as its level, it is automatically added by the `add_levelDefinition()` function. To follow the above example, R Example **??** adds a further attribute definition to the `Phonetic` level that contains the SAMPA versions of our annotations.

```
# add
add_attributeDefinition(dbHandle,
                        levelName = "Phonetic",
                        name = "SAMPA")
```

```
## NULL
```

```
# list attribute definitions of 'Phonetic' level
list_attributeDefinitions(dbHandle,
                          levelName = "Phonetic")
```

```
##       name    level   type hasLabelGroups hasLegalLabels
## 1 Phonetic Phonetic STRING          FALSE          FALSE
## 2    SAMPA Phonetic STRING          FALSE          FALSE
```

5.2.3.2 Legal labels

As can be inferred from the columns `hasLabelGroups` and `hasLegalLabels` of the output of the above `list_attributeDefinitions()` function, attribute definitions can also contain two further optional fields. The `legalLabels` field contains an array of strings that specifies the labels that are legal (i.e., allowed or valid) for the given attribute definition. As the `EMU-webApp` does not allow the annotator to enter any labels that are not specified in this array, this is a simple way of assuring that a level has a consistent label set. R Example **??** shows how the `set_legalLabels` and `get_legalLabels` functions can be used to specify a legal label set for the primary `Word` attribute definition of the `Word` level.

```
# define allowed word labels
wordLabels = c("amongst", "any", "are",
               "always", "and", "attracts")

# show empty legal labels
# for "Word" attribute definition
get_legalLabels(dbHandle,
                levelName = "Word",
                attributeDefinitionName = "Word")
```

```
## [1] NA
```

```
# set legal labels values
# for "Word" attribute definition
set_legalLabels(dbHandle,
                levelName = "Word",
                attributeDefinitionName = "Word",
                legalLabels = wordLabels)

# show recently added legal labels
# for "Word" attribute definition
get_legalLabels(dbHandle,
                levelName = "Word",
                attributeDefinitionName = "Word")
```

```
## [1] "amongst"  "any"      "are"      "always"   "and"      "attracts"
```

5.2.3.3 Label groups

A further optional field is the `labelGroups` field. It contains specifications of groups of labels that can be referenced by a name given to the group while querying the `emuDB`. R Example **??** shows how the `add_attrDefLabelGroup()` function is used to add two label groups to the `Phonetic` attribute definition. One of the groups is used to reference a subset of *longVowels* and the other to reference a subset of *shortVowels* on the `Phonetic` level.

```
# add long vowels label group
add_attrDefLabelGroup(dbHandle,
                      levelName = "Phonetic",
                      attributeDefinitionName = "Phonetic",
                      labelGroupName = "longVowels",
                      labelGroupValues = c("i:", "u:"))

# add short vowels label group
add_attrDefLabelGroup(dbHandle,
                      levelName = "Phonetic",
                      attributeDefinitionName = "Phonetic",
                      labelGroupName = "shortVowels",
                      labelGroupValues = c("i", "u", "@"))


# list current label groups
list_attrDefLabelGroups(dbHandle,
                        levelName = "Phonetic",
                        attributeDefinitionName = "Phonetic")
```

```
##          name  values
## 1  longVowels  i:; u:
## 2 shortVowels i; u; @
```

```
# query all short vowels
# Note the result of this query
# is empty as no annotations are present
# in the 'fromScratch' emuDB
query(dbHandle, "Phonetic == shortVowels")
```

```
## segment  list from database:  fromScratch
## query was:  Phonetic == shortVowels
## [1] labels  start   end     session bundle  level   type
## <0 rows> (or 0-length row.names)
```

For users who are familiar with or transitioning from the legacy EMU system, it is worth noting that the label groups correspond to the unfavorably named `Legal Labels` entries of the GTemplate Editor (i.e., legal entries in the `.tpl` file) of the legacy system. In the new system the `legalLabels` entries specify the legal or allowed label values of attribute definitions while the `labelGroups` specify groups of labels that can be referenced by the names given to the groups while performing queries.

A new feature of the EMU-SDMS is the possibility of defining label groups for the entire `emuDB` as opposed to a single attribute definition (see `?add_labelGroups` for further details). This avoids the redundant definition of label groups that should span multiple attribute definitions (e.g., a *longVowels* subset that is to be queried on a level called *Phonetic_1* as well as a level called *Phonetic_2*).

5.2.4 Link definitions

An essential and very powerful conceptual and structural element of any emuDB is its hierarchy. Using hierarchical structures is highly recommended but not a must. Hierarchical annotations allow for complex, rich data modeling and are often cleaner representations of the annotations at hand. As Chapter ?? contains in-depth explanations of the annotation modeling capabilities of the EMU-SDMS and Chapter 6 shows how these structures can be queried using emuR's query mechanics, this chapter will omit an explanation of hierarchical annotation structures. R Example ?? shows how a ONE_TO_MANY relationship between the Word and Phonetic in the form of a link definition is added to an emuDB.

```
# show that currently no link definitions
# are present
list_linkDefinitions(dbHandle)
```

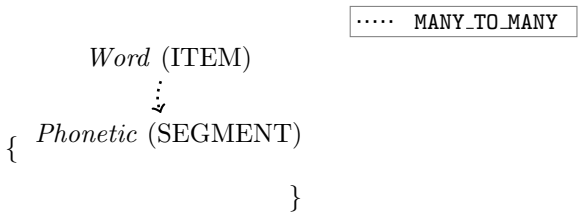
```
## NULL
```

```
# add new "ONE_TO_MANY" link definition
# between "Word" and "Phonetic" levels
add_linkDefinition(dbHandle,
                   type = "ONE_TO_MANY",
                   superlevelName = "Word",
                   sublevelName = "Phonetic")

# show newly added link definition
list_linkDefinitions(dbHandle)
```

```
##          type superlevelName sublevelName
## 1 ONE_TO_MANY           Word     Phonetic
```

A schematic of the simple hierarchical structure of the *fromScratch* created by R Example ?? is displayed in Figure 5.2.4.

\begin{figure}



\caption{A schematic representation of the simple hierarchical structure of the *fromScratch* created by the add_linkDefinition() function call in R Example ?(rexample:emuDB-addLinkDef).} \end{figure}

5.2.5 File handling

The previous sections of this chapter defined the simple structure of the *fromScratch* emuDB. An essential element that is still missing from the emuDB is the actual audio speech data[5]. R Example ?? shows how the import_mediaFiles() function can be used to import audio files, referred to as media files in the context of an emuDB, into the *fromScratch* emuDB.

```
# get the path to directory containing .wav files
wavDir = file.path(tempdir(), "emuR_demoData", "txt_collection")

# Import media files into emuDB session called fromWavFiles.
```

[5] As the EMU-webApp currently only supports mono 16 Bit .wav audio files, we currently recommend using this format only.

```
# Note that the txt_collection directory also contains .txt files.
# These are simply ignored by the import_mediaFiles() function.
import_mediaFiles(dbHandle,
                  dir = wavDir,
                  targetSessionName = "fromWavFiles",
                  verbose = F)

# list session
list_sessions(dbHandle)
```

```
##           name
## 1 fromWavFiles
```

```
# list bundles
list_bundles(dbHandle)
```

```
##        session     name
## 1 fromWavFiles msajc003
## 2 fromWavFiles msajc010
## 3 fromWavFiles msajc012
## 4 fromWavFiles msajc015
## 5 fromWavFiles msajc022
## 6 fromWavFiles msajc023
## 7 fromWavFiles msajc057
```

```
# show first two files in the emuDB
library(tibble) # convert to tibble only to prettify output
as_tibble(head(list_files(dbHandle), n = 2))
```

```
## # A tibble: 2 x 4
##   session      bundle   file                absolute_file_path
## * <chr>        <chr>    <chr>               <chr>
## 1 fromWavFiles msajc003 msajc003_annot.json /private/var/folders/yk/8z9tn~
## 2 fromWavFiles msajc003 msajc003.wav        /private/var/folders/yk/8z9tn~
```

The `import_mediaFiles()` call in R Example ?? added a new session called `fromWavFiles` to the *fromScratch* `emuDB` containing a new bundle for each of the imported media files. The annotations of every bundle, despite containing empty levels, adhere to the structure specified above. This means that every `_annot.json` file created contains an empty `Word` and `Phonetic` level array and the links array is also empty.

The `emuR` package also provides a mechanism for adding files to preexisting bundle directories, as this can be quite tedious to perform manually due to the nested directory structure of an `emuDB`. R Example ?? shows how preexisting `.zcr` files that are produced by `wrassp`'s `zcrana()` function can be added to the preexisting session and bundle structure. As the directory referenced by `wavDir` does not contain any `.zcr` files, R Example ?? first creates them and then adds them to the `emuDB` (see Chapter 8 for further details).

```
# load wrassp package
library(wrassp)

# list all wav files in wavDir
wavFilePaths = list.files(wavDir,
                          pattern = ".*.wav",
                          full.names = TRUE)

# calculate zero-crossing-rate files
# using zcrana function of wrassp package
```

```
zcrana(listOfFiles = wavFilePaths,
       verbose = FALSE)
```

```
## [1] 7
```

```
# add zcr files to emuDB
add_files(dbHandle,
          dir = wavDir,
          fileExtension = "zcr",
          targetSessionName = "fromWavFiles")

# show first three files in emuDB (convert to tibble only
# to prettify output)
as_tibble(head(list_files(dbHandle), n = 3))
```

```
## # A tibble: 3 x 4
##   session      bundle   file                absolute_file_path
## * <chr>        <chr>    <chr>               <chr>
## 1 fromWavFiles msajc003 msajc003_annot.json /private/var/folders/yk/8z9tn~
## 2 fromWavFiles msajc003 msajc003.wav        /private/var/folders/yk/8z9tn~
## 3 fromWavFiles msajc003 msajc003.zcr        /private/var/folders/yk/8z9tn~
```

5.2.6 SSFF track definitions

A further important structural element of any `emuDB` is use of the so-called SSFF tracks, which are often simply referred to as tracks. These SSFF tracks reference data that is stored in the SSFF (see Appendix **??** for a detailed description of the file format) within the `_bndl` directories. The two main types of data are:

- complementary data that was acquired during the recording such as by EMA or EPG; or
- derived data, that is data that was calculated from the original audio signal such as formant values and their bandwidths or the short-term Root Mean Square amplitude of the signal.

As Section **??** covers how the SSFF file output of a `wrassp` function can be added to an `emuDB`, an explanation will be omitted here. R Example **??** shows how the `.zcr` files added in R Example **??** can be added as an SSFF track definition (see Chapter 8 for further details).

```
# show that no SSFF track definitions
# are present
list_ssffTrackDefinitions(dbHandle)
```

```
## NULL
```

```
# add SSFF track definition to emuDB
add_ssffTrackDefinition(dbHandle,
                        name = "zeroCrossing",
                        columnName = "zcr",
                        fileExtension = "zcr")

# show newly added SSFF track definition
list_ssffTrackDefinitions(dbHandle)
```

```
##          name columnName fileExtension
## 1 zeroCrossing        zcr           zcr
```

5.2.7 Configuring the EMU-webApp and annotating the emuDB

As previously mentioned, the current *fromScratch* emuDB contains only empty levels. In order to start annotating the database, the EMU-webApp has to be configured to display the desired information. Although the configuration of the EMU-webApp is stored in the _DBconfig.json file and is therefore a part of the emuDB format, here we will omit an explanation of the extensive possibilities of configuring the web application (see Chapter 9 for an in-depth explanation). R Example ?? shows how the Phonetic level is added to the level canvases order array of the *default* perspective.

```
# show empty level canvases order
get_levelCanvasesOrder(dbHandle, perspectiveName = "default")
```

```
## NULL
```

```
# set level canvases order to display "Phonetic" level
set_levelCanvasesOrder(dbHandle,
                       perspectiveName = "default",
                       order = c("Phonetic"))

# show newly added level canvases order
get_levelCanvasesOrder(dbHandle, perspectiveName = "default")
```

```
## [1] "Phonetic"
```

As a final step before beginning the annotation process, the *fromScratch* emuDB has to be served to the EMU-webApp for annotation and visualization purposes. R Example ?? shows how this can be achieved using the serve() function.

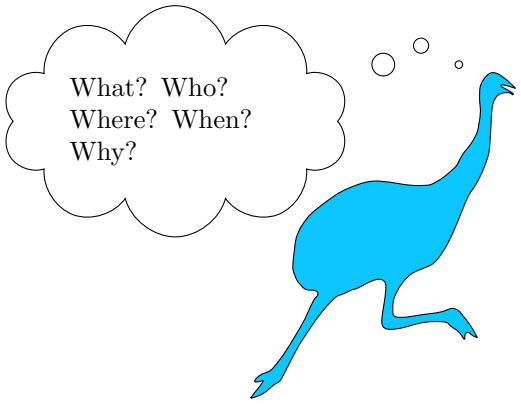
```
# serve "fromScratch" emuDB to the EMU-webApp
serve(dbHandle)
```

5.3 Conclusion

This chapter introduced the elements that comprise the new emuDB format and provided a practical overview of the essential database interaction functions provided by the emuR package. We feel the emuDB format provides a general purpose, flexible approach to storing speech databases with the added benefit of being able to directly manipulate and analyse these databases using the tools provided by the EMU-SDMS.

Chapter 6

The query system



This chapter describes the newly implemented query system of the `emuR` package. When developing the new `emuR` package it was essential that it had a query mechanism allowing users to query a database's annotations in a simple manner. The EMU query language (EQL) of the EMU-SDMS arose out of years of developing and improving upon the query language of the legacy system (e.g., Cassidy and Harrington (2001), Harrington (2010), John (2012)). As a result, today we have an expressive, powerful, yet simple to learn and domain-specific query language. The EQL defines a user interface by allowing the user to formulate a formal language expression in the form of a query string. The evaluation of a query string results in a set of annotation items or, alternatively, a sequence of items of a single annotation level in the `emuDB` from which time information, if applicable (see Section **??**), has been deduced from the time-bearing sub-level. An example of this is a simple query that extracts all strong syllables (i.e., syllable annotation items containing the label *S* on the *Syllable* level) from a set of hierarchical annotations (see Figure 6.1 for an example of a hierarchical annotation). The respective EQL query string `"Syllable == S"` results in a set of segments containing the annotation label *S*. Due to the temporal inclusion constraint of the domination relationship, the start and end times of the queried segments are derived from the respective items of the *Phonetic* level (i.e., the *m* and *H* nodes in Figure 6.1, as this is the time-bearing sub-level. The EQL described here allows users to query the complex hierarchical annotation structures in their entirety as they are described in Chapter **??**.

R Example **??** shows how to create the demo data that is provided by the `emuR` package followed by loading an example `emuDB` called *ae* into the current R session. This database will be used in all the examples throughout this chapter.

```
# load package
library(emuR)
```

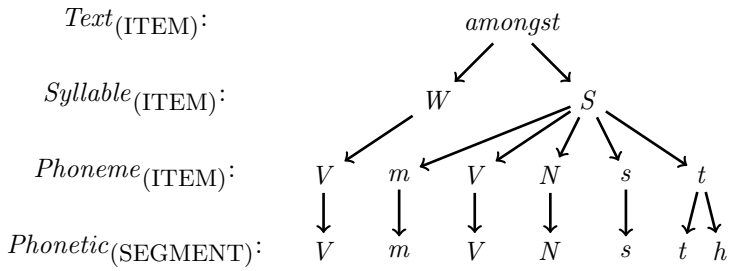


Figure 6.1: Simple partial hierarchy of an annotation of the word *amongst* in the *msajc003* bundle in the *ae* demo 'emuDB'.

```
# create demo data in directory
# provided by tempdir()
create_emuRdemoData(dir = tempdir())

# create path to demo database
path2ae = file.path(tempdir(), "emuR_demoData", "ae_emuDB")

# load database
ae = load_emuDB(path2ae, verbose = F)
```

6.1 emuRsegs: The resulting object of a query

In emuR the result of a query or requery (see Section 6.2.7) is a pre-specified object which is a superclass of the common data.frame. R Example ?? shows the result of a slightly expanded version of the above query ("Syllable == S"), which additionally uses the dominates operator (i.e., the ^ operator; for further information see Section @ref(subsubsec:query_dominationQueries)) to reduce the queried annotations to the partial hierarchy depicted in Figure @ref(fig:amongstHier} in the *ae* demo emuDB. In this example, the classes of the resulting object including its printed output are displayed. The class vector of a resulting emuRsegs object also contains the legacy EMU system's emusegs class, which indicates that this object is fully backwards compatible with the legacy class and the methods available for it (see Harrington (2010) for details). The printed output provides information about which database was queried and what the query was as well as information about the labels, start and end times (in milliseconds), session, bundle, level and type information. The call to colnames() shows that the resulting object has additional columns, which are ignored by the print() function. This somewhat hidden information is used to store information about what the exact items or sequence of items were retrieved from the emuDB. This information is needed to know which items to start from in a requery (see Section @ref(subsec:requery}) and is also the reason why an emuRsegs object should be viewed as a reference of sequences of annotation items that belong to a single level in all annotation files of an emuDB.

```
# query database
sl = query(ae, "[Syllable == S ^ Text == amongst]")

# show class vector
class(sl)
```

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

```
# show sl object
sl
```

```
## segment  list from database:  ae
## query was:  [Syllable == S ^ Text == amongst]
##   labels   start     end session   bundle    level type
## 1      S 256.925 674.175    0000 msajc003 Syllable ITEM
```

```
# show all (incl. hidden) column names
colnames(sl)
```

```
##  [1] "labels"             "start"              "end"
##  [4] "utts"               "db_uuid"            "session"
##  [7] "bundle"             "start_item_id"      "end_item_id"
## [10] "level"              "start_item_seq_idx" "end_item_seq_idx"
## [13] "type"               "sample_start"       "sample_end"
## [16] "sample_rate"
```

6.2 EQL: The EMU Query Language version 2

The EQL user interface was retained from the legacy system because it was sufficiently flexible and expressive enough to meet the query needs in most types of speech science research. The EQL parser implemented in `emuR` is based on the Extended Backus–Naur form (EBNF) (Garshol, 2003) formal language definition of John (2012), which defines the symbols and the relationship of those symbols to each other on which this language is built (see adapted version of entire EBNF in Appendix **??**). Here we will describe the various terms and components that comprise the slightly adapted version 2 of the EQL. It is worth noting that the new query mechanism uses a relational back-end to handle the various query operations (see Chapter **??** for details). This means that expert users, who are proficient in Structured Query Language (SQL) may also query this relational back-end directly. However, we feel the EQL provides a simple abstraction layer which is sufficient for most speech and spoken language research.

6.2.1 Simple queries

The most basic form of an EQL query is a simple equality, inequality, matching or non-matching query, two of which are displayed in R Example **??**. The syntax of a simple query term is `[L OPERATOR A]`, where `L` specifies a level (or alternatively the name of a parallel attribute definition); `OPERATOR` is one of `==` (equality), `!$=$` (inequality), `=~` (matching) or `!~` (non-matching); and `A` is an expression specifying the labels of the annotation items of `L` [1]. The second query in R Example **??** queries an event level. The result of querying an event level contains the same information as that of a segment level query except that the derived end times have the value zero.

```
# query all annotation items containing
# the label "m" on the "Phonetic" level
sl = query(ae, "Phonetic == m")

# query all items NOT containing the
# label "H*" on the "Tone" level
sl = query(ae, "Tone != H*")

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

```
## event  list from database:  ae
```

[1] The examples and syntax descriptions used in this chapter have been adapted from examples by Cassidy and Harrington (2001) and Harrington and Cassidy (2002) and were largely extracted from the `EQL` vignette of the `emuR` package. All of the examples were adapted to work with the supplied *ae* `emuDB`.

```
## query was:  Tone != H*
##   labels start end session   bundle level  type
## 1     L-  1107   0    0000 msajc003  Tone EVENT
```

R Example **??** queries two levels that contain time information: a segment level and an event level. As described in Chapter **??**, annotations in the EMU-SDMS may also contain levels that do not contain time information. R Example **??** shows a query that queries annotation items on a level that does not contain time information (the *Syllable* level) to show that the result contains deduced time information from the time-bearing sub-level.

```
# query all annotation items containing
# the label S on the Syllable level
sl = query(ae, "Syllable == S")

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

```
## segment  list from database:  ae
## query was:  Syllable == S
##   labels   start     end session   bundle    level type
## 1      S 256.925 674.175    0000 msajc003 Syllable ITEM
```

6.2.1.1 Queries using regular expressions

The slightly expanded version 2 of the EQL, which comes with the `emuR` package, introduces regular expression operators (`=~` and `!~`). These allow users to formulate regular expressions for more expressive and precise pattern matching of annotations. A minimal set of examples displaying the new regular expression operators is shown in Table **??**.

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

EQL V2: examples of simple and complex query strings using RegEx operators including their function descriptions.

Query

Function

`Phonetic =~ '[AIOUEV]'`

A disjunction of annotations using a RegEx character class

`Word =~ a.*`

All words beginning with *a*

`Word !~ .*st`

All words not ending in *st*

`[Phonetic == n ^ #Syllable =~ .*]`

All syllables that dominate an n segment of the Phonetic level

6.2.2 Combining simple queries

The EQL contains three operators that can be used to combine the simple query terms described above as well as position queries which we will describe below. These three operators are the sequence operator, `->`; the conjunction operator, `&`; and the domination operator, `^`, which is used to perform hierarchical queries. These three types of queries are described below. To start with, we describe the two types of queries that query more complex annotation structures on the same level (sequence and conjunction queries). This is followed by a description of domination queries that query hierarchically linked annotation structures, sometimes spanning multiple annotation levels.

6.2.2.1 Sequence queries

The syntax of a query string using the `->` sequence operator is `[L == A -> L == B]` where annotation item `A` on level `L` precedes item `B` on level `L`. For a sequence query to work, both arguments must be on the same level. Alternatively parallel attribute definitions of the same level may also be chosen (see Chapter **??** for further details). An example of a query string using the sequence operator is displayed in R Example **??**. All rows in the resulting segment list have the start time of @, the end time of n and their labels are @->n, where the -> substring denotes the sequence.

```
# query all sequences of items on the "Phonetic" level
# in which an item containing the label "@" is followed by
# an item containing the label "n"
sl = query(ae, "[Phonetic == @ -> Phonetic == n]")

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

```
## segment  list from database:  ae
## query was:  [Phonetic == @ -> Phonetic == n]
##   labels    start      end session   bundle    level    type
## 1   @->n 1715.425 1791.425    0000 msajc003 Phonetic SEGMENT
```

6.2.2.2 Result modifier

Because users are often interested in just one element of a compound query such as sequence queries (e.g., the @s in a @->n sequences), the EQL offers a so-called result modifier symbol, `#`. This symbol may be placed in front of any simple query component of a multi component query as depicted in R Example **??**. Placing the hashtag in front of either the left or the right simple query term will result in segment lists that contains only the annotation items of the simple query term that have the hashtag in front of it. Only one result modifier may be used per query.

```
# query the "@"s in "@->n" sequences
sl = query(ae, "[#Phonetic == @ -> Phonetic == n]")

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

```
## segment  list from database:  ae
## query was:  [#Phonetic == @ -> Phonetic == n]
##   labels    start      end session   bundle    level    type
## 1      @ 1715.425 1741.425    0000 msajc003 Phonetic SEGMENT
```

```
# query the "n"s in a "@->n" sequences
sl = query(ae, "[Phonetic == @ -> #Phonetic == n]")
```

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

```
## segment  list from database:  ae
## query was:  [Phonetic == @ -> #Phonetic == n]
##   labels    start      end session   bundle    level    type
## 1      n 1741.425 1791.425    0000 msajc003 Phonetic SEGMENT
```

6.2.2.3 Conjunction queries

The syntax of a query string using the conjunction operator can schematically be written as: `[L_a1 == A & L_a2 == B & L_a3 == C & L_a4 == D & ... & L_an == N]`, where annotation items on level `L` have the label `A` and also have the parallel labels `B`, `C`, `D`, ..., `N` (see Chapter **??** for more information about parallel labels). By analogy with the sequence operator, all simple query statements must refer to the same level (i.e., only parallel attributes definitions of the same level indicated by the `a1 - an` may to be chosen). Hence, the conjunction operator is used to combine query conditions on the same level. Using the conjunction operator is useful for two reasons:

- It combines different attributes of the same level: `[Text == always & Accent == S]` where *Text* and *Accent* are additional attributes of level *Word*; and
- It combines a simple query with a function query (see Position Queries Section **??**): *[Phonetic == l & Start(Word, Phonetic) == 1]*.

An example of a query string using the conjunction operator is displayed in R Example **??**.

```
# query all words with the orthographic transcription "always"
# that also have a strong word accent ("S")
query(ae, "[Text == always & Accent == S]")
```

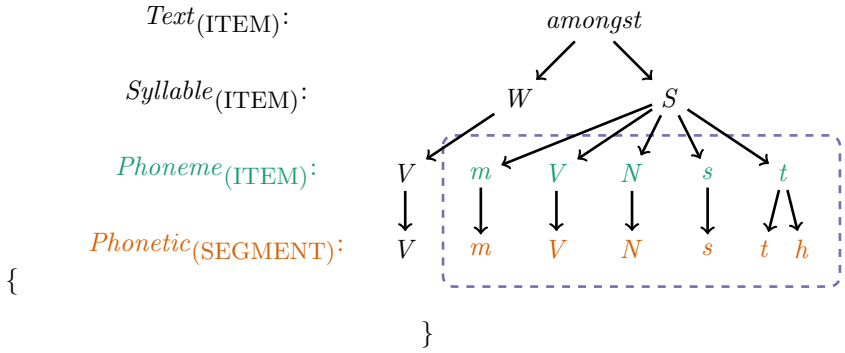
```
## segment  list from database:  ae
## query was:  [Text == always & Accent == S]
##   labels   start      end session   bundle level type
## 1 always 775.475 1280.175    0000 msajc022  Text ITEM
```

R Example **??** does not make use of the result modifier symbol. However, only the annotation items of the left simple query term (*Text == always*) are returned. This behavior is true for all EQL operators that combine simple query terms except for the sequence operator. As it is more explicit to use the result modifier to express the desired result, we recommend using the result modifier where possible. The more explicit variant of the above query which yields the same result is *"[#Text == always & Word == C]"*.

6.2.2.4 Domination/hierarchical queries

Compared to sequence and conjunction queries, a domination query using the operator ^ is not bound to a single level. Instead, it allows users to query annotation items that are directly or indirectly linked over one or more levels. Queries using the domination operator are often referred to as hierarchical queries as they provide the ability to query the hierarchical annotations in a vertical or inter-level manner. Figure 6.2.2.4 shows the same partial hierarchy as Figure 6.1 but highlights the annotational items that are dominated by the strong syllable (*S*) of the *Syllable* level. Such linked hierarchical sub-structures can be queried using hierarchical/domination queries.

\begin{figure}



{

}

\caption{Partial hierarchy depicting all annotation items that are dominated by the strong syllable (*S*) of the *Syllable* level (inside dashed box). Items marked extcolor{three_color_c1}{green} belong to the extcolor{three_color_c1}{*Phoneme*} level, items marked extcolor{three_color_c2}{orange} belong to the extcolor{three_color_c2}{*Phonetic*} level and the extcolor{three_color_c3}{purple} dashed box indicates the set of items that are dominated by *S*.} \end{figure}

A schematic representation of a simple domination query string that retrieves all annotation items *A* of level `L1` that are dominated by items `B` in level `L2` (i.e., items that are directly or indirectly linked) is `[L1 == A ^{ L2 == B]}`. Although the domination relationship is directed the domination operator is not. This means that either items in `L1` dominate items in `L2` or items in `L2` dominate items in `L1`. Note that link definitions that specify the validity of the domination have to be present in the `emuDB` configuration for this to work (see Chapter 5 for details). An example of a query string using the domination operator is displayed in R Example **??**.

```
# query all "p" phoneme items that belong
# to / are dominated by a strong syllable ("S")
sl = query(ae, "[Phoneme == p ^ Syllable == S]")

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

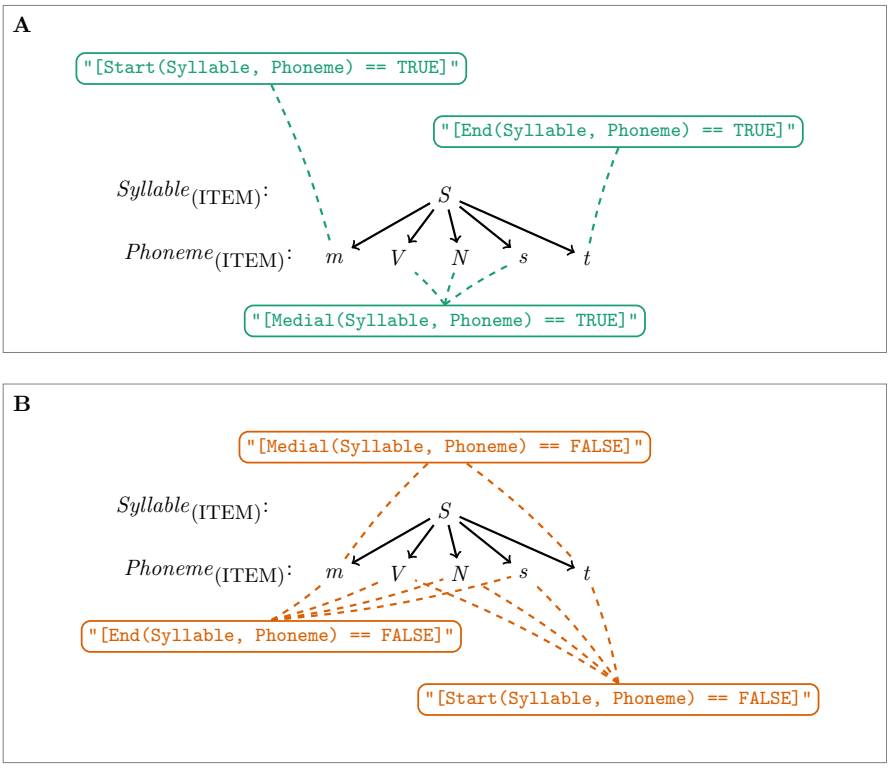
```
## segment  list from database:  ae
## query was:  [Phoneme == p ^ Syllable == S]
##   labels   start     end session   bundle   level type
## 1      p 558.575 639.575    0000 msajc015 Phoneme ITEM
```

As with the conjunction query, if no result modifier is present, a dominates query returns the annotation items of the left simple query term. Hence, the more explicit variant of the above query is `"[#Phoneme == p ^ Syllable == S]"`.

6.2.3 Position queries

The EQL has three function terms that specify where in a domination relationship a child level annotation item is allowed to occur. The three function terms are `Start()`, `End()` and `Medial()`. A schematic representation of a query string representing a simple usage of the `Start()`, `End()` and `Medial()` function would be: `POSFCT(L1, L2) == TRUE`. In this representation `POSFCT` is a placeholder for one of the three functions, at which level `L1` must dominate level `L2`. Where `L1` does indeed dominate `L2`, the corresponding item from level `L2` is returned. If the expression is set to `FALSE` (i.e., `POSFCT(L1, L2) == FALSE`), all the items that do not match the condition of `L2` are returned. An illustration of what is returned by each of the position functions depending on if they are set to `TRUE` or `FALSE` is depicted in Figure 6.2.3, while R Example **??** shows an example query using a position query term.

\begin{figure}



{

}

\caption{Illustration of what is returned by the `Start()`, `Medial()` and `End()` functions depending if they are set to extbf{A:} extcolor{three_color_c1}{`TRUE`} (green) or extbf{B:} extcolor{three_color_c2}{`FALSE`} (orange).} \end{figure}

```
# query all phoneme items that occur
# at the start of a syllable
sl = query(ae, "[Start(Syllable, Phoneme) == TRUE]")

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

```
## segment  list from database:  ae
## query was:  [Start(Syllable, Phoneme) == TRUE]
##   labels   start     end session   bundle   level type
## 1      V 187.425 256.925    0000 msajc003 Phoneme ITEM
```

6.2.4 Count queries

A further query component of the EQL are so-called count queries. They allow the user to specify how many child nodes a parent annotation item is allowed to have. Figure @ref(fig:query_amongstHierCount) displays two syllables, one containing one phoneme and one phonetic annotation item, the other containing five phoneme and six phonetic items. Using EQL's `Num()` function it is possible to specify which of the two syllables should be retrieved, depending on the number of phonemic or phonetic elements to which it is directly or indirectly linked. R Example @ref(rexample:query_countQuery) shows a query that queries all syllables that contain five phonemes.

A schematic representation of a query string utilizing the count mechanism would be `[Num(L1, L2) == N]`, where `L1` contains `N` annotation items in `L2`. For this type of query to work `L1` has to dominate `L2` (i.e., be

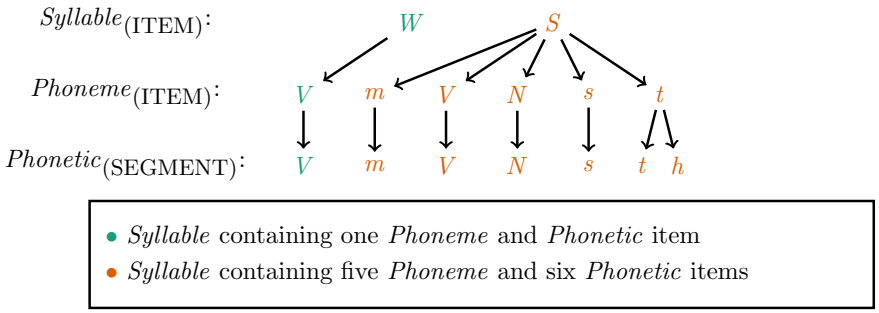


Figure 6.2: Partial hierarchy depicting a *Syllable* containing one *Phoneme* and *Phonetic* item (green) and a *Syllable* containing five *Phoneme* and six *Phonetic* items (orange).

a parent level to `L2`). As the query matches a number (`N`), it is also possible to use the operators `>` (more than), `<` (less than) and `!=` (not equal to). The resulting segment list contains items of `L1`.

```
# retrieve all syllables that contain five phonemes
query(ae, "[Num(Syllable, Phoneme) == 5]")
```

```
## segment  list from database:  ae
## query was:  [Num(Syllable, Phoneme) == 5]
##   labels    start      end session   bundle    level type
## 1      S  256.925  674.175    0000 msajc003 Syllable ITEM
## 2      S  739.925 1289.425    0000 msajc003 Syllable ITEM
## 3      W 2228.475 2753.975    0000 msajc010 Syllable ITEM
## 4      S 1890.275 2469.525    0000 msajc022 Syllable ITEM
## 5      S 1964.425 2554.175    0000 msajc023 Syllable ITEM
```

6.2.5 More complex queries

By using the correct bracketing, all of the above query components can be combined to formulate more complex queries that can be used to answer questions such as: *Which occurrences of the word "his" follow three-syllable words which contain a schwa (@) in the first syllable?* Such multi-part questions can usually be broken down into several sub-queries. These sub-queries can then be recombined to formulate the complex query. The steps to answering the above multi-part question are:

1. *Which occurrences of the word "his" …*: `[Text == his]`
2. *… three-syllable words …*: `[Num(Text, Syllable) == 3]`
3. *… contain a schwa (@) in the first syllable …*: `[Phoneme == @ ^ Start(Word, Syllable) == 1]`
4. All three can be combined by saying 2 dominates 3 (`[2 ^ 3]`) and these are followed by 1 (`[2 ^ 3] -> 1]`)

The combine query is depicted in R Example **??**. This complex query demonstrates the expressive power of the query mechanism that the EMU-SDMS provides.

```
# perform complex query
# Note that the use of paste0() is optional, as
# it is only used for formatting purposes
query(ae, paste0("[[[Num(Text, Syllable) == 3] ",
                 "^ [Phoneme == @ ^ Start(Word, Syllable) == 1]] ",
                 "-> #Text = his]"))
```

```
## segment  list from database:  ae
## query was:  [[[Num(Text, Syllable) == 3] ^ [Phoneme == @ ^ Start(Word, Syllable) == 1]] -> #Text = hi
```

```
##   labels    start      end session   bundle level type
## 1    his 2693.675 2780.725    0000 msajc015  Text ITEM
```

As mastering these complex compound queries can require some practice, several simple as well as more complex examples that combine the various EQL components described above are available in Appendix **??**. These examples provide practical examples to help users find queries suited to their needs.

6.2.6 Deducing time

The default behavior of the legacy EMU system was to automatically deduce time information for queries of levels that do not contain time information. This was achieved by searching for the time-bearing sub-level and calculating the start and end times from the left-most and right-most annotation items which where directly or indirectly linked to the retrieved parent item. This upward purculation of time information is also the default behavior of the new EMU-SDMS. However, a new feature has been added to the query engine which allows the calculation of time to be switched off for a given query using the `calcTimes` parameter of the `query()` function. This is beneficial in two ways: for one, levels that do not have a time-bearing sub-level may be queried and secondly, the execution time of queries can be greatly improved. The performance increase becomes evident when performing queries on large data sets on one of the top levels of the hierarchy (e.g., *Utterance* or *Intonational* in the *ae* `emuDB`). When deducing time information for annotation items that contain large portions of the hierarchy, the query engine has to walk down large partial hierarchies to find the left-most and right-most items on the time-bearing sub-level. This can be a computationally expensive operation and is often unnecessary, especially during data exploration. R Example **??** shows the usage of this parameter by querying all of the items of the *Intonational* level and displaying the `NA` values for start and end times in the resulting segment list. It is worth noting that the missing time information excluded during the original query can be retrieved at a later point in time by performing a hierarchical requery (see Section 6.2.7) on the same level.

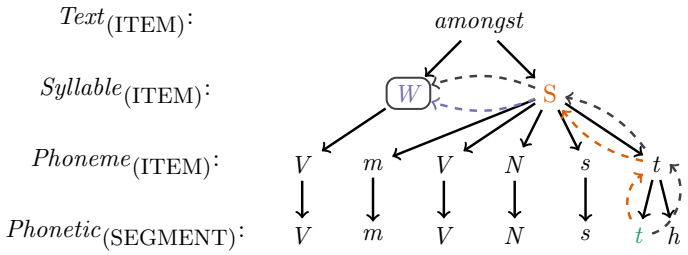
```
# query all intonational items
sl = query(ae, "Intonational =~ .*", calcTimes = F)

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

```
## segment  list from database:  ae
## query was:  Intonational =~ .*
##   labels start end session   bundle        level type
## 1     L%    NA  NA    0000 msajc003 Intonational ITEM
```

6.2.7 Requery

A popular feature of the legacy system was the ability to use the result of a query to perform an additional query, called a requery, starting from the resulting items of a query. The requery functionality was used to move either sequentially (horizontally) or hierarchically (vertically) through the hierarchical annotation structure. Although this feature technically does not extend the querying functionality (it is possible to formulate EQL queries that yield the same results as a query followed by $1 : n$ requeries), requeries benefit the user by breaking down the task of formulating long query terms into multiple, simpler queries. Compared with the legacy system, this feature is implemented in the `emuR` package in a more robust way, as unique item IDs are present in the result of a query, eliminating the need for searching the starting segments based on their time information. Examples of queries and their results within a hierarchical annotation based on a hierarchical and sequential requery as well as their EQL equivalents are illustrated in Figure 6.2.7.

\begin{figure}




```
sl1 = query(emuDBhandle, query = "Phonetic == t")
sl2 = requery_hier(emuDBhandle, sl1, level = "Syllable")
sl3 = requery_seq(emuDBhandle, sl2, offset = -1)
```

vs.

```
sl3 = query(emuDBhandle, query = "[#Syllable =~ .* ->
[Syllable =~ .* ^ Phonetic == t]]")
```

{

}

\caption{Three-step (extcolor{three_color_c1}{query} -> extcolor{three_color_c2}{requery_hier} -> extcolor{three_color_c3}{requery_seq}) requery procedure, its single extcolor{darkgray}{query} counterpart and their color coded movements within the annotation hierarchy.} \end{figure}

R Example **??** illustrates how the same results of the sequential query `[\#Phonetic =~ .* -> Phonetic == n]` can be achieved using the `requery_seq()` function. Further, it shows how the `requery_hier()` function can be used to move vertically through the annotation structure by starting at the *Syllable* level and retrieving all the *Phonetic* items for the query result.

```
#########################
# requery_seq()

# query all "n" phonetic items
sl_n = query(ae, "Phonetic == n")

# sequential requery (left shift result by 1 (== offset of -1))
# and hence retrieve all phonetic items directly preceeding
# all "n" phonetic items
sl_precn = requery_seq(ae, seglist = sl_n, offset = -1)

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

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

```
#########################
# requery_hier()

# query all strong syllables (S)
sl_s = query(ae, "Syllable == S")

# hierarchical requery
sl_phonetic = requery_hier(ae, seglist = sl_s,
                           level = "Phonetic")
```

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

```
## segment  list from database:  ae
## query was:  FROM REQUERY
##             labels   start     end session   bundle    level    type
## 1 m->V->N->s->t->H 256.925 674.175    0000 msajc003 Phonetic SEGMENT
```

6.3 Discussion

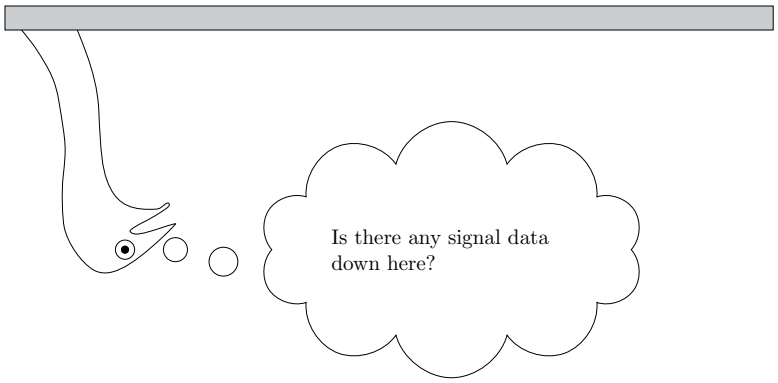
This chapter gave an overview of the abilities of the query system of the EMU-SDMS. We feel the EQL is an expressive, powerful, yet simple to learn and domain-specific query language that allows users to adequately query complex annotation structures. Further, the query system provided by the EMU-SDMS surpasses the querying capabilities of most commonly used systems. As the result of a query is a superclass of the common `data.frame` object, these results can easily be further processed using various R functions (e.g., to remove unwanted segments). Further, the results of queries can be used as input to the `get_trackdata()` function (see Chapter 7) which makes the query system a vital part in the default workflow described in Chapter 2.

Although the query mechanism of the EMU-SDMS covers most linguistic annotation query needs (including co-occurrence and domination relationship child position queries), it has limitations due to its domain-specific nature, its simplicity and its predefined result type. Performing more general queries such as: *What is the average age of the male speakers in the database who are taller than 1.8 meters?* is not directly possible using the EQL. Even if the gender, height and age parameters are available as part of the database's annotations (e.g., using the single bundle root node metadata strategy described in Chapter **??**) they would be encoded as strings, which do not permit direct calculations or numerical comparisons. However, it is possible to answer these types of questions using a multi-step approach. One could, for example, extract all height items and convert the strings into numbers to filter the items containing a label that is greater than 1.8. These filtered items could then be used to perform two requeries to extract all male speakers and their age labels. These age labels could once again be converted into numbers to calculate their average. Although not as elegant as other languages, we have found that most questions that arise as part of studies working with spoken language database can be answered using such a multi-step process including some data manipulation in R, provided the necessary information is encoded in the database. Additionally, from the viewpoint of a speech scientist, we feel that the intuitiveness of an EQL expression (e.g., a query to extract the sibilant items for the question asked in the introduction: `"Phonetic == s|z|S|Z"`) exceeds that of a comparable general purpose query language (e.g. a semantically similar SQL statement: `SELECT desired_columns FROM items AS i, labels AS l WHERE i.unique_bundle_item_id = l.uniq_bundle_item_id AND l.label = 's' OR l.label = 'z' OR l.label = 's' OR l.label = 'S' OR l.label = 'Z'`). This difference becomes even more apparent with more complex EQL statements, which can have very long, complicated and sometimes multi-expression SQL counterparts.

A problem which the EMU-SDMS does not explicitly address is the problem of cross-corpus searches. Different `emuDB`s may have varying annotation structures with varying semantics regarding the names or labels given to objects or annotation items in the databases. This means that it is very likely that a complex query formulated for a certain `emuDB` will fail when used to query other databases. If, however, the user either finds a query that works on every `emuDB` or adapts the query to extract the items she/he is interested in, a cross-corpus comparison is simple. As the result of a query and the corresponding data extraction routines are the same, regardless of database they where extracted from, these results are easily comparable. However, it is worth noting that the EMU-SDMS is completely indifferent to the semantics of labels and level names, which means it is the user's responsibility to check if a comparison between databases is justifiable (e.g., *are all segments containing the label "@" of the level "Phonetic"" in all `emuDB`s annotating the same type of phoneme?*).

Chapter 7

Signal data extraction [1]



As mentioned in the default workflow of Chapter @ref(chap:overview}, after querying the symbolic annotation structure and dereferencing its time information, the result is a set of items with associated time stamps. It was necessary that the `emuR` package contain a mechanism for extracting signal data corresponding to this set of items. As illustrated in Chapter @ref(chap:wrassp}, `wrassp` provides the R ecosystem with signal data file handling capabilities as well as numerous signal processing routines. `emuR` can use this functionality to either obtain pre-stored signal data or calculate derived signal data that correspond to the result of a query. Figure @ref(fig:sigDataExtr}A shows a snippet of speech with overlaid annotations where the resulting `SEGMENT` of an example query (e.g., `"Phonetic == ai"`) is highlighted in yellow. Figure 7.1B displays a time parallel derived signal data contour as would be returned by one of `wrassp`'s file handling or signal processing routines. The yellow segment in Figure 7.1B marks the corresponding samples that belong to the `ai` segment of Figure 7.1A.

R Example **??** shows how to create the demo data that will be used throughout this chapter.

```
# load the package
library(emuR)

# create demo data in directory provided by the tempdir() function
create_emuRdemoData(dir = tempdir())

# get the path to a emuDB called "ae" that is part of the demo data
path2directory = file.path(tempdir(), "emuR_demoData", "ae_emuDB")

# load emuDB into current R session
```

[1] Parts of this chapter have been published in Winkelmann et al. (2017).

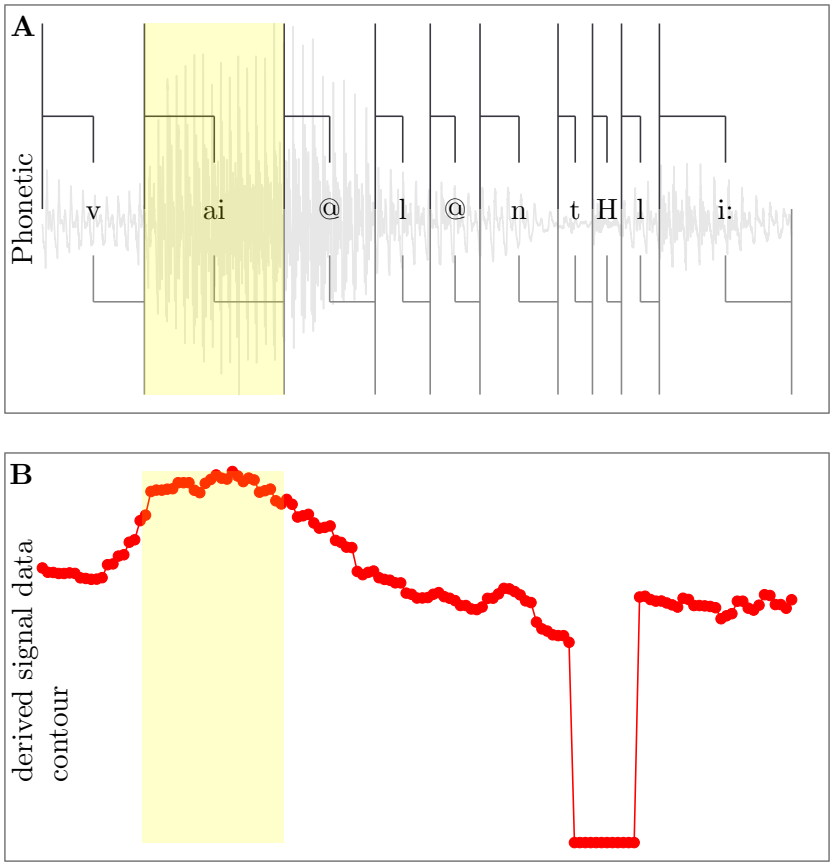


Figure 7.1: Segment of speech with overlaid annotations and time parallel derived signal data contour.

```
ae = load_emuDB(path2directory)
```

7.1 Extracting pre-defined tracks

To access data that are stored in files, the user has to define tracks for a database that point to sequences of samples in files that match a user-specified file extension. The user-defined name of such a track can then be used to reference the track in the signal data extraction process. Internally, `emuR` uses `wrassp` to read the appropriate files from disk, extract the sample sequences that match the result of a query and return values to the user for further inspection and evaluation. R Example **??** shows how a signal track that is already defined in the `ae` demo database can be extracted for all annotation items on the `Phonetic` level containing the label `ai`.

```
# list currently available tracks
list_ssffTrackDefinitions(ae)
```

```
##   name columnName fileExtension
## 1  dft        dft           dft
## 2   fm         fm           fms
```

```
# query all "ai" phonetic segments
ai_segs = query(ae, "Phonetic == ai")

# get "fm" track data for these segments
# Note that verbose is set to FALSE
# only to avoid a progress bar
# being printed in this document.
ai_td_fm = get_trackdata(ae,
                         seglist = ai_segs,
                         ssffTrackName = "fm",
                         verbose = FALSE)

# show summary of ai_td_fm
summary(ai_td_fm)
```

```
## Emu track data from 6 segments
##
## Data is  4 dimensional from track fm
## Mean data length is  30.5  samples
```

Being able to access data that is stored in files is important for two main reasons. Firstly, it is possible to generate files using external programs such as VoiceSauce (Shue et al., 2011), which can export its calculated output to the general purpose SSFF file format. This file mechanism is also used to access data produced by EMA, EPG or many other forms of signal data recordings. Secondly, it is possible to track, save and access manipulated data such as formant values that have been manually corrected. It is also worth noting that the `get_trackdata()` function has a predefined track which is always available without it having to be defined. The name of this track is `MEDIAFILE_SAMPLES` which references the actual samples of the audio files of the database. R Example **??** shows how this predefined track can be used to access the audio samples belonging to the segments in `ai_segs`.

```
# get media file samples
ai_td_mfs = get_trackdata(ae,
                          seglist = ai_segs,
                          ssffTrackName = "MEDIAFILE_SAMPLES",
                          verbose = FALSE)
```

```
# show summary of ai_td_fm
summary(ai_td_mfs)
```

```
## Emu track data from 6 segments
##
## Data is  1 dimensional from track MEDIAFILE_SAMPLES
## Mean data length is  3064.333  samples
```

7.2 Adding new tracks

As described in detail in Section **??**, the signal processing routines provided by the `wrassp` package can be used to produce SSFF files containing various derived signal data (e.g., formants, fundamental frequency, etc.). R Example **??** shows how the `add_ssffTrackDefinition()` can be used to add a new track to the `ae emuDB`. Using the `onTheFlyFunctionName` parameter, the `add_ssffTrackDefinition()` function automatically executes the `wrassp` signal processing function `ksvF0` (`onTheFlyFunctionName = "ksvF0"`) and stores the results in SSFF files in the bundle directories.

```
# add new track and calculate
# .f0 files on-the-fly using wrassp::ksvF0()
add_ssffTrackDefinition(ae,
                        name = "F0",
                        onTheFlyFunctionName = "ksvF0",
                        verbose = FALSE)

# show newly added track
list_ssffTrackDefinitions(ae)
```

```
##   name columnName fileExtension
## 1  dft        dft           dft
## 2   fm         fm           fms
## 3   F0         F0            f0
```

```
# show newly added files
library(tibble) # convert to tibble only to prettify output
as_tibble(list_files(ae, fileExtension = "f0"))
```

```
## # A tibble: 7 x 4
##   session bundle   file        absolute_file_path
## * <chr>   <chr>    <chr>       <chr>
## 1 0000    msajc003 msajc003.f0 /private/var/folders/yk/8z9tn7kx6hbcg_9n4c~
## 2 0000    msajc010 msajc010.f0 /private/var/folders/yk/8z9tn7kx6hbcg_9n4c~
## 3 0000    msajc012 msajc012.f0 /private/var/folders/yk/8z9tn7kx6hbcg_9n4c~
## 4 0000    msajc015 msajc015.f0 /private/var/folders/yk/8z9tn7kx6hbcg_9n4c~
## 5 0000    msajc022 msajc022.f0 /private/var/folders/yk/8z9tn7kx6hbcg_9n4c~
## 6 0000    msajc023 msajc023.f0 /private/var/folders/yk/8z9tn7kx6hbcg_9n4c~
## 7 0000    msajc057 msajc057.f0 /private/var/folders/yk/8z9tn7kx6hbcg_9n4c~
```

```
# extract newly added trackdata
ai_td = get_trackdata(ae,
                      seglist = ai_segs,
                      ssffTrackName = "F0",
                      verbose = FALSE)
```

```
# show summary of ai_td
summary(ai_td)
```

```
## Emu track data from 6 segments
##
## Data is  1 dimensional from track F0
## Mean data length is  30.5  samples
```

7.3 Calculating tracks on-the-fly

With the `wrassp` package, we were able to implement a new form of signal data extraction which was not available in the legacy system. The user is now able to select one of the signal processing routines provided by `wrassp` and pass it on to the signal data extraction function. The signal data extraction function can then apply this `wrassp` function to each audio file as part of the signal data extraction process. This means that the user can quickly manipulate function parameters and evaluate the result without having to store to disk the files that would usually be generated by the various parameter experiments. In many cases this new functionality eliminates the need for defining a track definition for the entire database for temporary data analysis purposes. R Example **??** shows how the `onTheFlyFunctionName` parameter of the `get_trackdata()` function is used.

```
ai_td_pit = get_trackdata(ae,
                          seglist = ai_segs,
                          onTheFlyFunctionName = "mhsF0",
                          verbose = FALSE)

# show summary of ai_td
summary(ai_td_pit)
```

```
## Emu track data from 6 segments
##
## Data is  1 dimensional from track pitch
## Mean data length is  30.5  samples
```

7.4 The resulting object: `trackdata` vs. `emuRtrackdata`

The default resulting object of a call to `get_trackdata()` is of class `trackdata` (see R Example **??**). The `emuR` package provides multiple routines such as `dcut()`, `trapply()` and `dplot()` for processing and visually inspecting objects of this type (see harrington:2010a and Section 3.5 for examples of how these can be used).

```
# show class vector of ai_td_pit
class(ai_td_pit)
```

```
## [1] "trackdata"
```

As the `trackdata` object is a fairly complex nested matrix object with internal reference matrices, which can be cumbersome to work with, the `emuR` package introduces a new equivalent object type called `emuRtrackdata` that essentially is a flat `data.frame` object. This object type can be retrieved by setting the `resultType` parameter of the `get_trackdata()` function to `emuRtrackdata`. R Example **??** shows how this can be achieved.

```
ai_emuRtd_pit = get_trackdata(ae,
                              seglist = ai_segs,
```

```
                                onTheFlyFunctionName = "mhsF0",
                                resultType = "emuRtrackdata",
                                verbose = FALSE)

# show first row (convert to tibble only to prettify output)
as_tibble(ai_emuRtd_pit[1, ])
```

```
## # A tibble: 1 x 21
##   sl_rowIdx labels start   end utts   db_uuid session bundle start_item_id
## *     <int> <chr>  <dbl> <dbl> <chr>  <chr>   <chr>   <chr>          <int>
## 1         1 ai      863. 1016. 0000:~ 0fc618~ 0000    msajc~           161
## # ... with 12 more variables: end_item_id <int>, level <chr>,
## #   start_item_seq_idx <int>, end_item_seq_idx <int>, type <chr>,
## #   sample_start <int>, sample_end <int>, sample_rate <int>,
## #   times_orig <dbl>, times_rel <dbl>, times_norm <dbl>, T1 <dbl>
```

```
# show relative time values of the first segment
# (relative time values always start at 0 for every segment)
ai_emuRtd_pit[ai_emuRtd_pit$sl_rowIdx == 1, ]$times_rel
```

```
##  [1]   0   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80
## [18]  85  90  95 100 105 110 115 120 125 130 135 140 145
```

```
# show original time values of the first segment
# (absolute time values always start at the original
# time stamp for that sample within the track)
ai_emuRtd_pit[ai_emuRtd_pit$sl_rowIdx == 1, ]$times_orig
```

```
##  [1]  867.5  872.5  877.5  882.5  887.5  892.5  897.5  902.5  907.5  912.5
## [11]  917.5  922.5  927.5  932.5  937.5  942.5  947.5  952.5  957.5  962.5
## [21]  967.5  972.5  977.5  982.5  987.5  992.5  997.5 1002.5 1007.5 1012.5
```

As can be seen by the first row output of R Example **??**, the `emuRtrackdata` object is an amalgamation of both a segment list and a `trackdata` object. The first `sl_rowIdx` column of the `ai_emuRtd_pit` object indicates the row index of the segment list the current row belongs to, the `times_rel` and `times_orig` columns represent the relative time and the original time of the samples contained in the current row (see R Example **??**) and T1 (to Tn in n dimensional `trackdata`) contains the actual signal sample values. It is also worth noting that the `emuR` package provides a function called `create_emuRtrackdata()`, which allows users to create `emuRtrackdata` from a segment list and a `trackdata` object. This is beneficial as it allows `trackdata` objects to be processed using functions provided by the `emuR` package (e.g., `dcut()` and `trapply()`) and then converts them into a standardized `data.frame` object for further processing (e.g., using R packages such as `lme4` or `ggplot2` which were implemented to use with `data.frame` objects). R Example **??** shows how the `create_emuRtrackdata()` function is used.

```
# create emuRtrackdata object
ai_emuRtd_pit  = create_emuRtrackdata(sl = ai_segs,
                                      td = ai_td_pit)

# show first row and
# selected columns of ai_emuRtd_pit
ai_emuRtd_pit[1, ]
```

```
##   sl_rowIdx labels   start      end session   bundle    level    type
## 1         1     ai 862.875 1015.825    0000 msajc010 Phonetic SEGMENT
##   times_orig times_rel times_norm       T1
## 1      867.5         0          0 134.7854
```

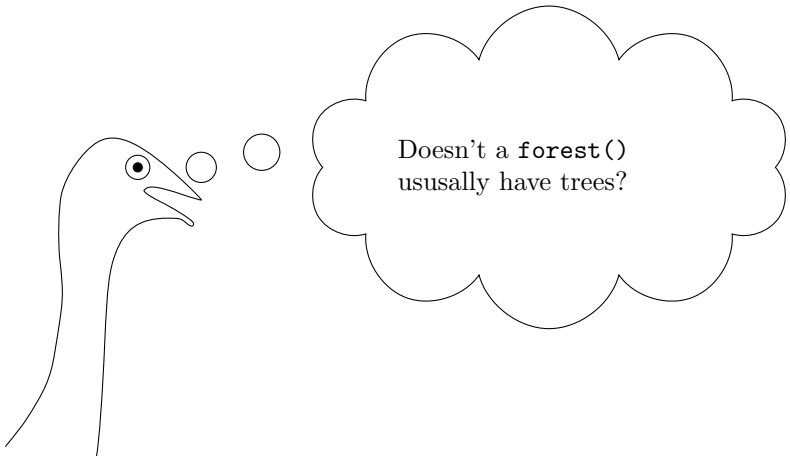
The general question remains as to when to use the `trackdata` and when to use the `emuRtrackdata` object and what the benefit of each class is. The `trackdata` object has a number of associated class functions (e.g. `trapply()`, `dcut()`, `dcut()` and `eplot()`) that ease data manipulation and visualization. Further, it avoids data redundancy and therefore has a smaller memory footprint than the `emuRtrackdata` object (this is usually negligible on current systems); however, this makes it rather difficult to work with. The `emuRtrackdata` object is intended as a long term replacement for the `trackdata` object as it contains all of the information of the corresponding `trackdata` object as well as its associated segment list. As is often the case with tabular data, the `emuRtrackdata` object carries certain redundant information (e.g. segment start and end times). However, the benefit of having a `data.frame` object that contains all the information needed to process the data is the ability to replace package specific functions (e.g. trapply() etc.) with standardized `data.frame` processing and visualization procedures that can be applied to any `data.frame` object independent of the package that generated it. Therefore, the knowledge that is necessary to process an `emuRtrackdata` object can be transferred to/from other packages which is not the case for `trackdata` object. Future releases of the `emuR` package as well as this manual will contain various examples of how to replace the functionality of the package-specific functions mentioned above with equivalent data manipulation and visualization using the `dplyr` as well as the `ggplot2` packages.

7.5 Conclusion

This chapter introduced the signal data extraction mechanics of the `emuR` package. The combination of the `get_trackdata()` function and the file handling and signal processing abilities of the `wrassp` package (see Chapter 8 for further details) provide the user with a flexible system for extracting derived or complementary signal data belonging to their queried annotation items.

Chapter 8

The R package `wrassp` [1]



8.1 Introduction

This chapter gives an overview and introduction to the `wrassp` package. The `wrassp` package is a wrapper for `R` around Michel Scheffers' `libassp` (Advanced Speech Signal Processor). The `libassp` library and therefore the `wrassp` package provide functionality for handling speech signal files in most common audio formats and for performing signal analyses common in the phonetic and speech sciences. As such, `wrassp` fills a gap in the R package landscape as, to our knowledge, no previous packages provided this specialized functionality. The currently available signal processing functions provided by `wrassp` are:

- `acfana()`: Analysis of short-term autocorrelation function
- `afdiff()`: Computes the first difference of the signal
- `affilter()`: Filters the audio signal (e.g., low-pass and high-pass)
- `cepstrum()`: Short-term cepstral analysis
- `cssSpectrum()`: Cepstral smoothed version of `dftSpectrum()`
- `dftSpectrum()`: Short-term DFT spectral analysis
- `forest()`: Formant estimation
- `ksvF0()`: F0 analysis of the signal
- `lpsSpectrum()`: Linear predictive smoothed version of `dftSpectrum()`
- `mhsF0()`: Pitch analysis of the speech signal using Michel Scheffers' Modified Harmonic Sieve algorithm

[1] Some examples of this chapter are adapted version of examples given in the `wrassp_intro` vignette of the `wrassp` package.

- `rfcana()`: Linear prediction analysis
- `rmsana()`: Analysis of short-term Root Mean Square amplitude
- `zcrana`(): Analysis of the averages of the short-term positive and negative zero-crossing rates

The available file handling functions are:

- `read.AsspDataObj()`: read a SSFF or audio file into an `AsspDataObj`, which is the in-memory equivalent of the SSFF or audio file.
- `write.AsspDataObj()`: write an `AsspDataObj` to file (usually SSFF or audio file formats).

See R's `help()` function for a comprehensive list of every function and object provided by the `wrassp` package is required (see R Example **??**).

```
help(package="wrassp")
```

As the `wrassp` package can be used independently of the EMU-SDMS this chapter largely focuses on using it as an independent component. However, Section **??** provides an overview of how the package is integrated into the EMU-SDMS. Further, although the `wrassp` package has its own set of example audio files (which can be accessed in the directory provided by `system.file('extdata', package='wrassp')`), this chapter will use the audio and SSFF files that are part of the `ae emuDB` of the demo data provided by the `emuR` package. This is done primarily to provide an overview of what it is like using `wrassp` to work on files in an `emuDB`. R Example **??** shows how to generate this demo data followed by a listing of the files contained in a directory of a single bundle called `msajc003` (see Chapter @ref(chap:emuDB} for information about the `emuDB` format). The output of the call to `list.files()` shows four files where the `.dft` and `.fms` files are in the SSFF file format (see Appendix **??** for further details). The `_annot.json` file contains the annotation information, and the `.wav` file is one of the audio files that will be used in various signal processing examples in this chapter.

```
# load the emuR package
library(emuR)

# create demo data in directory
# provided by tempdir()
create_emuRdemoData(dir = tempdir())

# create path to demo database
path2ae = file.path(tempdir(), "emuR_demoData", "ae_emuDB")

# create path to bundle in database
path2bndl = file.path(path2ae, "0000_ses", "msajc003_bndl")

# list files in bundle directory
list.files(path2bndl)
```

```
## [1] "msajc003_annot.json" "msajc003.dft"        "msajc003.fms"
## [4] "msajc003.wav"
```

8.2 File I/0 and the `AsspDataObj`

One of the aims of `wrassp` is to provide mechanisms for handling speech-related files such as audio files and derived and complementary signal files. To have an in-memory object that can hold these file types in a uniform way the `wrassp` package provides the `AsspDataObj` data type. R Example **??** shows how the `read.AsspDataObj()` can be used to import a `.wav` audio file.

```
# load the wrassp package
library(wrassp)
```

```
# create path to wav file
path2wav = file.path(path2bndl, "msajc003.wav")

# read audio file
au = read.AsspDataObj(path2wav)

# show class
class(au)
```

```
## [1] "AsspDataObj"
```

```
# show print() output of object
print(au)
```

```
## Assp Data Object of file /var/folders/yk/8z9tn7kx6hbcg_9n4c1sld980000gn/T//RtmpXrUeDA/emuR_demoData/a
## Format: WAVE (binary)
## 58089 records at 20000 Hz
## Duration: 2.904450 s
## Number of tracks: 1
##   audio (1 fields)
```

As can be seen in R Example **??**, the resulting `au` object is of the class `AsspDataObj`. The output of `print` provides additional information about the object, such as its sampling rate, duration, data type and data structure information. Since the file we loaded is audio only, the object contains exactly one track. Further, since it is a mono file, this track only has a single field. We will later encounter different types of data with more than one track and multiple fields per track. R Example **??** shows function calls that extract the various attributes from the object (e.g., duration, sampling rate and the number of records).

```
# show duration
dur.AsspDataObj(au)
```

```
## [1] 2.90445
```

```
# show sampling rate
rate.AsspDataObj(au)
```

```
## [1] 20000
```

```
# show number of records/samples
numRecs.AsspDataObj(au)
```

```
## [1] 58089
```

```
# shorten filePath attribute
# to 10 chars only to prettify output
attr(au, "filePath")  = paste0(substr(attr(au, "filePath"),
                                      start = 1,
                                      stop = 45), "...")
# show additional attributes
attributes(au)
```

```
## $names
## [1] "audio"
##
## $trackFormats
## [1] "INT16"
##
## $sampleRate
```

```
## [1] 20000
##
## $filePath
## [1] "/var/folders/yk/8z9tn7kx6hbcg_9n4c1sld980000g..."
##
## $origFreq
## [1] 0
##
## $startTime
## [1] 0
##
## $startRecord
## [1] 1
##
## $endRecord
## [1] 58089
##
## $class
## [1] "AsspDataObj"
##
## $fileInfo
## [1] 21  2
```

The sample values belonging to a trackdata objects tracks are also stored within an `AsspDataObj` object. As mentioned above, the currently loaded object contains a single mono audio track. Accessing the data belonging to this track, in the form of a matrix, can be achieved using the track's name in combination with the `$` notation known from R's common named `list` object. Each matrix has the same number of rows as the track has records and as many columns as the track has fields. R Example **??** shows how the `audio` track can be accessed.

```
# show track names
tracks.AsspDataObj(au)
```

```
## [1] "audio"
```

```
# or an alternative way to show track names
names(au)
```

```
## [1] "audio"
```

```
# show dimensions of audio attribute
dim(au$audio)
```

```
## [1] 58089     1
```

```
# show first sample value of audio attribute
head(au$audio, n = 1)
```

```
##      [,1]
## [1,]   64
```

This data can, for example, be used to generate an oscillogram of the audio file as shown in R Example **??**, which produces Figure 8.1.

```
# calculate sample time of every 10th sample
samplesIdx = seq(0, numRecs.AsspDataObj(au) - 1, 10)
samplesTime = samplesIdx / rate.AsspDataObj(au)

# extract every 10th sample using window() function
```

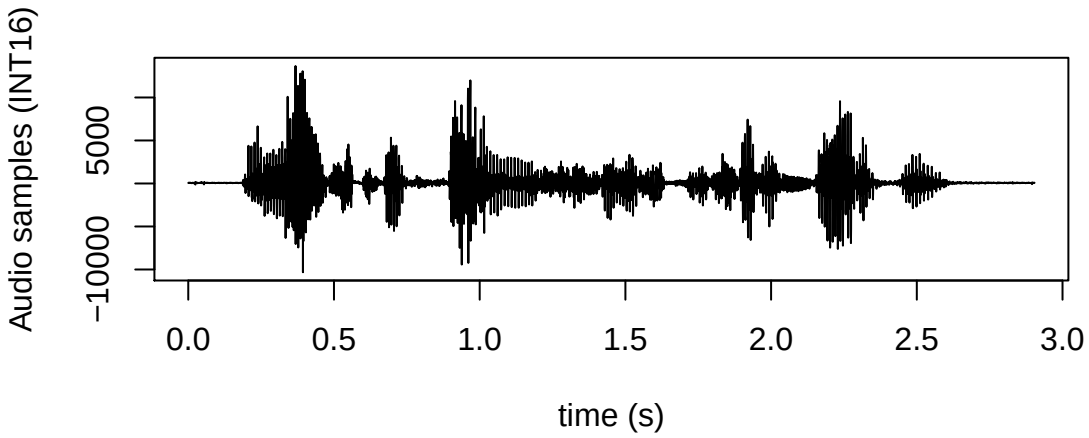


Figure 8.1: Oscillogram generated from samples stored in the `audio` track of the object `au`.

```
samples = window(au$audio, deltat=10)

# plot samples stored in audio attribute
# (only plot every 10th sample to accelerate plotting)
plot(samplesTime,
     samples,
     type = "l",
     xlab = "time (s)",
     ylab = "Audio samples (INT16)")
```

The export counterpart to `read.AsspDataObj()` function is `write.AsspDataObj()`. It is used to store in-memory `AsspDataObj` objects to disk and is particularly useful for converting other formats to or storing data in the SSFF file format as described in Section **??**. To show how this function can be used to write a slightly altered version of the `au` object to a file, R Example **??** initially multiplies all the sample values of `au$audio` by a factor of `0.5`. The resulting `AsspDataObj` is then written to an audio file in a temporary directory provided by `R`'s `tempdir()` function.

```
# manipulate the audio samples
au$audio = au$audio * 0.5
# write to file in directory
# provided by tempdir()
write.AsspDataObj(au, file.path(tempdir(), 'newau.wav'))
```

8.3 Signal processing

As mentioned in the introduction to this chapter, the `wrassp` package is capable of more than just the mere importing and exporting of specific signal file formats. This section will focus on demonstrating three of `wrassp`'s signal processing functions that calculate formant values, their corresponding bandwidths, the fundamental frequency contour and the RMS energy contour. Section 8.5 and **??** demonstrates signal processing to the audio file saved under `path2wav`, while Section 8.5.2 adresses processing all the audio files belonging to the *ae* `emuDB`.

8.4 The wrasspOutputInfos object

The wrassp package comes with the wrasspOutputInfos object, which provides information about the various signal processing functions provided by the package. The wrasspOutputInfos object stores meta information associated with the different signal processing functions wrassp provides. R Example ?? shows the names of the wrasspOutputInfos object which correspond to the function names listed in the introduction of this chapter.

```
# show all function names
names(wrasspOutputInfos)
```

```
##  [1] "acfana"      "afdiff"      "affilter"    "cepstrum"    "cssSpectrum"
##  [6] "dftSpectrum" "ksvF0"       "mhsF0"       "forest"      "lpsSpectrum"
## [11] "rfcana"      "rmsana"      "zcrana"
```

This object can be useful to get additional information about a specific wrassp function. It contains information about the default file extension ($ext), the tracks produced ($tracks) and the output file type ($outputType). R Example ?? shows this information for the forest() function.

```
# show output info of forest function
wrasspOutputInfos$forest
```

```
## $ext
## [1] "fms"
##
## $tracks
## [1] "fm" "bw"
##
## $outputType
## [1] "SSFF"
```

The examples that follow will make use of this wrasspOutputInfos object mainly to acquire the default file extensions given by a specific wrassp signal processing function.

8.5 Formants and their bandwidths

The already mentioned forest() is wrassp's formant estimation function. The default behavior of this formant tracker is to calculate the first four formants and their bandwidths. R Example ?? shows the usage of this function. As the default behavior of every signal processing function provided by wrassp is to store its result to a file, the toFile parameter of forest() is set to FALSE to prevent this behavior. This results in the same AsspDataObj object as when exporting the result to file and then importing the file into R using read.AsspDataObj(), but circumvents the disk reading/writing overhead.

```
# calculate formants and corresponding bandwidth values
fmBwVals = forest(path2wav, toFile=F)

# show class vector
class(fmBwVals)
```

```
## [1] "AsspDataObj"
```

```
# show track names
tracks.AsspDataObj(fmBwVals)
```

```
## [1] "fm" "bw"
```

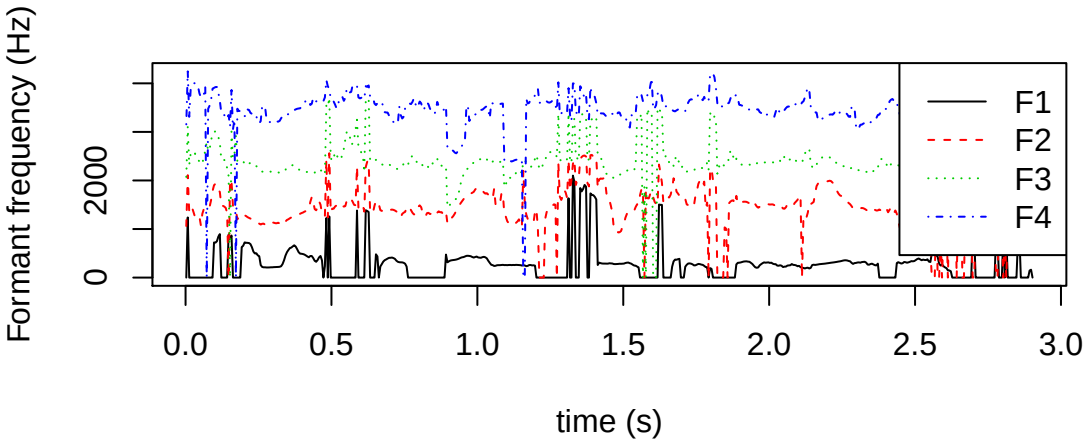


Figure 8.2: Matrix plot of formant values stored in the `fm` track of `fmBwVals` object.

```
# show dimensions of "fm" track
dim(fmBwVals$fm)
```

```
## [1] 581   4
```

```
# check dimensions of tracks are the same
all(dim(fmBwVals$fm) == dim(fmBwVals$bw))
```

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

As can be seen in R Example **??**, the object resulting from the `forest()` function is an object of class `AsspDataObj` with the tracks `"fm"` (formants) and `"bw"` (formant bandwidths), where both track matrices have four columns (corresponding to F1, F2, F3 and F4 in the `"fm"` track and $F1_{bandwidth}$, $F2_{bandwidth}$, $F3_{bandwidth}$ and $F4_{bandwidth}$ in the `"bw"` track) and 581 rows. To visualize the calculated formant values, R Example **??** shows how R's `matplot()` function can be used to produce Figure 8.2.

```
# plot the formant values
matplot(seq(0, numRecs.AsspDataObj(fmBwVals) - 1)
        / rate.AsspDataObj(fmBwVals)
        + attr(fmBwVals, "startTime"),
        fmBwVals$fm,
        type = "l",
        xlab = "time (s)",
        ylab = "Formant frequency (Hz)")

# add legend
startFormant = 1
endFormant = 4
legend("topright",
       legend = paste0("F", startFormant:endFormant),
       col = startFormant:endFormant,
       lty = startFormant:endFormant,
       bg = "white")
```

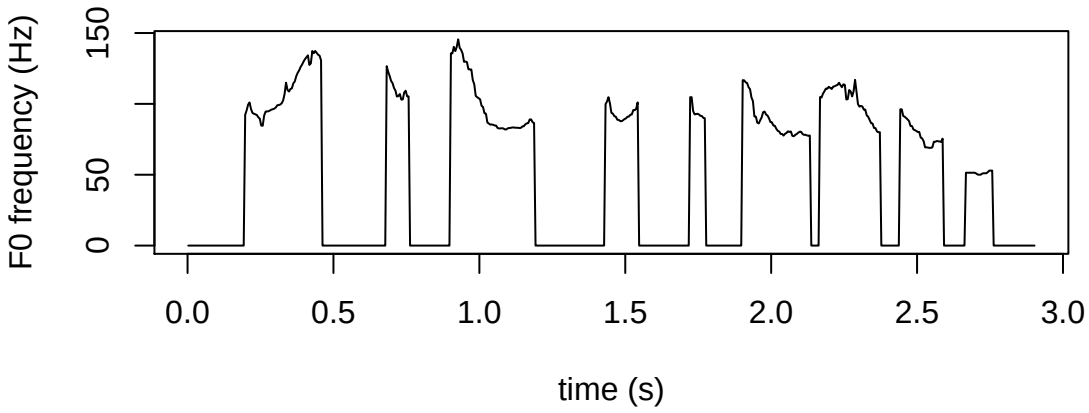


Figure 8.3: Plot of fundamental frequency values stored in the F0 track of f0vals object.

8.5.1 Fundamental frequency contour

The wrassp package includes two fundamental frequency estimation functions called ksvF0() and mhsF0(). R Example ?? shows the usage of the ksvF0() function, this time not utilizing the toFile parameter but rather to show an alternative procedure, reading the resulting SSFF file produced by it. It is worth noting that every signal processing function provided by wrassp creates a result file in the same directory as the audio file it was processing (except if the outputDirectory parameter is set otherwise). The default extension given by the ksvF0() is stored in wrasspOutputInfos\$ksvF0\$ext, which is used in R Example ?? to create the newly generated file's path.

```
# calculate the fundamental frequency contour
ksvF0(path2wav)

# create path to newly generated file
path2f0file = file.path(path2bndl,
                    paste0("msajc003.",
                           wrasspOutputInfos$ksvF0$ext))

# read file from disk
f0vals = read.AsspDataObj(path2f0file)
```

By analogy with to the formant estimation example, R Example ?? shows how the plot() function can be used to visualize this data as in Figure 8.3.

```
# plot the fundamental frequency contour
plot(seq(0,numRecs.AsspDataObj(f0vals) - 1)
     / rate.AsspDataObj(f0vals) +
       attr(f0vals, "startTime"),
     f0vals$F0,
     type = "l",
     xlab = "time (s)",
     ylab = "F0 frequency (Hz)")
```

8.5.2 RMS energy contour

The wrassp function for calculating the short-term root mean square (RMS) amplitude of the signal is called rmsana(). As its usage is analogous to the above examples, here we will focus on using it to calculate the RMS values for all the audio files of the ae emuDB. R Example ?? initially uses the

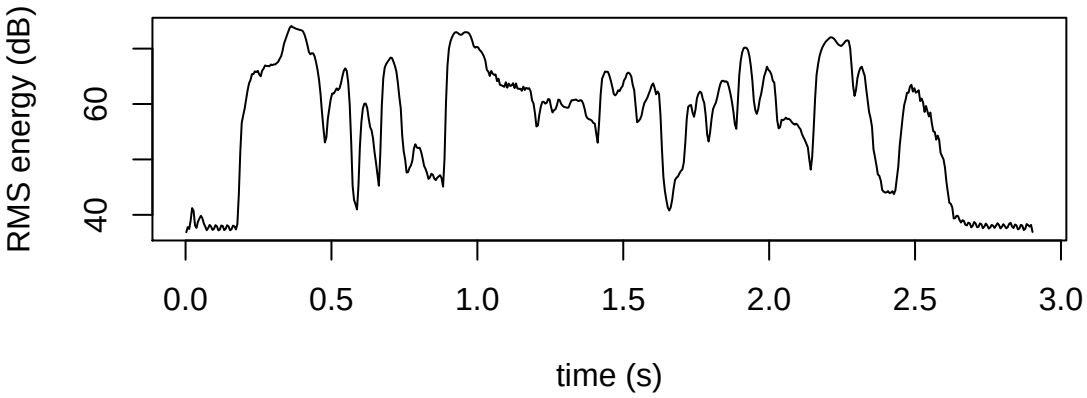


Figure 8.4: Plot of RMS values stored in `rms` track of the `rmsvals` object.

`list.files()` function to aquire the file paths for every `.wav` file in the `ae emuDB`. As every signal processing function accepts one or multiple file paths, these file paths can simply be passed in as the main argument to the `rmsana()` function. As all of `wrassp`'s signal processing functions place their generated files in the same directory as the audio file they process, the `rmsana()` function will automatically place every `.rms` into the correct bundle directory.

```
# list all .wav files in the ae emuDB
paths2wavFiles = list.files(path2ae, pattern = ".*wav$",
                            recursive = TRUE, full.names = TRUE)

# calculate the RMS energy values for all .wav files
rmsana(paths2wavFiles)

# list new .rms files using
# wrasspOutputInfos->rmsana->ext
rmsFPs = list.files(path2ae,
                    pattern = paste0(".*",
                                     wrasspOutputInfos$rmsana$ext),
                    recursive = TRUE,
                    full.names = TRUE)

# read first RMS file
rmsvals = read.AsspDataObj(rmsFPs[1])
```

R Example **??** shows how the `plot()` function can be used to visualize this data as in Figure 8.4.

```
# plot the RMS energy contour
plot(seq(0, numRecs.AsspDataObj(rmsvals) - 1)
     / rate.AsspDataObj(rmsvals)
     + attr(rmsvals, "startTime"),
     rmsvals$rms,
     type = "l",
     xlab = "time (s)",
     ylab = "RMS energy (dB)")
```

8.6 Logging wrassp's function calls

As it can be extremely important to keep track of information about how certain files are created and calculated, every signal processing function provided by the wrassp package comes with the ability to log its function calls to a specified log file. R Example @ref(rexample:wrassp-logging} shows a call to the ksvF0() function where a single parameter was changed from its default value (windowShift = 10). The content of the created log files (shown by the call to readLines()) contains the function name, time stamp, parameters that were altered and processed file path information. It is worth noting that a log file can be reused for multiple function calls as the log function does not overwrite an existing file but merely appends new log information to it.

```
# create path to log file in root dir of ae emuDB
path2logFile = file.path(path2ae, "wrassp.log")

# calculate the fundamental frequency contour
ksvF0(path2wav, windowShift = 10, forceToLog = T, optLogFilePath = path2logFile)
```

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

```
# display content of log file (first 8 lines)
readLines(path2logFile)[1:8]
```

```
## [1] ""
## [2] "###################################"
## [3] "###################################"
## [4] "######## ksvF0 performed ########"
## [5] "Timestamp:  2018-05-15 18:28:32 "
## [6] "windowShift : 10 "
## [7] "forceToLog : T "
## [8] " => on files:"
```

8.7 Using wrassp in the EMU-SDMS

As shown in Section 8.5.2, the wrassp signal processing functions can be used to calculate SSFF files and place them into the appropriate bundle directories. The only thing that has to be done to make an emuDB aware of these files is to add an SSFF track definition to the emuDB as shown in R Example ??. Once added, this SSFF track can be referenced via the ssffTrackName parameter of the get_trackdata() function as shown in various examples throughout this documentation. It is worth noting that this strategy is not necessarily relevant for applying the same signal processing to an entire emuDB, as this can be achieved using the on-the-fly add_ssffTrackDefinition() method described in R Example ??. However, it becomes necessary if certain bundles are to be processed using deviating function parameters. This can, for example, be relevant when setting the minimum and maximum frequencies that are to be considered while estimating the fundamental frequencies (e.g., the maxF and minF of ksvfF0()) for female versus male speakers.

```
# load emuDB
ae = load_emuDB(path2ae)

# add SSFF track defintion
# that references the .rms files
# calculated above
# (i.e. no new files are calculated and added to the emuDB)
ext = wrasspOutputInfos$rmsana$ext
colName = wrasspOutputInfos$rmsana$tracks[1]
```

```
add_ssffTrackDefinition(ae,
                        name = "rms",
                        fileExtension = ext,
                        columnName = colName)
```

A further way to utilize wrassp's signal processing functions as part of the EMU-SDMS is via the onTheFlyFunctionName and onTheFlyParams parameters of the add_ssffTrackDefinition() and get_trackdata() functions. Using the onTheFlyFunctionName parameter in the add_ssffTrackDefinition() function automatically calculates the SSFF files while also adding the SSFF track definition. Using this parameter with the get_trackdata() function calls the given wrassp function with the toFile parameter set to FALSE and extracts the matching segments and places them in the resulting trackdata or emuRtrackdata object. In many cases, this avoids the necessity of having SSFF track definitions in the emuDB. In both functions, the optional onTheFlyParams parameter can be used to specify the parameters that are passed into the signal processing function. R Example ?? shows how R's formals() function can be used to get all the parameters of wrassp's short-term positive and negative zero-crossing rate (ZCR) analysis function zrcana(). It then changes the default window size parameter to a new value and passes the parameters object into the add_ssffTrackDefinition() and get_trackdata() functions.

```
# get all parameters of zcrana
zcranaParams = formals("zcrana")

# show names of parameters
names(zcranaParams)
```

```
##  [1] "listOfFiles"     "optLogFilePath"  "beginTime"
##  [4] "centerTime"      "endTime"         "windowShift"
##  [7] "windowSize"      "toFile"          "explicitExt"
## [10] "outputDirectory" "forceToLog"      "verbose"
```

```
# change window size from the default
# value of 25 ms to 50 ms
zcranaParams$windowSize = 50

# to have a segment list to work with
# query all Phonetic 'n' segments
sl = query(ae, "Phonetic == n")

# get trackdata calculating ZCR values on-the-fly
# using the above parameters. Note that no files
# are generated.
td = get_trackdata(ae, sl,
                   onTheFlyFunctionName = "zcrana",
                   onTheFlyParams = zcranaParams,
                   verbose = FALSE)


# add SSFF track definition. Note that
# this time files are generated.
add_ssffTrackDefinition(ae,
                        name = "zcr",
                        onTheFlyFunctionName = "zcrana",
                        onTheFlyParams = zcranaParams,
                        verbose = FALSE)
```

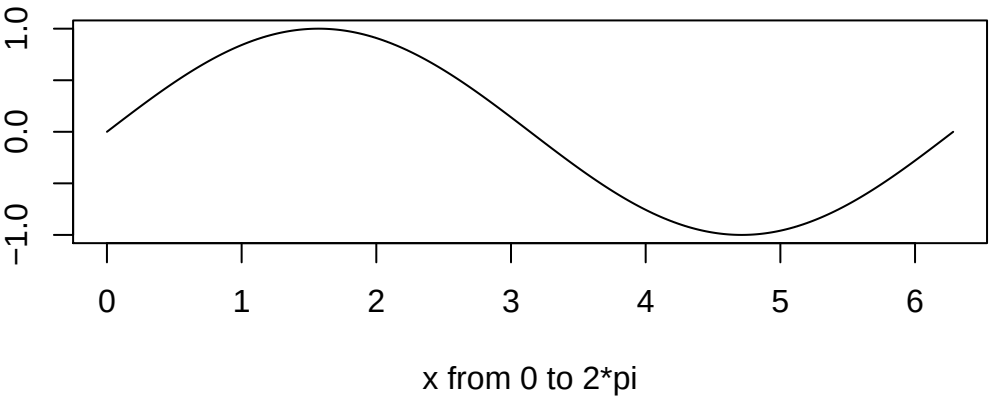


Figure 8.5: A single cycle sine wave consisting of 16000 samples.

8.8 Storing data in the SSFF file format

One of the benefits gained by having the `AsspDataObj` in-memory object is that these objects can be constructed from scratch in R, as they are basically simple `list` objects. This means, for example, that any set of n-dimensional samples over time can be placed in a `AsspDataObj` and then stored as an SSFF file using the `write.AsspDataObj()` function. To show how this can be done, R Example **??** creates an arbitrary data sample in the form of a single cycle sine wave between 0 and $2 * pi$ that is made up of 16000 samples and displays it in Figure 8.5.

```
x = seq(0, 2 * pi, length.out = 16000)
sineWave = sin(x)
plot(x, sineWave, type = 'l',
     xlab = "x from 0 to 2*pi",
     ylab = "")
```

Assuming a sample rate of 16 kHz `sineWave` would result in a sine wave with a frequency of 1 Hz and a duration of one second. R Example **??** shows how a `AsspDataObj` can be created from scratch and the data in `sineWave` placed into one of its tracks. It then goes on to write the `AsspDataObj` object to an SSFF file.

```
# create empty list object
ado = list()

# add sample rate attribute
attr(ado, "sampleRate") = 16000

# add start time attribute
attr(ado, "startTime") = 0

# add start record attribute
attr(ado, "startRecord") = as.integer(1)

# add end record attribute
attr(ado, "endRecord") = as.integer(length(sineWave))

# set class of ado
class(ado) = "AsspDataObj"

# show available file formats
AsspFileFormats
```

```
##     RAW   ASP_A   ASP_B   XASSP  IPDS_M  IPDS_S    AIFF    AIFC     CSL
##       1       2       3       4       5       6       7       8       9
##    CSRE    ESPS     ILS     KTH   SWELL   SNACK     SFS     SND      AU
##      10      11      12      13      13      13      14      15      15
##    NIST  SPHERE PRAAT_S PRAAT_L PRAAT_B    SSFF    WAVE  WAVE_X  XLABEL
##      16      16      17      18      19      20      21      22      24
##    YORK     UWM
##      25      26
```

```
# set file format to SSFF
# NOTE: assignment of "SSFF" also possible
AsspFileFormat(ado) = as.integer(20)

# set data format (1 == 'ascii' and 2 == 'binary')
AsspDataFormat(ado) = as.integer(2)

# set track format specifiers
# (available track formats for numbers
# that match their C equivalent are:
# "UINT8"; "INT8"; "UINT16"; "INT16";
# "UINT24"; "INT24"; "UINT32"; "INT32";
# "UINT64"; "INT64"; "REAL32"; "REAL64");
attr(ado, "trackFormats") = c("REAL32")

# add track
ado = addTrack(ado, "sine", sineWave, "REAL32")

# write AsspDataObj object to file
write.AsspDataObj(dobj = ado,
                  file = file.path(tempdir(), "example.sine"))
```

```
## NULL
```

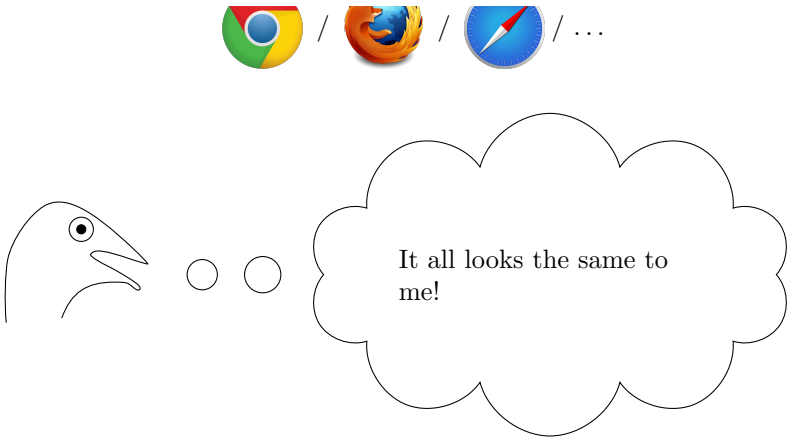
Although somewhat of a generic example, R Example **??** shows how to generate an `AsspDataObj` from scratch. This approach can, for example, be used to read in signal data produced by other software or signal data acquisition devices. Hence, this approach can be used to import many forms of data into the EMU-SDMS. Appendix @ref(sec:app-chap-wrassp-praatsSigProc} shows an example of how this approach can be used to take advantage of Praat's signal processing capabilities and integrate its output into the EMU-SDMS.

8.9 Conclusion

The `wrassp` packages enriches the R package landscape by providing functionality for handling speech signal files in most common audio formats and for performing signal analyses common in the phonetic and speech sciences. The EMU-SDMS utilizes the functionality that the `wrassp` package provides by allowing the user to calculate signals that match the segments of a segment list. This can either be done in real time or by extracting the signals from files. Hence, the `wrassp` package is an integral part of the EMU-SDMS but can also be used as a standalone package if so desired.

Chapter 9

The EMU-webApp [1]



The EMU-SDMS has a unique approach to its GUI in that it utilizes a web application as its primary GUI. This is known as the EMU-webApp (Winkelmann and Raess, 2015). The EMU-webApp is a fully fledged browser-based labeling and correction tool that offers a multitude of labeling and visualization features. These features include unlimited undo/redo, formant correction capabilities, the ability to snap a preselected boundary to the nearest top/bottom boundary, snap a preselected boundary to the nearest zero crossing, and many more. The web application is able to render everything directly in the user's browser, including the calculation and rendering of the spectrogram, as it is written entirely using HTML, CSS and JavaScript. This means it can also be used as a standalone labeling application, as it does not require any server-side calculations or rendering. Further, it is designed to interact with any websocket server that implements the EMU-webApp websocket protocol (see Section 13.1). This enables it to be used as a labeling tool for collaborative annotation efforts. Also, as the EMU-webApp is cached in the user's browser on the first visit, it does not require any internet connectivity to be able to access the web application unless the user explicitly clears the browser's cache. The URL of the current live version of the EMU-webApp is: http://ips-lmu.github.io/EMU-webApp/.

9.1 Main layout

The main screen of the EMU-webApp can be split into five areas. Figure 9.1 shows a screenshot of the EMU-webApp's main screen displaying these five areas while displaying a bundle of the *ae* demo database.

[1] Sections of this chapter have been published in (Winkelmann and Raess, 2015) and some descriptions where taken from the EMU-webApp's own manual.

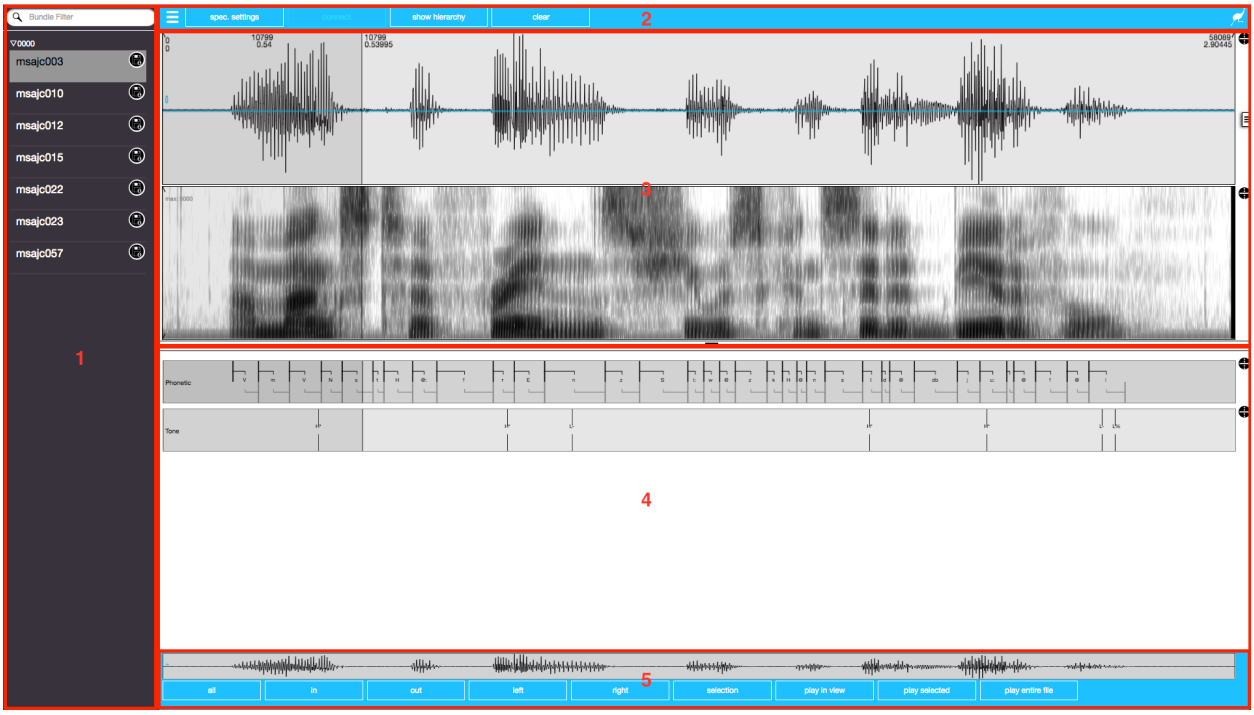


Figure 9.1: Screenshot of 'EMU-webApp' displaying the *ae* demo database with overlaid areas of the main screen of the web application (see text).

This database is served to the `EMU-webApp` by invoking the `serve()` command as shown in R Example **??**. The left side bar (area marked 1 in Figure 9.1) represents the bundle list side bar which, if connected to a database, displays the currently available bundles grouped by their sessions. The top and bottom menu bars (areas marked 2 and 5 in Figure 9.1) display the currently available menu options, where the bottom menu bar contains the audio navigation and playback controls and also includes a scrollable mini map of the oscillogram. Area 3 of Figure 9.1 displays the signal canvas area currently displaying the oscillogram and the spectrogram. Other signal contours such as formant frequency contours and fundamental frequency contours are also displayed in this area. Area 4 of Figure 9.1 displays the area in which levels containing time information are displayed. It is worth noting that the main screen of the `EMU-webApp` does not display any levels that do not contain time information. The hierarchical annotation can be displayed and edited by clicking the `show hierarchy` button in the top menu bar (see Figure 9.6 for an example of how the hierarchy is displayed).

```
# serve ae emuDB to EMU-webApp
serve(ae)
```

9.2 General usage

This section introduces the labeling mechanics and general labeling workflow of the `EMU-webApp`. The `EMU-webApp` makes heavy use of keyboard shortcuts. Is is worth noting that most of the keyboard shortcuts are centered around the `WASD` keys, which are the navigation shortcut keys (`W` to zoom in; `S` to zoom out; `A` to move left and `D` to move right). For a full list of the available keyboard shortcuts see the `EMU-webApp`'s own manual, which can be accessed by clicking the EMU icon on the right hand side of the top menu bar (area 2 in Figure 9.1).

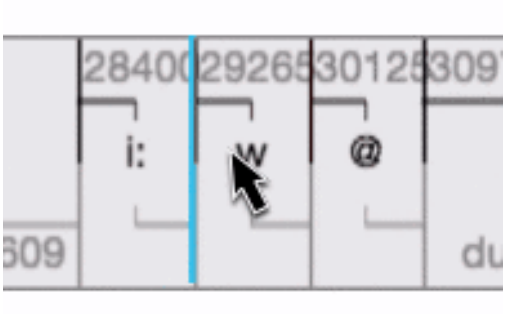


Figure 9.2: Screenshot of segment level as displayed by the 'EMU-webApp' with superimposed mouse cursor displaying the automatic boundary preselection of closest boundary (boundary marked blue).

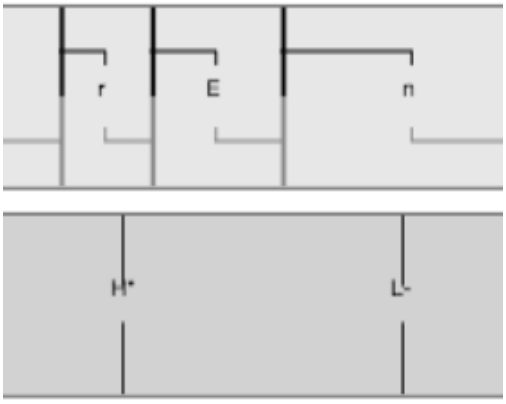


Figure 9.3: Screenshot of two levels as displayed by the 'EMU-webApp', where the lower level is preselected (i.e., marked in a darker shade of gray).

9.2.1 Annotating levels containing time information

9.2.1.1 Boundaries and events

The `EMU-webApp` has slightly different labeling mechanics compared with other annotation software. Compared to the usual click and drag of segment boundaries and event markers, the web application continuously tracks the movement of the mouse in levels containing time information, highlighting the boundary or event marker that is closest to it by coloring it blue. Figure 9.2 displays this automatic boundary preselection.

Once a boundary or event is preselected, the user can perform various actions with it. She or he can, for example, grab a preselected boundary or event by holding down the `SHIFT` key and moving it to the desired position, or delete the current boundary or event by hitting the `BACKSPACE` key. Other actions that can be performed on preselected boundaries or events are:

- snap to closest boundary or event in level above (Keyboard Shortcut `t`),
- snap to closest boundary or event in level below (Keyboard Shortcut `b`), and
- snap to nearest zero crossing (Keyboard Shortcut `x`).

To add a new boundary or event to a level the user initially has to select the desired level she or he wishes to edit. This is achieved either by using the up and down cursor keys or by single-left-clicking on the desired level. The current preselected level is marked in a darker shade of gray, as is displayed in Figure 9.3.

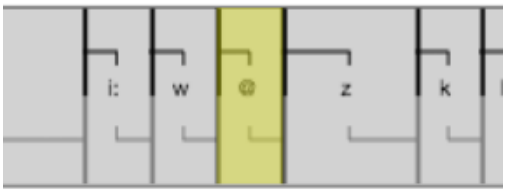


Figure 9.4: Screenshot of level as displayed by the 'EMU-webApp', where the /@/ segment is currently preselected as it is marked yellow.

Figure 9.5: Screenshot of segment level with three attribute definitions. The radio buttons that switch between the parallel labels are highlighted by a red square.

To add a boundary to the currently selected level one first has to select a point in time either in the spectrogram or the oscillogram by single-left-clicking on the desired location. Hitting the enter/return key adds a new boundary or event to the preselected level at the selected time point. Selecting a stretch of time in the spectrogram or the oscillogram (left-click-and-drag) and hitting enter will add a segment (not a boundary) to a preselected segment level.

9.2.1.2 Segments and events

The `EMU-webApp` also allows segments and events to be preselected by single-left-clicking the desired item. The web application colors the preselected segments and events yellow to indicate their pre-selection as displayed in Figure 9.4.

As with preselected boundaries or events the user can now perform multiple actions with these preselected items. She or he can, for example, edit the item's label by hitting the enter/return key (which can also be achieved by double-left-clicking the item). Other actions that can be performed on preselected items are:

- Select next item in level (keyboard shortcut `TAB`),
- Select previous item in level (keyboard shortcut `SHIFT` plus `TAB`),
- Add time to selected item(s) end (keyboard shortcut `+`),
- Add time to selected item(s) start (keyboard shortcut `SHIFT` plus `+`),
- Remove time to selected item(s) end (keyboard shortcut `-`),
- Remove time to selected item(s) start (keyboard shortcut `SHIFT` plus `-`), and
- Move selected item(s) (hold down `ALT` Key and drag to desired position).

By right-clicking adjacent segment or events (keyboard shortcut `SHIFT` plus left or right cursor keys), it is possible to select multiple items at once.

9.2.1.3 Parallel labels in segments and events

If a level containing time information has multiple attribute definitions (i.e., multiple parallel labels per segment or event) the `EMU-webApp` automatically displays radio buttons underneath that level (see red square in Figure 9.5) that allow the user to switch between the parallel labels. Figure 9.5 displays a segment level with three attribute definitions.

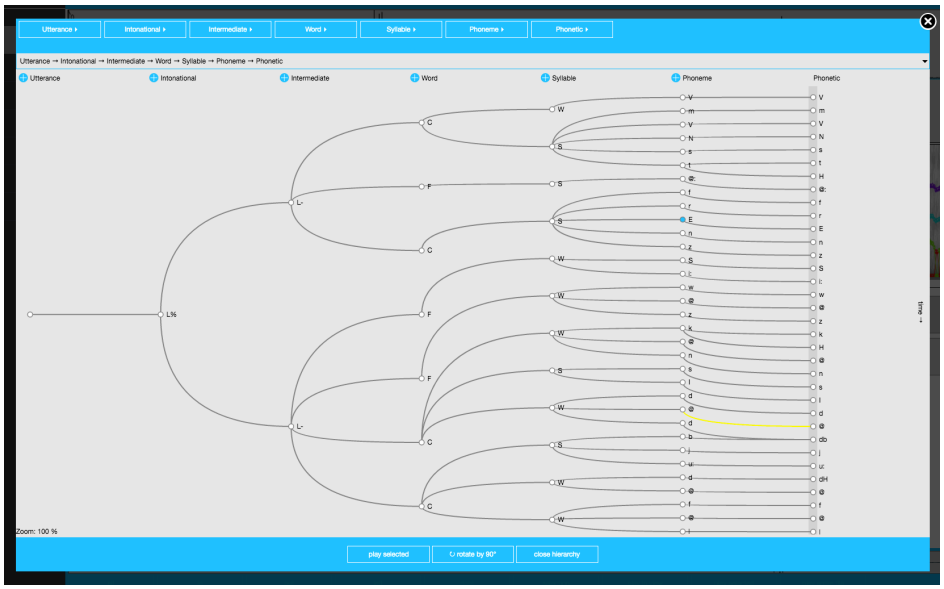


Figure 9.6: Screenshot of the hierarchy modal window level displaying a path through the hierarchy of the *ae* 'emuDB' in its horizontal form.

9.2.1.4 Legal labels

As mentioned in Section 5.2.3.2, an array of so-called legal labels can be defined for every level or, more specifically, for each attribute definition. The `EMU-webApp` enforces these legal labels by not allowing any other labels to be entered in the label editing text fields. If an illegal label is entered, the text field will turn red and the `EMU-webApp` will not permit this label to be saved.

9.2.2 Working with hierarchical annotations [2]

9.2.2.1 Viewing the hierarchy

As mentioned in Section 9.1, pressing the `show hierarchy` button (keyboard shortcut `h`) in the top menu bar opens the hierarchy view modal window[3]. As with most modal windows in the `EMU-webApp`, it can be closed by clicking on the `close` button, clicking the X circle icon in the top right hand corner of the modal or by hitting the `ESCAPE` key. By default, the hierarchy modal window displays a horizontal version of the hierarchy for a spatially economical visualization. As most people are more familiar with a vertical hierarchical annotation display, the hierarchy can be rotated by hitting the `rotate by 90°` button (keyboard shortcut `r`). Zooming in and out of the hierarchy can be achieved by using the mouse wheel, and moving through the hierarchy in time can be achieved by holding down the left mouse button and dragging the hierarchy in the desired direction. Figure 9.6 shows the hierarchy modal window displaying the hierarchical annotation of a single path (*Utterance* -> *Intonational* -> *Intermediate* -> *Word* -> *Syllable* -> *Phoneme* -> *Phonetic*) through a multi-path hierarchy of the *ae* `emuDB` in its horizontal form.

[2] This section is an updated version of the *The level hierarchy* section of the *General Usage* chapter that is part of the `EMU-webApp` own brief manual by Markus Jochim.

[3] The term modal window is used in user interface design to refer to pop-up windows that force the user to interact with the window before returning back to the main application.

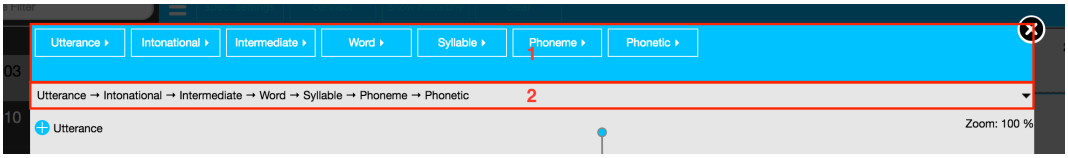


Figure 9.7: Screenshot of top of hierarchy modal window of the 'EMU-webApp' in which the area marked 1 shows the drop-down menus for selecting the parallel label for each level and area 2 marks the hierarchy path drop-down menu.

9.2.2.2 Selecting a path through the hierarchy

As more complex databases have multiple hierarchical paths through their hierarchical annotation structure (see Figure 4.2 for an example of a multi-dimensional hierarchical annotation structure), the hierarchy modal offers a drop-down menu to choose the current path to be displayed. Area 2 in Figure 9.7 marks the hierarchy path drop-down menu of the hierarchy modal.

It is worth noting that only non-partial paths can be selected in the hierarchy path drop-down menu.

9.2.2.3 Selecting parallel labels in timeless levels

As timeless levels may also contain multiple parallel labels, the hierarchy path modal window provides a drop-down menu for each level to select which label or attribute definition is to be displayed. Area 1 of Figure 9.7 displays these drop-down menus.

9.2.2.4 Adding a new item

The hierarchy modal window provides two methods for adding new annotation items to a level. This can either be achieved by pressing the blue and white + button next to the level's name (which appends a new item to the end of the level) or by preselecting an annotation item (by hovering the mouse over it) and hitting either the `n` (insert new item before preselected item) or the `m` key (insert new item after preselected item).

9.2.2.5 Modifying an annotation item

An item's context menu [4] is opened by single-left-clicking its node. The resulting context menu displays a text area in which the label of the annotation item can be edited, a play button to play the audio section associated with the item and a collapse arrow button allowing the user to collapse the sub-tree beneath the current item. Collapsing a sub-tree can be useful for masking parts of the hierarchy while editing. A screenshot of the context menu is displayed in Figure 9.8.

9.2.2.6 Adding a new link

Adding a new link between two items can be achieved by hovering the mouse over one of the two items, holding down the `SHIFT` key and moving the mouse cursor to the other item. A green dashed line indicates that the link to be added is valid, while a red dashed line indicates it is not. A link's validity is dependent on the database's configuration (i.e., if there is a link definition present and the type of link definition) as well as the *non-crossing constraint* (Coleman and Local, 1991) that essentially implies that links are not

[4] The term context menu is used in user interface design to refer to a pop-up menu or pop-up area that provides additional information for the current state (i.e., the current item).

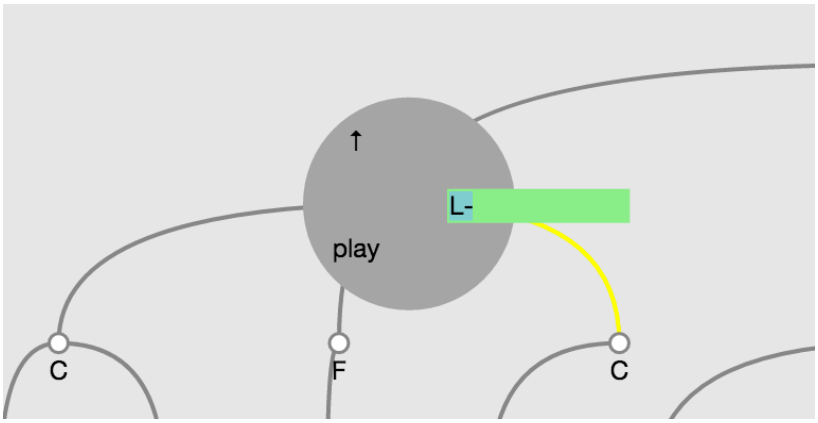


Figure 9.8: Screenshot of the hierarchy modal window of the 'EMU-webApp' displaying an annotation item's context menu.

allowed to cross each other. If the link is valid (i.e., a green dashed line is present), releasing the SHIFT key will add the link to the annotation.

9.2.2.7 Deleting an annotation item or a link

Items and links are deleted by initially preselecting them by hovering the mouse cursor over them. The preselected items are marked blue and preselected links yellow. A preselected link is removed by hitting BACKSPACE and a preselected item is deleted by hitting the y key. Deleting an item will also delete all links leading to and from it.

9.3 Configuring the EMU-webApp

This section will give an overview of how the EMU-webApp can be configured. The configuration of the EMU-webApp is stored in the EMUwebAppConfig section of the _DBconfig.json of an emuDB (see Appendix ?? for details). This means that the EMU-webApp can be configured separately for every emuDB. Although it can be necessary for some advanced configuration options to manually edit the _DBconfig.json using a text editor (see Section 9.3.3), the most common configuration operations can be achieved using functions provided by the emuR package (see Section 9.3.1).

A central concept for configuring the EMU-webApp are so-called perspectives. Essentially, a perspective is an independent configuration of how the EMU-webApp displays a certain set of data. Having multiple perspectives allows the user to switch between different views of the data. This can be especially useful when dealing with complex annotations where only showing certain elements for certain labeling tasks can be beneficial. Figure 9.9 displays a screenshot of the perspectives side bar menu of the EMU-webApp which displays the three perspectives of the *ae* emuDB. The *default* perspective displays both the *Phonetic* and the *Tone* levels where as the *Phonetic-only* and the *Tone-only* only display these levels individually.

9.3.1 Basic configurations using emuR

R Example ?? shows how to create and load the demo data that will be used throughout the rest of this chapter.

```
# load package
library(emuR)
```

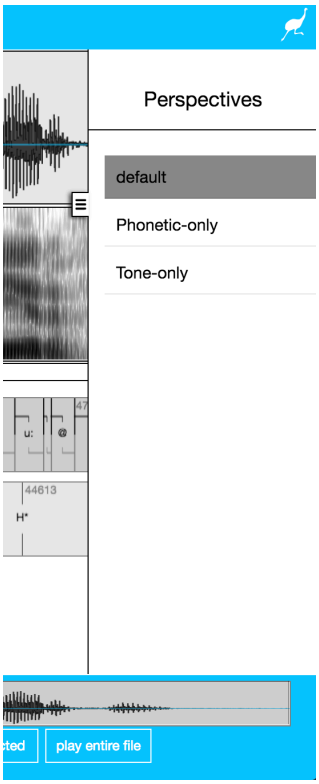


Figure 9.9: Screenshot of the hierarchy modal window of the 'EMU-webApp' displaying an annotation item's context menu.

```
# create demo data in directory provided by tempdir()
create_emuRdemoData(dir = tempdir())

# create path to demo database
path2ae = file.path(tempdir(), "emuR_demoData", "ae_emuDB")

# load database
ae = load_emuDB(path2ae, verbose = F)
```

As mentioned above, the EMU-webApp subdivides different ways to look at an emuDB into so-called perspectives. Users can switch between these perspectives in the web application. They contain, for example, information on what levels are displayed, which SSFF tracks are drawn. R Example ?? shows how the current perspectives can be listed using the list_perspectives() function.

```
# list perspectives of ae emuDB
list_perspectives(ae)
```

```
##            name signalCanvasesOrder levelCanvasesOrder
## 1       default          OSCI; SPEC     Phonetic; Tone
## 2 Phonetic-only          OSCI; SPEC           Phonetic
## 3     Tone-only          OSCI; SPEC               Tone
```

As it is sometimes necessary to add new or remove existing perspectives to or from a database, R Example ?? shows how this can be achieved using emuR's add/remove_perspective() functions.

```
# add new perspective to ae emuDB
add_perspective(ae,
                name = "tmpPersp")

# show added perspective
list_perspectives(ae)
```

```
##            name signalCanvasesOrder levelCanvasesOrder
## 1       default          OSCI; SPEC     Phonetic; Tone
## 2 Phonetic-only          OSCI; SPEC           Phonetic
## 3     Tone-only          OSCI; SPEC               Tone
## 4      tmpPersp          OSCI; SPEC
```

```
# remove newly added perspective
remove_perspective(ae,
                   name = "tmpPersp")
```

9.3.2 Signal canvas and level canvas order

As mentioned above, R Example ?? shows that the *ae* emuDB contains three perspectives. The first perspective (*default*) displays the oscillogram (OSCI) followed by the spectrogram (SPEC) in the signal canvas area (area 3 of Figure 9.1) and the *Phonetic* and *Tone* levels in the level canvas area (area 4 of Figure 9.1). It is worth noting that OSCI (oscillogram) and SPEC (spectrogram) are predefined signal tracks that are always available. This is indicated by the capital letters indicating that they are predefined constants. R Example ?? shows how the order of the signal canvases and level canvases can be changed using the get/set_signalCanvasesOrder() and get/set_levelCanvasesOrder().

```
# get order vector of signal canvases of default perspective
sco = get_signalCanvasesOrder(ae,
                              perspectiveName = "default")
```

```
# show sco vector
sco
```

```
## [1] "OSCI" "SPEC"
```

```
# reverse sco order
# using R's rev() function
scor = rev(sco)

# set order vector of signal canvases of default perspective
set_signalCanvasesOrder(ae,
                        perspectiveName = "default",
                        order = scor)

# set order vector of level canvases of default perspective
# to only display the "Tone" level
set_levelCanvasesOrder(ae,
                       perspectiveName = "default",
                       order = c("Tone"))

# list perspectives of ae emuDB
# to show changes
list_perspectives(ae)
```

```
##            name signalCanvasesOrder levelCanvasesOrder
## 1       default          SPEC; OSCI               Tone
## 2 Phonetic-only          OSCI; SPEC           Phonetic
## 3     Tone-only          OSCI; SPEC               Tone
```

After the changes made in R Example **??**, the default perspective will show the spectrogram above the oscillogram in the signal canvas area and only the *Tone* level in the level canvas area. Only levels with time information are allowed to be displayed in the level canvas area, and the `set_levelCanvasesOrder()` will print an error if a level of type `ITEM` is added (see R Example **??**).

```
# set level canvas order where a
# level is passed into the order parameter
# that is not of type EVENT or SEGMENT
set_levelCanvasesOrder(ae,
                       perspectiveName = "default",
                       order = c("Syllable"))
```

```
## Error in set_levelCanvasesOrder(ae, perspectiveName = "default", order = c("Syllable")): levelDefini
```

The same mechanism used above can also be used to display any SSFF track that is defined for the database by referencing its name. R Example **??** shows how the existing SSFF track called *fm* (containing formant values calculated by `wrassp`'s `forest()` function) can be added to the signal canvas area.

```
# show currently available SSFF tracks
list_ssffTrackDefinitions(ae)
```

```
##   name columnName fileExtension
## 1  dft        dft           dft
## 2   fm         fm           fms
```

```
# re-set order vector of signal canvases of default perspective
# by appending the fm track
set_signalCanvasesOrder(ae,
```

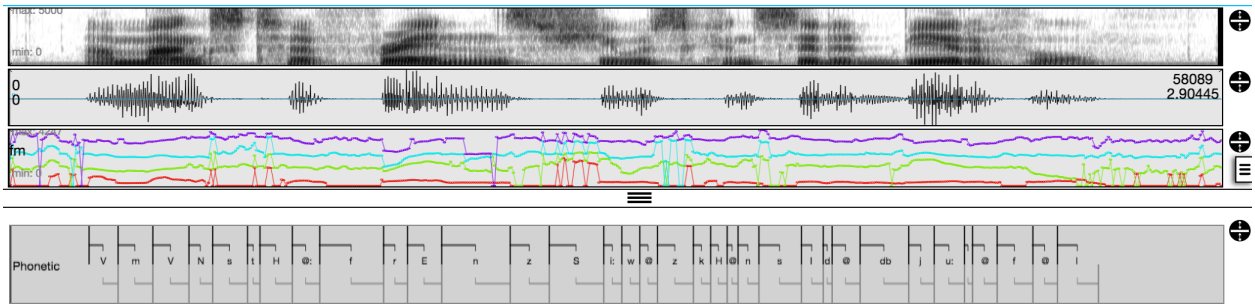


Figure 9.10: Screenshot of signal and level canvases displays of the 'EMU-webApp' after the changes made in R Examples ?? and ??.

```
                    perspectiveName = "default",
                    order = c(scor, "fm"))
```

A screenshot of the current display of the *default* perspective can be seen in Figure 9.10.

9.3.3 Advanced configurations made by editing the `_DBconfig.json`

Although the above configuration options cover the most common use cases, the `EMU-webApp` offers multiple other configuration options that are currently not configurable via functions provided by `emuR`. These advanced configuration options can currently only be achieved by manually editing the `_DBconfig.json` file using a text editor. As even the colors used in the `EMU-webApp` and every keyboard shortcut can be reconfigured, here we will focus on the more common advanced configuration options. A full list of the available configuration fields of the `EMUwebAppConfig` section of the `_DBconfig.json` including their meaning, can be found in Appendix ??.

9.3.3.1 Overlaying signal canvases

To save space it can be beneficial to overlay one or more signal tracks onto other signal canvases. This can be achieved by manually editing the `assign` array of the `EMUwebAppConfig:perspectives[persp_idx]:signalCanvases` field in the `_DBconfig.json`. Listing ?? shows an example configuration that overlays the *fm* track on the oscillogram where the `OSCI` string can be replaced by any other entry in the `EMUwebAppConfig:perspectives[persp_idx]:signalCanvases:order` array. Figure 9.11 displays a screenshot of such an overlay.

```
...
"assign": [{
    "signalCanvasName": "OSCI",
    "ssffTrackName": "fm"
}],
...
```

9.3.3.2 Frequency-aligned formant contours spectrogram overlay

The current mechanism for laying frequency-aligned formant contours over the spectrogram is to give the formant track the predefined name *FORMANTS*. If the formant track is called *FORMANTS* and it is assigned to be laid over the spectrogram (see Listing ??) the `EMU-webApp` will frequency-align the contours to the current minimum and maximum spectrogram frequencies (see Figure 9.12).

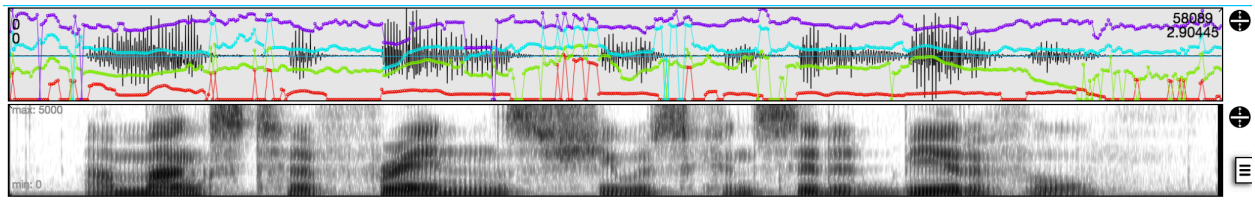


Figure 9.11: Screenshot of signal canvases display of the 'EMU-webApp' after the changes made in R Examples **??** and **??**.

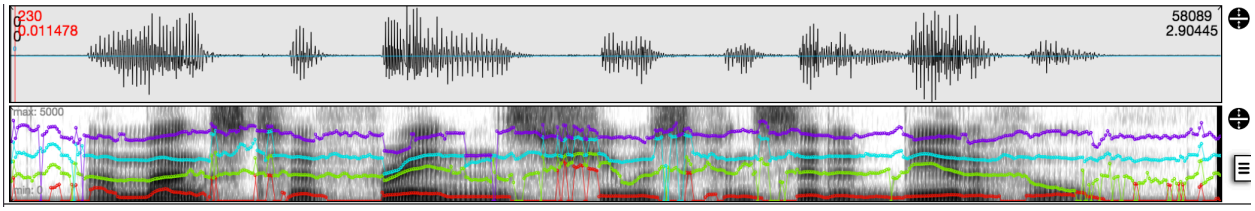


Figure 9.12: Screenshot of signal canvases area of the 'EMU-webApp' displaying formant contours that are overlaid on the spectrogram and frequency-aligned.

```
...
"assign": [{
    "signalCanvasName": "SPEC",
    "ssffTrackName": "FORMANTS"
}],
...
```

9.3.3.3 Correcting formants

The above configuration of the frequency-aligned formant contours will automatically allow the *FORMANTS* track to be manually corrected. Formants can be corrected by hitting the appropriate number key (`1` = first formant, `2` = second formant, …). Similar to boundaries and events, the mouse cursor will automatically be tracked in the `SPEC` canvas and the nearest formant value preselected. Holding down the `SHIFT` key moves the current formant value to the mouse position, hence allowing the contour to be redrawn and corrected.

9.3.4 2D canvas

The `EMU-webApp` has an additional canvas which can be configured to display two-dimensional data. Figure 9.13 shows a screenshot of the 2D canvas, which is placed in the bottom right hand corner of the level canvas area of the web application. The screenshot shows data representing EMA sensor positions on the mid sagittal plane. Listings **??** shows how the 2D canvas can be configured. Essentially, every drawn dot is configured by assigning a column in an SSFF track that specifies the X values and an additional column that specifies the Y values.

```
...
"twoDimCanvases": {
    "order": ["DOTS"],
    "twoDimDrawingDefinitions": [{
        "name": "DOTS",
        "dots": [{
            "name": "tt",
```

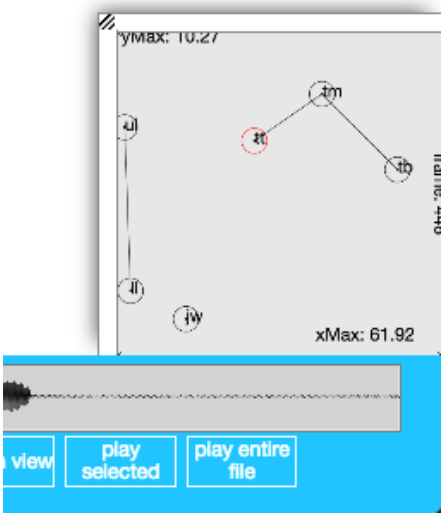


Figure 9.13: Screenshot of 2D canvas of the 'EMU-webApp' displaying two-dimensional EMA data.

```
            "xSsffTrack": "tt_posy",
            "xContourNr": 0,
            "ySsffTrack": "tt_posz",
            "yContourNr": 0,
            "color": "rgb(255,0,0)"
        },
...
    "connectLines": [{
        "fromDot": "tt",
        "toDot": "tm",
            "color": "rgb(0,0,0)"
    },
...
```

9.3.4.1 EPG

The 2D canvas of the `EMU-webApp` can also be configured to display EPG data as displayed in Figure 9.14. The SSFF file containing the EPG data has to be formated in a specific way. The format is a set of eight bytes per point in time, where each byte represents a row of electrodes on the artificial palate. Each binary bit value per byte indicates whether one of the eight sensors is activated or not (i.e., tongue contact was measured). If data in this format and an SSFF track with the predefined name *EPG* referencing the SSFF files are present, the 2D canvas can be configured to display this data by adding the *EPG* to the `twoDimCanvases:order` array as shown in Listing **??**.

```
"twoDimCanvases": {
    "order": ["EPG"]
}
```

9.3.4.2 EMA gestural landmark recognition

The `EMU-webApp` can also be configured to semi-automatically detect gestural landmarks of EMA contours. The functions implemented in the `EMU-webApp` are based on various Matlab scripts by Phil Hoole. For a

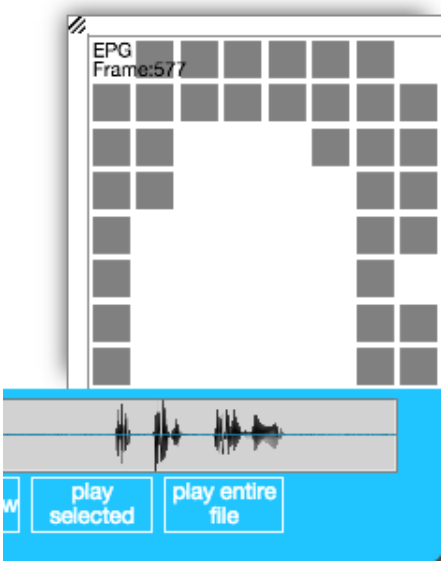


Figure 9.14: Screenshot of 2D canvas of the 'EMU-webApp' displaying EPG palate traces.

description of which gestural landmarks are detected and how these are detected, see Bombien (2011) page 61 ff.

Compared to the above configurations, configuring the `EMU-webApp` to semi-automatically detect gestural landmarks of EMA contours is done as part of the level definition's configuration entries of the `_DBconfig.json`. Listing **??** shows the `anagestConfig` entry, which configures the *tongueTipGestures* event level for this purpose. Within the web application this level has to be preselected by the user and a region containing a gesture in the SSFF track selected (left click and drag). Hitting the `ENTER/RETURN` key then executes the semi-automatic gestural landmark recognition functions. If multiple candidates are recognized for certain landmarks, the user will be prompted to select the appropriate landmark.

```
...
"levelDefinitions": [{
  {
    "name": "tongueTipGestures",
    "type": "EVENT",
    "attributeDefinitions": [{
        "name": "tongueTipGestures",
        "type": "STRING"
    }],
    "anagestConfig": {
        "verticalPosSsffTrackName": "tt_posz",
        "velocitySsffTrackName": "t_tipTV",
        "autoLinkLevelName": "ORT",
        "multiplicationFactor": 1,
        "threshold": 0.2,
        "gestureOnOffsetLabels": ["gon", "goff"],
        "maxVelocityOnOffsetLabels": ["von", "voff"],
        "constrictionPlateauBeginEndLabels": ["pon", "poff"],
        "maxConstrictionLabel": "mon"
    }
...
```

The user will be prompted to select an annotation item of the level specified in `anagestConfig:autoLinkLevelName` once the gestural landmarks are recognized. The `EMU-webApp` then automatically links all gestural landmark events to that item.

9.4 Conclusion

This chapter provided an overview of the `EMU-webApp` by showing the main layout and configuration options and how its labeling mechanics work. To our knowledge, the `EMU-webApp` is the first client-side web-based annotation tool that is this feature rich. Being completely web-based not only allows it to be used within the context of the EMU-SDMS but also allows it to connect to any web server that implements the `EMU-webApp-websocket-protocol` (see Appendix **??** for details). This feature is currently being utilized, for example, by the `IPS-EMUprot-nodeWSserver.js` server side software package (see https://github.com/IPS-LMU/IPS-EMUprot-nodeWSserver), which allows `emuDB`s to be served to any number of clients for collaborative annotation efforts. Further, by using the URL Parameters (see Chapter 13 for details) the web application can also be used to display annotation data that is hosted on any web server [5]. Because of these features, we feel the `EMU-webApp` is a valuable contribution to the speech and spoken language software tool landscape.

[5] See the BAS CLARIN Repository for a further example of an application using the `EMU-webApp-websocket-protocol` to display repository data in the `EMU-webApp`. See the BAS Web Services for an example of an application that creates links that utilize the URL parameters.

(PART) Main `emuR` function and object index

Chapter 10

emuR - package functions

This chapter gives an overview of the essential functions and central objects provided by the emuR package. It is not meant as a comprehensive list of every function and object provided by emuR, but rather tries to group the essential functions into meaningful categories for easier navigation. The categories presented in this chapter are:

- Import and conversion routines (Section 10.1),
- emuDB interaction and configuration routines (Section 10.2),
- EMU-webApp configuration routines (Section 10.3),
- Data extraction routines (Section 10.4),
- Central objects in emuR (Section 10.5), and
- Export routines (Section 10.6).

If a comprehensive list of every function and object provided by the emuR package is required, R's help() function (see R Example ??) can be used.

```
help(package="emuR")
```

10.1 Import and conversion routines

As most people that are starting to use the EMU-SDMS will probably already have some form of annotated data, we will first show how to convert existing data to the emuDB format. For a guide to creating an emuDB from scratch and for information about this format see Chapter 5.

10.1.1 Legacy EMU databases

For people transitioning to emuR from the legacy EMU system, emuR provides a function for converting existing legacy EMU databases to the new emuDB format. R Example ?? shows how to convert a legacy database that is part of the demo data provided by the emuR package.

```
# load the package
library(emuR)

# create demo data in directory provided by the tempdir() function
create_emuRdemoData(dir = tempdir())

# get the path to a .tpl file of
# a legacy EMU database that is part of the demo data
```

```
tplPath = file.path(tempdir(),
                    "emuR_demoData",
                    "legacy_ae",
                    "ae.tpl")

# convert this legacy EMU database to the emuDB format
convert_legacyEmuDB(emuTplPath = tplPath, targetDir = tempdir())
```

This will create a new emuDB in a temporary directory, provided by R's tempdir() function, containing all the information specified in the .tpl file. The name of the new emuDB is the same as the basename of the .tpl file from which it was generated. In other words, if the template file of the legacy EMU database has path A and the directory to which the converted database is to be written has path B, then convert_legacyEmuDB(emuTplPath = "A", targetdir = "B") will create an emuDB directory in B from the information stored in A.

10.1.2 TextGrid collections

A further function provided is the convert_TextGridCollection() function. This function converts an existing .TextGrid and .wav file collection to the emuDB format. In order to pair the correct files together the .TextGrid files and the .wav files must have the same name (i.e., file name without extension). A further restriction is that the tiers contained within all the .TextGrid files have to be equal in name and type (equal subsets can be chosen using the tierNames argument of the function). For example, if all .TextGrid files contain the tiers Syl: IntervalTier, Phonetic: IntervalTier and Tone: TextTier the conversion will work. However, if a single .TextGrid of the collection has the additional tier Word: IntervalTier the conversion will fail, although it can be made to work by specifying the equal tier subset equalSubset = c('Syl', 'Phonetic', 'Tone') and passing it into the function argument convert_TextGridCollection(..., tierNames = equalSubset, ...). R Example ?? shows how to convert a TextGrid collection to the emuDB format.

```
# get the path to a directory containing
# .wav & .TextGrid files that is part of the demo data
path2directory = file.path(tempdir(),
                           "emuR_demoData",
                           "TextGrid_collection")

# convert this TextGridCollection to the emuDB format
convert_TextGridCollection(path2directory, dbName = "myTGcolDB",
                           targetDir = tempdir())
```

R Example ?? will create a new emuDB in the directory tempdir() called myTGcolDB. The emuDB will contain all the tier information from the .TextGrid files but will not contain hierarchical information, as .TextGrid files do not contain any linking information. It is worth noting that it is possible to semi-automatically generate links between time-bearing levels using the autobuild_linkFromTimes() function. An example of this was given in Chapter 3. R Example ?? creates a new emuDB in the directory tempdir() called myTGcolDB. The emuDB contains all the tier information from the .TextGrid files no hierarchical information, as .TextGrid files do not contain any linking information. Further, it is possible to semi-automatically generate links between time-bearing levels using the autobuild_linkFromTimes() function. An example of this was given in Chapter 3.

10.1.3 BPF collections

Similar to the convert_TextGridCollection() function, the emuR package also provides a function for converting file collections consisting of BPF and .wav files to the emuDB format. R Example ?? shows how

this can be achieved.

```
# get the path to a directory containing
# .wav & .par files that is part of the demo data
path2directory = file.path(tempdir(),
                           "emuR_demoData",
                           "BPF_collection")

# convert this BPFCollection to the emuDB format
convert_BPFCollection(path2directory, dbName = 'myBPF-DB',
                      targetDir = tempdir(), verbose = F)
```

As the BPF format also permits annotation items to be linked to one another, this conversion function can optionally preserve this hierarchical information by specifying the `refLevel` argument.

10.1.4 txt collections

A further conversion routine provided by the `emuR` package is the `convert_txtCollection()` function. As with other file collection conversion functions, it converts file pair collections but this time consisting of plain text `.txt` and `.wav` files to the `emuDB` format. Compared to other conversion routines it behaves slightly differently, as unformatted plain text files do not contain any time information. It therefore places all the annotations of a single `.txt` file into a single timeless annotation item on a level of type `ITEM` called *bundle*.

```
# get the path to a directory containing .wav & .par
# files that is part of the demo data
path2directory = file.path(tempdir(),
                           "emuR_demoData",
                           "txt_collection")

# convert this txtCollection to the emuDB format
convert_txtCollection(sourceDir = path2directory,
                      dbName = "txtCol",
                      targetDir = tempdir(),
                      attributeDefinitionName = "transcription",
                      verbose = F)
```

Using this conversion routine creates a bare-bone, single route node `emuDB` which either can be further manually annotated or automatically hierarchically annotated using the `runBASwebservice_*`[1] functions of `emuR`. It is worth noting that these functions are already part of the `emuR` package; however, they are still considered to have a beta status which is why they are omitted from this documentation. In future versions of this documentation a section or chapter will be dedicated to using the BAS Webservices (Kisler et al., 2012) to automatically generate a hierarchical annotation structure for an entire `emuDB`.

10.2 emuDB interaction and configuration routines

This section provides a tabular overview of all the `emuDB` interaction routines provided by the `emuR` package and also provides a short description of each function or group of functions.

```
##
## Attaching package: 'dplyr'
```

[1] Functions contributed by Nina Pörner.

```
## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

Overview of the emuDB interaction routines provided by emuR.

Functions

Description

add/list/remove_attrDefLabelGroup()

Add / list / remove label group to / of / from attributeDefinition of emuDB

add/list/remove_labelGroup()

Add / list / remove global label group to / of / from emuDB

add/list/remove_levelDefinition()

Add / list / remove level definition to / of / from emuDB

add/list/remove_linkDefinition()

Add / list / remove link definition to / of / from emuDB

add/list/ remove_ssffTrackDefinition()

Add / list / remove SSFF track definition to / of / from emuDB

add/list/rename/remove_attributeDefinition()

Add / list / rename / remove attribute definition to / of / from emuDB

add_files()

Add files to emuDB

autobuild_linkFromTimes()

Autobuild links between two levels using their time information emuDB

create_emuDB()

Create empty emuDB

duplicate_level()

Duplicate level

import_mediaFiles()

Import media files to emuDB

list_bundles()

List bundles of emuDB

list_files()

List files of emuDB

list_sessions()

List sessions of emuDB

load_emuDB()

Load emuDB

replace_itemLabels()

Replace item labels

set/get/remove_legalLabels()

Set / get / remove legal labels of attribute definition of emuDB

rename_emuDB()

Rename emuDB

10.3 EMU-webApp configuration routines

This section provides a tabular overview of all the EMU-webApp configuration routines provided by the emuR package and also provides a short description of each function or group of functions. See Chapter 9 for examples of how to use these functions.

Overview of the EMU-webApp configuration functions provided by emuR.

Functions

Description

add/list/remove_perspective()

Add / list / remove perspective to / of / from emuDB

set/get_levelCanvasesOrder()

Set / get level canvases order for EMU-webApp of emuDB

set/get_signalCanvasesOrder()

Set / get signal canvases order for EMU-webApp of emuDB

It is worth noting that the legal labels configuration of the emuDB configuration will also affect how the EMU-webApp behaves, as it will not permit any other labels to be entered except those defined as legal labels.

10.4 Data extraction routines

This section provides a tabular overview of all the data extraction routines provided by the emuR package and also provides a short description of each function or group of functions. See Chapter 6 and Chapter 7 for multiple examples of how the various data extraction routines can be used.

Overview of the data extraction functions provided by emuR.

Functions

Description

query()

Query emuDB

requery_hier()

Requery hierarchical context of a segment list in an emuDB

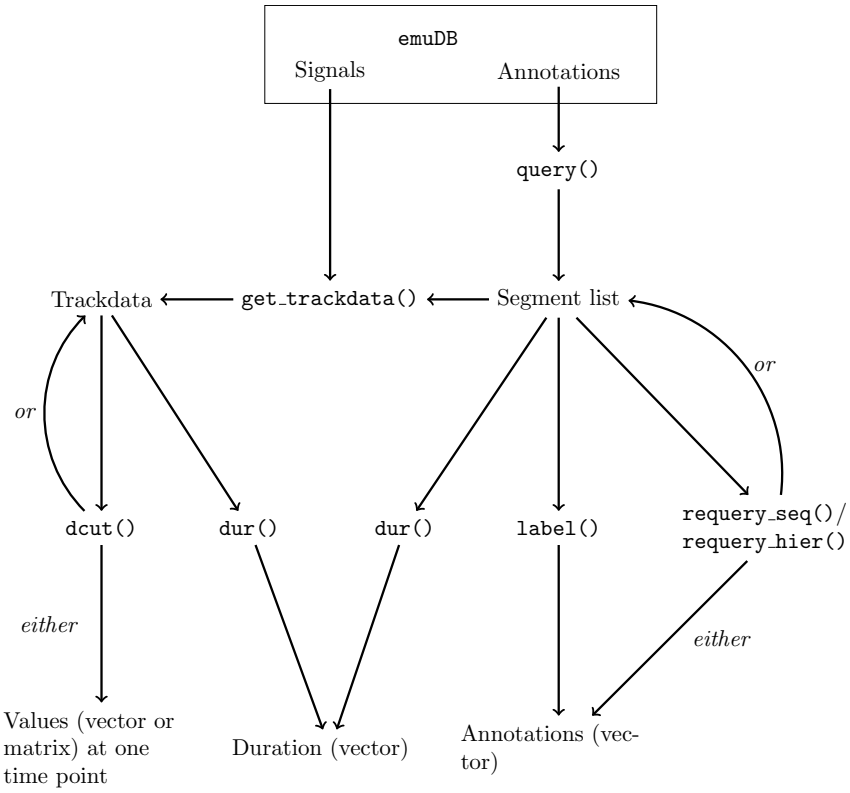


Figure 10.1: Relationship between various key functions in 'emuR' and their output. Figure is an updated version of Figure 5.7 in @harrington:2010a on page 121.

`requery_seq()`

Requery sequential context of segment list in an `emuDB`

`get_trackdata()`

Get trackdata from loaded `emuDB`

An overview of how the various data extraction functions in the `emuR` package interact is displayed in Figure 10.1. It is an updated version of a figure presented in Harrington (2010) on page 121 that additionally shows the output type of various post-processing functions (e.g., `dcut()`).

10.5 Central objects

This section provides a tabular overview of the central objects provided by the `emuR` package and also provides a short description of each object. See Chapter 6 and 7 for examples of functions returning these objects and how they can be used.

Overview of the central objects of the `emuR` package.

Object

Description

`emuRsegs`

A emuR segment list is a list of segment descriptions. Each segment descriptions describes a sequence of annotation items. The list is usually a result of an emuDB query using the query() function.

trackdata

A track data object is the result of get_trackdata() and usually contains the extracted signal data tracks belonging to segments of a segment list.

emuRtrackdata

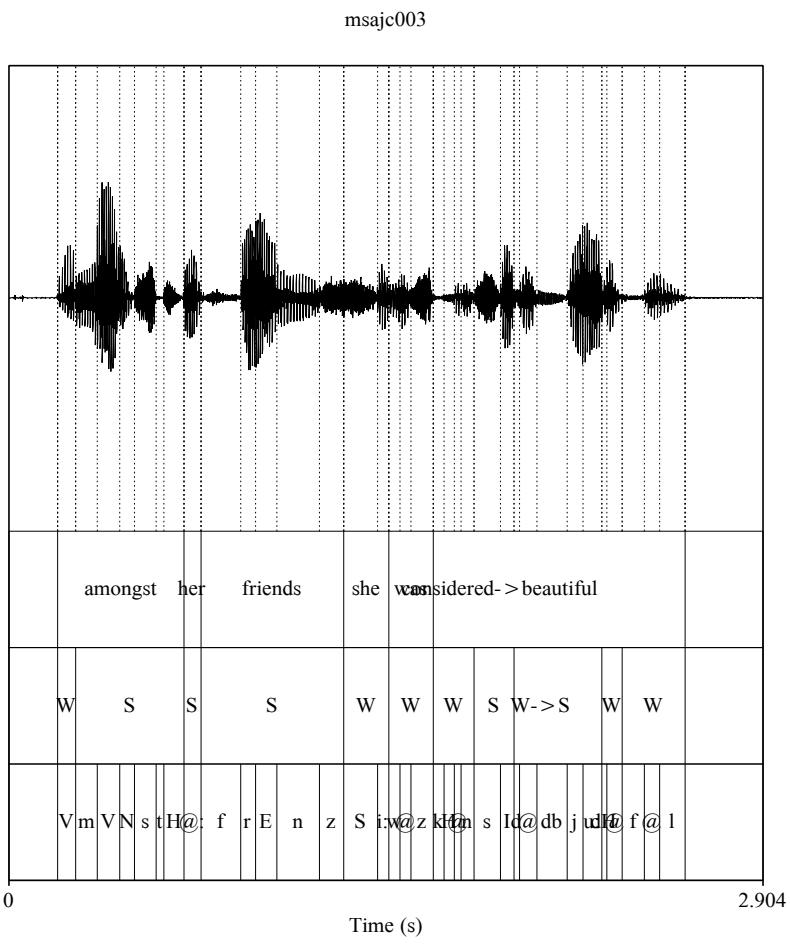
A emuR track data object is the result of get_trackdata() if the resultType parameter is set to emuRtrackdata or the result of an explicit call to create_emuRtrackdata. Compared to the trackdata object it is a sub-class of a data.table/data.frame which is meant to ease integration with other packages for further processing. It can be viewed as an amalgamation of an emuRsegs and a trackdata object as it contains the information stored in both objects (see also ?create_emuRtrackdata()).

10.6 Export routines

Although associated with data loss, the emuR package provides an export routine to the common TextGrid collection format called export_TextGridCollection(). While exporting is sometimes unavoidable, it is essential that users are aware that exporting to other formats which do not support or only partially support hierarchical annotations structures will lead to the loss of the explicit linking information. Although the autobuild_linkFromTimes() can partially recreate some of the hierarchical structure, it is advised that the export routine be used with extreme caution. R Example **??** shows how export_TextGridCollection() can be used to export the levels *Text*, *Syllable* and *Phonetic* of the *ae* demo emuDB to a TextGrid collection. Figure 10.6 show the content of the created msajc003.TextGrid file as displayed by Praat's "Draw visible sound and Textgrid..." procedure.

```
# get the path to "ae" emuDB
path2ae = file.path(tempdir(), "emuR_demoData", "ae_emuDB")

# load "ae" emuDB
ae = load_emuDB(path2ae)

# export the levels "Text", "Syllable"
# and "Phonetic" to a TextGrid collection
export_TextGridCollection(ae,
                          targetDir = tempdir(),
                          attributeDefinitionNames = c("Text",
                                                       "Syllable",
                                                       "Phonetic"))
```

\begin{figure}



{

}

\caption{TextGrid annotation generated by the `export_TextGridCollection()` function containing the tiers (from top to bottom): *Text*, *Syllable*, *Phonetic.*} \end{figure}

Depending on user requirements, additional export routines might be added to the `emuR` in the future.

10.7 Conclusion

This chapter provided an overview of the essential functions and central objects, grouped into meaningful categories, provided by the `emuR` package. It is meant as a quick reference for the user to quickly find functions she or he is interested in.

Part III

Implementation

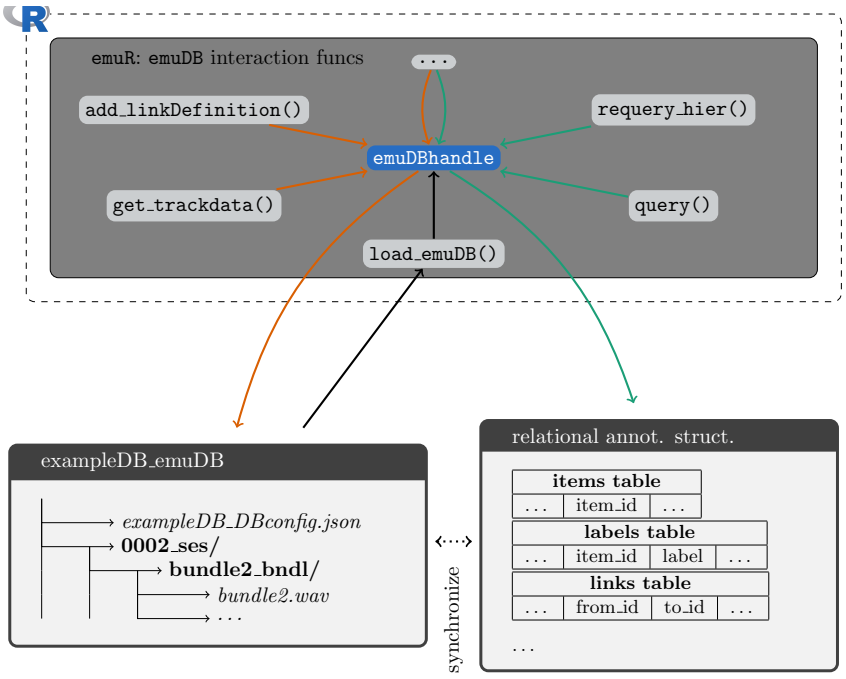
Chapter 11

Implementation of the query system [1]

Compatibly with other query languages, the EQL defines the user a front-end interface and infers the query's results from its semantics. However, a query language does not define any data structures or specify how the query engine is to be implemented. As mentioned in Chapter 2, a major user requirement was database portability, simple package installation, tolerable run times over complex queries, and a system that did not rely on external software at runtime. The only available back-end implementation that met those needs and was also available as an R package at the time was (R)SQLite (Hipp and Kennedy (2007), Wickham et al. (2014)). As (R)SQLite is a relational database management system, `emuR`'s query system could not be implemented so as to use directly the primary data sources of an `emuDB`, that is, the JSON files described in Chapter 5. A syncing mechanism that maps the primary data sources to a relational form for querying purposes had to be implemented. This relational form is referred to as the `emuDBcache` in the context of an `emuDB`. The data sources are synchronized while an `emuDB` is being loaded and when changes are made to the annotation files. To address load time issues, we implemented a file check-sum mechanism which only reloads and synchronizes annotation files that have a changed MD5-sum (Rivest, 1992). Figure 11 is a schematic representation of how the various `emuDB` interaction functions interact with either the file representation or the relational cache.

\begin{figure}

[1] Sections of this chapter have been published in Winkelmann et al. (2017).



{

}

\caption{Schematic architecture of `emuDB` interaction functions of the `emuR` package. textcolor{three-color-c2}{Orange} paths show examples of functions interacting with the files of the `emuDB`, while extcolor{three-color-c1}{green} paths show functions accessing the relational annotation structure. Actions like saving a changed annotation using the `EMU-webApp` first save the `_annot.json` to disk then update the relational annotation structure.} \end{figure}

Despite the disadvantages of cache invalidation problems, there are several advantages to having an object relational mapping between the JSON-based annotation structure of an `emuDB` and a relation table representation. One is that the user still has full access to the files within the directory structure of the `emuDB`. This means that external tools can be used to script, manipulate or simply interact with these files. This would not be the case if the files were stored in databases in a way that requires (semi-)advanced programming knowledge that might be beyond the capabilities of many users. Moreover, we can provide expert users with the option of using other relational database engines such as PostgreSQL, including all their performance-tweaking abilities, as their relational cache. This is especially valuable for handling very large speech databases.

The relational form of the annotation structure is split into six tables in the relational database to avoid data redundancy. The six tables are:

1. `emu_db`: containing `emuDB` information (columns: `uuid`, `name`),

uuid

name

0fc618dc-8980-414d-8c7a-144a649ce199

ae

2. `session`: containing `session` information (columns: `db_uuid`, `name`),

db_uuid

name

0fc618dc-8980-414d-8c7a-144a649ce199

0000

3. `bundle`: containing `bundle` information (columns: `db_uuid`, `session`, `name`, `annotates`, `sample_rate`, `md5\_annot\_json`),

db_uuid

session

name

annotates

sample_rate

md5_annot_json

0fc618dc-8980-414d-8c7a-144a649ce199

0000

msajc003

msajc003.wav

20000

785c7cdb6d4bd5e8b5cd7c56a5946ddf

...

...

...

...

...

...

4. `items`: containing all annotation items of `emuDB` (columns: `db_uuid`, `session`, `bundle`, `item_id`, `level`, `type`, `seq_idx`, `sample_rate`, `sample_point`, `sample_start`, `sample_dur`),

db_uuid	session	bundle	item_id	level	type	seq_idx	sample_rate
0fc618dc-8980-414d-8c7a-144a649ce199	0000	msajc003	147	Phonetic	SEGMENT	1	20000
...	...	...	...	...	...	...	...

5. `labels`: containing all labels belonging to all items (columns: `db_uuid`, `session`, `bundle`, `item_id`, `label_idx`, `name`, `label`), and

db_uuid	session	bundle	item_id	label_idx	name	label
0fc618dc-8980-414d-8c7a-144a649ce199	0000	msajc003	147	1	Phonetic	V
...	...	...	...	...	...	...

6. `links`: containing all links between annotation items of `emuDB` (columns: `db_uuid`, `session`, `bundle`, `from_id`, `to_id`, `label`).

db_uuid	session	bundle	from_id	to_id	label
0fc618dc-8980-414d-8c7a-144a649ce199	0000	msajc003	8	7	NA
...	...	...	...	7	NA

While performing a query the engine uses an aggregate key to address every annotation item and its labels (`db_uuid`, `session`, `bundle`, `item_id`) and a similar aggregate key to dereference the links (`db_uuid`, `session`, `bundle`, `from_id` / `to_id`) which connect items. As the records in relational tables are not intrinsically ordered a further aggregate key is used to address the annotation item via its index and level (`uuid`, `session`, `bundle`, `level` / `seq_idx`). This is used, for example, during sequential queries to provide

an ordering of the individual annotation items. It is worth noting that a plethora of other tables are created at query time to store various temporary results of a query. However, these tables are created as temporary tables during the query and are deleted on completion which means they are not permanently stored in the `emuDBcache`.

11.1 Query expression parser

The query engine parses an EQL query expression while simultaneously executing partial query expressions. This ad-hoc string evaluation parsing strategy is different from multiple other query systems which incorporate a query planner stage to pre-parse and optimize the query execution stage (e.g., Hipp and Kennedy (2007), Conway et al. (2016)). Although no pre-optimization can be performed, this strategy simplifies the execution of a query as it follows a constant heuristic evaluation strategy. This section describes this heuristic evaluation and parsing strategy based on the EQL expression `[[Syllable == W -> Syllable == W] ^ [Phoneme == @ -> #Phoneme == s]]`.

The main strategy of the query expression parser is to recursively parse and split an EQL expression into left and right sub-expressions until a so-called Simple Query (SQ term is found and can be executed (see EBNF in Appendix **??** for more information on the elements comprising the EQL). This is done by determining the operator which is the first to be evaluated on the current expression. This operator is determined by the sub-expression grouping provided by the bracketing. Each sub-expression is then considered to be a fully valid EQL expression and once again parsed. Figure 11.1, which is split into seven stages (marked S1-S7), shows the example EQL expression being parsed (S1-S3) and the resulting items being merged to meet the requirements of the individual operator (S4-S6) of the original query. S1 to S3 show the splitting operator character (e.g., textcolor{three_color_c3}{`->`} in purple) which splits the expression into a textcolor{three_color_c1}{left} (green) and textcolor{three_color_c2}{right} (orange) sub-expression.

The result modifier symbol (`#`) is noteworthy for its extra treatment by the query engine as it places an exact copy of the items marked by it into its own intermediary result storage (see $\#s_{\text{items}}$ node on S7 in Figure 11.1). After performing the database operations necessary to do the various merging operation which are performed on the intermediary results, this storage is updated by removing items from it that are no longer present due to the merging operation. As a final step, the query engine evaluates if there are items present in the intermediary result storage created by the presence of the result modifier symbol. If so, these items are used to create an `emuRsegs` object by deriving the time information and extracting the necessary information from the intermediate result storage. If no items are present in the result modifier storage, the query engine uses the items provided by the final merging procedure in S3 instead (which is not the case in the example used in Figure 11.1).

A detailed description of how this query expression parser functions is presented in a pseudo code representation in Algorithms **??** and **??**[2]. For simplicity, this representation ignores the treatment of the result modifier symbol (#) and focuses on the parsing and evaluation strategy of the query expression parser. As stated previously, the presence of the result modifier before an SQ triggers the query engine to place a copy of the result of that SQ into an additional result table, which is then updated throughout the rest of the query. The starting point for every query is the `query()` function (see line **??** in Algorithm **??**). This function places the filtered items, links and labels entries that are relevant for the current query into temporary tables. Depending on which query terms and operators are found, the EQL query engine uses the various sub-routines displayed in Algorithms **??** and **??** to parse and evaluate the EQL expression.

[2]The R code that implements this pseudo code can be found here: https://github.com/IPS-LMU/emuR/blob/master/R/emuR-query.database.R.

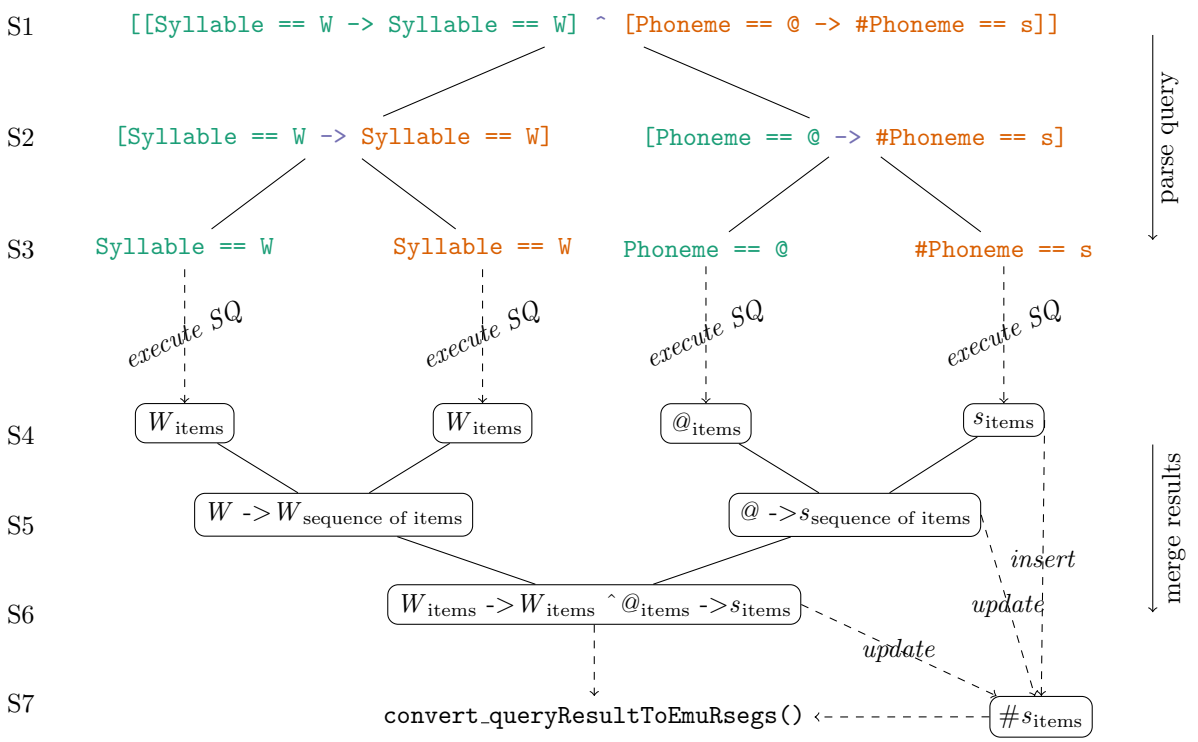


Figure 11.1: Example of how the query expression parser parses and evaluates an EQL expression and merges the result according to the respective EQL operators.

11.2 Redundant links

A noteworthy difference between the legacy and the new EMU system is how hierarchies are stored. The legacy system stored the linking information of a hierarchy in so-called hierarchical label files, which were plain text files that used the `.hlb` extensions. Within the label files this information was stored in space/blank separated lines:

```
\textcolor{three_color_c1}{111} \textcolor{three_color_c2}{139 140 141 173 174 175 185}
\textcolor{three_color_c1}{112} \textcolor{three_color_c2}{142 143 176 177}
\textcolor{three_color_c1}{113} \textcolor{three_color_c2}{144 145 146 178 179 180}
\textcolor{three_color_c1}{114} \textcolor{three_color_c2}{147}
\textcolor{three_color_c1}{115} \textcolor{three_color_c2}{148}
\textcolor{three_color_c1}{116} \textcolor{three_color_c2}{149},
```

where the textcolor{three_color_c1}{first number} (green) of each line was the parent's ID and the textcolor{three_color_c2}{following numbers} (orange) indicated the annotation items the parent was linked to. However, it was not just links to the items on the child level that were stored in each line. Rather, a link to all children of all levels below the parent level was stored for each parent item. This was likely due to performance benefits in parsing and mapping onto the internal structures used by the legacy query engine. A schematic representation of this form of linking is displayed in Figure 11.4A. As these redundant links are prone to errors while updating the data model and lead to a convoluted annotation structure models (see excessive use of dashed lines in Figure 11.4A), we chose to eliminate them and opted for the cleaner, non-redundant representation displayed in Figure 11.4B. Although this led to a more complex query parser engine for hierarchical queries and functions, we feel it is a cleaner, more accurate and more robust data representation.

```
1: function QUERY_DBEQLFUNCQ(query)
2:     place all parent level items into tmp table
3:     place all child level items into tmp table
4:     QUERY_DBHIER(parentItemsTable, childItemsTable)
5:     if Start End or Medial query then
6:         extract parent items and place in tmp result table
7:     else
8:         extract child items and place in tmp result table
9:     end if
10: end function
11: function QUERY_DBEQLLABELQ(query)
12:     splitLabels ← split labels at |
13:     for all splitLabels do
14:         if operator is ==, = or != then
15:             extract items that contain labels which are equal or unequal to label
16:         else if operator is =~ or !~ then
17:             extract items that contain labels that match or don't match RegEx
18:         end if
19:         merge results in tmp table
20:     end for
21: end function
22: function QUERY_DBEQLSQ(query)
23:     if query contains round brackets then
24:         QUERY_DBEQLFUNCQ(query)
25:     else
26:         QUERY_DBEQLLABELQ(query)
27:     end if
28: end function
29: function QUERY_DBEQLCONJQ(query)
30:     splitItems ← split query at &
31:     for all splitItems do
32:         QUERY_DBEQLSQ(splitItems)
33:         merge results in tmp table
34:     end for
35: end function
36: function QUERY_DBHIER(leftTable, rightTable)
37:     hp ← extract hier. paths conn. leftTable and rightTable level names
38:     for all child and parent level pairs in hp do
39:         connect child and parent items using links table
40:         reduce to min seq. idx (left side of trapeze)
41:         and to max seq. idx (right side of trapeze)
42:     end for
43: end function
```

Figure 11.2: Pseudo Code for Query Engine Algorithm - Part 1

```
44: function QUERY_DBEQLINBRACKET(query)
45:     qTrim ← remove outer square brackets
46:     leftQuery, rightQuery ← split qTrim at cur. operator
47:     QUERY_DATABASEWITHEQL(leftQuery)          ▷ recursive part of query
48:     QUERY_DATABASEWITHEQL(rightQuery)         ▷ recursive part of query
49:     if cur. operator is domintation operator then
50:         QUERY_DBHIER(leftQueryResultTable, rightQueryResultTable)
51:     else if cur. operator is seq. operator then
52:         find seq. of leftQueryResultTable and rightQueryResultTable items
53:     else
54:         QUERY_DATABASEWITHEQL(qTrim)
55:     end if
56: end function
57: function QUERY_DBWITHEQL(query)
58:     if query isn't wrapped in brackets then
59:         QUERY_DBEQLCONJQ(query)
60:     else
61:         QUERY_DBEQLINBRACKET(query)
62:     end if
63: end function
64: function QUERY(query, sesPattern, bndlPattern)
65:     filter items in relational tables by sesPattern
66:     filter items in relational tables by bndlPattern
67:     QUERY_DBWITHEQL(query)
68:     seglist ← CONVERT_QUERYRESULTTOEMURSEGS(tmpResultTableName)
69:     return seglist
70: end function
```

Figure 11.3: Pseudo Code for Query Engine Algorithm - Part 2

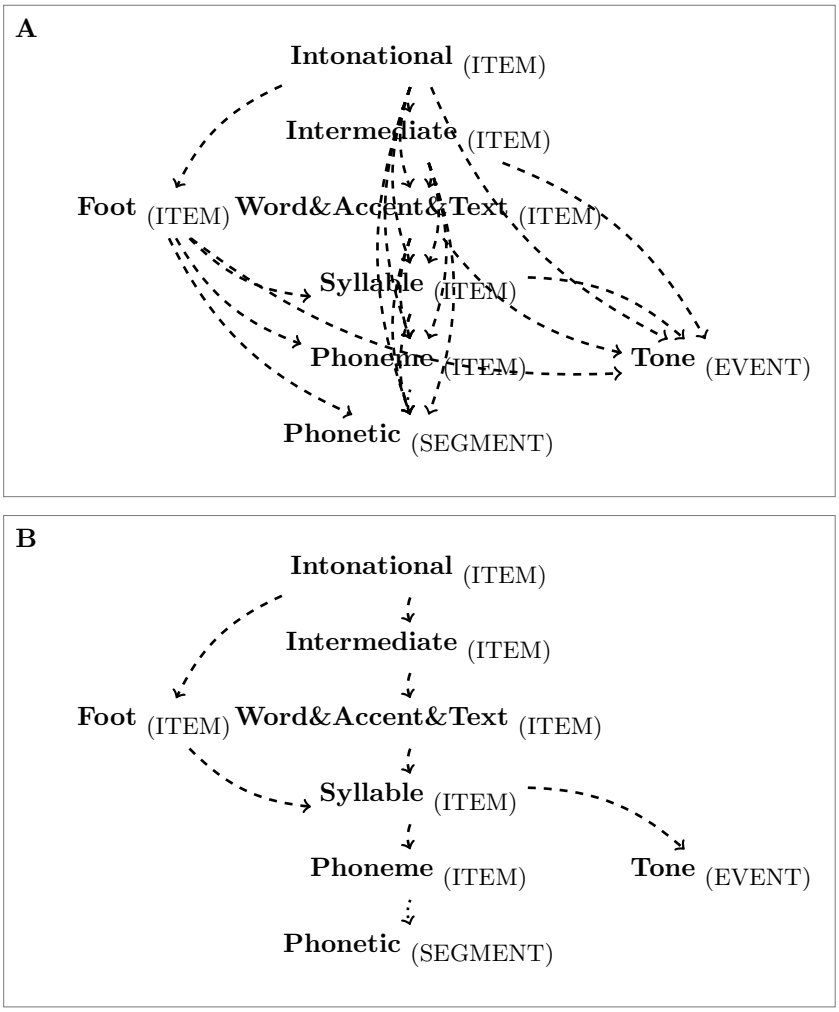


Figure 11.4: Schematic of hierarchy graph *ae*; extbfA: legacy redundant strategy vs. extbfA: cleaner non-redundant strategy.

Chapter 12

`wrassp` implementation

The `libassp` was originally written by Michel Scheffers as a C library which could be linked against or compiled into separate executable signal processing command line tools. To extend the legacy EMU system, the `libassp` it was integrated into it by using the Tcl Extension Architecture (TEA) to create a native extension to the Tcl programming language. The bulk of this work was done by Lasse Bombien in collaboration with Michel Scheffers. Lasse Bombien also implemented the `tkassp` user interface module as part of the legacy EMU system to allow the user full access to the functionality of the `libassp` from a GUI. The `wrassp` R package was written by Lasse Bombien and Raphael Winkelmann based on a similar approach as the `tclassp` port using the TEA. Since the `libassp` was put under the GPL version 3 (see https://www.gnu.org/licenses/gpl-3.0.en.html) by Michel Scheffers, the `wrassp` also carries this license.

12.1 The `libassp` port

Here, we briefly describe our strategy for porting the `libassp` to R. The port of the `libassp` to the R eco-system was achieved using the foreign language interface provided by the R system as is described in the R Extensions manual (see https://cran.r-project.org/doc/manuals/r-release/R-exts.htmlWriting). To port the various signal processing routines provided by the `libassp` and to avoid code redundancy a single C function called `performAssp()` was created. This function acts as a C wrapper function interface to `libassp`'s internal functions and handles the data conversion between `libassp`'s internal and R's data structures. However, to provide the user with a clear and concise API we chose to implement separate R functions for every signal processing function. This also allowed us to formulate more concise manual entries for each of the signal processing function provided by `wrassp`. R Example **??** is a pseudo-code example of the layout of each signal processing function `wrassp` provides.

```
##' roxygen2 documentation for genericWrasspFun
genericWrasspSigProcFun = function(listOfFiles,
                                   ...,
                                   forceToLog = useWrasspLogger){

  ############################
  # perform parameter checks
  if (is.null(listOfFiles)) {
        stop(paste("listOfFiles is NULL! ..."))
  }
  # ...

  # call performAssp
```

```
  externalRes = invisible(.External("performAssp", listOfFiles,
                                    fname = "forest", ...))


  ############################
  # write options to options log file
  if (forceToLog){
      optionsGivenAsArgs = as.list(match.call(
        expand.dots = TRUE))
      wrassp.logger(optionsGivenAsArgs[[1]],
                    optionsGivenAsArgs[-1],
                    optLogFilePath, listOfFiles)
  }

  return(externalRes)
}
```

To provide access to the file handling capabilities of the `libassp`, we implemented two C interface functions called `getDObj2()` (where `2` is simply used as a function version marker) and `writeDObj()`. These functions use `libassp`'s `asspFOpen()`, `asspFFill()`, `asspFWrite()` and `asspFClose()` function to read and write files supported by the `libassp` from and to files on disk into R. The public API functions `read.AsspDataObj()` and `write.AsspDataObj()` are the R wrapper functions around `getDObj2()` and `writeDObj()`.

To be able to access some of `libassp`'s internal variables further wrapper functions were implemented. It was necessary to have access to these variables to be able to perform adequate parameter checks in various functions. R Example **??** shows these functions.

```
# load the wrassp package
library(wrassp)

# show AsspWindowTypes
AsspWindowTypes()
```

```
##  [1] "RECTANGLE" "PARABOLA"  "COS"       "HANN"      "COS_4"
##  [6] "HAMMING"   "BLACKMAN"  "BLACK_X"   "BLACK_M3"  "BLACK_M4"
## [11] "NUTTAL_3"  "NUTTAL_4"  "KAISER2_0" "KAISER3_0" "KAISER4_0"
```

```
# show wrasspOutputInfos
AsspLpTypes()
```

```
## [1] "ARF" "LAR" "LPC" "RFC"
```

```
# show wrasspOutputInfos
AsspSpectTypes()
```

```
## [1] "DFT" "LPS" "CSS" "CEP"
```

The `wrassp` package provides two R objects that contain useful information regarding the supported file format types (`AsspFileFormats`) and the output created by the various signal processing functions. R Example **??** shows the content of these two objects.

```
# show AsspFileFormats
AsspFileFormats
```

```
##     RAW   ASP_A   ASP_B   XASSP  IPDS_M  IPDS_S    AIFF    AIFC     CSL
##       1       2       3       4       5       6       7       8       9
##    CSRE    ESPS     ILS     KTH   SWELL   SNACK     SFS     SND      AU
```

```
##      10      11      12      13      13      13      14      15      15
##    NIST  SPHERE PRAAT_S PRAAT_L PRAAT_B    SSFF    WAVE  WAVE_X  XLABEL
##      16      16      17      18      19      20      21      22      24
##    YORK     UWM
##      25      26
```

```
# show first element of wrasspOutputInfos
wrasspOutputInfos[[1]]
```

```
## $ext
## [1] "acf"
##
## $tracks
## [1] "acf"
##
## $outputType
## [1] "SSFF"
```

As a final remark, it is worth noting that porting the C library `libassp` to R enables the functions provided by the `wrassp` package to run at near native speeds on every platform supported by R and avoids almost any interpreter overhead.

Chapter 13

EMU-webApp implementation

Here, we briefly describe our strategy for implementing the EMU-webApp. The EMU-webApp is written entirely in HTML, Javascript and CSS. To ease testing and to enable easy integration and extendability we chose to use the AngularJS Javascript framework (Google, 2014). Most of the components of the EMU-webApp (e.g., the spectrogram display) are implemented as so-called Angular directives. This means that, apart from dependencies on data service classes that have to be made available, these components are reusable and can be integrated into other web applications. The EMU-webApp makes extensive use of Angular data bindings to keep the display and the various data services in sync with each other. It is also worth noting that we chose to use the SASS (see http://sass-lang.com/) preprocessor to compile .sass files to CSS. This enabled us to use things like mixins, variables and inheritance for a more concise stylesheet management and generation.

The main reason we chose the JSON file format as the main file type for the EMU-SDMS is because we wanted a web application as the main GUI of the new system. Using JSON files enables the EMU-webApp to directly use the annotation and configuration files that are part of an emuDB without manipulating or reformatting the data.

The rest of this chapter will focus on the communication protocol and the URL parameters provided by the EMU-webApp. These should be of special interest to developers as they describe how to communicate with the web application and how to use the web application to display data that is hosted on other http web servers.

13.1 Communication protocol[1]

A large benefit gained by choosing the browser as the user interface is the ability to easily interact with a server using standard web protocols, such as http, https or websockets. In order to standardize the data exchange with a server, we have developed a simple request-response communication protocol on top of the websocket standard. This decision was strongly guided by the availability of the httpuv R package (RStudio and Inc., 2015). Our protocol defines a set of JSON objects for both the requests and responses. A subset of the request-response actions, most of them triggered by the client after connection, are displayed in Table ??.

Protocol_Command

Comments

GETPROTOCOL

[1]This section has been published in Winkelmann and Raess (2015).

Check if the server implements the correct protocol

`GETDOUSERMANAGEMENT`

See if the server handles user management (if yes, then this prompts a login dialog $ ightarrow$ `LOGONUSER`

`GETGLOBALDBCONFIG`

Request the configuration file for the current connection

`GETBUNDLELIST`

Request the list of available bundles for current connection

`GETBUNDLE`

Request data belonging to a specific bundle name

`SAVEBUNDLE`

Save data belonging to a specific bundle name

This protocol definition makes collaborative annotation efforts possible, as developers can easily implement servers for communicating with the `EMU-webApp`. Using this protocol allows a database to be hosted by a single server anywhere on the globe that then can be made available to a theoretically infinite number of users working on separate accounts logging individual annotations, time and date of changes and other activities such as comments added to problematic cases. Tasks can be allocated to and unlocked for each individual user by the project leader. As such, user management in collaborative projects is substantially simplified and trackable compared with other currently available software for annotation.

The `emuR` package implements this websocket protocol as part of the `serve()` function utilizing the `httpuv` package. Further example implementations of this websocket protocol are provided as part of the source code repository of the `EMU-webApp` (see https://github.com/IPS-LMU/EMU-webApp/tree/master/exampleServers). A in-depth description of the protocol which includes descriptions of each request and response JSON object can be found in Appendix ??.

13.2 URL parameters

The `EMU-webApp` currently implements several URL parameters (see https://en.wikipedia.org/wiki/Query_string for more information) as part of its URL query string. This section describes the currently implemented parameters and gives some accompanying examples.

13.2.1 Websocket server parameters

The current URL parameters that affect the websocket server connection are:

- **serverUrl**=*URL* is a URL pointing to a websocket server that implements the EMU-webApp websocket protocol, and
- **autoConnect**=*true / false* automatically connects to a websocket server URL specified in the **serverUrl** parameter. If the **serverUrl** parameter is not set the web application defaults to the entry in its `default_emuwebappConfig.json`.

13.2.2 Examples

- auto connect to local wsServer: http://ips-lmu.github.io/EMU-webApp/?autoConnect=true&serverUrl=ws:%2F%2Flocalhost:17890

13.2.3 Label file preview parameters

The current URL parameters for using the `EMU-webApp` to visualize files that are hosted on other http servers are:

- **audioGetUrl**=*URL* GET URL that will respond with `.wav` file,
- **labelGetUrl**=*URL* GET URL that will respond with label/annotation file,
- **DBconfigGetURL**=*URL* GET URL that will respond with `_DBconfig.json` file, and
- **labelType**=*TEXTGRID / annotJSON* specifies the type of annotation file.

This mechanism is, for example, currently being used by the WebMAUS web-services of the BASWebServices (see https://clarin.phonetik.uni-muenchen.de/BASWebServices) to provide a preview of the automatically segmented speech files.

13.2.4 Examples

- TextGrid example: http://ips-lmu.github.io/EMU-webApp/?audioGetUrl=https://raw.githubusercontent.com/IPS-LMU/EMU-webApp/master/app/testData/oldFormat/msajc003/msajc003.wav&labelGetUrl=https://raw.githubusercontent.com/IPS-LMU/EMU-webApp/master/app/testData/oldFormat/msajc003/msajc003.TextGrid&labelType=TEXTGRID
- annotJSON example: http://ips-lmu.github.io/EMU-webApp/?audioGetUrl=https://raw.githubusercontent.com/IPS-LMU/EMU-webApp/master/app/testData/newFormat/ae/0000_ses/msajc003_bndl/msajc003.wav&labelGetUrl=https://raw.githubusercontent.com/IPS-LMU/EMU-webApp/master/app/testData/newFormat/ae/0000_ses/msajc003_bndl/msajc003_annot.json&labelType=annotJSON

Bibliography

Abercombie, D. (1967). *Elements of general phonetics*. Aldine Pub. Company.

Beckman, M. E. and Ayers, G. (1997). Guidelines for ToBI labelling. *The OSU Research Foundation*, 3.

Bird, S. and Liberman, M. (2001). A formal framework for linguistic annotation. *Speech communication*, 33(1):23–60.

Boersma, P. and Weenink, D. (2016). Praat: doing phonetics by computer (Version 6.0.19). http://www.fon.hum.uva.nl/praat/.

Bombien, L. (2011). *Segmental and prosodic aspects in the production of consonant clusters: On the goodness of clusters*. PhD thesis, München, Univ., Diss., 2011.

Bombien, L., Cassidy, S., Harrington, J., John, T., and Palethorpe, S. (2006). Recent developments in the Emu speech database system. In *Proc. 11th SST Conference Auckland*, pages 313–316.

Cassidy, S. (2013). The Emu Speech Database System Manual: Chapter 9. Simple Signal File Format. http://emu.sourceforge.net/manual/chap.ssff.html.

Cassidy, S. and Harrington, J. (1996). Emu: An enhanced hierarchical speech data management system. In *Proceedings of the Sixth Australian International Conference on Speech Science and Technology*, pages 361–366.

Cassidy, S. and Harrington, J. (2001). Multi-level annotation in the Emu speech database management system. *Speech Communication*, 33(1):61–77.

Coleman, J. and Local, J. (1991). The "no crossing constraint" in autosegmental phonology. *Linguistics and Philosophy*, 14(3):295–338.

Conway, J., Eddelbuettel, D., Nishiyama, T., Prayaga, S. K., and Tiffin, N. (2016). *RPostgreSQL: R interface to the PostgreSQL database system*. R package version 0.4-1 package version 0.4-1.

Draxler, C. and Jänsch, K. (2004). SpeechRecorder - a Universal Platform Independent Multi-Channel Audio Recording Software. In *Proc. of the IV. International Conference on Language Resources and Evaluation*, pages 559–562, Lisbon, Portugal.

Fromont, R. and Hay, J. (2012). LaBB-CAT: An annotation store. In *Australasian Language Technology Association Workshop 2012*, volume 113. Citeseer.

Garshol, L. M. (2003). BNF and EBNF: What are they and how do they work. *acedida pela última vez em*, 16.

Google (2014). AngularJS. http://angularjs.org/.

Harrington, J. (2010). *Phonetic analysis of speech corpora*. John Wiley & Sons.

Harrington, J. and Cassidy, S. (2002). The emu-query language (anhang).

Harrington, J., Cassidy, S., Fletcher, J., and Mc Veigh, A. (1993). The mu+ system for corpus based speech research. *Computer Speech & Language*, 7(4):305–331.

Hipp, D. R. and Kennedy, D. (2007). Sqlite. https://www.sqlite.org/.

Ide, N. and Romary, L. (2004). International standard for a linguistic annotation framework. *Natural language engineering*, 10(3-4):211–225.

ISO (2012). Language resource management — Linguistic annotation framework (laf). ISO 24612:2012, International Organization for Standardization, Geneva, Switzerland.

John, T. (2012). *Emu speech database system*. PhD thesis, Ludwig Maximilian University of Munich.

Kisler, T., Schiel, F., Reichel, U. D., and Draxler, C. (2015). Phonetic/linguistic web services at BAS. ISCA.

Kisler, T., Schiel, F., and Sloetjes, H. (2012). Signal processing via web services: the use case WebMAUS. In *Proceedings Digital Humanities 2012, Hamburg, Germany*, pages 30–34, Hamburg.

Knuth, D. E. (1968). The Art of Computer Programming Vol. 1, Fundamental Algorithms. *Addison-Wesley, Reading, MA*, 9:364–369.

McAuliffe, M. and Sonderegger, M. (2016). Speech Corpus Tools (SCT). http://speech-corpus-tools.readthedocs.io/.

Ooms, J. (2014). The jsonlite package: A practical and consistent mapping between json data and r objects. *arXiv:1403.2805 [stat.CO]*.

R Core Team (2016). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

R Special Interest Group on Databases (R-SIG-DB), Wickham, H., and Müller, K. (2016). *DBI: R Database Interface*. R package version 0.4.

Rivest, R. (1992). The md5 message-digest algorithm. https://tools.ietf.org/html/rfc1321.

Rose, Y., MacWhinney, B., Byrne, R., Hedlund, G., Maddocks, K., O'Brien, P., and Wareham, T. (2006). Introducing phon: A software solution for the study of phonological acquisition. In *Proceedings of the... Annual Boston University Conference on Language Development. Boston University Conference on Language Development*, volume 2006, page 489. NIH Public Access.

RStudio and Inc. (2015). *httpuv: HTTP and WebSocket Server Library*. R package version 1.3.3.

Shue, Y.-L., P., K., C., V., and K., Y. (2011). VoiceSauce: A program for voice analysis. In *Proceedings of the ICPhS*, volume XVII, pages 1846–1849.

Wells, J. C. et al. (1997). Sampa computer readable phonetic alphabet. *Handbook of standards and resources for spoken language systems*, 4.

Wickham, H., James, D. A., and Falcon, S. (2014). *RSQLite: SQLite Interface for R*. R package version 1.0.0.

Winkelmann, R., Harrington, J., and Jänsch, K. (2017). EMU-SDMS: Advanced speech database management and analysis in R. *Computer Speech & Language*, pages –.

Winkelmann, R. and Raess, G. (2015). EMU-webApp. http://ips-lmu.github.io/EMU-webApp/.

Wittenburg, P., Brugman, H., Russel, A., Klassmann, A., and Sloetjes, H. (2006). Elan: a professional framework for multimodality research. In *Proceedings of LREC*, volume 2006.

Zipser, F. and Romary, L. (2010). A model oriented approach to the mapping of annotation formats using standards. In *Workshop on Language Resource and Language Technology Standards, LREC 2010*, La Valette, Malta.