*Glottometrics 49, 2020, 66–86*

# Tools for Semi-Automatic Analysis of Sound Correspondences: The *soundcorrs* Package for *R*

*Kamil Stachowski*[1]

**Abstract.** *soundcorrs* is a small R library of functions intended to facilitate computer-aided analysis of sound correspondences between languages. It is not designed to draw its own conclusions, merely to automate labour-intensive tasks and furnish the linguist with sifted and processed data for him or her to interpret. To make use of its basic functionality, *soundcorrs* requires only a very rudimentary knowledge of R, and no understanding of statistics at all. More advanced functions can be accessed and more involved results obtained after only a brief course of the two.

**Keywords:** software, sound correspondences, loanword adaptation, letter-grapheme-phone correspondences, phonology.

## 1 Introduction

The traditional method of investigating sound correspondences between languages is to spend long hours filling shoeboxes with flashcards, and then longer hours still excavating from them pairs of words which exemplify the currently discussed problem. The *soundcorrs* library can help reduce the time and effort involved in this kind of analysis. By design, it does not attempt to draw any conclusions on its own, a relatively recent practice which the more traditionally-minded linguists find well-justified reasons to distrust; it merely automates the management of data, acting more as a secretary than an assistant.

It is a library for R, which means that, although effort has been made to render it as easy to use as possible, some rudimentary knowledge of the language and the environment is required. The necessary topics include: basic data structures, function invocation, and variable assignment. A degree of understanding of regular expressions is highly recommended, and any additional knowledge can surely be put to good use, too. In the least effort scenario, simply repeating the example (sec. 4)] should prove useful, hopefully as an incentive to embark on a more in-depth exploration.

The *soundcorrs* library exports a number of functions, some serving an analytic purpose, others being just convenient helpers. The former are discussed in the sections below; the latter are only mentioned, but I trust that the documentation provided in the package will prove

1 Jagiellonian University, ul. Gołębia 24, PL – 31-007 Cracow, Poland; https://orcid.org/0000-0002-5909-035X; kamil.stachowski@gmail.com.

sufficient for a researcher to utilize them. As a quick reference, the analytic functions are listed in tab. 1, organized by their output.

**Tab.1.**
A quick reference to *soundcorrs*' most important functions. For a full list, see the vignette (run `vignette ("soundcorrs")`).

| Output | Details | Function | Section |
|---|---|---|---|
| contingency table | segment-to-segment | `summary()` | 3.1 |
| contingency table | correspondence-to-correspondence | `table()` | 3.2 |
| contingency table | correspondence-to-metadata | `table()` | 3.2 |
| contingency table | *n*-gram-to-*n*-gram | `ngrams()` | 4 |
| contingency tables | all, as a list | `allTables()` | 3.2 |
| fitting | one dataset, multiple models | `multiFit()` | 3.3 |
| fitting | multiple datasets, multiple models | `fitTable()` | 3.3 |
| *n*-grams | table with counts | `ngrams()` | 3.3 |
| pairs | single, unformatted | `findPairs()` | 3.1 |
| pairs and tables | all, formatted | `allPairs()` | 3.1 |
| segments | in relation to a correspondence | `findSegments()` | 3.2 |

The present paper has been written primarily with loanword adaptation in mind. However, *soundcorrs* can be also used to explore other topics, both in purely qualitative linguistics (e.g., sound correspondences between related languages, morphological correspondences), as well as in more quantitatively-oriented research (e.g., grapheme-phoneme correspondences; Altmann/Fengxiang, 2008).

The organization of this paper is as follows: in sec. 2, data preparation; in sec. 3, a discussion of the most important of *soundcorrs*' functions, ordered from the more qualitative to the more quantitative approach to research; in sec. 4, a simple sample session with *soundcorrs*, and in sec. 5, a brief discussion of the errors and warnings issued by *soundcorrs*, as well as a caveat concerning encoding.

This paper describes *soundcorrs* as of version 0.1.1.

## 2 Preparation

Before work with *soundcorrs* can begin, the user needs to prepare a definition of the transcription or transcriptions in which the data are recorded, and the data themselves. Especially the latter can be a lengthy process, but it is unfortunately unavoidable. Both are described separately in subsections below.

One remark, however, is common to them, and it concerns encoding. The recommended choice is UTF-8. It has not been found to cause any issues under BSD, Linux, and macOS, but it has under Windows, and for this reason it is suggested that Windows users limit their transcriptions to plain ASCII. This is a harsh restriction; hopefully, future versions of *soundcorrs* will be able to do without it.

### 2.1 Transcription

There are two reasons why *soundcorrs* needs to know about the transcription in which the data are recorded. Firstly, without this knowledge, traditional linguistic regular expressions ("wildcards") would not be possible; and secondly, it allows to involve phonetics and other

aspects in the analysis performed using *soundcorrs*.

The transcription is stored in a tsv file in the form of a table with two or three columns, as shown in Fig. 1, and can be read using the `read.transcription()` function. The only required argument is the name of the file; optionally, custom names for columns can also be provided. The return value is an object of class `transcription`.

```
GRAPHEME    VALUE                    META
b           cons,stop,lab,vd         b
p           cons,stop,lab,vl,mark1   p
f           cons,fric,lab,vl,mark1   f
B           cons,stop,lab            [pb]
P           mark1                    [pf]
-           NULL                     -
```

**Fig. 1.** Sample transcription file.

The first column, `GRAPHEME`, contains a list of graphemes. It is recommended that the transcription only employ single characters, as multigraphs may cause *soundcorrs* to yield unexpected and incorrect results, even if they are always isolated into separate segments. This restriction applies especially to metacharacters ("wildcards"). If a character missing from the Unicode is necessary, e.g. *b* with acute, it is generally recommended that it be not composed using a combining diacritical mark, but rather replaced with another single character, such as Б, ב, ب, etc. Characters used by R as metacharacters in regular expressions (`. + * ^ \ $ ? | ( ) [ ] { }`) cannot be used as graphemes. When reading a dataset, *soundcorrs* will warn about segments not covered by the provided transcription.

The second column, `VALUE`, contains comma-separated (without spaces) features of individual graphemes: phonetic attributes, formant frequencies, etc. The intention behind this column is to help analyze phonetics, but it is not actually necessary that the features be phonetic. They can be thought of as markers or labels required to generate the `META` column. Grapheme(s) which represent "linguistic zero" should be given the value of `NULL`. Such graphemes will be ignored, among others, by the function `findPairs()` if the argument `exact` is set to `FALSE` (which is the default).

The third column, `META`, is optional. If it is not given in the transcription file, *soundcorrs* will generate it automatically based on the `VALUE` column (the recommended method) – but if it is there, *soundcorrs* will not check its accordance with the `VALUE` column. The `META` column can be used to extend R's in-built set of metacharacters to include symbols which are conventionally used in linguistics. Technically, the values in this column are regular expressions which are substituted for the characters from the `GRAPHEME` column when a query with `findPairs()` is performed. Graphemes which are not meant to be used as metacharacters, including the linguistic zero, should be simply repeated; those that are need to be expanded to an enumeration. For example, if ‹N› is to represent 'any nasal consonant', it should be expanded to `[mḿnńŋ]`, or whatever other set of nasal consonants is available in the given transcription. When the `META` column is generated automatically, the expansion will include all graphemes with values that form a superset of the value of the given grapheme; e.g. if `N` is given the value of `cons,nasal`, it will be expanded to an enumeration of all graphemes which contain both `cons` and `nasal` in the `VALUE` column.

Phonetic analysis is the primary intended use for a transcription, but it can as well be used for morphology, and perhaps other angles of inquiry as well. The general rule is that

transcription should match segmentation (see sec. 2.2): if the latter is phonetic, so should the transcription be; if it is morphological, so should the transcription be.

## 2.2 Data

The same as the transcription, the data are stored in text files in the form of tables. They can be in what will be referred to as the *long format* or the *wide format*. The "long format" is a table with at least two columns: `ALIGNED` and `LANGUAGE`, and each entry occupying its own row, as in Fig. 2a. The "wide format" is perhaps less convenient for people, but it is used internally and required by *soundcorrs*. In it, corresponding pairs / triples / … of words are placed in a single row which, therefore, contains at least two columns, each holding words from a single language: `ALIGNED.x` and `ALIGNED.y`, or `LATIN` and `GERMAN`, etc. – as in Fig. 2b.

Data frames can be converted from one format to the other using functions `long2wide()` and `wide2long()`. Partial conversion is also possible, for metadata which are more conveniently viewed as describing entire pairs than individual words. For this purpose, `long2wide()`'s argument `skip` is used.

(a) The "long format".

| LANGUAGE | WORD | ALIGNED |
|---|---|---|
| Latin | mūsica | m\|ū\|s\|i\|k\|a |
| English | music | m\|jū\|z\|i\|k\|- |
| German | Musik | m\|u\|z\|ī\|k\|- |
| Polish | muzyka | m\|u\|z\|y\|k\|a |
| Latin | prōvincia | p\|r\|ō\|v\|i\|n\|s\|i\|a |
| English | province | p\|r\|ɒ\|v\|i\|n\|s\|-\|- |
| German | Provinz | p\|r\|o\|v\|i\|n\|c\|-\|- |
| Polish | provincja | p\|r\|o\|v\|i\|n\|c\|j\|a |

(b) The "wide format".

| WORD.LAT | ALIGNED.LAT | WORD.ENG | ALIGNED.ENG |
|---|---|---|---|
| mūsica | m\|ū\|s\|i\|k\|a | music | m\|jū\|z\|i\|k\|- |
| prōvincia | p\|r\|ō\|v\|i\|n\|s\|ia | provnice | p\|r\|ɒ\|v\|i\|n\|s\|-\|- |

| WORD.GER | ALIGNED.GER | WORD.POL | ALIGNED.POL |
|---|---|---|---|
| Musik | m\|u\|z\|ī\|k\|- | muzyka | m\|u\|z\|y\|k\|a |
| Provinz | p\|r\|o\|v\|i\|n\|c\|- | prowincja | p\|r\|o\|v\|i\|n\|c\|j\|a |

**Fig. 2**. Sample data file.

The `ALIGNED` column contains words divided into segments using a fixed separator (`"|"`, by default). Segments can be simply single graphemes, but in some cases it may be more useful to separate entire morphemes or affixes into individual segments. It is necessary that all words in each pair / triple /… have the same number of segments and that the corresponding segments are aligned, though each segment can be composed of a different number of characters. Linguistic zeros (see sec. 2.1) can be used to create empty segments that are

required to preserve the alignment. For example, if the German word *Junker* 'landowner' was rendered in Polish as *junkier* 'Prussian landowner, …' (de Vincenz − Hentschel, 2010), this can be encoded, e.g., as `j|u|n|k|e|r : j|u|n|ḱ|e|r`, but also `j|u|n|k|-|e|r : j|u|n|k|j|e|r` etc., depending on the preferred phonological interpretation. Segmentation and alignment can be either performed manually, or using one of the automated tools, such as *alineR*, *LingPy*, or *PyAline* (Downey – Sun − Norquest, 2017; List – Greenhill − Forkel, 2018; Huff, 2010). It is, however, recommended that their results be thoroughly inspected by a human, if for no other reason than to allow the researcher to acquaint themselves with the material and its specificity. *soundcorrs* only offers a very simple function `addSeparators()`, which intersperses a vector of words with a separator character, providing a convenient starting point for manual alignment. As was mentioned in sec. 2.1, segmentation does not necessarily need to be phonetic; it can follow morphology, or any other kind of boundaries.

The second column, `LANGUAGE`, contains the name of the language from which the given word has been taken.

Data files can contain any number of additional columns, e.g., for comments, references to sources, etc. Single rows in them can be hidden from *soundcorrs* by placing a number sign (#) at the beginning of the line. A `soundcorrs` object can also be subsetted using the function `subset.soundcorrs()`.

To allow a greater degree of flexibility, data from various languages are read in individually, using the `read.scOne()` function, which requires four arguments: the path to the file, the name of the language, the name of the column with the aligned words, and the path to the transcription file. If the segment separator is different from the default `"|"`, it should also be specified. Objects returned by `read.scOne()` can then be merged into a single `soundcorrs` object (see sec. 4). It is perfectly possible to create a `soundcorrs` object out of data for a single language, read the data out of a single file, only using different columns as the designated 'aligned' column.

It is not advised to store data from different languages in separate files, but it is possible. In such case, care needs to be taken to avoid conflicts between column names, and it is required that words from each language are recorded in the same order, or that each file contains a column with matching IDs (this column must have the same name in all the files). It is still recommended that the final, merged dataset be inspected before analysis.

## 3 Analysis

The *soundcorrs* library provides tools for linguistic analysis on the spectrum ranging from the traditional, purely qualitative approach to the more recent, statistical and wholly quantitative perspective. For easier orientation, they have been divided here into three uneven groups: the qualitative approach in sec. 3.1, the intermediate one in 3.2, and the quantitative one in 3.3.

Only the more important functions (cf. Tab. 1) are discussed in detail. Each description in sec. 3.1 and 3.2 is followed by a very basic example. The functions discussed in 3.3 cannot be illustrated so succinctly. Examples of their usage, as well as more complex examples of functions explained in sec. 3.1 and 3.2, are included within a sample *soundcorrs* session, which is presented in sec. 4. Further examples, as well as the documentation of all of *soundcorrs*' functions are available in the vignette (run `vignette("soundcorrs")`), and through the in-built help (run `?NameOfTheFunction`).

### 3.1 The qualitative approach

Let us begin with three entirely qualitative functions: `findPairs()`, which looks for

examples of specific sound correspondences, `summary()`, which describes a single language or a dataset as a whole, and `allPairs()`, which produces a formatted listing of the entire dataset.

*

In practice, `findPairs()` is perhaps the most frequently used of all *soundcorrs* functions . What it does is it searches – in a dataset of two languages – for pairs of words which exhibit a specific sound correspondence. The function has three obligatory arguments: `data`, which is the dataset to be searched (a `soundcorrs` object), and `x` and `y`, which are the sounds to look for. It also has two optional arguments: `exact`, and `cols`.

The function operates in two modes: the exact mode (when the argument `exact` is set to `TRUE`), and the inexact one (when it is set to `FALSE`, or just omitted). In the exact mode, the comparison is performed on a strict segment-to-segment basis: `x` must occupy exactly the same segments as `y`, and both must be the entire segments. In the inexact mode, `x` is allowed to start or end one segment earlier or later than `y`, and they can be just parts of the specific segments. Let us use the following pair as an example: (`a|bc`, `ab|c`). In the exact mode, such a pair will only be matched in two cases (not counting queries with regular expressions): if `x=="a"` and `y=="ab"` – or if `x=="bc"` and `y=="c"`. The inexact mode, being more liberal, will also match this pair when `x=="a"` and `y=="a"`, when `x=="ab"` and `y=="a"`, and in several more cases. In addition, in the exact mode, linguistic zeros count as any other character would, while in the inexact mode, they are entirely disregarded.

In both modes, both `x` and `y` can be regular expressions: as provided by R (the default, 'extended' type, not Perl-like), or as defined in the transcription (see sec. 2.1). For example, to find all cases where *a* : *a* or *e*, one might run `findPairs (data, "a", "[ae]")`, and to find all cases of diphthongization in general, one might first define `V` to represent 'any vowel or semivowel', and then run `findPairs(data, "V", "VV")`. It should be noted that searching is performed on whole words, so, e.g., `findPairs(data, "a", ".*")` will cause *a* in the first language to be compared to the entire word in the second language – and will therefore only rarely return a match. To find all pairs in which one of the words contains a specific segment, regardless of what it corresponds to in the other word, `x` or `y` should be an empty string; for example, to find all pairs in which there is an *a* in the first language, without checking what it corresponds to in the second, one needs to run `findPairs(data, "a", "")`.

The function that translates metacharacters, as defined in the transcription, to regular expressions can also be used outside of `findPairs()`. It is called `expandMeta()`, and it takes two arguments: a `transcription` object, and the string to be transcribed. This first argument is necessary, but should it prove cumbersome in practice, a wrapper function can be defined as follows (assuming `ipa` is a `transcription` object): `expandIpa <- function(x) expandMeta(ipa, x)`.

The core of the return value of `findPairs()` is a subset of the provided data frame, which contains the matching pairs. By default, it is limited to only two columns which hold the aligned words, but this can be customized using the `cols` argument. It needs to be emphasized that pairs are only included once in the result, even when the specified correspondence appears multiple times in them (as, e.g., *lo* : *lo* in the German word *Haplologie* < Greek/Latin). For this reason, the number of rows in the result may be lower than the total number of occurrences of the given sound correspondence. The latter can be obtained using the function `summary()` [see below].

Technically, the return value of `findPairs()` is a list of class `df.FindPairs`. The

mentioned subset is stored in the field named `data`; the `found` field holds a data frame with the exact positions of the matching segments; and the field named `which` is a vector of logical values which can be used to turn the output of `findPairs()` into a new `soundcorrs` object, as shown in sec. 4.

Example:

```
findPairs (sampleSoundCorrsData.abc, "a", "[ou]")
```

will look for all pairs in which L1 *a* : L2 *o* or L2 *u*, and print:

```
  ALIGNED.L1 ALIGNED.L2
3      a|b|c      o|b|c
4    a|b|a|c    u|w|u|c
```

*

The `summary()` function provides a more general overview by producing a contingency table of all the segment correspondences attested in a dataset. The default layout is with segments from the first language (= *L1*) in rows, and segments from the second language (= *L2*) in columns. The values represent in how many words the given correspondence occurs in the dataset. Thus, for example, to see all the renderings that L1 *a* has in L2, one would either issue `summary(data)` and look for the row named *a*, or run `summary(data)["a",]` and have only this row printed. And conversely, to see all the L1 sounds which yield L2 *a*, one would run `summary(data)[,"a"]`. (Assuming that *a* is always separated into an individual segment.)

The direction of the table can be modified using the argument `direction`. The default value is `1`, which is the "*x* yields *y*"-perspective, while `2` stands for "*y* stems from *x*". Another argument `summary()` can take is `unit`. This defines whether the values in the table represent the number of times, or the number of words in which the given correspondence occurs, i.e. whether *lo* : *lo* in *Haplologie* is counted twice or once, respectively. The accepted values are: `"o"`, `"occ"`, `"occurrence"`, `"occurrences"`, and `"w"`, `"wor"`, `"word"`, `"words"` (the default). Lastly, the argument `count` determines whether values in the table are absolute or relative – with relation to the entire row. The values it accepts are: `"a"`, `"abs"`, `"absolute"`, and `"r"`, `"rel"`, `"relative"`.

Example:

```
summary (sampleSoundCorrsData.abc)
```

will print a contingency table of segment-to-segment correspondences:

```
    L2
L1   a b c ə o u w
  - 0 0 0 2 0 0 0
  a 4 0 0 0 1 1 0
  b 0 5 0 0 0 0 1
  c 0 0 6 0 0 0 0
```

*

The function `allPairs()` combines `findPairs()` with `summary()` – and a little automation, in order to produce a nicely formatted digest of the entire dataset. Its output is very similar to the material part of many a work dealing with loanword adaptation or sound correspondences in general, such as Pekaçar (2006) or Pomorska (2018). It is divided into sections, one for each segment attested in either L1 (the default), or L2. Sections open with a table with counts of all the renderings of the given segment; they are followed by subsections, each listing all the pairs with the given rendering.

Like in `summary()`, the perspective can be switched by changing the `direction`

argument from the default `1` ("$x \rightarrow y$") to `2` ("$y \leftarrow x$"). Likewise, values in tables can represent either the number of words, or the number of occurrences of the given correspondence, and can either be absolute or relative (arguments `unit` and `count`). The listings of pairs are taken from the output of `findPairs()`, which means that by default, they are the aligned columns `ALIGNED.x` and `ALIGNED.y`, containing both linguistic zeros and separators (see sec. 2 above). While both can be easily removed in any text editor or word processor, it may be more convenient to use the `cols` argument to fine-tune the output of `allPairs()`, in the same way as it is used with `findPairs()`. By default, `allPairs()` will print to the screen, but it can be made to write to a file by setting the `file` argument to the desired path in place of `NULL`.

The output of `allPairs()` is formatted by a specialized function, defined through the `formatter` argument. The *soundcorrs* library provides three such functions: `formatter.none()`, which does almost no formatting at all (the default), `formatter.html()`, which outputs HTML code, and `formatter.latex()`, which returns LaTeX code. For users of LibreOffice, Microsoft Word, or another word processor, HTML may prove the most convenient option, as it can be opened in any web browser and simply copied to the processor without losing the formatting.

The provided formatters are not customizable, but it is not too difficult to write a custom one using one of them as a template (see sec. 4). Such a function needs to take at least three arguments: `what`, `x`, and `direction`. The last one is simply `1` or `2`, the middle one is the data sent by `allPairs()`, and `what` defines the type of `x` as `"section"`, `"subsection"`, `"table"`, or `"data.frame"`. Additional arguments can also be used, and will be sent to the formatter function directly from the call to `allPairs()`.

Example:

```
allPairs (sampleSoundCorrsData.abc)
```

will print all segment-to-segment correspondences, with only the most basic formatting:

```
section [1] "-"
table   ə
table   2
subsection   [1] "-" "ə"
data.frame     ALIGNED.L1 ALIGNED.L2
data.frame   5    a|b|c|-    a|b|c|ə
data.frame   6  a|b|a|c|-  a|b|a|c|ə
```

etc.

## 3.2 The intermediate approach

Next, functions which stand halfway between qualitative and quantitative linguistics – i.e., those which use statistical methodology, but return results which describe qualitative features – will be presented. Four functions are discussed in this subsection: `table()`, which builds contingency tables of sound correspondences; `findSegments()`, which creates metadata for use with `table()`; `binTable()`, which collapses tables to single correspondences; and `allTables()`, which acts as a wrapper for the two, with additional functionality.

*

First, the `table()` function will be discussed; as its name suggests, it generates contingency tables. Unlike `summary()`, however, it cross-tabulates not segments, but correspondences – with themselves or with metadata.

The default mode is the former. It is invoked by setting the argument `column` to `NULL`.

The output is a table in which both rows and columns are sound correspondences, and the values represent the number, or the percentage of times or of words in which they co-occur. The names of rows and columns are composed from segments and separated with an underscore, so, e.g., L1 *a* : L2 *e* would be notated `a_e`.

The other mode is invoked when the argument `column` is set to the name of one of the columns in the dataset. (As was mentioned in 2.2, the data are internally stored in the "wide format", i.e., with suffixes appended to column names, unless the `skip` argument was used with `read.soundcorrs()`.) The output in this mode is very similar, only columns hold the metadata from the indicated column, instead of correspondences. For `table()`, it is irrelevant whether the metadata are numeric, or categorical.

As in `summary()` [sec. 3.1 above], the units and the direction can be changed using the arguments `unit` and `direction`. They accept the same values, and the defaults are likewise `"w"` and `1`. Similarly, `table()` also takes the argument `count`, which accepts the same values and defaults to `"a"`. The difference is that its mode of operation with `summary()` was only a special case. As a general rule, the table is always divided into blocks: in the external mode, those blocks are made up of rows which share the same initial segment; in the internal mode, they are the intersection of rows which share the same initial segment, and of columns which share the same initial segment. For example, in `summary()`, each row summed up to 1; in the external mode of `table()`, all rows beginning with `"a_"` will sum up to 1, as will all rows beginning with `"b_"`, `"c_"`, etc.; on the other hand, in the internal mode of `table()`, the rectangle made of all rows beginning with `"a_"` and all columns beginning with `"b_"` will sum up to 1, while rows beginning with `"a_"` in their entirety will sum up to more – if, that is, L2 has more segments than just *b*. The reason for the distinction between absolute and relative counting is more clear with `table()` than it was with `summary()`. On their own, absolute numbers can be very misleading; for example, if it is found that L1 *a* : L2 *e* co-occurs multiple times with L1 *o* : *ö* but only rarely with L1 *u* : L2 *ü*, the reason may be that whatever palatalizing factor was present in those words, it does not affect *u*; but it may also be that *a* and *u* almost never appear in the same words in L1. With relative counting, the output may contain empty places. This means that the two segments just never appear together; their relative frequency is 0/0, which R represents as `NaN` and, in a table, prints as an empty space.

It should be noted that in the correspondence-to-correspondence mode, co-occurrence with itself is also counted. Values in tables produced in this mode may thus not be immediately understandable, especially if `unit` is set to `"o"`, and one or more of the correspondences appear multiple times in a single word (as in *Haplologie*). Let us consider two pairs of words: L1 *abc* : L2 *abc*, and L1 *aba* : L2 *aba*. In *abc*, we have three combinations: (*a*:*a*, *b*:*b*), (*a*:*a*, *c*:*c*), and (*b*:*b*, *c*:*c*). As shown in Fig. 3a, this would be represented as a 3×3 table (to accommodate all the three different combinations), filled with 1's (because each combination only appears once). With *aba*, the situation is in essence the same. We have three combinations: (*a*:*a*, *b*:*b*), (*a*:*a*, *a*:*a*), and (*b*:*b*, *a*:*a*), and would likewise use a 3×3 table filled with 1's. But here, because the first and last row and column are the same, we can simplify the table and combine the two *a*-rows and the two *a*-columns by simply adding them together, as is demonstrated in Fig. 3b. Hence, there are two co-occurrences of L1 *a* : L2 *a* with L1 *b* : *b* – ($a_1$ : $a_1$, *b* : *b*), and (*b* : *b*, $a_2$ : $a_2$) – and hence, there are four co-occurrences between L1 *a* : L2 *a* and itself.

(a) L1 *abc* : *abc*

| | *a*:*a* | *b*:*b* | *c*:*c* |
|---|---|---|---|
| *a*:*a* | 1 | 1 | 1 |
| *b*:*b* | 1 | 1 | 1 |
| *c*:*c* | 1 | 1 | 1 |

(b) L1 *aba* : *aba*

| | *a*:*a* | *b*:*b* | *a*:*a* |
|---|---|---|---|
| *a*:*a* | 1 | 1 | 1 |
| *b*:*b* | 1 | 1 | 1 |
| *a*:*a* | 1 | 1 | 1 |

→

| | *a*:*a* | *b*:*b* | *a*:*a* |
|---|---|---|---|
| *a*:*a* | 2 | 2 | 2 |
| *b*:*b* | 1 | 1 | 1 |

→

| | *a*:*a* | *b*:*b* |
|---|---|---|
| *a*:*a* | 4 | 2 |
| *b*:*b* | 2 | 1 |

**Fig. 3.** Counting the number of co-occurrences with `table()`.

Example:

```
table (sampleSoundCorrsData.abc)
```

will print a correspondence-to-correspondence contingency table:

```
      L1→L2
L1→L2 -_ə a_a a_o a_u b_b b_w c_c
  -_ə   2   2   0   0   2   0   2
  a_a   2   4   0   0   4   0   4
  a_o   0   0   1   0   1   0   1
  a_u   0   0   0   1   0   1   1
  b_b   2   4   1   0   5   0   5
  b_w   0   0   0   1   0   1   1
  c_c   2   4   1   1   5   1   6
```

*

The metadata used with `table()` can be virtually anything, including phonetics. The *soundcorrs* library provides the function `findSegments()` to help make use of this kind of data. In short, it generates a list of segments preceding or following the matches found by `findPairs()` [sec. 3.1]. It takes four arguments: `data`, `x`, and `y` – just like `findPairs()`, and, in addition, `segment`, which determines which segment to extract, in relation to the segments which realize the L1 *x* : L2 *y* correspondence. For example, to extract the segments which directly precede L1 *a* : L2 *e*, one would run `findSegments(data,"a","e",-1)`.

The output of `findSegments()` is a list of two vectors: one for segments taken from L1, and the other for those from L2. Both vectors are of the same length as the original dataset so as to be easily attachable to it: `data.new <- cbind(data, BEFORE.A.E=findSegments(data,"a","e",-1)$L1)`. Naturally, not every pair in the dataset must necessarily realize the *a* : *e* correspondence; those that do not are represented as `NA`'s. The lists produced by `findSegments()` can also be translated into phonetics using the function `char2value()`; an example of this is given in sec. 4.

Example:

```
findSegments (sampleSoundCorrsData.abc, "a", "u", 1)
```

will find segments that directly follow the L1 *a* : L2 *u* correspondence (cf. the example for `findPairs()` in sec. 3.1):

```
$L1
[1] NA    NA    NA    "c,b" NA    NA
$L2
[1] NA    NA    NA    "c,w" NA    NA
```

*

One of the possible uses for a contingency table is a test of independence. However, some of the most popular ones require that the sample be relatively large, and in linguistics, such data may not always available. The `binTable()` function attempts to alleviate this problem by selecting from a table only those rows and columns which are to be investigated, and combining (summing) all the others so that the initial table is reduce, as illustrated in Fig. 4. `binTable()` takes three arguments: `x`, which is the table or matrix to be collapsed, and `row` and `col`, which are the numbers of the rows and columns that are to be spared. The latter two can be single integers, or vectors of integers.

A side effect of this procedure is that the binned table contains one comparison where previously there were many, which makes it possible to ask more specific questions. One just needs to be careful to make sure that binning of given rows or columns makes sense from the linguistic point of view. For example, it may be reasonable to wish to compare how often L1 *a* : L2 *e* coincides with L1 *o* : L2 *ö*, versus all the other possible renderings of L1 *a* and *o*, but it will take quite specific circumstances to justify cross-tabulating these two correspondences against, e.g., all the other correspondences combined, including all the consonants, suffixes, and whatever else may have been separated into its own segment in the dataset.

| | *o*:*o* | *o*:*ö* |
|---|---|---|
| *a*:*a* | 10 | 1 |
| *a*:*e* | 2 | 10 |
| *a*:*o* | 3 | 0 |

→

| | *o*:*o* | non-*o*:*o* |
|---|---|---|
| *a*:*a* | 10 | 1 |
| non-*a*:*a* | 5 | 10 |

…

| | *o*:*ö* | *non-o*:*ö* |
|---|---|---|
| *a*:*o* | 0 | 3 |
| non-*a*:*o* | 11 | 12 |

**Fig. 4.** Binning of a 2×3 table into 2×2 tables.

Example:

```
binTable (table(sampleSoundCorrsData.abc), row=7, col=6)
```

will collapse the table that we saw above in the example for `table()` to its last but one cell:

```
          b_w non-b_w
c_c         1      19
non-c_c     2      46
```

*

The `allTables()` function automates the use of `table()` and `binTable()`, and generates a list of all contingency tables for the given dataset. The result has a form that can be easier to read for a person, but its primary intended use is to make easier the application of tests of independence – for which task the function `lapplyTest()` [see below] can be employed.

The function takes several arguments: `data` is, as usual, the dataset; `column`, `unit`, `count`, and `direction` work as with `table()` above; in addition, `bin` determines whether to bin through all the produced tables (defaults to `TRUE`), or whether to content itself with slicing the general contingency table (as produced by `table()`) into blocks devoted to single segments.

The return value of `allTables()` is a list containing all the resulting tables. It is named using the same logic as with `table()`. Specifically, if `bin` is `FALSE`, the names will be simply the segments attested in L1 or L2, depending on the value of `direction`; and when `bin` is `TRUE`, they will be composed of correspondences and values taken from `column` or, if that is `NULL`, correspondences again, all separated by underscores. For example, `allTables(data,"DIALECT")$a_e_D1` will hold the table for L1 *a* : L2 *e* with dialect D1, and `allTables(data)$a_e_o_ö` for L1 *a* : L2 *e* with L1 *o* : L2 *ö*. (Or with the languages swapped, if `direction=2`.) Cross-tabulations of correspondences with themselves are skipped, that is, e.g., L1 *a* : L2 *a* would be compared with the correspondences of L1 *b*, *c*, etc., but not with those of L1 *a* itself, which is why, e.g., the field `$a_a_a_o` (L1 *a* : L2 *a* × L1 *a* : L2 *o*) will be missing from the result.

Example:

```
allTables (sampleSoundCorrsData.abc)
```

will generate a binned table for each cell of the table we saw in the example for `table()` above – except, as explained above, for the mutually exclusive ones:

```
$`-_ə_a_a`
    a_a non-a_a
-_ə   2       4
$`-_ə_a_o`
    a_o non-a_o
-_ə   0       6
```

etc.

*

Another function is `lapplyTest()`, which applies a function to a list. The main difference between it and regular `lapply()` is the handling of warnings and errors. Its main intended use is to apply a test of independence to a list of contingency tables, such as produced by `allTables()`.

This function can take two or more arguments: `x`, which is the list of tables, `fun`, which determines what function is to be applied (the default is `chisq.test`), as well as all additional arguments to that function.

The return value is a list of the outputs of `fun`. It is of the `list.lapplyTest` class, so it can be passed to `summary()` to be turned into a brief overview of the results. In the report printed by `lapplyTest()`, only results below a specific *p*-value are included, with the default being 0.05. (It is for this reason that the return value of `fun` must contain an element named `p.value`, as it is from this field that the *p*-values are extracted. If the desired function has an incompatible return value, it will be necessary to write a wrapper around it). An exclamation mark at the beginning of a line in the output means that `fun` returned a warning. The specific message is attached to the given element of the list as an attribute named

`"warning"`. If `fun` returned an error, the return value is a list with an attribute `"error"`.

The list returned by `lapplyTest()` preserves the naming scheme of the list passed to it as `x` so, for example, in order to see the result of the $\chi^2$ test applied to the contingency table of L1 *a* : L2 *e* and L1 *o* : L2 *ö*, one can run

```
lapplyTest(allTables(data))$a_e_o_ö,
```

and to see if it produced a warning –

```
attr(lapplyTest(allTables(data))$a_e_o_ö, "warning").
```

In the case of the default chi-squared test, the message "Chi-squared approximation may be incorrect" often indicates insufficient data, and may be helped by the application of binning in the `allTables()` function.

With `allTables()` and `lapplyTest()`, as opposed to `table()` above, attention needs to be paid to whether the chosen test is compatible with the metadata (i.e., the values of the columns). Contingency tables are primarily used for categorical data, such as the names of the consultants who provided the given pronunciations, or the dialects they spoke. Numeric data, such as the year when the given word was recorded, may require an entirely different approach, perhaps one from beyond what is offered directly by *soundcorrs*.

Example:

```
res <- lapplyTest (allTables(sampleSoundCorrsData.abc))
```

will apply the $\chi^2$ test to all the tables we saw in the example for `allTables()` above, and store the result in a variable called `res`. Then

```
summary (res)
```

will display a brief summary:

```
Total results: 34; with p-value ≤ 0.05: 7.
! -:ə with a:o: p-value = 0.014
! -:ə with a:u: p-value = 0.014
! -:ə with b:w: p-value = 0.014
  c:c with -:ə: p-value = 0.008
  c:c with a:o: p-value = 0.001
  c:c with a:u: p-value = 0.001
  c:c with b:w: p-value = 0.001
```

and

```
res$c_c_b_w
```

will display the result for the table we saw in the example for binTable() above:

```
   Chi-squared test for given probabilities
data:  tab
X-squared = 10.286, df = 1, p-value = 0.001341
```

### 3.3 The quantitative approach

Lastly, let us examine the three functions that will be mostly useful to quantitative linguists: `ngrams()`, which produces a table with counts of *n*-grams, and two functions which fit multiple models to one or more datasets: `multiFit()` and `fitTable()`.

*

Let us begin with `ngrams()`, a simple function that extracts *n*-grams or, more accurately, *n*-segments. The first argument is a `scOne` object (not a `soundcorrs` one, as is the case with

nearly all the functions discussed here); the second is `n`, the length of subsequences to be extracted. Its default value is `1`, in which case `ngrams()` produces simply a frequency list of all segments; if it is larger than the number of segments in one of the words, that word is ignored in the final calculation. The third argument is `zeros`, which determines whether linguistic zeros (see sec. 2.1) are to be included (defaults to `TRUE`); the fourth is `as.table`, about which see below.

The return value of `ngrams()` are absolute counts of *n*-grams in the data for one language. The default format is a table. A table is legible, and it can be converted quite easily into a data frame with ranks (see the vignette; `vignette("soundcorrs")`), but it cannot be cross-tabulated with another language. For this purpose, the `as.table` argument can be set to `FALSE` to make `ngrams()` return the result in the form of a list; see an example of this in sec. 4. Note that cross-tabulation is only possible when the lists for both languages have the matching number of *n*-grams for each word – which is an alignment that setting the argument `zeros` to `FALSE` may destroy.

*

The function `multiFit()` fits multiple models to a single dataset. The first argument is a list of models. Each of its elements needs to contain two named fields: `formula`, and `start`. The latter contains the starting estimates for the fitting function. It is possible to include several sets of estimates, but even when there is only one, `start` needs to be a list of lists [e.g., `list(list(a=1))`]. The second argument is the dataset, in the form of a data frame or a list, or potentially any other that the fitting function accepts (see below). The column names in the dataset must correspond to the names given in the formulae in `models`. The third argument is the fitting function. It defaults to R's built-in `nls()`, but functions from external packages, such as `nlsLM` (Elzhov et al., 2016), might prove to be more convenient, especially when it comes to the accuracy of the starting estimates. Lastly, `multiFit()` can take some additional arguments and pass them to the fitting function.

The return value of `multiFit()` is a list with the outputs of the fitting function. When fitting failed to produce a result, and `multiFit()` suppresses the printing of errors, the value is `NA`, and the error or the warning are attached to it as attributes (in the same way that `lapplyTest()` does, see sec. 3.2). Technically, the output is of the `list.multiFit` class, so that it can be passed as an argument to `summary()` to produce a table for a more convenient comparison of the results. The metric can be set using the argument `metric`; the available options are: `"aic"`, `"bic"`, `"rss"` (the default), and `"sigma"`.

*

Similarly to `multiFit()`, `fitTable()` fits multiple models, the difference being that it fits them to multiple datasets. The first argument, `models`, is the same. The second, `data`, requires a matrix or a table, such as the ones produced by `summary.soundcorrs()` or `table.soundcorrs()` [sec. 3.1 and 3.2, respectively]. The third argument is `margin`, and it is nearly the same as with `apply()`: `1` for rows, or `2` for columns. The fourth argument, `conv`, is a function that will be applied to `data` in order to turn individual rows or columns (i.e., vectors) into data frames. The *soundcorrs* library offers three such functions: `vec2df.id` (only adds a column of subsequent numbers starting with 1; the default choice), `vec2df.hist` (creates a data frame from midpoints and counts extracted from a histogram), and `vec2df.rank` (sorts the data and adds a column with ranks). A custom function can be defined quite easily; it needs to take exactly one argument, a numeric vector, and return

whatever format the fitting function can accept. The three converters provided by *soundcorrs* return data frames with two columns named `X` and `Y`. The custom function does not have to follow this convention, but the names must correspond to the variables given in the formulae in the `models` argument. To this, additional arguments can be added, which `fitTable()` will pass on to `multiFit()` [this includes the fitting function], and which `multiFit()` will then pass on to the fitting function.

The return value of `fitTable()` is a nested list with the outputs of the fitting function, or `NA`'s. As with `multiFit()`, its class is `list.multiFit`, so it can be passed to `summary()` to generate a table for convenient comparison.

## 4 Example

Now, let us put all of the above into practice. The *soundcorrs* package contains two sample transcription files and three sample datasets. The transcription files are named `trans-common.tsv` and `trans-ipa.tsv`, and contain parts of the common tradition of linguistic transcriptions (as used in the Americanist phonetic notation, the Finno-Ugric transcription, and most others) and of the International Phonetic Alphabet. Neither are full, as they are intended only as samples, based on which users will be able to craft a set specifically to their needs. The sample datasets are `data-abc.tsv`, `data-capitals.tsv`, and `data-ie.tsv`. The first is entirely fabricated; the second contains the names of EU capitals in German, Polish, and Spanish[2] (linguistically, of course, it has no reason to be, for methodological reasons; it is only meant to serve as an example that stands on the common ground of a highly specialized field); lastly, the third contains a dozen words showcasing the Grimm's and Verner's laws (adapted from Campbell, 2013: 136f). Here, we will mostly use "abc", and leave the other two for the user to explore.

The three datasets are preloaded as `sampleSoundCorrsData.abc`, `sampleSoundCorrsData.capitals`, and `sampleSoundCorrsData.ie`, but here, we will read them from files. Let us assume that R and *soundcorrs* are installed, and begin by loading *soundcorrs* and the data. The paths to the sample files can be found using `system.file()`.

```
# Load soundcorrs.
#     The warning is correct, but no cause for alarm.
library (soundcorrs)

# Find the path to a sample transcription.
path.trans.com <- system.file ("extdata", "trans-common.tsv",
     package="soundcorrs")
path.trans.ipa <- system.file ("extdata", "trans-ipa.tsv",
     package="soundcorrs")

# Find the paths to the two sample datasets.
path.abc <- system.file ("extdata", "data-abc.tsv",
     package="soundcorrs")
path.ie <- system.file ("extdata", "data-ie.tsv",
     package="soundcorrs")

# The "ie" set is in the wide format, it can be read as it is.
```

2 I would like to express my gratitude to José Andrés Alonso de la Fuente, Ph.D. (Cracow, Poland), for his help with the Spanish data.

```
#     Different languages can use different transcriptions.
#     Regarding the warnings, see sec. 5.
d.ie.lat <- read.scOne (path.ie, "Lat", "LATIN", path.trans.com)
d.ie.eng <- read.scOne (path.ie, "Eng", "ENGLISH", path.trans.ipa)
d.ie <- soundcorrs (d.ie.lat, d.ie.eng)

# The "abc" set needs to be first converted to the wide format.
#     The "ID" column refers to pairs as a whole,
#     so it will not be converted.
tmp <- long2wide (read.table(path.abc,header=T), skip=c("ID"))
d.abc.l1 <- scOne (tmp, "L1", "ALIGNED.L1",
     read.transcription(path.trans.com))
d.abc.l2 <- scOne (tmp, "L2", "ALIGNED.L2",
     read.transcription(path.trans.com))
d.abc <- soundcorrs (d.abc.l1, d.abc.l2)
```

The calls to `read.scOne()` and `scOne()` will cause *soundcorrs* to show warnings about certain segments in both datasets not being covered by the transcription (cf. sec. 5). Since we will not be performing an in-depth phonetic analysis here, these warnings can be safely ignored. To inspect the loaded data, one can simply run `d.abc` and `d.ie`, which will print a brief summary, or to see all the individual examples, `d.abc$data`, and `d.ie$data`.

Thus prepared, let us proceed to simulate a brief working session with *soundcorrs*, including all the analytic functions, though not in the same order in which they were discussed in sec. 3.

```
# First let us prepare the material part of the paper,
#     printing all words in the appropriate orthography
#     rather than in the working, segmented form,
#     and format the output in HTML.
allPairs (d.abc, file="~/Desktop/abc.html",
     cols=c("ORTHOGRAPHY.L1","ORTHOGRAPHY.L2"),
     formatter=formatter.html)

# Now, let us see a general overview of the "abc" dataset as a whole.
summary (d.abc)

# Does counting words and occurrences make
#     a considerable difference?
summary (d.abc, unit="w") # same as above, since "w" is the default
summary (d.abc, unit="o")

# Let us take a closer look at "a" because this seems to be
#     the most complex one.
summary (d.abc, unit="o") ["a", ]

# Does it seem that the rendering of "a" may be
#     tied to some piece of metadata?
#     Let us create a convenience variable for column names.
myCols <- c("ALIGNED.L1","ALIGNED.L2","DIALECT.L2")

# Let us see the rounded correlates.
findPairs (d.abc, "a", "O", cols=myCols)

# Do there appear to be any regularities?
```

```
table (d.abc, unit="occ")
round (table (d.abc, unit="occ", count="rel"), 3)

# Perhaps "a" with "b". Let us only see this part of the table.
tab <- table (d.abc, unit="occ")
rows <- which (rownames(tab) %hasPrefix% "a")-->
cols <- which (colnames(tab) %hasPrefix% "b")-->
tab [rows, cols]

# The number of examples is low, but the result seems promising.
chisq.test (tab[rows,cols])

# Will we be able to find any more?
tabs <- allTables (d.abc)
chisq <- lapplyTest (tabs)
summary (chisq)

# Unfortunately, L1 - (= linguistic zero) and L1 c
#     only correspond to L2 ə and c, so these results
#     provide little insight.
tabs$`-_ə_a_o`
tabs$`c_c_-_ə`

# Considering that this is only an example, and the number of
#     examples is very low, let us be generous.
chisq <- lapplyTest (tabs)
summary (chisq, p.value=0.3)
chisq$a_u_b_w
attr (chisq$a_u_b_w, "warning")
tabs$a_u_b_w

# Now let us look at the cases of apocope.
#     'exact' must be explicitly set to 'TRUE' because in the
#     default inexact mode, linguistic zeros (here '-') are ignored.
findPairs (d.abc, "-", "", cols=myCols)
findPairs (d.abc, "-", "", cols=myCols, exact=T)

# Does it appear in all the "southern" pairs?
apocopes <- subset (d.abc, findPairs(d.abc,"-","",exact=T)$which)
southern <- subset (d.abc, DIALECT.L2=="south")
identical (apocopes, southern)

# Is there any particular environment in which it appears?
#     "NA"'s mean that the word does not exemplify the given
#     correspondence – which cannot be the case here, since
#     we are using the "apocopes" set –, or that the segment that
#     is looked for falls outside of the word – as with the second
#     command here.
findSegments (apocopes, "-", "", -1)
findSegments (apocopes, "-", "", +1)

# The output can be easily turned into its phonetic value.
before.apocope <- findSegments (apocopes, "-", "", -1)
char2value (apocopes, "L1", before.apocope$L1)

# Most correspondences in "d.abc" seem to be quite one-sided.
```

```
summary (d.abc)

# Let us see if they follow a simple power law.
models <- list (
"model A" = list( formula="Y ~ X^a", start=list(list(a=-1))),
"model B" = list( formula="Y ~ a*X^b", start=list(list(a=1,b=-1))))
fit <- fitTable (models, summary(d.abc), 1, vec2df.rank)
summary (fit)

# Now, let us see if any patterns can be found in n-grams.
bigrams.l1 <- ngrams (d.abc.l1, n=2, zeros=T, as.table=F)
bigrams.l2 <- ngrams (d.abc.l2, n=2, zeros=T, as.table=F)
table (unlist(bigrams.l1), unlist(bigrams.l2))
```

Several more examples, together with an alternative explanation of the workings of each function, can be found in *soundcorrs*' vignette, which is accessible via `vignette("soundcorrs")`. Further details about each function are available in the package documentation, which can be accessed via `?findPairs`, etc.

## 5 Errors and how to solve them

Below is a near-comprehensive list of warning and error messages displayed by *soundcorrs* (abbreviated to *W* and *E*, respectively), together with brief explanations of possible causes and recommended solutions.

One issue that may appear without a message is encoding. Tests under BSD, Linux, and macOS did not reveal any problems with UTF-8, but they did under Windows. Some issues could be helped using `iconv()` to convert data from UTF-8 to UTF-8 (sic!), but other problems proved to be more resilient. Since no complete solution could be found, and a partial one would be misleading, *soundcorrs* does not contain any mechanism at all to remedy this situation. It is recommended that under Windows, only plain ASCII characters be used. This is quite unfortunate, and a priority for future versions of *soundcorrs*.

**(E) At least two "scOne" objects are required.** A `soundcorrs` object can only be created for two or more languages. It is perfectly acceptable, however, to pass data from one language, just with different 'aligned' columns and different names, as different 'languages'.

**(E) Differing column names for different suffixes.** The columns defined for one language do not match the columns defined for the other language. See sec. 2.2 and inspect the data file for typos in column names.

**(E) Differing number of *X*.** The data from two languages do not match. This error may occur when one converts between the "long" and "wide formats", or when one combines `scOne` objects into a `soundcorrs` object. If `X` is `columns` or `entries`, see sec. 2.2; if it is `segments`, check the specified lines in the data file and make sure that both words in the pair are divided into the same number of segments.

**(E) Differing values between columns specified in "skip".** A column listed in the `skip` argument of the `long2wide()` function must have identical values for each pair / triple / … of words. See 2.2 and inspect the data file for any mismatches.

**(E) Extended regular expressions metacharacters are used as graphemes: …** Characters `. + * ^ \ $ ? | ( ) [ ] { }` have special meanings in *R*'s extended regular expressions, and they cannot be used as graphemes. See sec. 2.1 and replace them with

other characters in the transcription file.

**(E) Extended regular expressions metacharacters are used in the data: …** Characters `. + * ^ \ $ ? | ( ) [ ] { }` have special meanings in *R*'s extended regular expressions, and they cannot be used in the column with segmented words. See sec. 2.1 and replace them with other characters in the data file.

**(E) Incompatible datasets. Perhaps conflicting column names?** This error is shown when the user attempts to combine into a `soundcorrs` object two or more `scOne` objects with incompatible data in them. The reasons can be varied, e.g., mismatched column names or differing numbers of examples in the datasets. It is usually easier to spot inconsistencies when all the data are stored in the same file.

**(E) It is required that 0 < *row* ≤ nrow(*x*) and 0 < *col* < ncol(*x*).** This error is shown by `binTable()` when the argument `row` or `col` is beyond the limits of the specified table `x`. The number of rows and columns of `x` can be checked with `nrow(x)` and `ncol(x)`, respectively.

**(E) Linguistic zero is not defined in the transcription.** Some of *soundcorrs* functions, e.g. `findPairs()`, can only work correctly if the transcription defines a symbol to denote linguistic zero. Sec. 2.1 explains how to do it.

**(E) Linguistic zeros must be separate segments: …** In general, *soundcorrs* allows multiple characters within a single segment, but the character used to denote linguistic zero is an exception. Linguistic zeros can only fulfil their intended role in *soundcorrs* when they are isolated into separate segments of their own.

**(E) Multiple definitions for graphemes: …** The same character is defined more than once in the transcription file. The specified graphemes should be checked and redundant definitions removed.

**(E) One or more column names are missing from *X*.** (Variants: Column *Y* is missing from *X*.) The column names given as the argument to the function cannot be found in the given dataset. Inspect the line for typos, or check the available columns by running `colnames(X$data)`.

**(E) The specified "language" is missing from *X*.** The value of the argument `language` to the function `char2value()` is not compatible with the dataset *X*. To check the available names, simply run `X`.

**(E) This function does not know how to handle an object of class *X*.** The given function has only been defined for objects of the `scOne` or `soundcorrs` classes. If the user would like to define it for a different class, this should not collide with the working of *soundcorrs*.

**(E) Transcription metacharacters are used in the data: …** Characters defined in the transcription as metacharacters cannot be used in the column with segmented words. See sec. 2.1 and replace them with other characters in the data file.

**(E) *X* cannot be empty string or NA.** Argument `X` needs to be given a concrete value in order for the function to be able to perform its function. The user is directed to the documentation of the specific function (run `?NameOfTheFunction`).

**(E) *X* must be exactly one column name.** Only one column can be designated as the one that holds the segmented words. It is perfectly possible to have in a dataset multiple columns with segmented words, potentially each segmented according to a different set of rules, but they must be all read into separate variables.

**(E) *X* must be *Y*.** (Variants: *X* must be of class *Y*, *X* must refer to *Y*.) Argument `X` can only take values of certain type or from a limited range. That range may be specified in the warning message in full or, in the 'refer'-variant, in short. The user is directed to the documentation of the specific function (run `?NameOfTheFunction`).

**(W) Missing the metacharacters column. The "*X*" column was generated.** Leaving the generation of metacharacters to *soundcorrs* is the recommended method, but at the same time, it is always safest to check manually any content that has been generated automatically. Transcription is easy to find and explore – it is simply a data frame passed as the output of the `read.transcription()` function.

**(W) The following object is masked from 'package:base': table.** This should not interfere with the working of `table()` for all applications beyond *soundcorrs*.

**(W) The following segments are not covered by the transcription: …** The data contain segments that are not listed in the transcription. However, not all tasks that can be performed with *soundcorrs* require the transcription. As long as it is not explicitly made use of, this warning can be safely ignored. See 2.1.

**(W) This function only supports two languages.** As of version 0.1, some functions provided by *soundcorrs* only accept datasets of no more than two languages. This is planned to change in future releases.

## 6 Summary

The paper presents an R library by the name of *soundcorrs*. The goal of this package is to facilitate the analysis of sound correspondences between languages by automating the most tedious of tasks involved in this kind of investigation. In opposition to cladistics and other computerized methods that have gained a degree of popularity among linguists in recent years, *soundcorrs* is not intended to produce any conclusions, merely to extract information from a dataset while leaving interpretation entirely to the user.

This paper discusses in some detail the most important functions, the applications of which range from a purely qualitative approach to a more quantitatively-oriented one. It also presents a sample session with *soundcorrs*, and explains the meaning of warnings and errors issued by *soundcorrs*.

Plans for future versions of *soundcorrs* include: solution of the problem of encoding under Windows, implementation of support for multiple languages in those functions which currently only accept pairs of languages, addition of functions to simulate phonetic changes, and more. Users are asked to address all requests, as well as bug reports, to this author. If *soundcorrs* is used in published research, please cite this paper.

## References

**Altmann, G. & Fengxiang, F.** (2008). *Analyses of Script. Properties of Characters and Writing Systems* (= *Quantitative Linguistics* 63). Berlin – New York: Mouton de Gruyter.

**Campbell, L.** (2013). *Historical Linguistics. An Introduction*. Edinburgh: Edinburgh University Press.

**Downey, S. S. & Sun, G. & Norquest, P.** (2017). alineR: an R Package for Optimizing Feature-Weighted Alignments and Linguistic Distances. *The R Journal* 9(1), 138–152.

**Elzhov, T. V. & Mullen, K. M. & Spiess, A.-N. & Bolker, B.** (2016). *minpack.lm: R Interface to the Levenberg-Marquardt Nonlinear Least-Squares Algorithm Found in MINPACK, Plus Support for Bounds*. https://CRAN.R-project.org/package=minpack.lm.

**Huff, P.** (2010). *PyAline*. http://pyaline.sourceforge.net.

**List, J.-M. & Greenhill, S. & Forkel, R.** (2018). *LingPy. A Python library for historical linguistics*. http://lingpy.org.

**Pekaçar, Ç.** (2006). Kumuk Türkçesine Arapça ve Farsçadan Geçen Kelimelerdeki Ses Olayları. *Selçuk Üniversitesi Türkiyat Araştırmaları Dergisi* 19, 53–71.

**Pomorska, M.** (2018). *Russian Loanwords in the Chulym Turkic Dialects*. Kraków: Księgarnia Akademicka.

**R Core Team** (2019). *R: A language and environment for statistical computing*. Vienna: R Foundation for Statistical Computing. https://www.R-project.org/.

**de Vincenz, A. & Hentschel, G.** (2010). *Wörterbuch der deutschen Lehnwörter in der polnischen Schrift- und Standardsprache* (= *Studia Slavica Oldenburgensia* 20). Oldenburg: BIS-Verlag.