An Environment for Acoustic Phonetic Research

APS
User Guide

Jonathan M. Harrington
Gordon S. Watson

The Centre for Speech Technology Research
University of Edinburgh

APS An Environment for Acoustic Phonetic Research

Jonathan M. Harrington[1] Gordon S. Watson

July 11, 1990

[1]Now at the School of English and Linguistics, Macquarie University, Sydney, N.S.W. 2109, Australia.

The APS code contains three compiled FORTRAN-77 functions taken from a public domain library. The following is its copyright notice.

Preface

APS is a tool that enables acoustic phonetic experiments on speech databases. Suppose someone wanted to compute the average value of the first formant in the middle third of all /a/ phonemes that occur in the database, or suppose we require a histogram plot of the fundamental frequency of all back vowels in the database which follow oral stops and precede voiceless fricatives for two speakers. These are the kind of operations APS is designed to do.

The database that APS analyses must be structured in terms of label files and track files. The label files contain phonemic transcriptions of each utterance with a start time and an end time for each phonetic symbol. The track files contain analysis data which have been created by applying signal processing algorithms over digitised speech. APS enables statistical operations to be applied to any of the track files with respect to the boundaries of the phonetic symbols in the label files.

At, CSTR, the S language and APS run on four Sun 4 machines: Watt, Brodie, Scott and Phoenix. The database at CSTR has track files with values at 5ms intervals. There are various acoustic parameters such as *f1* (first formant), *syncspt* (fundamental frequency) and *E0-300* (energy in the 0 - 300 Hz band).

The tutorial chapters were written by Jonathan Harrington. I have updated them to reflect the changes to APS since since these were written in April 1989. The other chapters provide information for those wishing to install APS, how to make APS run on a new database, and documentation for all APS functions.

Gordon S. Watson
CSTR, July 1990

Contents

Chapter 1

What is APS?

Research into acoustic phonetics attempts to explain how the characteristic cues of specific phonemes can be recovered from the acoustic signal.

The relationship between phonetic segments and their acoustic values is complicated by a number of factors that cause variability. Variability can exist due to phonetic context (compare the acoustics of the /k/ in cool with the /k/ in *keel*). The position of the segment in a syllable or sentence also effects its acoustic characteristics. Variability exists between speakers and even for a single speaker when repeating the same syllable [1].

The complexity of the task is compounded by the fact that the acoustic characteristics of phonetic segments are defined by a time-series of multi-dimensional vectors.

1.1 Overview

In the face of these complexities a software environment called APS was designed with the following features

1. The ability to select tokens of speech which can be grouped in various ways according to the factors that cause variability.

2. A means of measuring the acoustic characteristics of each token.

3. Appropriate analysis apparatus.

APS (*A*coustic *P*honetics in *S*) consists of a set of extensions to the S system, a programming environment for data analysis and graphics [2]. All APS functions are written either in the S language and or in C and then

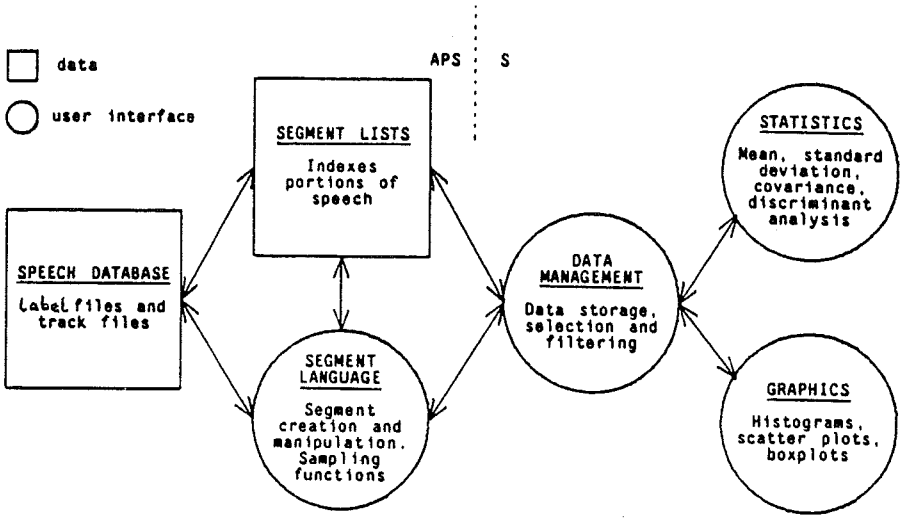

Figure 1.1: APS Functional Overview

integrated into the S system. Fig 1.1 gives a functional view of the system. Component parts are discussed in the sections below.

1.2 Speech Database

The speech database currently used by the APS package consists of a series of utterances, typically whole sentences of continuous speech of between 1.5 and 10 seconds in duration. Each utterance is stored in a pair of files called the label file and track file.

The label file contains a phonemic transcription made either by trained phoneticians using a segmentation station or by an automatic segmentation and labeling program. Each phonemic segment is assigned a label, a start time and an end time.

The track file contains output from a variety of digital signal processing operations, each of which is called a track. Examples of tracks used at CSTR are: energy in various frequency bands, formant frequency, the first moment of the power spectrum. Each track is stored as one value for each frame of data, where a frame is (at CSTR) 5 milliseconds in duration.

APS provides *a means of investigating how the phonemes, stored as segments in the label files, relate to the acoustics, stored as tracks in the track files.*

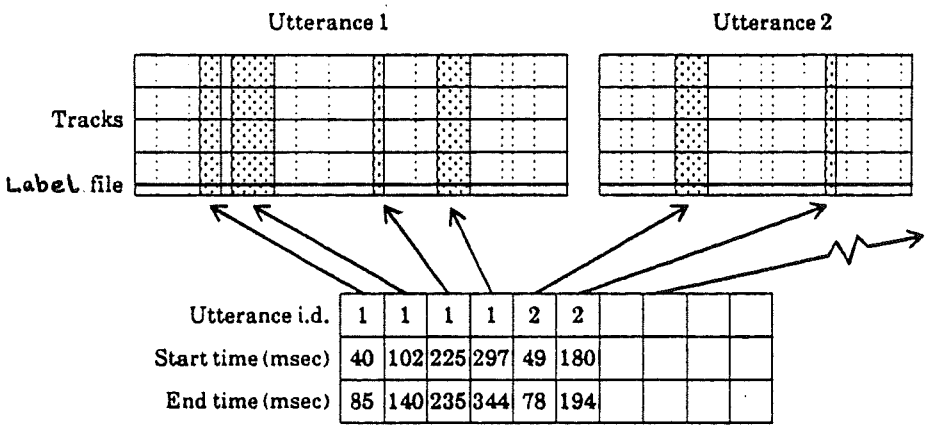

Figure 1.2: Segment Lists

1.3 Segment Lists

APS employs the notion of a segment, an index into a single, contiguous portion of speech within the bounds of a single utterance in the speech database. When many such segments are stored together it is called a segment list (Fig 1.2). A segment list might, for example, refer to all the oral stops in the database, or it could refer to all the instants at which the rate-of-change of high-frequency energy is particularly great.

Hendrix and Boves [3], in an implementation based on a relational database, make explicit the difference between acoustic segments and phonemic segments. In APS the interpretation of the segment list is left to the discretion of the user. This type-less representation allows the operations described in the next section to be applied to any segment list.

1.4 Segment Language

1.4.1 Creating Segment Lists

Segment lists may be created either a) on the basis of the hand-labelled data in the label files or b) from the acoustics stored in the track files.

The operation **phon** reads all the label files in the database and returns the segments in the form of a segment list. An optional argument to **phon** restricts the segment list to those whose label in the label file belongs to a given set of phonemes. It is also possible to select phonemes according to left and right context up to an arbitrary distance away.

Operation	*Description*
meanv	Mean value
medianv	Median value
minv	Minimum value
minp	Point in time at which min. occurs
maxv	Maximum value
maxp	Point in time at which max. occurs

Table 1.1: Some Sampling Operations

Segments may be generated on the basis of track files by means of the two operations above and below. The user specifies a track and a single constant number, the threshold. For a given threshold and track a test is performed between the threshold and the values in each frame of all the utterances. The operation above returns segments that span consecutive frames whose values are all greater than or equal to the threshold. below does the same operation for those frames whose values are less than the threshold.

Segment lists obtained from tracks may be matched against segment lists obtained from label files. In this way acoustic events can be identified from the corresponding hand transcription.

1.4.2 Measuring the Acoustic Characteristics of Segments

The most complete description of the contour of a track across each segment is obtained by plotting the values at each frame against a time axis. See for example Fig 1.3.

The acoustic data is usually reduced before further analysis by a process called sampling. Sampling is the means by which all the frames for a particular track and segment are reduced to a single numerical value.

Given a number of speech segments in the form of a segment list and a track upon which to operate, a sampling operator returns as many values as there are segments in the segment list. For example, meanv computes the mean value of a track across each segment in a segment list. Other sampling operations are shown in Table 1.1. All sampling operations return as many values as there are segments in the segment list.

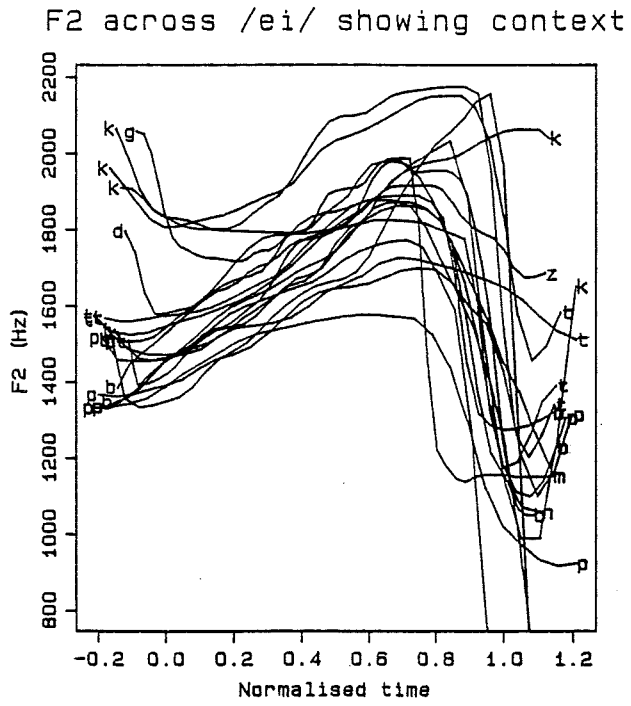

Figure 1.3: F2 Contours

1.4.3 Segment Shifting

It is often desirable to adjust the start time and/or the end times of segments. For example, in an attempt to reduce the effect of context it may be beneficial to exclude the first and last portions of a segment and perform the remaining analysis on the middle portion. The operation midslice reduces segments to a proportion of the original duration, removing equal portions from the beginning and end. More general adjustments to start times and end times may be made using arithmetic expressions.

1.5 Data Management

Much acoustic phonetic data is categorical - the speech domain commonly assigns objects into classes. S provides 1) functions for indexing and ranking data, 2) a mechanism for filtering out data elements according to arithmetic or logical criteria or by set membership and 3) functions that give tabular summaries.

1.6 Statistics

Investigations with a large speech database require access to statistical functions. Due to the inherent variability of speech, a sufficiently large number of speech tokens must be considered in order to obtain a representative sample. Statistical functions are required for summarising and modelling the data. Some statistical functions in S are: median, standard deviation, covariance, discriminant analysis, and PDF modelling.

1.7 Graphics

Data characteristics are more easily seen with the use of modern graphics [4]: graphical displays take advantage of the ability of the human visual system to process a large quantity of data, perceiving patterns and exceptions to patterns. Graphical capabilities in S include histograms, scatter plots (Fig 1.4), 3D perspective plots and boxplots (Fig 1.5).

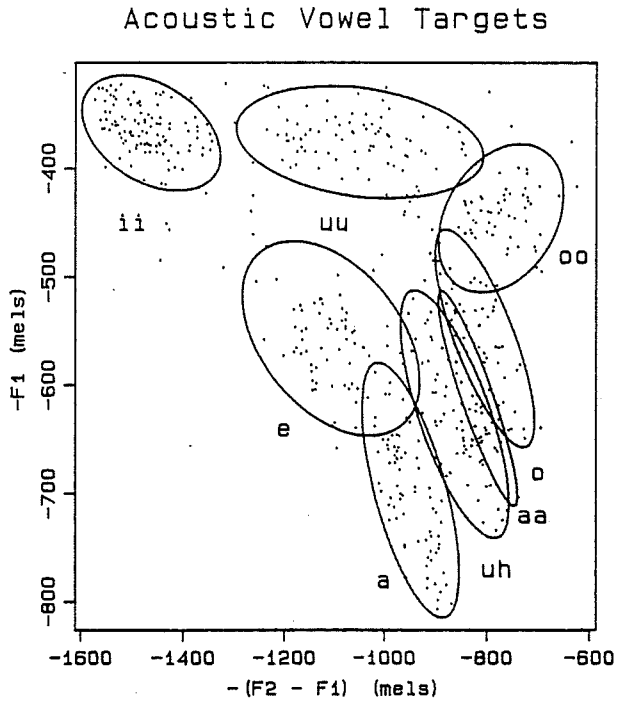

Figure 1.4: Scatter Plot

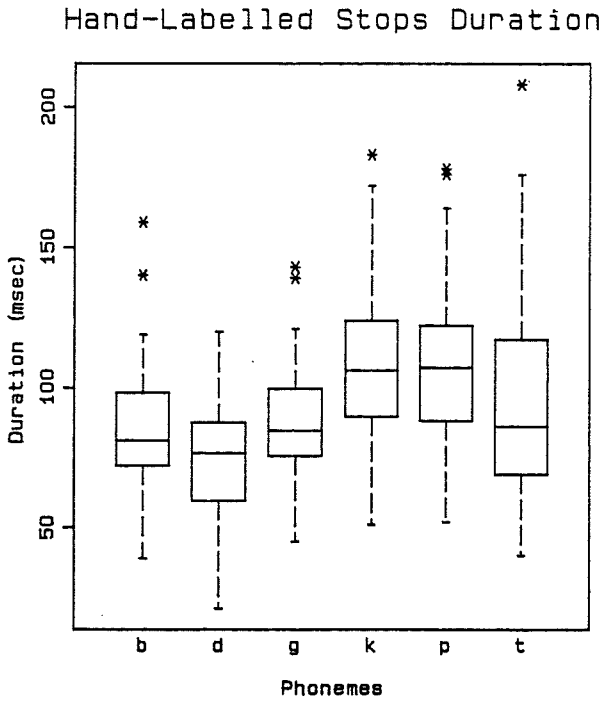

Figure 1.5: Boxplot Plot

Bibliography

[1] G.E. Peterson and H.L. Barney *Control Methods Used in a Study of the Vowels*, JASA Vol. 24, No. 2, pp175-184. 1952.

[2] R.A. Becker, J.M. Chambers and A.R. Wilks, *The New S Language.* Pacific Grove, California: Wadsworth and Brooks/Cole, 1988.

[3] J.P.M. Hendrix and L. Boves *Definition of Relations in an Acoustic Phonetic Database* Proc ICASSP 1988, pp. 643-646.

[4] J.M. Chambers, W.S. Cleveland, B. Kleiner and P.A. Tukey, *Graphical Methods for Data Analysis.* Belmont, California: Wadsworth, 1983.

Chapter 2

Tutorial Introduction to the S Language

This chapter and the following two chapters are a compilation of notes which have been used for the 1988/89 M.Sc *Speech Computing* course in the Department of Linguistics. It was originally intended for postgraduate students who may have had absolutely no previous experience of using computers and the style so much of the material will seem very elementary to anyone who is familiar with programming of any kind (especially in 'C').

This chapter gives a general introduction to the S language independently of APS.

2.1 Start Up

In order to be able to write expressions which perform operations such as those suggested in the introduction, we shall begin by giving a brief introduction to the S language. This introduction should be supplemented by reading *The New S Language* by Becker R., Chambers J. & Wilks A. (1988, Wadsworth & Brooks: Pacific Grove, Ca.). Chapters 1 to 3 are sufficient to use S effectively.

It is preferable if the user can login and work at the computer while working through the examples in this document. At CSTR login to either Watt, Brodie, Scott or Phoenix and in answer to the UNIX prompt %, type

```
% Splus
```

The reply from S will look something like this

```
Initializing login directory for new S-PLUS user.
S-PLUS : Copyright 1988, 1989 Statistical Sciences
S : Copyright AT&T.
Version 2.2 Release 1 for Sun4 :  1989
Working data will be in /u5/sip/gsw/.Data
>
```

The first line means that S has created a directory called `.Data`. All S data will be stored in there and remains even until the next S session. After printing version numbers of the program S prints > which is the S prompt. You are now able to start using S. (Henceforth, commands to be entered at the terminal will be indented as above and any comments which follow the commands will be appended with #).

2.2 Creating Objects: Vectors

(It is assumed that the user has now run 'Splus').

A vector can be defined as a single list of data entries. The following is an example of a vector

```
123 45 2 127
```

Vectors can also be other than numbers in which case each data entry is surrounded by double quotes " ". The following is also a vector.

```
"Phonetics" "37" "entry" "msc"
```

Storing the vector of numbers in an object called `x1` can be achieved in S by entering

```
> x1 ← c(123, 45, 2, 127)
```

Note that the ← (assignment arrow) is obtained by typing _ (underscore).

In order to inspect the contents of `x1`, simply enter `x1`.

```
> x1
[1] 123 45 2 127
>
```

Don't worry about the `[1]` for now. The character vector can be entered in a similar manner

```
> x2 ← c("Phonetics", "37", "entry", "misc")
> x2
[1] "Phonetics" "37" "entry" "msc"
>
```

A list of your S objects can be obtained by entering

```
> ls()
[1] "x1" "x2"
```

The command to delete object **x2** is

```
> rm(x2)
```

For the next example, create a vector of numbers from 1000 to 1200 as follows

```
> values ← c(1000:1200)
```

A display of the entire vector can of course be obtained by entering **values**. But if we only wanted to display the 89th entry, we would have to enter

```
> values[89]
```

Alternatively, entries from 89 up to 100 are obtained by entering

```
> values[89:100]
```

In order to store these entries in a new object called values1, enter

```
> values1 ← values[89:100]
```

which creates another vector called **values1** with these data entries.

Two or more vectors can also be combined into a single vector. First create two objects which contain entries 27 through to 30 and 35 through to 40 of the object **values**.

```
> values1 ← values[27:30]
> values2 ← values[35:40]
```

To combine these two vectors into a single vector, enter

```
> valuesnew ← c(values1, values2)
```

2.3 Creating Objects: Matrices

A matrix consists of two or more columns of data entries (a vector can be thought of as a single columned matrix). Begin by creating five vectors which have an equal number of entries, e.g.

```
> vec1 ← c(110, 28, 39, 49)
> vec2 ← c(123, 560, 2, 11)
> vec3 ← c(110, 34, 50, 98)
> vec4 ← c(59, 102, 421, 61)
> vec5 ← c(121, 322, 87, 23)
```

These vectors can be combined into a matrix using either `rbind` or `cbind`. `rbind` concatenates the vectors as rows and `cbind` as columns. Compare the results of the following

```
> newdata ← rbind(vec1, vec2, vec3, vec4, vec5)
> newdata1 ← cbind(vec1, vec2, vec3, vec4, vec5)
```

Any entry in a matrix is given by `objectname[r, c]` where `r` is the row number and `c` is the column. You will see from `newdata` that 50 is stored in row 3 column 3; this can be obtained by entering

```
> newdata[3,3]
[1] 50
```

Now try the following

```
> objectname[r,] # to read row 'r'
> objectname[,c] # to read column 'c'
```

So we could store column 2 of `newdata` in the object `w1` by entering

```
> w1 ← newdata[,2]
```

(w1 will now be a vector). It is also possible to give ranges, as in

`objectname[`$r(a)$`:`$r(b)$`,]`

to read from row $r(a)$ to $r(b)$

`objectname[,`$c(a)$`:`$c(b)$`]`

to read from column $c(a)$ to $c(b)$. So, rows 2 through to 4 of `newdata` are given by

```
> newdata[2:4,]
     [,1] [,2] [,3]
[1,]   28   39   49
[2,]  560    2   11
[3,]   34   50   98
[4,]  102  421   61
[5,]  322   87   23
>
```

and columns 2 through to 5 by

```
> newdata[,2:5]
```

To read from rows 1 through to 3 and columns 1 through to 4, we would have to enter

```
> newdata[1:3, 1:4]
```

2.4 Arithmetic Operators

The arithmetic operators are as follows

```
+       # addition
-       # subtraction
*       # multiplication
/       # division
^       # raise to the power of
```

Try these out. For example, to compute the arithmetic expression $((13-45)(17+3))^3$, enter

```
> ((13 - 45) * (17 + 3))^3
```

We can apply any of these operators to vectors. For example, to subtract 7 from each entry in **vec1** (and store the results in **newvec**), enter

```
> newvec ← vec1 - 7
> vec1
[1] 110 28 39 49
> newvec # all equal vec1 - 7
[1] 103 21 32 42
>
```

To multiply the third column of newdata by 2.6

```
> newdata[,3] * 2.6
```

It is also possible to perform arithmetic operations on two vectors or matrices. For example, to subtract **vec2** from **vec1**

```
> vec2 - vec1
[1] 13 532 -37 -38
> vec2
[1] 123 560 2 11
> vec1
[1] 110 28 39 49
>
```

To add **vec5** to the third column of **newdata**

```
> newdata[,3] + vec5
```

2.5 Comparison Operators

The following is a list of the comparison operators

```
>       # greater than
>=      # greater than or equal to
<       # less than
<=      # less than or equal to
==      # equal to
!=      # not equal to
```

These comparison operators return a logical vector consisting of either **TRUE** (**T**) or **FALSE** (**F**).

For example,

```
> vec1 < 40
```

returns **F T T F** because only the second and third entries of this vector are less than 40 (the statement **vec1 < 40** is only **TRUE** for entries 2 and 3).

Again, we can compare vectors with other vectors (preferably of the same length). For example, to find out whether the entries in **vec3** are equivalent to those in the third column of **newdata**, enter

```
> vec3 == newdata[,3]
```

which returns **T T T T** because the entries in **vec3** are identical to the those in third column of **newdata**.

The importance of these operators lies in the fact that they provide a convenient way of extracting data. Suppose we wanted to extract all the entries from the object **values** which were less than 1080. First, create an appropriate logical vector and write this to an object (in this case **which**)

```
> which ← values < 1080
```

Those entries which are less than 1080 are obtained by entering

```
> values[which]
```

which lists all the entries where **which** is **TRUE**. If we just wanted to know how many entries are less than 1080

```
> sum(which)
```

and if we wanted to know whether there were any entries less than 1080, enter

```
> any(which)
```

which returns **TRUE** if there are any entries less than 1080, otherwise **FALSE**. We can also obtain the inverse, i.e. a list of all the entries which are not less than 1080 by entering

```
> values[!which]
```

which lists all the entries where **which** is **FALSE**.

We can combine one or more logical vectors in a single expression. For example, suppose we want to extract those entries from **values** which are both less than 1080 and greater than 1020 (i.e. the range 1020 ¡ values ¡ 1080). This can be done using **&** (and operator)

```
> which ← values < 1020 & values > 1080
> values[which]
```

Suppose we would like the entries for which **values** is either less than 1100 or greater than 1150 (this is equivalent to a listing of both **values < 1100** and **values > 1150**). This could be done using the | (or) operator

```
> which ← values < 1100 | values > 1150
> values[which]
```

Logical vectors are also useful in extracting data from matrices. For the next example, create another vector, **friends**, with the names John, Michael, Mary, Cathy and Colin

```
> friends ← c("John", "Michael", "Mary", "Cathy",
                "Colin")
```

Now suppose that the data in the third column of **newdata** represents the number of times a year these four people go to work by car (i.e. the first entry in **newdata[,3]** corresponds to how many times John goes to work by car, the second corresponds to Michael etc.) and that we would like to find out which of them drives to work more than 55 times a year. This is given by

```
> which ← newdata[,3] > 55
> friends[which]
```

This returns "Cathy" and "Colin". The first line creates a logical vector; the second lists the members of friends which are T (i.e. for which the entries in newdata[,3] are greater than 55). For another example, suppose that the second column in newdata denotes the number of times the people in friends go to work by train. We would now like to find out which of them go to work more often by car than by train. This would be given by

```
> which ← newdata[,3] > newdata[,2]
> friends[which]
```

which should return "John", "Mary" and "Cathy".

2.6 Functions

A number of functions are available in S for performing, amongst other things, arithmetic and statistical operations. Only a few will be listed here

```
mean(x)  # calculate the mean of x
min(x)   # calculate the minimum of x
max(x)   # calculate the maximum of x
len(x)   # calculate the length of x
sqrt(x)  # calculate the square root of x
var (x)  # calculate the variance of x
log(x)   # calculate the natural log of x
exp(x)   # calculate the exponential of x
log10(x) # calculate the log to the base 10 of x
```

Normally, these functions would be applied to data stored in objects. For example, we could calculate the mean of vec2 and store the results in the object y1 by typing

```
> y1 ← mean(vec2)
```

2.7 Using an Editor Inside S: Writing Functions

This section assumes that you are able to use an editor. If you use an editor other than the default Emacs, then specify your own editor by assigning to the object .Editor. If you use vi, for example, say

```
> .Editor ← "vi"
```

As we saw above, one way of storing a list of numbers in a vector would be to type

```
> newvec1 ← c(123, 34, 46.8, -19, 37.....  etc)
```

Another way, which is more convenient when the list is long, is to use an editor inside S. If we want to store the list of numbers in `newvec1`, begin by typing

```
> newvec1 ← ed( as.numeric() )
```

Having typed this, you will automatically enter the editor. A character vector can be created by typing `as.character` in place of `as.numeric`. Numbers (or characters) should be separated by spaces or new lines. Write out the file in the usual manner. Verify that the numbers have been stored in `newvec1` by entering `newvec1`. The object `newvec1` could now be edited using `ed`, as follows

```
> newvec1 ← ed(newvec1)
```

You can also use `ed` to create your own functions. For example, to create a function which computed the difference between the maximum and minimum values of a data set, begin by entering

```
> myfun ← ed(FUNCTION)
```

Never forget to assign the result of `ed` to an object; if you simply say

```
> ed(FUNCTION) # Wrong!
```

or

```
> ed(myfun) # Wrong!
```

you will lose your edits.

When you are in the editor you should see this on your screen

```
function()
{
}
```

This is a skeleton function. The material inside the "`()`" defines how many arguments the function is to have. In this case, we want the function to apply to a single object and thus one argument will suffice. We can type any name inside the "`()`" e.g. "`x`", "`y`", "`data`" or whatever. The purpose of the function is to work out the difference between the maximum and minimum for a particular object. As we saw above, this is given by

	Italy	France	Spain	U.S.A.	Germany
Heathrow	235	389	269	1035	412
Gatwick	289	412	893	984	349
Edinburgh	88	179	7	0	18
Glasgow	192	315	216	0	368
Manchester	130	163	271	58	290

Table 2.1: Flights Table

```
max(x) - min(x)
```

where **x** is the name of the object. Accordingly, use **ed** to edit **FUNCTION** as follows

```
function(x)
{
max(x) - min(x)
}
```

The braces must always surround the expression(s) and follow **function()**. When the function has been written, exit from **ed** in the normal way. The function will now be stored under **myfun**. This can be verified by entering

```
> myfun
```

following which the material you typed in using **ed** should appear on the screen. If you want to edit the function, enter

```
> myfun ← ed(myfun)
```

To apply this function to a data set such as **values**, simply enter

```
> myfun(values)
```

Note that in writing **myfun** we have produced our own version of an S function called **range**. Do **myfun** and **range** return the same values?

2.8 Exercises

1. Vectors and Matrices

   (a) The following is a table of number of flights to different destinations from four different airports over a period of time.

   Create five vectors corresponding to the number of flights from each airport (i.e. vector 1, which might be named **Heathrow** should be a vector containing 235, 389, 269, 1035, 412).

(b) Use the vectors created in the previous question to form an appropriate matrix for Table 2.1 stored under the object name `flights`

(c) Write an S expression for the following

i. a list of the flights to France (from all destinations)

ii. a list of the flights from Edinburgh (to all destinations)

iii. a list of the flights from Glasgow to France and Spain

iv. a list of the flights from Heathrow and Gatwick to Spain, the U.S.A. and Germany

(d) Create objects for the following

i. a single vector of flights to France, Spain and Germany

ii. a two columned matrix with the flights to the USA in the first column and the flights to France in the second

iii. a matrix of three rows and two columns containing the flights from Heathrow, Gatwick and Manchester to Italy and Germany. Hint: create three temporary matrices each with the flights from Heathrow, Gatwick and Manchester (use `cbind`). Then collect these objects into a complete matrix using rbind.

2. Arithmetic Operators

(a) Convert the following into S expressions

i. $(31 + 17)(25 - 3)$

ii. $5((76.3 - 13)^2 + 11/3)$

(b) Write an S expression for

i. the difference between the number of flights from Heathrow to Spain and the number of flights from Manchester to Germany

ii. The number of flights to the USA divided by 5

3. Comparison Operators

(a) Create two vectors called `airports` and `destinations` containing the names of the airports and destinations in Table 2.1.

(b) Write an S expression for

i. the destinations to which the number of flights from Edinburgh is greater than 100

ii. the airports which have less than 60 flights to the USA

iii. the airports which have both less than 200 flights to Italy and more than 300 flights to France

iv. the airports which have between 100 and 300 flights to Germany

v. the airports which have more flights to France than Spain

vi. the airports whose combined flights to Italy and France are less than those to the USA

4. Functions

   (a) Write an S expression for

      i. the difference between the maximum number of flights to Italy and the maximum number of flights to Spain

      ii. the sum of the average number of flights to the USA and the average number of flights to Germany

      iii. the average of the flights from Glasgow and Manchester combined. Hint: use `c()` to combine the flights from Glasgow and those from Manchester into a single vector, and then work out the average value of this vector

      iv. the airport which has the maximum number of flights to Italy (this problem also uses some of the techniques in the next chapter)

   (b) write a function which returns `TRUE` for entries which are greater than the mean value of the entries. So for example, we would like the function to return

      ```
      F F F F T T T
      ```

      to the vector

      ```
      2 4 6 8 10 12 14
      ```

      (the average is 8 and the last three entries are greater than 8).

   (c) Use your function to find out the destinations which have a greater than average number of flights from Gatwick

2.9 Answers

1. Vectors and Matrices

(a) `Heathrow ← c(235, 389, 269, 1035, 412)`
`Gatwick ← c(289, 412, 893, 984, 349)` etc
or, use ed as follows
`Heathrow ← ed(as.numeric())` etc.

(b)
```
flights ← rbind (Heathrow, Gatwick, Edinburgh,
                 Glasgow, Manchester)
```

(c) i. `flights[,2]`
ii. `flights[3,]`
iii. `flights[4, 2:3]`
iv. `flights[1:2, 3:5]`

(d) i. `obj1 ← c(flights[,2:3], flights[,5])`
or,
`obj1 ← c(flights[,2], flights[,3], flights[,5])`
ii. `obj2 ← cbind (flights[,4], flights[,2])`
iii. First create three temporary objects
```
x1 ← cbind(flights[1,1], flights[1,5])
x2 ← cbind(flights[2,1],flights[2,5])
x3 ← cbind(flights[5,1], flights[5,5])
```
Then bind them into one matrix
```
rbind(x1, x2, x3)
```

2. Arithmetic Operations

(a) i. `(31 + 17) * (25 - 3)`
ii. `5 * ((76.3 - 13)^2 + (11/3))`

(b) i. `flights[1,3] - flights[5,5]`
ii. `flights[,4]/5`

3. Comparison operators

(a)
```
airports ← c("Heathrow", "Gatwick",
             "Edinburgh", "Glasgow", "Manchester")
destinations ← c("Italy", "France",
                 "Spain", "USA", "Germany")
```

(b) i.
```
which ← flights[3,] > 100
destinations[which]
```
ii.
```
which ← flights[,4] < 60
airports[which]
```

iii.
```
which ← flights[,1] < 200 &
        flights [,2] > 300
airports[which]
```

iv.
```
which ← flights[,5] > 100 &
        flights [,5] < 300
destinations[which]
```

v.
```
which ← flights[,2] > flights[,3]
airports[which]
```

vi.
```
which ← (flights[,1] + flights[,2]) <
        flights[,4]
airports[which]
```

4. Functions

(a) i. `max(flights[,1]) - max(flights[,3])`

ii. `mean(flights[,4]) + mean(flights[,5])`

iii.
```
obj1 ← c(flights[4,], flights[5,])
mean(obj1)
```

iv.
```
which ← flights[,1] == max(flights[,1])
airports[which]
```
(NB no spaces between the equals signs)

(b)
```
function (x)
{
  x > mean (x)
}
```

(c) If you called the function **myfun**, then the following should give you the right answer

```
which ← myfun(flights[2,])
destinations[which]
```

Chapter 3

Tutorial Introduction to APS

This chapter provides an introduction to some of the more important functions inside APS enabling segment lists and basic calculations such as means and standard deviations to be made.

APS (*A*coustic *P*honetics in *S*) is a set of S functions and datasets designed for reading label file and track files. See the function documentation or the on-line help pages for descriptions of these functions. To gain access the APS, while running S issue the command **aps**

```
> aps()
APS : Copyright 1990 CSTR, University of Edinburgh
>
```

If you know you will always want to use APS functions when running S, then APS may be made perminently available like so

```
> .First ← function()
{
  aps()
}
>
```

(When you enter S it looks for a function called `.First` and if it is found `.First` is called as if it were typed by the user).

Take a look at the names of the APS functions and datasets.

```
> ls(pos=2)
```

This will simply list names. To see documentation for an APS object use the help function.

```
> help(Utterances)
```

3.1 Utterances

At CSTR there is database of analysed speech upon which APS operates on by default. At other sites, if the person who installed APS has not provided such a default the user will have to follow the instructions given in Chapter 6 "Speech Databases for APS".

The default utterances at CSTR consist of 200 sentences spoken by one speaker. Enter

```
> Utterances
```

Each element of this character vector is an utterance in the database. APS functions that read the database use this object to find the label files and track files. If you have your own label files and track files you may use these as your database instead; this is done by creating your own version of `Utterances` (for details see Chapter 6 "Speech Databases for APS" and documentation for the function **mkdb**) in Chapter 7.

It is also possible to operate on only part of the default database. For example, the command

```
> Utterances ← Utterances.db[1:50]
```

will create a local copy of the first 50 Utterances in the standard database and causes operations to be performed on this subset only. (`Utterances.db` is a copy of the public version of `Utterances`).

To list the tracks which are available, enter

```
> tracks()
```

The meaning of some of the more important tracks is given in Table 3.1.

3.2 Segment Lists

APS describes slices of speech from the speech database by means of *segment lists*. A segment list is a matrix of three columns containing the start times and end times of each segment and the names of the utterances from which they come. In the following segment list

Track name	*Contents*
eX_Y	energy from X to Y Hz
f1	first formant
f2	second formant
f3	third formant
a1	amplitude of first formant
a2	amplitude of second formant
a3	amplitude of third formant
bw1	bandwidth of first formant
bw2	bandwidth of second formant
bw3	bandwidth of third formant
rms	total energy
r1	first autocorrelation coefficient
zcr	zero crossing density
syncspt	fundamental frequency (F0) values in Hz

Table 3.1: Some of the more important tracks in the CSTR default database

```
"240" "290" "/DB/atr20/be_dr1/mgsw0/sc016/sc016"
"1090" "1150" "/DB/atr20/be_dr1/mgsw0/sc021/sc021"
```

there are two segments. The first occurs in utterance `"sc016"` and has a start time at 240ms (relative to the beginning of the utterance) and extends for 50ms until end time 290ms. The second segment occurs in utterance `"sc021"` with a start time of 1090ms and an end time of 1150ms.

In order to extract a segment list for every occurrence of a particular segment, use the function **phon**. For example,

```
> s.b ← phon("b")
> s.ii ← phon("ii")
```

would extract all the /b/ and /ii/ segments which occur in `Utterances` and store them in the objects (segment lists) `s.b` and `s.ii` respectively. To extract all voiced stop segments, first create a vector with the relevant labels

```
> VSTOP ← c("b", "d", "g")
```

and then give `VSTOP` as an argument to **phon**

```
> s.voicedstop ← phon(VSTOP)
```

Some vectors with segment labels have already been created. These are `STOP` (oral stops), `NAS` (nasal consonants), `GLI` (/w/ and /y/), `LIQ` (/l/ and

/r/) and **FRIC** (all fricatives). You can see which segment labels are in these vectors by entering e.g.

```
> STOP
```

Therefore, in order to make a segment list of all the oral stops in Utterances, enter

```
> s.stop ← phon(STOP)
```

The length of any segment list is given by **slen**. For example, if we count the number of stops in the default database

```
> slen(s.stop)
```

returns 573. A subsection of this segment list can be extracted from **s.stop** as follows

```
> newseg ← s.stop[1:5,]
```

which extracts the first five entries in **s.stop**. (The comma is to ensure that entire rows and therefore utterance name, start and end times are obtained. If we had entered **s.stop[1:5]**, only the start times would be found).

The functions **start** and **end** extract the start and end times of segments. The start times of the five segments in **newseg** would be given by

```
> start(newseg)
```

The difference between end and start times gives, of course, the segment durations

```
> end(newseg) - start(newseg)
```

However, there is an equivalent function **dur**. Thus **dur(newseg)** is equivalent to **end(newseg) - start(newseg)**. If we wanted to get the average duration of a segment list, this would be given by

```
> mean(dur(newseg))
```

We often need to create a second segment list which has different start and end times from an original segment list. Suppose, for example, we wanted to create a new segment list from **s.stop** whose start and stop times were 10ms greater and less respectively than those of **s.stop**. This can be done as follows

```
> newstart ← start(s.stop) + 10
> newend ← end(s.stop) - 10
> newstop ← cbind(newstart, newend, utt(s.stop))
```

In the above, **newstart** is a vector consisting of start times which are 10ms greater than those of **s.stop** while **newend** has stop times which are 10ms less. The object **newstop** is created from the start times **newstart**, the end times **newend** and the utterance names **utt(s.stop)**; these are bound together in a three columned matrix using cbind.

The start times and end times can also be changed in the original segment list like this

```
> start(newstop) ← start(newstop) + 10
> end(newstop) ← end(newstop) - 10
```

As another example, supposing we wanted to create a segment list whose start time are 15% greater and whose stop times are 10% less than those of **s.stop** (i.e. if the original time goes from 0 to 100ms, the new time should run from 15ms to 90ms). This can be done as follows

```
> newstop ← s.stop
> stop(newstop) ← stop(newstop) + dur(s.stop)/15
> end(newstop) ← end(newstop) + dur(s.stop)/10
```

3.3 Labels

When label files are read by the function **phon** the labels are also stored in the segment list. When you print a segment list you will see these printed at the start of each line, one label per segment. To obtain a vector of these labels create, for example, a segment list containing all the vowels

```
> s.vow ← phon(VOW)
```

We can obtain the label files that correspond to the segment lists in either of these cases using the function **label**

```
> label(s.vow)
```

Using the function **table** see the totals for each phoneme

```
> table(label(s.vow))
```

Labels in conjunction with the function **class** are extremely useful in acoustic-phonetic analysis because they enable us to extract segments lists of a particular label from a larger segment list. For example, suppose we wanted to extract all /ii/ segments from **s.vow**. This would be done as follows

```
> l.vow ← label(s.vow)
> which ← class(l.vow, "ii")
```

The function **class** takes two arguments, the first of which must be a vector of labels and the second the set of labels we wish to extract. **class** returns a logical vector (in this case stored in **which**) which is **TRUE** for all segments in **s.vow** which are /ii/, otherwise **FALSE**. It is an easy matter to create a new segment list corresponding to all /ii/ labels

```
> s.ii ← s.vow[which,]
```

You can verify that **s.ii** consists entirely of /ii/ segments by using the function **all** which returns **TRUE** only if all the elements in its argument are **TRUE**

```
> all(which)
```

Note that all segments other than /ii/ would be obtained by

```
> s.vow[!which,]
```

which returns all segments in **s.vow** for which the logical vector **which** is **FALSE**.

There are at least two ways of extracting more than two segments from a long segment list. The first uses — (or) to extract (in this case) all /au/ and /oi/ segments from **s.vow**

```
> which ← class(l.vow, "au") | class(l.vow, "oi")
> s.vow[which,]
```

Alternatively, the comparison operator == could be used

```
> which ← l.vow == "au" | l.vow == "oi"
> s.vow[which,]
```

The second method binds the labels into a vector first

```
> diphlab ← c("au", "oi")
> which ← class(l.vow, diphlab)
> s.vow[which,]
```

If we had simply wanted to add up how many /au/ and /oi/ segments there were in **s.vow**, this could be done by using **sum**

```
> sum(which)
```

It is also possible to extract the labels of left and right contexts. For example, suppose we wanted to extract from `s.vow` all vowels which were preceded by /m/. To do this, create a label file of all segments preceding vowels

```
> ll.vow ← label(s.vow, -1)
```

The `-1` causes access to the labels one position to the left. Now create a logical vector for all /m/ labels in `ll.vow`

```
> rhs.m ← class(ll.vow, "m")
```

The `s.vow` segments preceded by /m/ are given by

```
> newsegs ← s.vow[rhs.m,]
```

All vowels which are followed by e.g. fricatives can be obtained as follows

```
> lr.vow ← label(s.vow, +1)
> which ← class(lr.vow, FRIC)
> newsegs ← s.vow[which,]
```

If we wanted all /@/ vowels which occur between nasal stops, the operator & (and) must be used

```
> # labels of vowels:
> l.vow ← label(s.vow)
> # labels of segments to the left of vowels:
> ll.vow ← label(s.vow,-1)
> # labels of segments to the right of vowels:
> lr.vow ← label(s.vow, +1)
> which ← class(ll.vow, NAS) &
          class(l.vow, "@") &
          class(lr.vow, NAS)
> schwanas ← s.vow[which,]
```

Finally, the identification of labels from a segment list which has start and end times which are different from those in the original utterances must be obtained using the function `luniq` rather than `label`. In the example below, a segment list is created whose start time is 10ms greater and whose end time is 20ms less than the start and end times of `s.vow`. The labels of this new segment list are then determined using `luniq`

```
> smallsegs ← cbind( start(s.vow) + 10, end(s.vow) - 20,
                     utt(s.vow))
> l.smallsegs ← luniq(smallsegs)
```

If you had used `label(smallsegs)` instead, no labels would appear (only "" - try this).

3.4 Frames

For the purposes of illustration, extract a single vowel segment (the 75th entry) from the segment list `s.vow` and store it in a segment list `a.seg`

```
a.seg ← s.vow[75,]
```

A segment can thought to be composed of frames of data which occur at equal intervals between a segment's start and end times. The interval at which the frames occur depends on the sampling frequency; in this database, the frames occur at intervals of 5ms (sampling frequency of 20kHz). Accordingly, since `a.seg` has a start time at 1200ms and a stop time at 1325ms (verify this by typing `a.seg`) it is 125ms long. Since the frames occur at 5ms, we can expect this segment to be composed of $125/5 = 25$ frames of data.

Each frame of data is associated with a value for each parameter track which has been run over the database. There will therefore be 25 values for e.g. `"f1"`, `"f2"`, `"f3"`, `"r1"`. If we wanted to inspect the `"f1"` (1st formant frequency values) for example, we can use the function `frames` which takes two arguments

```
> frames(a.seg, "f1")
```

25 `"f1"` values will now appear on the screen. The average F1 value can of course be obtained as follows

```
> mean(frames(a.seg, "f1"))
```

but there is another function `meanv` which computes the same result. It takes two arguments, a segment list and a track name

```
> meanv(a.seg, "f1")
```

The following functions operate in a similar way

```
> maxv (a.seg, "f1") # maximum value of a.seg on f1
> minv (a.seg, "f1") # minimum value of a.seg on f1
```

These are of course equivalent to

```
> max(frames(a.seg, "f1"))
> min(frames(a.seg, "f1"))
```

These functions have so far only been applied to a single segment. It is more usual to consider the values of parameters over a whole range of segments. For example, suppose we wanted to work out the average value of `"f2"` over all /a/ segments in Utterances. First, obtain all /a/ segments in the normal way

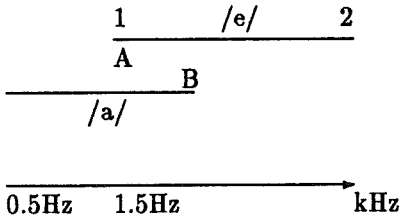

Figure 3.1: Example overlap

```
> s.a ← phon("a")
```

The mean value for each segment is given by

```
> meanv(s.a, "f2")
```

This produces 151 values because there are 151 /a/ segments in the database (the values represent the average of the `"f2"` frames for each segment). It is a straightforward matter to obtain the average of all these 151 values

```
> mean(meanv(s.a, "f2"))
```

The middle of the above expression may be written to an intermediate object

```
> file1 ← meanv(s.a, "f2")
> mean(file1)
```

The functions discussed above can be used to get a rough idea of the degree of overlap between two different segments types of a particular parameter. For example, we know from acoustic-phonetics that F2 of /a/ is generally lower than F2 of /e/. But when we examine many segments in continuous speech, there is likely to be some overlap between the two. Thus, we might expect /e/ and /a/ to overlap on F2 as shown in Fig 3.1.

Suppose we wanted to find out precisely how many /a/ and /e/ fall in the region of overlap – i.e. from A to B in Fig 3.1. Let us in fact consider how to determine the extent of this overlap on F2 for the middle third of these segments (the middle third is chosen to try to exclude the coarticulatory effects from neighbouring segments). First obtain the middle third of all /a/ and /e/ segments from `s.vow` and deposit the contents in `mids.a` and `mids.e` respectively

```
> # seg.  list for /a/:
> s.a ← s.vow[label(s.vow) == "a",]
```

```
> # duration of /a/ segs divided by three:
> middur ← dur(s.a)/3
> # copy of s.a:
> mids.a ← s.a
> # new start time:
> start(mids.a) ← start(s.a) + middur
> # new end time:
> end(mids.a) ← end(s.a) - middur
```

(Perform an analogous operation to obtain `mids.e`). Next obtain the average values of F2 for these segment lists

```
> f2.a ← meanv(mids.a, "f2")
> f2.e ← meanv(mids.e, "f2")
```

Confirm that the mean of all the /a/ segments is less than that of the /e/ segments.

```
> mean(f2.a)
> mean(f2.e)
```

It sometimes happens that we get spurious data; that is F2 values with ridiculous values like 0 or 100,000 Hz. If this were to happen at this point (it doesn't in the current database) then before proceeding any further, they should be removed from consideration by constraining F2 to occur within a sensible range like 400 Hz to 2800 Hz

```
> which ← f2.a > 400 & f2.a < 2800
> f2.a ← f2.a[which]
```

and perform the same operation on `f2.e`.

We now need to find point B in the above figure i.e. the maximum value of /a/ on F2 and point A i.e. the minimum value of /e/ on F2. These are obtained straightforwardly as follows

```
> max(f2.a)
> min(f2.e)
```

which I make to be 1638.22 Hz and 1162.175 Hz respectively. The question is now: how many /a/ segments fall in this range? and how many /e/ segments? These solutions can be determined by creating a logical vector in the appropriate range and summing for all `TRUE`

```
> which ← f2.a > 1162
> which1 ← f2.e < 1638
> sum(which)
> sum(which1)
```

which produces 151 for /a/ and 24 for /e/. We should really express these as a percentage of the total number of /a/ and /e/ segments

```
> # no.  of /a/ segs greater than 1162 Hz on F2:
> x1 ← sum(which)
> # no.  of /e/ segs less than 1638 Hz on F2:
> x2 ← sum(which1)
> # total number of /a/ segs:
> x3 ← slen(s.a)
> # total number of /e/ segs:
> x4 ← slen(s.e)
> # result:
> ((x1 + x2)/(x3 + x4)) * 100
```

which shows that 69.9% of all /a/ and /e/ segments overlap on F2 - not very encouraging!

Finally, **maxp**, **minp** and **meanp** return the times at which the maximum, minimum and mean values of a parameter in a segment list occur. Like **meanv** etc., they also take two arguments: a segment list and a parameter. Thus, suppose we wanted to change the segments in **s.a** to start at the time at which the F2 value was maximum. This can be done by

```
> start(s.a) ← maxp(s.a, "f2")
```

3.5 Plotting Facilities

Before calling a plotting function we need to declare which graphical device we wish to draw on. Some of the devices S can plot on and their corresponding functions are

```
> X11()      # X Windows
> suntools() # Sun View
> tek4014()  # Tektronics 4014
> hpgl()     # Hewlett Packard plotter
> printer()  # character terminal
```

Simply call one of these functions before calling a plotting function. If you use **printer** you will have to call **show** after the plotting command to see the result on your screen. See the S book or program documentation for details on each function.

One of the most useful plotting facilities is **bathtub** which plots a parameter track against time and also labels the left and right context of such a plot. Consider for example plotting F2 against time for the first 20 entries of the segment list **s.e**. This would be specified as follows

```
> bathtub(s.e[1:20,], "f2")
```

Note also that normalised time is plotted, that is, segments of different durations appear as if they are of the same duration on this plot.

Another useful plot is the histogram. If we want to make a histogram plot of the mean value of F2 over all /e/ segments, we first need to create the appropriate data set

```
> s.e ← phon("e")
> f2data ← meanv(s.e, "f2")
> hist(f2data, 20)
```

The second argument to `hist` specifies the number of bars of the histograms. If you leave out the second argument, a reasonable number of bars is provided.

3.6 Exercises

1. Write S expressions for the following
   (a) a segment list of all /ei/ segments
   (b) a segment list of all vowels
   (c) a segment list of all fricatives
   (d) a segment list of all front vowels /ii/, /e/ and /a/
2. Assume that all /ii/ segments have been stored as a segment list `s.ii`. How could we alter the start times to be
   (a) 20ms greater?, and
   (b) 20% of the total duration greater than the current start time?
3. Assume that no segment lists have been created. Write S expressions for
   (a) the labels of all the vowels
   (b) the labels of the first 30 vowels in the database
   (c) the labels of all the voiceless stops /p/, /t/, /k/
4. Assume a segment list for all vowels has been created and stored as s.vow. Using only this segment list, could we obtain the following?
   (a) a new segment list of all /ii/ segments

(b) a new segment list for the back vowels /uu/, /oo/, /o/, /aa/

(c) a label file for the diphthongs /ai/ and /ei/

(d) a label file of everything except the diphthongs /ai/ and /ei/

(e) the total number of /a/ segments in s.vow

(f) a label file for all voiced stops that precede vowels

(g) the number of back vowels /uu/ and /u/ that follow the glides /y/ and /w/ and precede the liquids /l/ and /r/

5. (You can assume in this question that the necessary segment lists are already in existence; prefix the segment list by "`s.`" in each case e.g. `s.ii` (segment list for /ii/), `s.vow` (segment list for vowels) etc.)

   Write S expressions for

   (a) the mean value of F2 plus the mean value of F1 for all /oi/ segments

   (b) the average F0 values (across all frames) of /e/ multiplied by 2

   (c) the mean values of the bandwidth of F3 across the middle third of all /ei/ segments

   (d) the number of /a/ segments whose average F1 falls in the region 600Hz to 1100Hz

   (e) the average of the maximum value of energy in the 2700Hz - 8000Hz band for all /s/ segments

3.7 Answers

1. (a) `phon("ei")`

   (b) `phon(VOW)`

   (c) `phon(FRIC)`

   (d)
   ```
   front ← c("ii", "e", "a")
   phon(front)
   ```

2. (a) `start(s.ii) ← start(s.ii) + 20`

   (b) `start(s.ii) ← start(s.ii) + dur(s.ii) / 5`

3. (a) `label(phon(VOW))`

   (b) `label((phon(VOW)[1:30,]))`

   (c) `label(phon(c("p", "t", "k")))`

4. (a)
```
which ← class(label(s.vow), "ii")
```
or
```
which ← label(s.vow) == "ii"
newsegs ← s.vow[which,]
```

(b)
```
BACK ← c("uu", "oo", "o", "aa")
which ← class(label(s.vow), BACK)
newsegs ← s.vow[which,]
```

(c)
```
l.vow ← label(s.vow)
which ← class(l.vow, c("ai","ei"))
l.diph ← l.vow[which]
```
or
```
l.vow ← label(s.vow)
l.diph ← l.vow[class(l.vow, "ai") | class(l.vow, "ei")]
```
or
```
l.diph ← l.vow[l.vow == "ai" | l.vow == "ei"]
```

(d)
```
l.vow ← label(s.vow)
which ← class(l.vow, c("ai","ei"))
l.ndiph ← l.vow[!which]
```

(e)
```
sum( label(s.vow) == "a" )
```
or
```
sum( class( label(s.vow), "a" ) )
```

(f)
```
l.lhs ← label(s.vow, -1)
VSTOP ← c("b", "d", "g")
l.vstop ← l.lhs[ class(l.lhs, VSTOP) ]
```

(g)
```
l.lhs ← label(s.vow, -1)
l.rhs ← label(s.vow, +1)
is.back ← class(label(s.vow), c("uu","u"))
which ← is.back & class(l.lhs, c("y","w")) &
class(l.rhs, c("l","r"))
sum(which)
```

5. (a)
```
meanv(s.oi, "f2") + meanv(s.oi, "f1")
```

(b)
```
meanv(s.e, "syncspt") * 2
```

(c)
```
thirdof ← dur(s.ei)/3
newei ← s.ei
start(newei) ← start(newei) + thirdof
end(newei) ← end(newei) - thirdof
meanv(newei, "bw3")
```

```
(d) which ← meanv(s.a, "f1") > 600 &
            meanv(s.a, "f1") < 1100
    sum(which)
(e) mean(maxv(s.s, "e2700-8000"))
```

Chapter 4

Using APS

Familiarity with an editor such as Emacs or `vi` *is assumed in this chapter.*

This chapter covers material that will enable students to write their own functions inside APS. Some acoustic phonetic experiments are introduced and functions are described to perform them.

At some stage, the user may wish to create their own functions: as we saw earlier, this can be conveniently done inside S using an editor. The default editor is Emacs. If you wish to use another editor such as `vi`, then assign the object `.Editor` to the name of the editor as in

```
> .Editor ← "vi"
```

This assignment will be effective even in later sessions of S.

The general form of a function is

```
function (arguments)
{
  expression 1
  expression 2
  ....
  expression n
}
```

The arguments specify to the user how many variables must be entered for example, `meanv` has two arguments, the name of a segment list and the name of a parameter track e.g. `meanv(s.vow, "f1")`. Arguments must always be separated by commas. Some examples of functions are given below.

4.1 Example I: Removing values

Suppose we have calculated the mean of F2 over a segment list and we would now like to remove all segments which have F2 less than 400 Hz and greater than 2800 Hz. For a segment list **x**, this could be done as follows

```
> which ← meanv (x, "f2") < 400 |
          meanv (x, "f2") > 2800
> x ← x[!which,]
```

If we want to incorporate this into a function, we would begin by using **ed**

```
> Remove ← ed(FUNCTION)
```

(The function will be written to an object called **Remove**). This will put you into the editor containing a skeleton function. Edit the function to look like this

```
function(x)
{
  which ← meanv(x, "f2") < 400 |
           meanv(x, "f2") > 2800
  x[!which,]
}
```

The value returned by a function is the value of the last expression it executes, in this case the new segment list. Write out the file from the editor in the normal way. The function will now be stored under the name **Remove**. If we now want to remove these F2 values from a segment list called **s.vow**, we would enter

```
s.vow ← Remove(s.vow)
```

However, it is more common to write the results of applying the function to an object

```
newseg ← Remove(s.vow)
```

s.vow will now have no segments whose F2 values are less than 400Hz or greater than 2800Hz.

A slight problem with the above function is that we are calculating the mean value of F2 twice (which could be time consuming). A better solution would be to calculate F2 only once as follows

```
function (x)
{
```

```
  f2data ← meanv(x, "f2")
  which ← f2data < 400 | f2data > 2800
  x[!which]
}
```

4.2 Example II: Herz to Mel

In acoustic phonetics, we often need to convert Hz to the Mel scale. The relationship between Hz and Mel is given as follows (m is the frequency in Mels, h the frequency in Herz)

$$m = \frac{1000}{\log 2 \log(1 + h/1000)}$$

We can convert this into an S expression as follows

```
1000/log(2) * log(1 + (h/1000))
```

Putting this inside a function we have

```
function (h)
{
  1000/log(2) * log(1 + (h/1000))
}
```

If we call this function **Mel**, then **Mel(data)** will convert values in a vector from Hz to Mel.

4.3 Example III: Plotting -F1 Against -(F2 - F1)

A plot of against -F1 against -(F2 - F1) is often very useful when examining vowel data because the positions of the vowels in this chart resemble those of the vowel quadrilateral. However, in order to plot values on these axes using a suitable plotting function inside S (e.g. **eplot**), we often need to prepare the data in a single two columned matrix, e.g.

```
     [,1] [,2]
[1,] -1345 -456
[2,] -1214 -987
[3,] -1548 -298
[4,] -1231 -892
[5,] -1812 -990
```

(-(F2-F1) is in the left hand column, -F1 in the right hand column)

In order to prepare data in this way, first calculate (for any segment list) F1 and F2

```
> f1 ← meanv(x, "f1")
> f2 ← meanv(x, "f2")
> left ← -(f2 - f1)
> right ← -f1
```

In the third line, `left` is made equal to -(F2 - F1) while `right` has values of -F1. We must now bind these into a single matrix

```
> mat ← cbind(left, right)
```

The whole thing inside a function would be

```
function (x)
{
f1 ← meanv(x, "f1")
f2 ← meanv(x, "f2")
left ← -(f2 - f1)
right ← -f1
cbind(left, right)
}
```

Suppose we additionally wanted to remove all F1 values which were less than 300Hz or greater than 1200Hz and also to remove F2 values which were less than 500Hz or greater than 2800Hz. Such an operation could be accomplished as follows

```
> which1 ← f1 < 300 | f1 > 1200
> which2 ← f2 < 500 | f2 > 2800
> f1 ← f1[!which]
> f2 ← f2[!which]
```

Incorporating this into the function we have

```
function (x)
{
  f1 ← meanv(x, "f1")
  f2 ← meanv(x, "f2")
  which1 ← f1 < 300 | f1 > 1200
  which2 ← f2 < 500 | f2 > 2800
  f1 ← f1[!which]
  f2 ← f2[!which]
```

```
  left ← -(f2 - f1)
  right ← -f1
  cbind (left, right)
}
```

We can also call other functions inside functions. For example, suppose we had written the Mel function (described above) which converted Hz to Mel and would like to plot -F1 against -(F2 - F1) in Mels. This could be done straightforwardly by converting `left` and `right` into Mels

```
function (x)
{
  f1 ← meanv(x, "f1")
  f2 ← meanv(x, "f2")
  which1 ← f1 < 300 | f1 > 1200
  which2 ← f2 < 500 | f2 > 2800
  f1 ← f1[!which]
  f2 ← f2[!which]
  left ← -Mel(f2 - f1)
  right ← -Mel(f1)
  cbind(left, right)
}
```

4.4 Example IV: Calculating the Slope of F2

In this example, the object is to calculate the slope of F2 from the midpoint of a segment to the end of a segment. In fact, we shall calculate (a) the average F2 value from the midpoint for 15ms and (b) the average F2 value for the last 15ms and then connect (a) and (b) by a straight line. The slope (in Hz/ms) is then the degree to which the line deviates from the horizontal. Begin by creating two segment lists. The first extends from the midpoint for 15ms and the last from 15ms before the end to the end of the segment

```
> midpoint ← dur(x)/2
> msegs ← cbind( midpoint, midpoint + 15, utt(x) )
> esegs ← cbind( start(x), end(x) - 15, utt(x) )
```

Next calculate the average value of F2 for both segment lists

```
> f2m ← meanv(msegs, "f2")
> f2e ← meanv(esegs, "f2")
```

The slope will be given by the difference between the average values of `f2e` and `f2m` divided by the duration from the end of the segment to the midpoint

```
> (f2e - f2m) / (end(x) - midpoint)
```

The above only applies to F2; we could also make it apply to either F1 or F3 by creating a second argument in the function and replacing "`f2`" by this argument name. The whole function then becomes

```
function(x, trackname)
{
  midpoint ← dur(x)/2
  msegs ← cbind( midpoint, midpoint + 15, utt(x) )
  esegs ← cbind( start(x), end(x) - 15, utt(x) )
  fm ← meanv(msegs, trackname)
  fe ← meanv(esegs, trackname)
  (fe - fm) / (end(x) - midpoint)
}
```

4.5 Example V: Vowel Normalisation

Different speakers have different vocal tract shapes and so even though two speakers may produce the same phonetic type like /i/, the formant frequency values will be different. In particular, the formant frequency values of female speakers tend to be higher than those of male speakers. Accordingly, a comparison of formant frequency values across different vowel types from a large corpus of male and female speakers requires some kind of normalisation in order to make the comparison meaningful.

One possible vowel normalisation technique, outlined in Gerstman (1968), is to rescale minimum and maximum F1 values in the range 250 Hz - 750 Hz and minimum and maximum F2 values in the range 850 Hz - 2250 Hz. Thus, if a speaker has a minimum value of F1 at 400 Hz, a maximum value of F1 at 1200 Hz, then a speaker's F1 value which is exactly halfway at 800 Hz would be translated into the halfway value on the scale 250 - 750 Hz i.e. 500 Hz. How could we write a function to perform this kind of normalisation?

We begin by noting that if a range which extends from R_{min} to R_{max} is to be translated linearly into another range S_{min} to S_{max}, then a value r on the old range has a value on the new range, s, which is given by

$$s = S_{min} + 2\frac{(r - R_{min})(S_{max} - S_{min})}{(R_{max} - R_{min})}$$

Thus if a speaker's F1 range extended from 400 to 1200 Hz and the new scale extended from 250 to 750 Hz, the rescaled F1 values r_{res} would be given by

$$r_{res} = 250 + \frac{(r - 400)(750 - 250)}{1200 - 400}$$

where r is an F1 value of a speaker. The following function rescales according to the Gerstman technique and takes two arguments for the F1 and F2 values of a given speaker

```
function(f1, f2)
{
  f1min ← min(f1)
  f1max ← max(f1)
  f2min ← min(f2)
  f2max ← max(f2)
  newf1 ← 250 + ((f1 - f1min) * (750 - 250) /
                 (f1max - f1min))
  newf2 ← 850 + ((f2 - f2min) * (2250 - 850) /
                 (f2max -f2min))
  cbind(newf1, newf2)
}
```

The rescaled F1 and F2 values are then bound together in a single two columned matrix. More usually, we would want to rescale F1 and F2 directly from the mean values of F1 and F2 derived from a segment list. Such a function, in which **x** can be any segment list, is shown below

```
function(x)
{
  f1 ← meanv(x, "f1")
  f2 ← meanv(x, "f2")
  f1min ← min(f1)
  f1max ← max(f1)
  f2min ← min(f2)
  f2max ← max(f2)
  newf1 ← 250 + ((f1 - f1min) * (750 - 250) /
                 (f1max - f1min))
  newf2 ← 850 + ((f2 - f2min) * (2250 - 850) /
                 (f2max - f2min))
  cbind(newf1, newf2)
}
```

Better still, let's leave it to the user to decide whether F1 or F2 is to be rescaled. In this case, we would need to incorporate a conditional

if expression into the function which depended on whether the user had chosen F1 or F2. Informally, we would like to say

> "If the user decided on F1, then S_{min} and S_{max} (see the formula above) should be set to 250 and 750 Hz respectively; otherwise, if the user decided on F2, S_{min} and S_{max} should be set to 850 and 2250 Hz respectively."

In S the syntax for an **if** expression is the following

```
if(statement)
{
  expression 1
  expression 2
  ...
  expression n
}
```

The expressions are the ones which will be evaluated if **statement** is TRUE. Note that if there is more than one expression, they must be inside braces { } and the statement must be enclosed by parentheses. In the context of the following function, we could say

```
function(segs, formant)
{
  f ← meanv(segs, formant)
  Rmin ← min(f)
  Rmax ← max(f)
  if (formant == "f1")
  {
    Smin ← 250
    Smax ← 750
  }
  if (formant == "f2")
  {
    Smin ← 850
    Smax ← 2250
  }
  Smin + ((f - Rmin) * (Smax - Smin) / (Rmax - Rmin))
}
```

(Note the braces are automatically indented when you quit from **ed**). The first argument of the function, **segs**, is a segment list and the second argument, **formant**, is a parameter track which should be either F1 or F2. **f**

is the mean value of either F1 or F2 and `Rmin` and `Rmax` the minimum or maximum values. We now test whether the second argument specified by the user is `"f1"`. If it is, then the minimum and maximum of the scales are set to 250 and 750 Hz respectively; if `y` is `"f2"`, the minimum and maximum values are set to 850 and 2250 Hz. The final statement is the normalisation which takes the relevant `Smin` and `Smax` values.

4.6 Exercises

1. The relationship between the Hz and Bark scale is defined as

$$b = 13 \arctan 0.76f + 3.5 \arctan(f/7.5)2$$

   where b is frequency in Bark and f is frequency in kHz and arctan is arc tangent in radians. Write a function which takes a single argument and converts from Herz to Bark. (NB. arctan in radians is defined inside S as `atan` e.g. arctan 12 radians is `atan(12)` in S. Note also that 1kHz = 1000Hz).

2. Write a function which takes a segment list as a single argument and performs vowel normalisation according to the algorithm set out in Nearey (1977) in which the average value of the log frequencies of F1 and F2 across all vowels is subtracted from each F1 and F2 value of each vowel. More specifically, find the logarithm of F1 and F2 for all vowels (where `x` is a segment list of vowels)

```
G1 ← log(meanv(x, "f1"))
G2 ← log(meanv(x, "f2"))
```

   Then work out `Gbar`, the average value of `G1` and `G2` together

```
Gbar ← mean(c(G1, G2))
```

   Then subtract `Gbar` from `G1` and `G2`

```
G1new ← G1 - Gbar G2new ← G2 - Gbar
```

   Finally, rescale `G1new` to fall within 250 Hz to 750 Hz and `G2new` to fall within 850 to 2250 Hz using the formula

$$s = s_{min} + \frac{(r - r_{min})(s_{max} - s_{min})}{s_{max} - r_{min}}$$

   where (for F1), s_{min} is 250, s_{max} is 750, r is `G1new`, r_{min} is `min(G1new)` and r_{max} is `max(G1new)`. The output of the function should be a two columned matrix with normalised F1 and F2 values.

3. Modify the function for vowel normalisation you wrote above to remove all segments which have either

   F1 < 300
   F1 > 1200
   F2 < 500
   F2 > 2800

   These segments should be removed before you compute `G1` and `G2`.

4.7 Answers

These are of course only possible solutions; you may well have other better ones.

1.
```
function(x)
{
  kHz ← x/1000
  const ← (kHz/7.5) * (kHz/7.5)
  (13 * atan(0.76 * kHz)) + (3.5 * atan(const))
}
```

2.
```
function(x)
{
  G1 ← log(meanv(x, "f1"))
  G2 ← log(meanv(x, "f2"))
  Gbar ← mean(c(G1, G2))
  G1n ← G1 - Gbar
  G2n ← G2 - Gbar
  G1n.min ← min(G1n)
  G1n.max ← max(G1n)
  G2n.min ← min(G2n)
  G2n.max ← max(G2n)
  G1norm ← 250 + (((G1new - G1n.min) * (750 - 250)) /
  (G1n.max - G1min))
  G2norm ← 850 + (((G2new - G2n.min) * (2250 - 850)) /
  (G2n.max - G2min))
  cbind(G1norm, G2norm)
}
```

3.
```
function(x)
{
  f1 ← meanv(x, "f1")
```

```
    f2 ← meanv(x, "f2")
    whichf1 ← f1 < 300 | f1 > 1200
    whichf2 ← f2 < 500 | f2 > 2800
    f1new ← f1[!whichf1]
    f2new ← f2[!whichf2]
    G1 ← log(f1new)
    G2 ← log(f2new)
    Gbar ← mean(c(G1, G2))
    G1n ← G1 - Gbar
    G2n ← G2 - Gbar
    G1n.min ← min(G1n)
    G1n.max ← max(G1n)
    G2n.min ← min(G2n)
    G2n.max ← max(G2n)
    G1norm ← 250 + (((G1new - G1n.min) * (750 - 250)) /
    (G1n.max - G1min))
    G2norm ← 850 + (((G2new - G2n.min) * (2250 - 850))
    (G2n.max - G2min))
    cbind(G1norm, G2norm)
}
```

Chapter 5

Installing APS

To run APS you will need: 1) an APS distribution package, 2) a Sun4, and 3) the S system. APS is an add-on package to the S system. S is a programming environment for data analysis and graphics. APS runs on S-PLUS, a version of S from Statistical Sciences Inc., version 2.2 or 2.3. Although it has not beed tested and therefor cannot be guaranteed, it is expected that APS would also run on AT&T S, July 1989 release, as S-PLUS is based on AT&T S.

1. *Reading the Distribution*

   It is suggested that you install APS under your existing S directory structure. You will initially need 2.5Mb of space which can later be reduced to 0.5Mb. APS is distributed either on a DOS $3\frac{1}{2}$ inch disc or on a $\frac{1}{4}$ inch tape cartridge. The installation can be performed by any user who has write permission in the S home directory.

   (a) If you have a *DOS disc*, first copy the file `APS.TAR` from the DOS disc to a file in the S home directory called `aps.tar` (if you use `ftp`, use binary mode). In the S home directory execute the command

   ```
   tar xvf aps.tar
   ```

   Then remove `aps.tar`

   ```
   rm aps.tar
   ```

   (b) If you have a *tape cartridge*, first move to the S home directory. Mount the tape cartridge in the tape drive. Enter the command

```
tar xpf /dev/rst8
```

You will now have created a directory called `aps` under the S home directory. We will call this the APS home directory. The APS home directory in turn contains the following subdirectories: 1) `src`, containing all `.o` files and `.c` files necessary for making a new version of S with the APS code compiled into it, 2) `.Data` which is the public directory of S objects for APS including all functions and datasets, 3) `.Data/.Help` containing S 'help' pages for APS objects, and 4) `emacs` containing a public-domain package that allows S to run in an Emacs buffer, providing a convenient method of editing and re-executing the history of S commands.

2. *Modifying S scripts*

   Three S files need to be modified in order to make and run APS. Filenames are all relative to the S home directory.

   (a) `newfun/lib/S_makefile`. Change the line that looks like this

   ```
   SYSLIB = -lF77 -lI77 -lm
   ```

   to look like this

   ```
   SYSLIB = -lm # -lF77 -lI77 -lm
   ```

   This removes the need of a FORTRAN-77 library. The APS distribution contains public domain object code for the necessary functions.

   (b) `cmd/NEW`. After the line which start with this (line 4 in our version):

   ```
   -o) which=o; S_FUNCTIONS=$SHOME/os;
   ```

   add the line

   ```
   -aps) which=aps.; shift;;
   ```

   This will cause the start-up script to look for a special version of S which contains the APS code.

   (c) `S` or `Splus`. This is the shell script placed in the users' search path which starts the S process. It's name depends on which version of S you are using. In order to get it to look for an APS version of S change the line

   ```
   0) exec $SHOME/cmd/$S_DEFAULT
   ```

   to look like this

   ```
   0) exec $SHOME/cmd/$S_DEFAULT -aps
   ```

3. *Making APS*

   (a) Change to the subdirectory `src`

   ```
   cd src
   ```

   (b) If your command to start up S is not `S`, then edit the last line of the file `load` to start with the appropriate command, so instead of

   ```
   S LOAD $f77stuff $obj $src
   ```

   put, for example,

   ```
   Splus LOAD $f77stuff $obj $src
   ```

   (c) Enter the command

   ```
   load
   ```

   This will take a few minutes to run. When it is finished you should see the message

   ```
   Local version of S loaded
   ```

   You will now have a large file called `local.Sqpe`. It contains all the S code and the APS code linked together.

4. *Installing APS*

   (a) To put `local.Sqpe` in the correct place for execution by all users, while in the directory `aps/src` enter

   ```
   install
   ```

   (b) We will now create a new S function, `aps`, which connects the APS directory to the S search path. From `aps/src` start up S in the usual way

   ```
   S # or Splus, etc.
   ```

   Then, in reply to the S prompt >, enter the S command

   ```
   mkaps()
   ```

   The function `aps` will now have been created and put in the public directory of S functions. An S user who calls this function will now gain access to all the APS functionality.

   At this point it is advisable to set up some defaults for the new user of APS. Before we do that, use the function we just created to gain access to the APS functions. Enter

   ```
   aps()
   ```

5. *Local Setup*

   (a) To get APS to read your local format label files and/or track files, follow the procedure described in Chapter 6 "Speech Databases for APS".

   (b) In `.Data`, the public directory of S objects for APS, there are some datasets containing the sets of phoneme labels that occur in the label files. The dataset `STOP`, for example, contains the labels of the six English stops /p/, /t/, /k/, /b/, /d/ and /g/. These datasets are distributed as CSTR MRPA labels (*M*achine *R*eadable *P*honetic *A*lphabet). You may wish to change these to reflect the contents of your own database. See `Phonemes` in the Chapter 7 "APS Functions" for a full list of these datasets. To change one of them, start S and APS in the APS home directory and assign, for example

```
> VOW ← c("A", "E", "&", "I", "O", "Q",
          "U", "#", "J", "W")
```

   (c) The object `.Editor` defines the default editor when using the function `ed`. You may wish to change the public copy by running S and APS in the APS home directory and assigning thus, for example

```
> .Editor ← "vi"
```

6. *S in Emacs*

   The directory `emacs` contains public domain Emacs code for running S in an Emacs buffer. It is particularly useful for allowing easy editing of your history of S commands. The package includes an S mode for editing S data and functions. To install it follow the instructions in S.el.

   If your S is called by means of a name other than "S", eg. "Splus", then edit the line in S.el

```
(defvar inferior-S-program "S"
```

   to look like

```
(defvar inferior-S-program "Splus"
```

   or as appropriate.

Chapter 6

Speech Databases for APS

How does APS read a speech database? This question is answered below. This should be of interest to someone who wishes to start using APS on an existing database and to the person who is installing APS for the first time.

Speech databases for APS consist of label files and track files. Label files contain lists of start-time/end-time/label triples; track files contain vectors of values at a regular interval (eg. 5 msec) generated by various signal processing algorithms. There must be a label file and a track file for each utterance in the database.

Included here is a description of how APS reads a database, a list of accepted formats for label files and track files, how to make APS read a new database and, if APS does not recognise your label file or track file format, how to write a new label file and track file reader for APS.

6.1 How The Speech Database Is Read

APS recognises various database formats. Currently recognised formats are indicated in Table 6.1.

If you have label files and track files in the internal formats shown in the table, you may now wish to read Section 6.3 "Indexing a New Database".

How does APS determine the database format? This is how it works: It constructs the name of the file using the database format specified in the first label file entry in Table 6.1. Exactly how a filename is constructed for a particular database format is described in Section 6.2. If this file does not

Label/Track	*Extension*	*Filename Construction*	*Internal Format*
Label	`.mrp`	Standard	AUDLAB label
Label	`.syl`	Standard	AUDLAB label
Label	`.trn`	Standard	SAM-BA, OSPREY
Label	`.seg`	Standard	CSTR ISD
Label	`.lab`	Olivetti	Olivetti label
Track	`.trk`	Standard	AUDLAB track
Track	`.otrk`	Standard	AUDLAB track
Track	`.otr`	Standard	AUDLAB track
Track	`.dbtrk`	Standard	Old AUDLAB track
Track	`.dbt`	Standard	Old AUDLAB track
Track	`.lpc`	Olivetti	Olivetti track

Table 6.1: Database formats for label files and for track files. The entries for label files and track files form separate lists. The top section shows database formats for label files and the bottom section shows database formats for track files. Each database format has a filename extension which determines the internal format of the files. As indicated, there are two ways in which filenames are constructed.

exist then try constructing a new filename using the the second entry in the table. Continue trying all the database formats for label files until one is found that generates a filename that exists. This then becomes the selected database format for label files. The same method is used to determine the database format for track files except that the search spans the track file database formats shown in the bottom half of Table 6.1.

When a file is found, the internal file type is determined by the extension on the filename. On this basis the appropriate read function for this type of file is selected, the function is then used to read the file and the data is made available to the calling function.

To save performing the search on every utterance the selected database format is remembered for subsiquent label file requests. A separate note is kept of the selected database format for track files.

While reading the database, if a database format ever fails by generating a filename that does not exist, the search for a database format is started again from the top of the list. This allows database formats to be mixed in a single database. Note that the order of the search becomes significant when there exists more than one database format. When this case it is the format that occurs first in Table 6.1. which is used and any others are never seen. It is possible to change the order of these entries in the internal table; see Section 6.4 "Adding New Database Formats",

6.2 How Filenames Are Constructed

APS accesses the speech database by means of an S object called `Utterances`. This is a character vector in which each element describes the location of a single utterance in the database. APS functions such as `phon`, `tracks` and `above` use this object to locate the label files or track files it needs.

There will be default version of `Utterances` available to you. The contents will vary according to your local installation, but typically it will contain all the utterances available in some public database. To inspect the first six utterances while in S (after issuing the command `aps()` if you have not already done so) enter `Utterances[1:6]`. At CSTR and on the distribution copy of APS you will get

```
> Utterances[1:6]
[1] "/DB/atr20/be_dr1/mgsw0/sc001/sc001"
[2] "/DB/atr20/be_dr1/mgsw0/sc002/sc002"
[3] "/DB/atr20/be_dr1/mgsw0/sc003/sc003"
[4] "/DB/atr20/be_dr1/mgsw0/sc004/sc004"
[5] "/DB/atr20/be_dr1/mgsw0/sc005/sc005"
[6] "/DB/atr20/be_dr1/mgsw0/sc006/sc006"
>
```

For most database formats `Utterances` gives the complete pathnames of the label files and track files, but with the filename's extension stripped off. In the current version of APS there are, however, two possible ways in which filenames are constructed, Standard and Olivetti. Refer to Table 6.1. to see which applies to your database format. We will now describe these in turn. If neither of these methods are suitable for your installation then it is possible to define your method of filename construction; see Section 6.4 "Adding New Database Formats".

6.2.1 Standard Filename Construction

Each element in `Utterances` is the complete pathname of the label file or track file, except for the filename extension which has been stripped off. When an APS function wishes to read the label file associated with the first utterance in the database, it takes the string from the first element of `Utterances` and concatinates a label file extension onto the end of the string to form the complete pathname of the label file. The extension is determined by the selected database format for label files as described in Section 6.2.

For a function like `phon`, which by default reads all the utterances in the database, this operation is repeated for all the elements in `Utterances`. An

equivalent procedure is followed when forming track file names for functions such as **above**; in this case a track file extension is concatinated on the end of the string instead of a label file extension.

To illustrate, if we are using the first database format for label files shown in Table 6.1., then filename construction is Standard and the filename extension is **.mrp**. The label files associated with the first ten utterances shown above would therefor be

```
/DB/atr20/be_dr1/mgsw0/sc001/sc001.mrp
/DB/atr20/be_dr1/mgsw0/sc002/sc002.mrp
/DB/atr20/be_dr1/mgsw0/sc003/sc003.mrp
/DB/atr20/be_dr1/mgsw0/sc004/sc004.mrp
/DB/atr20/be_dr1/mgsw0/sc005/sc005.mrp
/DB/atr20/be_dr1/mgsw0/sc006/sc006.mrp
```

The **utt** fields of segment lists are decoded in the same way as **Utterances**. Suppose for example that the segment list

```
    start    end                                    utt
@ "89"   "101"  "/DB/atr20/be_dr1/mgsw0/sc001/sc001"
@ "1192" "1359" "/DB/atr20/be_dr1/mgsw0/sc001/sc001"
@ "1269" "1308" "/DB/atr20/be_dr1/mgsw0/sc002/sc002"
@ "1807" "1867" "/DB/atr20/be_dr1/mgsw0/sc002/sc002"
@ "90"   "122"  "/DB/atr20/be_dr1/mgsw0/sc003/sc003"
```

was passed to the function **maxv**, then the track files corresponding to each segment would be, respectively,

```
/DB/atr20/be_dr1/mgsw0/sc001/sc001.trk
/DB/atr20/be_dr1/mgsw0/sc001/sc001.trk
/DB/atr20/be_dr1/mgsw0/sc002/sc002.trk
/DB/atr20/be_dr1/mgsw0/sc002/sc002.trk
/DB/atr20/be_dr1/mgsw0/sc003/sc003.trk
```

It is quite possible to have relative pathnames in **Utterances**. This means that if you have a private database of a few files in, say, a neigbouring directory called **data** then **Utterances** may contain something like

```
[1] "../data/utt01" "../data/utt02" "../data/utt03"
[4] "../data/utt04"
```

The same rules for forming filenames apply to these as to complete pathnames so that the **Utterances** shown above would index the label files and track files

```
../data/utt01.mrp ../data/utt02.mrp
../data/utt03.mrp ../data/utt04.mrp
../data/utt01.trk ../data/utt02.trk
../data/utt03.trk ../data/utt04.trk
```

Files can be stored in the current directory by indexing with an `Utterances` such as

```
[1] "utt01" "utt02" "utt03" "utt04"
```

to access the files

```
utt01.mrp utt02.mrp utt03.mrp utt04.mrp
utt01.trk utt02.trk utt03.trk utt04.trk
```

6.2.2 Olivetti Filename Construction

APS recognises Olivetti internal formats for label files and track files. The manner in which filenames are constructed for these internal formats are somewhat different. Table 6.1. shows that there is currently one internal format for both label files and track files which use Olivetti filename construction.

Unlike Standard filename construction pairs of label files and track files with Olivetti filename construction are not stored in the same directory but are stored in parallel directories. The `Utterances` shown earlier would expect the following label files

```
/DB/atr20/be_dr1/mgsw0/sc001/lab/sc001.lab
/DB/atr20/be_dr1/mgsw0/sc002/lab/sc002.lab
/DB/atr20/be_dr1/mgsw0/sc003/lab/sc003.lab
/DB/atr20/be_dr1/mgsw0/sc004/lab/sc004.lab
/DB/atr20/be_dr1/mgsw0/sc005/lab/sc005.lab
/DB/atr20/be_dr1/mgsw0/sc006/lab/sc006.lab
```

The track files would be

```
/DB/atr20/be_dr1/mgsw0/sc001/lpc/sc001.lpc
/DB/atr20/be_dr1/mgsw0/sc002/lpc/sc002.lpc
/DB/atr20/be_dr1/mgsw0/sc003/lpc/sc003.lpc
/DB/atr20/be_dr1/mgsw0/sc004/lpc/sc004.lpc
/DB/atr20/be_dr1/mgsw0/sc005/lpc/sc005.lpc
/DB/atr20/be_dr1/mgsw0/sc006/lpc/sc006.lpc
```

As with Standard filename construction `Utterances` can have simple pathnames relative to the current directory.

6.3 Indexing a New Database

To get APS to read a new database you will have to create an `Utterances` dataset that indexes the appropriate files. This procedure would also be followed when installing APS for the first time in order to create a default database for APS users.

Take a look at Table 6.1. Do the internal formats and filename extensions of your label files and track files appear in the table? If they do, then things are fine – continue reading this section. If your the internal formats are in the table but the filename extensions are not the same, then you can correct this by creating symbolic links to your files with the extension shown in the table; alternatively you can add a new entry to the APS internal copy of Table 6.1. and recompile APS (the section below, "Adding New Database Formats", explains how to do this.) If your internal format is not in the table you will have to write some C functions that read your file format; follow the procedure described in the next section.

Assuming that you have one of the recognised internal formats and that you have the appropriate extension on the filenames, do the files appear in the correct place in the directory structure? If your internal format uses Standard filename construction then each label file must occur in the same directory as the corresponding track file. If your internal format uses Olivetti filename construction then corresponding label files and track files must be stored in parallel subdirectories whose names are the same as the filename extensions (`lab` for label files and `lpc` for track files). Label files and track files for separate utterances may be stored anywhere – at different levels of the directory structure or even different filesystems. The above restrictions apply only to matching pairs of label files and track files.

Once all the label files and track files are in place in an internal format recognised by APS it is time to create the dataset `Utterances` that will index these files. This is done using the APS function `mkdb`. Start APS and enter the command `mkdb` which will look something like this

```
> mkdb("/DB","syl","s")
```

Using the `find` command `mkdb` will create `Utterances` to index all files with a given filename extension in a given part of the filesystem.

The first argument to `mkdb` indicates the root of the directory structure that contains all the label files and track files. In the example `/DB` is at the root of a filesystem, but this could also be a pathname relative to the current directory, or the current directory itself. Because the elements of `Utterances` are propagated into the segments of segment lists it will save space if they can be kept as short as possible, either by using pathnames

relative to the current directory or by placing the database close to the root directory on the filesystem.

The second argument is the filename extension. Every file with this extension found beneath the given directory generates an entry in `Utterances`. The third argument indicates the filename construction. Refer to Table 6.1. Further details on `mkdb` may be obtained from the APS function documentation or by saying `help(mkdb)` while in S.

6.4 Adding New Database Formats

The table of database types shown in Table 6.1. is compiled into the executable file which includes both S and APS code. In order to alter this table it is necessary to make a new executable file for APS. This section describes how to do this.

In reverse order of difficulty these are the alterations which are likely to be made to the table. The easiest thing is to simply change the order of the entries in the table. This has the effect of assigning priority when more that one format exists. (See earlier sections.) To do this requires editing the table and recompiling APS.

A little harder would be defining a new method of filename construction in addition to the Standard and Olivetti filename constructions already defined. This requires writing one C function which takes the name of the utterance returns the actual filename.

Hardest is adding a new internal format to the list. This requires one C function for label files and five C functions for track files.

After a description of the internal table of database formats each of these tasks are described below. You will need to know how to perform the former tasks to perform the latter tasks.

6.4.1 Internal Table of Database Formats

The directory `src` in the APS distribution contains the internal copy of Table 6.1. in a file called `types.c`. This contains C source code that including two structures called `Labtypes` and `Tracktypes` which define the database formats for label files and track files respectively. With some `#ifdef` statements removed they look something like this

```
struct ftype Labtypes[] = {
/* First type is a cache for saving the previous type read */
"lab", ufilename_o, readlab,  0, 0, 0, 0,
"mrp", ufilename, readmrpa, 0, 0, 0, 0, /* MRPA */
"syl", ufilename, readmrpa, 0, 0, 0, 0, /* MRPA */
```

```
"trn", ufilename, readtrn,  0, 0, 0, 0, /* SAM-BA, OSPREY */
"seg", ufilename, readseg,  0, 0, 0, 0, /* ISD ASCII */
"lab", ufilename_o, readlab,  0, 0, 0, 0, /* Olivetti */
0, 0, 0, 0, 0, 0
};

struct ftype Tracktypes[] = {
/* First type is a cache for saving the previous type read */
"lpc", ufilename_o,  readlpc, nextlpc, tracknames_lpc,
  ntracks_lpc, frameshift_lpc,

/* AUDLAB track file */
"trk", ufilename,  readtrk, nexttrk, tracknames_trk,
  ntracks_trk, frameshift_trk,

/* AUDLAB track file */
"otrk", ufilename, readtrk, nexttrk, tracknames_trk,
  ntracks_trk, frameshift_trk,

/* AUDLAB track file with truncated PC filename extension */
"otr", ufilename,  readtrk, nexttrk, tracknames_trk,
  ntracks_trk, frameshift_trk,

/* Old "denjlsa001"-type AUDLAB track file */
"dbtrk", ufilename, readotrk, nextotrk, tracknames_otrk,
  ntracks_otrk, frameshift_otrk,

/* Old "denjlsa001"-type AUDLAB track file with truncated
 * PC filename extension */
"dbt", ufilename,  readotrk, nextotrk, tracknames_otrk,
  ntracks_otrk, frameshift_otrk,

/* Olivetti track file */
"lpc", ufilename_o,  readlpc, nextlpc, tracknames_lpc,
  ntracks_lpc, frameshift_lpc,

0, 0, 0, 0, 0, 0, 0
};
```

Both `Labtypes` and `Tracktypes` are arrays of the structure `ftype`. Each element is a database format that corresponds directly to a line in Table 6.1. The order in which database formats are tried is as given in the elements of

this array. Note that the first entry is a cache where the current database format is stored for speedy access. Do not store the only definition of a database format in this first element.

The definition of the structure **ftype** is found in the file **types.h**. This is it:

```
struct ftype {
  char *extn;           /* Filename extension */
  char *(*decoder)();   /* Function that translates an Utterance
                         * into a filename */
  int (*reader)();      /* Function that reads the file. Will read
                         * the whole file for label files and a
                         * selected track for track files */
  int (*nexttrack)();   /* Function that simply reads the next
                         * track that happens to be in the current
                         * track file */
  int (*tracknames)(); /* Function that gets the track names */
  int (*ntracks)();     /* Function that gives the number
                         * of tracks */
  int (*frameshift)(); /* Function that returns (and sets)
                         * FrameShift */
};
```

The field **extn** points to the filename extension for the database format. The remaining fields are all pointers to functions. These functions get called when APS needs to read a label file or track file. Most of the functions are used for reading track files only – those functions that do not apply to label files are set to zero in **Labtypes**.

For a definition of what these functions do see the comments associated with the structure **ftype** as shown above. The aps distribution directory **src** contains the source code for the Olivetti label file database format (filename extension **.lab** and Olivetti track file database format (filename extension **.lpc** in the files **readlab.c**, **readlpc.c** and **ufilename_o.c**.

6.4.2 Altering the Table of Database Formats

The order of the internal copy of Table 6.1. defines the priority of the database formats. If there are label files (or track files) for an utterance in two database formats then only the format which appears first in the table is ever read.

To change the order of the entries in the table you must edit the structure **Labtype** or **Tracktype** in the file **types.c** as appropriate. The order of the elements in these structures define the order of the table. Note that the

first database format in each structure is the cache for the current database format. Do not store the only copy of a database format here – it will be overwritten after every successful file access. The first, cache entry should contain a *copy* of the database format most often used at your installation.

After creating a new version of `types.c` it is necessary to recompile a new version of S with APS. Simply enter `load`. This will make a new version of the S executive which includes all the APS code and the new `types.c`. When this has completed after a few minutes you should see the message

```
Local version of S loaded
```

You will now have a large file called `local.Sqpe`. It contains all the S code and the APS code linked together. When you run S in the current directory you will use this version instead of the public version. To install this version in the public place simply move `local.Sqpe` to the S subdirectory `cmd` and rename it `aps.Sqpe`.

6.4.3 Adding a New Method of Filename Construction

The function pointer `decoder` pointer to the function that defines the method for filename construction, that is, how to convert the utterance name in `Utterances` to an actual label file name or track file name. The file `ufilename_o.c` in the APS distribution directory `src` contains the function `ufilename_o` which defines Olivetti filename construction. The form of this function, which is the same for all `decoder` functions, is as follows

```
char *
ufilename_o( utterance, type, dbdir )
char *utterance;
char *type;
char *dbdir;
```

It takes the utterance name, an element of `Utterances`, in the argument `utterance`. The filename extension is given in `type`. The argument `dbdir` is historical. It contains the value of the S dataset `Dbdirs` which may be used to specify a database in some way. The `decoder` function then constructs the filename from these arguments and returns this as the value of the function.

A new method of filename construction can be defined by modifying this function or by writing a new function that conforms to the specification just described. It can be used as part of a new database format by adding a new entry in the structure(s) `Labtypes` and/or `Tracktypes`. If you put a

function into a new file, then edit the file **load** in the directory **src** and add the new filename to the list

```
src="types.c ufilename_o.c readlab.c readlpc.c"
```

Follow the procedure described in the previous section to create a new **local.Sqpe**.

You should enhance **mkdb** so that it can create **Utterances** in the correct format for the new method for filename construction.

6.4.4 Adding New Internal Formats

If you have label files or track files in a format other than those shown in Table 6.1. you will have to write some functions and compile them into S. Refer to the comments in **types.h** as quoted above and to the example source code provided for Olivetti formats.

For label files you will have to write one function, pointed to by the function pointer **reader**, that reads a label file and returns the contents via one of its arguments. The file **readlab.c** contains the source for the function that reads Olivetti label files.

Track files require more work. You must provide a function for all the function pointers described in **types.h**. See the file **readlpc.c** for the source to all the necessary functions for Olivetti track files.

Follow the procedures described in the previous sections to create a new **local.Sqpe**.

Chapter 7

APS Functions

The following pages contain documentation for all APS functions and datasets. They are also available online from within S by typing `help(phon)` for example. Future changes to these functions will be reflected in the online copy.

above — Run Threshold on Track to Create Segments — above

above(trackname, threshold, difference = FALSE)
below(trackname, threshold, difference = FALSE)

ARGUMENTS

trackname character string indicating the track (acoustic parameter) on which to threshold. Possible values are given by the function **tracks**.

threshold value for the threshold. A number.

difference If set to TRUE will perform a first difference on the values in the track before thresholding. Default is FALSE - track is unmodified.

VALUE

An APS segment list where the track values of all the frames within the segments are less than (**below**) or greater-than-or-equal-to (**above**) the threshold.

For a description of the format of a segment list see the function **segs**.

EXAMPLES

```
below("f1",610.5)        # generate all the segments whose frames contain values
                         # less than 610.5 on the track "f1".
above("ediffnls",6,T)    # segments with continuous increase of 6.0 or more on
                         # track "ediffnls".
```

aps — Start Up APS — aps

aps()

When called within S, the function **aps** gives access to all APS functions and data. It does this by adding the appropriate directory to the search list of data directories.

SEE ALSO

attach

EXAMPLES

```
aps()
```

as.segs — see segs — as.segs

bathtub — Plot Superimposed Track Contours — bathtub

bathtub(segs, trackname, difference = FALSE, lhs = 20, rhs = 20, ymin = <min value>, ymax = <max value>, labx = "Normalised time", laby = trackname, color = 4)

ARGUMENTS

segs Segment list.

trackname character string indicating the track (acoustic parameter) on which to threshold. Possible values are given by the function **tracks**.

difference If this is TRUE then a first-difference is performed on the track.

lhs Amount of left context in milliseconds. As this is an absolute amount and the plot is produced with normalised time, short segments appear longer than shorter segments.

rhs Similar to **lhs** but for right context.

ymin The minimum track value to be plotted. All low values are plotted if **ymin** is not specified.

ymax The minimum track value to be plotted. All high values are plotted if **ymax** is not specified.

labx Character string for the x-axis.

laby Character string for the y-axis.

color Colour for the contour lines.

EXAMPLES

```
# Plot contours of closure segments.
bathtub( closure, "ediffnls" )
# Show positive values of rate of change of ediffnls.
bathtub( closure, "ediffnls", difference = T, ymin = 0.0 )
```

below — see **above** — below

cderr — Cumulative Distribution — cderr

cderr(x, y)

ARGUMENTS

x A set of values.

y Another set of values.

VALUE

A structure with two components. **vals** contains the values taken from **x** and **y** sorted in order. **err** contains the cumulative discrimination corresponding to the elements in **vals**.

EXAMPLES

```
cd ← cderr( stopvals, nonstopvals )
plot( cd$vals, cd$err )
```

class — Which Elements Belong to a Set? — class

class(x, set)

ARGUMENTS

x a vector.

set a vector.

VALUE

Returns a logical vector of the same length as **x** which is TRUE wherever the corresponding element of **x** occurs in **set**.

SEE ALSO

match

EXAMPLES

```
labels ← label( segs )
is.stop ← class( labels, STOP ) # Find the stops
segs[is.stop,]    # Select the stop segments
```

cm — Confusion Matrix — cm

cm(actual, confusions, classes, total)

ARGUMENTS

actual What phonemes they were. These appear as the rows of the matrix.

confusions What phonemes they were thought to be. These are columns of the matrix.

classes Only consider this subset of phonemes. Default is the complete set from **actual** and **confusions**.

total Are totals required at the ends of the rows and columns Default is T.

VALUE

A confusion matrix with labels suitable for printing with **tprint**.

SEE ALSO

tprint

EXAMPLES

```
hypotheses ← myPhonemeClassifier()
correctAnswers ← getAnswers()
cm( correctAnswers, hypotheses )
cm
```

Confusions:	m	n	ng	FA	Total
Actuals:					
m	44	4	5	2	55
n	14	53	7	12	86
ng	0	2	14	1	17
FA	1	0	1	119	121
Total	59	59	27	134	279

confirm	Prompt for Yes or No	confirm

confirm(str)

ARGUMENTS

str character string prompt message.

VALUE

TRUE if user replied with "y" or "yes"; FALSE if "n" or "no". **confirm** insists on getting one of these replies.

EXAMPLES

```
if(confirm("Do you really want to delete all your files?"))
        remove(ls())
```

Dbdirs	see **Utterances**	Dbdirs

diac	Diacritics	diac

diac(segs, diacs = "", offset = 0)
isdiac(segs, diacs = "", offset = 0)
Diacritics

ARGUMENTS

segs APS segment list

diacs Matching diacritics. This is a character string. Each character is matched against the diacritics in the label file. A zero length string (the default) matches any diacritic.

offset specifies neibouring segments in the label files. A positive value selects right neighbours (segments that follow in time): if offset is 1, then select the segment immediately to the right; if offset is 2, then select the segment 2 positions to the right; etc. A negative value selects a left neighbour. The default, 0, selects the segment indicated, unmodified.

VALUE

diacs returns the diacritic characters in the label file for the segments given by **segs**. Diacritics are returned as a character string, one character for each diacritic (see DIACRITICS below). Segments with no diacritics cause **diacs** to return zero length strings.

isdiac returns TRUE or FALSE depending on whether any of the set of diacritics given by the argument **diacs** occurs in the segment.

Diacritics is a dataset containing the table shown below.

DIACRITICS

These are the set of possible diacritics in CSTR label files. Only those marked with Y in the Used column are used by ATR. This list is available in the dataset **Diacritics**.

Diacritic	Used	Meaning
c	Y	Stop closure
!	Y	Stop burst
h	Y	Aspiration (includes any burst)
o		Devoicing
v		Voicing
r		Retroflexion
~	Y	Nasalisation
:		Lengthening
1		First portion of diphthong, or first (stop) portion of affricate
2		Second portion of diphthong, or second (fric) portion of affricate
'	Y	Primary stress
"	Y	Secondary stress
=	Y	Syllabic
f		Frication
[		Dentalisation
w		Labialisation
j		Palatalisation
x		Velarisation
?		Glottalisation
*		Syllable-boundary (on last phoneme of the syllable)

SEE ALSO

label, **Diacritics**

EXAMPLES

```
stops ← phon(STOP)

diac(stops)              # diacritics of all the stops
is.released ← isdiac(stops,"!h") # which stops are released
```

Diacritics — see diac — Diacritics

dur — Duration of Speech Segments — dur

```
dur(segs)
```

ARGUMENTS

segs An segment list.

VALUE

integer vector of segments' end-times minus the segments' start-times.

EXAMPLES

```
stop ← phon( STOP )
stem( dur( stop ) )      # Histogram of durations of all stops.
```

end — End Time of Segments — end

end(segs)
end(segs) ← values

ARGUMENTS

segs An APS segment list.

VALUE

The end time, in milliseconds from the start of the utterance, of all the segments in the given segment list. This function is equivalent to **segs[,2]**.

The form **end(segs) ← values** will set all the end times of segs to values.

SEE ALSO

start, **utt**, **dur**

EXAMPLES

```
# Get the closure part of all the stops
stop ← phon(stop)
burst ← maxp( stop, "ediffnls", T )
closure ← stop
end(stop) ← burst
```

eplot — Scatter Plots with Ellipses — eplot

eplot(..., nsdev = 1.96, dopoints = TRUE)

ARGUMENTS

... Any number of 2 columned matrices. Column 1 gives values for the x coordinate, column 2 the y coordinate. Each matrix is considered as a separate population and appears with its own ellipse.

nsdev An integer specifying at how many standard deviations to plot the ellipse(s).

dopoints if TRUE, then plot the points, otherwise just plot the ellipse(s).

Plots a scatter plot with fitted ellipses.

EXAMPLES

```
eplot(idata)

# Plot the ellipses only and at 2 times s.d.
eplot(idata,idata2,idata3,nsdev=X,dop=F)
```

findex — Index Segments From Frames — findex

findex(segs, lhs = 0, rhs = 0, markseg = F)

ARGUMENTS

segs segment list.

lhs an integer number. Include this much left context from each token in milliseconds.

rhs as **lhs** but for right context.

markseg If TRUE then will add an NA after the frames for each segment. This is for compatability with the related functions listed under SEE ALSO.

VALUE

Returns an integer vector with as many elements as there are frames for the segment list. Each integer indexes the segment in the segment list associated with each frame.

SEE ALSO

frames, ftfrac, bathtub

EXAMPLES

```
# Get frame values and labels at each frame.
fvalues ← frames(stop,"ediffnls")
stop.labs ← label(stop)
findx ← findex(stop)
frame.labs ← stop.labs[findx]
```

frames — Get Values in Every Frame of Segments — frames

frames(segs, trackname, difference = FALSE, lhs = 0, rhs = 0, markseg = FALSE)

ARGUMENTS

segs An APS segment list.

trackname character string specifying the track. Possible values are given by the function **tracks**.

difference If TRUE then **frames** performs a first-difference on the values in the frames. Default is FALSE - tracks are unmodified.

lhs an integer number. Include this much left context from each token in milliseconds.

rhs as **lhs** but for right context.

markseg If TRUE then will add an NA after the frames for each segment. This is useful when drawing using **lines**.

VALUE

A floating point vector. Contains the value of the given track from every frame in the segment list.

SEE ALSO

ftfrac, findex, bathtub

EXAMPLES

```
stop ← phon(STOP)
f ← frames(stop, "f1")
```

```
# Draw segment tracks on top of one another.
fvalues ← frames(stop,"ediffnls",,10,10,T)
ftimes ← ftfrac(stop,10,10,T)
plot(ftimes,fvalues,type="n")
lines(ftimes,fvalues)
```

FRIC — see Phonemes — FRIC

fround — Round Milliseconds to Nearest Frame — fround

fround(times, frameshift = 5)

ARGUMENTS

times Vector of times in milliseconds.

frameshift Time interval between frames in milliseconds. Note that this value is NOT obtained from the track files. If a frameshift is required other than the default of 5 milliseconds, then the user must set it accordingly.

VALUE

Times rounded to nearest frame boundary time.

EXAMPLES

```
fround( end(segs) )      # Get end times rounded to nearest frame.
fround( start(segs), 10 ) # Get start times rounded to nearest multiple
                  # of 10 msecs.
```

ftfrac — Times of Frames in Segments — ftfrac

ftfrac(segs, lhs = 0, rhs = 0, markseg = FALSE)
ftimes(segs, lhs = 0, rhs = 0, markseg = FALSE, fromzero = TRUE)

ARGUMENTS

segs Segment list.

lhs an integer number. Include this much left context from each token in milliseconds.

rhs as **lhs** but for right context.

markseg If TRUE then will add an NA after the frames for each segment. This is useful when drawing using **lines**.

fromzero (**ftimes** only) If set TRUE (the default) then frame times start at 0 for each segment. If set FALSE then frame times are taken directly from the segment and measure time from the start of the utterance.

VALUE

Returns a floating point vector representing the time at which each frame in the segment occurs.

ftimes returns the time measured in milliseconds for each frame. The argument **fromzero** specifies whether time is to be measured from the start of the segment or

from the start of the utterance.

`ftfrac` returns the time expressed as a fraction into the current segment. Values range from 0.0 to 1.0.

These functions are useful in conjunction with `frames` for plotting curves of tracks inside segments. See example below.

SEE ALSO

`frames`, `findex`, `bathtub`

EXAMPLES

```
# Draw segment tracks on top of one another.
fvalues ← frames(stop,"ediffnls",,10,10,T)
ftimes ← ftfrac(stop,10,10,T)
plot(ftimes,fvalues,type="n")
lines(ftimes,fvalues)
```

ftimes — see ftfrac — ftimes

GLI — see Phonemes — GLI

hist2 — Double Histogram From Two Sets — hist2

hist2(x, class)

ARGUMENTS

x Values for histograms.

class A logical vector of the same length as **x**. Elements in **x** are assigned to a histogram according to whether the corresponding elements of **class** are TRUE or FALSE.

`hist2` plots two histograms on top of each other.

EXAMPLES

```
hist2( stopduration, isvoiced ) # Plot stop durations for voiced and
                                # unvoiced stops.
```

indeces — Indexes from Logical Vector — indeces

indeces(x)

ARGUMENTS

x Logical vector

VALUE

returns indexes that correspond to TRUE elements of **x**.

EXAMPLES

which ← indeces[is.strange]

is.segs	see **segs**	is.segs

isdiac	see **diac**	isdiac

label	Label Segments	label

label(segs, offset = 0)

ARGUMENTS

segs APS segment list.

offset allows neighbouring phonemes to be returned in preference to the matching phoneme. A positive value selects a right neighbour: if offset is 1, then select the phoneme immediately to the right; if offset is 2, then select the phoneme 2 positions to the right; etc. A negative value selects a left neighbour. The default, 0, selects the matching phoneme.

VALUE

Returns the phoneme in the label file found at each segment. Start and end times must match exactly. See **luniq** if this is not suitable. Result is a character vector of phoneme strings. The possible set of phonemes is found in the object called **Phonemes**.

EXAMPLES

```
stop ← phon(STOP)
stoplist ← label(stop)          # identify the stops
lcontext ← label(stop,-1)       # identify the left context
rcontext ← label(stop,1)        # identify the right context
table(stoplist)                 # Produce a table of the phonemes
```

LIQ	see **Phonemes**	LIQ

luniq	Get Closest Matching Label	luniq

luniq(segs, verbose = FALSE)

ARGUMENTS

segs APS segments to be labeled.

verbose logical. If there is a tie for the most overlapping segment, then the first segment is indexed. This flag indicates whether a message is printed to warn the user of such an occurance.

VALUE

A character vector of phoneme strings. The possible set of phonemes are found in the object called **Phonemes**.

luniq matches the segment in the label file which has the greatest overlap. The amount of overlap for each potential matching label is the fraction of the total extent of the segment.

SEE ALSO

olap

EXAMPLES

```
clos ← below("ediffnls",808.0)
lab ← luniq(clos)
table(lab)
```

maxp — Time of Extreme Values Inside Segments — maxp

maxp(segs, trackname, difference=FALSE)
minp(segs, trackname, difference=FALSE)

ARGUMENTS

segs APS segments. See the function **segs** for a description.

trackname Character string specifying the track. Possible values aregiven by the function **tracks**.

difference If TRUE perform a 1st difference on the track values. Default is FALSE, the track is unmodified.

VALUE

Returns the time inside each segment that correspond to the maximum (minimum) value of the track. Values are in milliseconds measured from the start of the utterance - not the start of the segment. Result is a vector with length the same as the number of rows in **segs**. Missing values (NAs) indicate an error occured in reading the track file.

SEE ALSO

Functions **maxv** and **minv** return respectively the actual values at the maximum and minumum points.

EXAMPLES

```
minp(segs,"zcr")          # returns time of minimum for track "zcr"
maxp(segs,"ediffnls",T) # returns time of maximum rate-of-change in track "ediffnls"
```

maxv — Extreme Values Inside Segments — maxv

maxv(segs, trackname, difference = FALSE)
minv(segs, trackname, difference = FALSE)

ARGUMENTS

segs APS segments. See the function **segs** for a description.

trackname Character string indicating the track to operate on. Possible values are given by the function **tracks**.

difference If TRUE perform a 1st difference on the track values. Default is FALSE, the track is unmodified.

VALUE

Returns the maximum (minimum) value of the track inside each segment. Result is a real vector with length the same as the number of rows in **segs**. Missing values (NAs) indicate an error occured in reading the track file.

SEE ALSO

Functions **maxp** and **minp** return respectively the points in time where the maximum and minumum occur in segments.

EXAMPLES

```
minv(segs,"zcr")      # returns for each segment the minimum value in track "zcr"
maxv(segs,"hfd")      # returns for each segment the maximum rate-of-change
                      # in track "hfd"
```

meanv — Mean/Median Value in Segments — meanv

meanv(segs, trackname, difference = FALSE)
medianv(segs, trackname, difference = FALSE)

ARGUMENTS

segs APS segments. See the function **segs** for a description.

trackname character string indicating the track (acoustic parameter) on which to threshold. Possible values are given by the function **tracks**.

difference If **T** then perform a first difference on the values in the track. Deafult is **F** - tracks unmodified.

VALUE

Returns the mean (median) of the frames in each segment. Result is a real vector with length the same as the number of rows in **segs**. Missing values (NAs) indicate an error occured in reading the track file or, in the case of **medianv** allocating memory for sorting the data prior to finding the median.

SEE ALSO

maxv, **maxp**

EXAMPLES

```
meanv(segs,"f1")  # returns the mean for given segments in track "f1"
```

medianv(segs,"f2") # returns the median for given segments on track "f2"

medianv	see **meanv**	medianv

META	see **Phonemes**	META

midslice	Extract Middle Portion of Segments	midslice

midslice(segs, fraction)

ARGUMENTS

segs Segment list.

fraction A number between 0.0 and 1.0, the fraction of the segment.

VALUE

A segment list that is the middle part of the segments in **segs**.

EXAMPLES

midslice(phon(NAS), 0.5) # Nasals with the 1st and last quarters removed.

mincderr	Minimum Cumulative Discrimination Error	mincderr

mincderr(x, y)

ARGUMENTS

x One set of values.

y Another set of values.

VALUE

Returns the value at which the cumulative discrimination error is at minumum.

SEE ALSO

cderr

EXAMPLES

midcderr(stopvals, nonstopvals) # Minumum error for stop vs. non-stops.

minp	see **maxp**	minp

minv	see **maxv**	minv

mkdb	Set Up Utterances From a Database	mkdb

mkdb(directory = "", extn = "syl", fc = "s")

ARGUMENTS

directory character string of directory where data files are to be found. If this argument not given then files are searched for in the current directory.

extn character string giving the filename extension for matching files that define the database.

fc character string indicating the method of filename construction. Two methods are recogised: Standard indicated by "s" (the default, and Olivetti indicated by "o". See the APS User Guide for an explaination of filename construction.

mkdb creates **Utterances**, the data object used by APS functions to interprete a segment list (see **segs**).

SEE ALSO

Utterances, segs

EXAMPLES

```
# Create Utterances from all files with extension ".syl"
# in the current directory
mkdb()

# Create Utterances according to ".trk" files in /u2/sip/gsw/data
mkdb( "/u2/sip/gsw/data", ".trk" )

# Create Utterances using Olivetti filename construction
mkdb( "/DB/corpus", ".lab", "o" )
```

mplot	Plot Many Histograms By Phoneme Class	mplot

mplot(x, labs, classes = Phonemes$names[class(Phonemes$names, uniq(labs))])

ARGUMENTS

x a numeric vector of values.

labs a character vector of phoneme labels corresponding to **x**.

classes the uniq set of labels to plot. The default set are all of the labels given by **labs** in the order they occur in **Phonemes$names** so that each class of phoneme, such as all the stops, appear together.

This function will plot many histograms together. It is useful for comparing distributions of some values for different classes of phoneme.

EXAMPLES

```
all ← phon()    # Get all phonemes from the database
mplot( dur(all), label(all) )      # Compare duration distributions
```

mrpa — Obsolete — mrpa

mrpa is now called **phon**. See help for **phon**.

NAS — see Phonemes — NAS

phon — Make Segments From Phoneme Label Files — phon

phon(phonemes = "*", offset = 0)

ARGUMENTS

phonemes character vector describing the phonemes required for collection. The default is "*" which matches all phonemes and is therfore equivalent to **phon(Phonemes$names)**.

offset allows neighbouring phonemes to be returned in preference to the matching phoneme. A positive value selects a right neighbour (the phoneme that follows in time): if offset is 1, then select the phoneme immediately to the right; if offset is 2, then select the phoneme 2 positions to the right; etc. A negative value selects a left neighbour. The default, 0, selects the matching phoneme.

VALUE

An APS segment list corresponding to the phonemes in the label files.

Segments may be restricted to those whose label matches one of the strings given by the argument **phonemes** (see examples). For convinience there exist data objects such as **STOP**, **VOW** which index the more common phoneme classes. These sets may be concatinated, as in **SON ← c(VOW, NAS, LIQ, GLI)**, and/or created from strings, as in **NASw ← c("m","n","ng","w")**. The data object **Phonemes** contains the set of all the phonemes stored in the label files.

SEE ALSO

label

EXAMPLES

```
phon()          # get all phonemes
phon(STOP)      # get all stops
phon(STOP,+1)       # get segments following all the stops
phon( c("p","t","k") )    # get all voiceless stops
phon( c("s","sh") )       # get all /s/ and /sh/
```

Phonemes — The Set of Possible Phonemes — Phonemes

Phonemes has the following attributes:

names the character strings that occur in the label files. These are the possible strings returned by the functions **label**, **luniq**, etc.

freq the frequency with which the phonemes occur in a lexical corpus.

The following datasets contain useful subsets of **Phonemes$names**.

Dataset	Phoneme Set
VOW	Vowels
STOP	Stops
FRIC	Fricatives
NAS	Nasals
LIQ	Liquids
GLI	Glides
META	Special set of annotations:

##	- pause
.	- NOT ALLOCATED
-	- NOT ALLOCATED
*	- NOT ALLOCATED
<	- before start of utterance
>	- after end of utterance
FA	- false alarm

SOURCE

Phoneme frequencies were collected by M. Cooper from [XXX]

samplev — Sample Track Values — samplev

samplev(segs, trackname, points = c(0, 0.5, 1), smooth = TRUE)

ARGUMENTS

segs a segment list

trackname name of track to sample

points points within segments to sample. These are expressed as a fraction of the duration of the segment. The default - 0.0, 0.5 and 1.0 - returns the values at the start, middle and end of the segments.

smooth if TRUE, then does a Gaussian smoothing on the frame values. After smoothing the frame value for frame number N is equal to

$$0.25*value(N-1) + 0.5*value(N) + 0.25*value(N+1)$$

VALUE

A matrix of values from the track. The matrix has as many rows as there are segments and as many columns as the length of **points**. The values are read from the track file at the point in time given by **segs** and **points**.

EXAMPLES

```
# Sample F1 at the start, the middle and the end of vowels.
vowels ← phon(VOW)
```

f1 ← samplev(vowels,"f1")

segs	Segment Lists	segs

segs(start = rep(0,length), end = rep(0,length), utt = rep(0,length), length = 0)
as.segs(x)
is.segs(x)

ARGUMENTS

start integer vector of start times from the beginning of the utterance in milliseconds.
end like **start** but for the end times.
utt utterance names, a character vector that identifies the utterance as a filename without an extension. This is a copy of the appropriate element of the dataset **Utterances**.
length in the absence of the other arguments this gives the number of segments to initialise.

VALUE

segs returns an APS segment list. Its format is recognised by all APS functions as the means of describing segments of speech.

An APS segment list is an integer matrix where each row represents one segment. The first column, named "start", is the start time in milliseconds counted from the start of the utterance; the second column, named "end", is the end time in milliseconds. The third column, names "utt", is references the utterance as the filename without its extension. This is a copy of an element in the specially named dataset **Utterances**;

The function **segs** creates a segment list according to the values given in the arguments **start**, **end** and **utt**. If the only argument given is **length**, **segs** will create a segment list of **length** segments with all elements set to zero.

is.segs returns TRUE if **x** is a segment list and returns FALSE otherwise.

as.segs returns **x** as a segment list. **x** is treated as a vector, start times come first, followed by end times and then utts.

SEE ALSO

phon

EXAMPLES

 # Create 5 msec segments from time instants.
burst.segs ← segs(burst.time, burst.time + 5, utt(stop))

segsort	Sort a Segment List	segsort

segsort(segs)

ARGUMENTS

segs Segment list.

VALUE

The same segments sorted in the following order: firstly by utterance id, secondly by

start time, thirdly by end time. This is the standard order for segment lists returned by functions like **phon**, **above** and **below**.

SEE ALSO

segorder

EXAMPLES

```
weakFrics ← findwfrics()        # Get weak fricatives
strongFrics ← findsfrics()      # Get strong fricatives
frics ← segsort( c( weakFrics, strongFrics ) )   # Join them together
```

shift — Ajust Segment Times — shift

shift(segs, startshift = 0, endshift = 0)

ARGUMENTS

segs Segments to be shifted.

startshift An integer which indicates how many milliseconds to shift the start-time of segments. Negative values shift earlier, positive values later.

endshift As **startshift**, but ajusts the end-time.

VALUE

Result is a new segment dataset with start and end times altered.

EXAMPLES

```
shift(segs,-5,-5) # Return segments 5 msecs earlier
shift(segs,0,10)  # Move all end-time 10 msecs later
```

slen — Number of Segments in Segment List — slen

slen(segs)

ARGUMENTS

segs An APS segment list.

VALUE

The number of segments in the segment list.

EXAMPLES

```
slen(phon(NAS))        # How many nasals in the database?
```

slice — Chop Segments into Equal Pieces — slice

slice(segs, npieces)

ARGUMENTS

segs An APS segment list.

npieces Split segments into this many pieces.

Creates as many datasets as specified by **npieces**. Each new dataset has a name of the form segs.N, where segs is the name of the argument **segs** and N is a number from 1 to **npieces**.

EXAMPLES

```
slice(clos, 3)          # Makes three files, clos.1, clos.2 and clos.3
                        # for the each third part of clos.
```

splitdb — Split a Database into Sets — splitdb

splitdb(n)

ARGUMENTS

n an integer. How many pieces to make.

Creates **n** new datasets named USet.1, USet.2, ..., USet.N each of which is a subset of **Utterances**. The members of **Utterances** are distributed on a round-robin basis. For example, if **n** is 3 then USet.1 will contain elements 1, 4, 7, ..., of **Utterances**, USet.2 elements 2, 5, 8, ..., and USet.3 elements 3, 6, 9, ...

EXAMPLES

splitdb(3)

start — Start time of Segments — start

start(segs)
start(segs) ← values

ARGUMENTS

segs An APS segment list.

VALUE

The start time, in milliseconds from the start of the utterance, of all the segments in the given segment list. This function is equivalent to **segs[,1]**.

The form **start(segs) ← values** will set all the start times of segs to values.

SEE ALSO

end, utt, dur

EXAMPLES

```
# Get the closure part of all the stops
stop ← phon(stop)
burst ← maxp( stop, "ediffnls", T )
closure ← cbind( start(stop), burst, utt(stop) )
```

STOP	see **Phonemes**	STOP

tracks	Track Names	tracks

tracks(utterance = Utterances[1], duplicates = FALSE)

ARGUMENTS

utterance Utterance from which tracks are to be read. This is a character string in the same format as **Utterances**.

duplicates When there is more than one track file for the utterance, it is possible that tracks are duplicated. If **duplicates** is TRUE, then return all the track names regardless whether they duplicated or not. Note, however, that APS only ever reads the first track that matches a trackname. If **duplicates** is FALSE, the default, then duplicates are removed.

VALUE

vector of character strings containing the track names in the track file. Normally **tracks** is called with no argument causing it to return those tracks available in the current default database.

A call with no argument causes **tracks** to read the track names from the first utterances in **Utterances**. All APS functions consider the first utterance as defining the set of tracks in all the remaining utterances.

EXAMPLES

```
# List the available tracks.
tracks()

# Store track names from given track file.
mytracks ← tracks( "/u2/sip/gsw/data/jl10s" )
```

EXAMPLES

```
# How many utterances are there?
length(Utterances)

# Reduce working database to the first 98 utterances
Utterances ← Utterances[1:98]
all ← phon() # ... and get all the phonemes there.

# Utterances passed as an argument
stops ← phon(STOP,u="denjlsb045")
labs ← label(stops)
```

Utterances.db	see **Utterances**	Utterances.db

VOW	see **Phonemes**	VOW

wmrpa	Write Out AUDLAB MRPA Label Files	wmrpa

wmrpa(segs, labels = label(segs), desc="S output", dir=".", extn=".mrp")

ARGUMENTS

segs Standard APS segment list.

labels Character string indicating the labels which correspond to the **segs** list.

desc A quoted character string which is stored in the MRPA fileheader field descr (max length is 16 characters).

dir The UNIX directory in which to write the files.

extn The extension to give to the filenames.

This function writes out a set of MRPA label files using data stored in S lists. Each output file takes its name (without any path information) from **utt** in the segment list and has the extension ".mrp" attached to it. The output directory path may be set by the argument **dir** or the environmental varible WMDIR; the current directory is the default.

SEE ALSO

phon

EXAMPLES

```
wmrpa(detected.segs,matched.labs,"BC_NASALS")
```

wsegs	Write Segment Files for Lisp	wsegs

wsegs(segs, labels = label(segs), dir = ".", extn = ".seg")

ARGUMENTS

segs APS segment list.

labels character strings of labels corresponding to the segments. This is obtained via **label** by default. If only one string is given, then this is used for all segments.

dir directory in which to write files.

extn extension for filenames.

wsegs writes segments out to files in a form accepted by Lisp.

A segment appears as a line in the appropriate file of the form

(<start-time> <end-time> <label>)

The form of each filename is

dir/<stem-of-utterance-name>**extn**

EXAMPLES

wsegs(clos, dir = "../segs")