This document is under construction,
and will be updated during the course.
Last updated: March 11, 2013

Hints for using R

Bart de Boer
Vrije Universiteit Brussel
University of Amsterdam
bart@ai.vub.ac.be

R is a tool for doing **statistical analysis**. It is somewhat less user friendly than graphical user-interface packages like SPSS and SAP (but remember, those started life without a graphical user interface, too) but it is more flexible, it is free, it is continually being extended by a large user base and it is therefore rapidly becoming the academic standard for doing statistics. However, if you are comfortable using these other tools, feel free to continue to use them. The somewhat steep learning curve of R may not pay off in that case.

R can be **downloaded** from: http://www.r-project.org/ Because there are so many people downloading R, they ask you to choose a server that is close to you. There are several servers in the Netherlands. Follow the instructions after choosing your computer type (Windows, mac or linux). You should generally be fine with installing the latest version.

It is also useful to have a **spreadsheet program** to use for user-friendly data entry. Microsoft Excel is fine, but if you want a free alternative, you can download and install open office Calc: http://www.openoffice.org/product/calc.html

If you start R, you get at least one window that is called the **R Console**. This is where you type in commands (shown in `typewriter font` in this document) and where results will be printed. While you are making graphs, or editing files, other windows may appear.

This document is meant to be a **get-started-quickly** guide to using the bare basics of R, but should nevertheless give you access to enough functionality to do most statistical things you want to do. Commands have been tested on a PC (R version 2.14.0) and on a Mac (R version 1.40). I haven't tested it on Linux, but chances are that if you are a Linux-user you don't need this manual anyway. It is meant to be usable by people with very little computer experience, but it does require some intelligence and some playing around by yourself. Study the examples, and try small variations on them by yourself. Also use the excellent built-in help functions of R, and especially the examples provided there!

R as a calculator

One way to use R (and the only one discussed here) is like a large calculator in which you can type in all kinds of commands, but more importantly, store lots of intermediate values of your calculations. I will call such stored values "variables" (this is computer science terminology). You can name the values yourself. For example:

```
fancyVariableName <- 2
```

(don't forget to hit the Enter key after each command) stores the value 2 and labels it fancyVariableName if you then type `fancyVariableName` and hit Enter, R prints:

```
[1] 2
```

which means that the first value in `fancyVariableName` is 2.
You can do all kinds of calculations on a variable. For example,

```
sqrt(fancyVariableName)
```

gives you the square root of the value stored:

```
[1] 1.414214
```

In fact, all variables in R can be list of values (hence the [1]). Try:

```
x<-seq(1,10)
sqrt(x)
```

Basically anything you can do on a single value, you can do on a sequence of values as well. Of course, you can store the outcome of a calculation again. Try:

```
y<-sqrt(x)
y*y-x
```

Note that, although you would expect the final result to be a list of zeros, in fact you get a list of very small values in "scientific notation". A value of `4.440892e-16` (the second value in my list, but you may get another value) means 4.440892×10^{-16}, or:
0.0000000000000004440892
Such small values can generally be taken to be equal to zero, but the reason that they sometimes appear is that the computer has limited precision. This is a bit of a technical detail, just be prepared to sometimes see values that look like this.
Values in a list can also be looked at individually. If you have created `y` using the previous examples

```
y[2]
```

Will give you:

```
[1] 1.414214
```

you can also set values this way

```
y[2] <- 2
```

will set the second element of `y` to 2, and will erase the previous value. You can also look at a range of values, Try:

```
y[2:4]
y[-3]
y[c(2,4,6)]
```

and try to figure out what `:` and `c(...)` do.
You can also store text in variables, but then you need to surround the text with quote marks. Single quotes and double quotes are allowed, but don't mix them:

```
x<-'hallo'
```

and

```
x<-"hallo"
```

are fine, but:

```
x<-"hallo'
```

is not.

Functions

You have seen three *functions* in the examples above: `c`, `seq` and `sqrt`. Functions take input (called input parameters) and produce output (which can be printed or stored in a variable, as you have seen). A function like `sqrt` is like a mathematical function: its output is the list

of square roots of the values in the list that you gave it as input. Many standard mathematical functions can be found in R, such as `sin` for sine, `log` for logarithm, etc.
The `seq` function is an example of a slightly different kind of function: it takes as input parameters a number of parameters that and then does something with these values to produce a list, or another kind of output. For example, the function `seq` as shown above takes as input parameters the starting value of the list, and the end value of the list and then creates the list of integer (whole) values in between. For example, `seq(1,10)` gives you the values from 1 to 10. The function `seq` is slightly more flexible, though. You can specify the step size between the values. Try:

```
seq(1,10, by=4)
```

Alternatively, you can specify the number of steps between the start and the end points. Try:

```
seq(1,10,length.out=5)
```

Note how you specify the value of these parameters by name. This is in general the way things are done in R. In fact, you can specify all values for the `seq` function like this:

```
seq( from=1, to=10, length.out=5 )
```

This is a lot of typing, but it is easier to understand. Also, now the order in which you specify the parameters doesn't matter anymore:

```
seq(to=10, length.out=5, from=1  )
```

does exactly the same thing.
Functions are also used for all kinds of practical matters in R, such as plotting. Try:

```
x<-seq(0,10, length.out=100)
plot( x, sqrt(x))
```

Typing less

If you use the same commands a lot (and rest assured, you will). It pays to store the ones you use a lot in a file. You can do this by choosing "New script" in the "File" menu of R. A new window (the R Editor) will open, and you can type and save your favorite commands there. If you want to execute a command in the R editor, move your cursor onto the command and type Ctrl-R on windows or cmd-Enter on the Mac. You can also select multiple lines and execute them all at once. Once you have saved your commands, you can re-open it using "Open script" in the "File" menu. You can in fact automate a lot of processing using scripts in R, but these hints are not to place to go into that.
It is strongly recommended, though, to store your commands in a script, especially when you are making plots. This way you can fiddle around until your plot looks great, without having to retype lots and lots of commands.

Helping yourself

Once you have figured out the name of a function, it is easy to get help through the R help function (under the menu "Help" in windows, while for Mac OS there is a special Help Search box, at least in my version of R). Alternatively, you can type `help("plot")` (where of course you can replace `plot` by any name of a function you want help about). Figuring out the name of the correct function can be pretty frustrating, but this hints file is meant to get you started. For more in depth knowledge, there are excellent manuals and tutorials on the R website, while Google can be your friend, too.

Loading data

```
yourTable <- read.csv( file.choose())
```

Reads a comma separated value (a human readable file format that is somewhat of a standard for open source packages) using a file that you select using the graphical user interface. The data are stored in a variable called `yourTable.` If you want to see what is in it type `yourTable` and hit Enter. If you use the table that is given in the next paragraph, you should get something like:

```
  Type1 Type2
1   3.8   1.6
2   4.1   1.9
3   3.3   2.1
4   2.5   1.5
```

Usually data files tend to be so long that they don't fit on your screen. In that case, just use: `yourTable[1:5,]` (don't forget the comma!). Some versions of Microsoft Excel use the semicolon (;) to separate columns in a comma separated value file. If you try to read such a file with the above commands, you get columns that have semicolons in them, something like:

```
  Type1.Type2
1     3.8;1.6
2     4.1;1.9
3     3.3;2.1
4     2.5;1.5
```

No worries, if you see something like this, use:

```
yourTable <- read.csv( file.choose(), sep=";")
```

to set the column separator to the semicolon.

Note that if you use the (for example) Dutch locality settings in Microsoft Excel, you tend to get problems like the one mentioned above. Perhaps the simplest solution is to change the locality settings to those of the USA, as software tends to be a bit US-o-centric anyway.

Try typing in a small data file in your favorite spreadsheet program, for example:

Type1	Type2
3.8	1.6
4.1	1.9
3.3	2.1
2.5	1.5

Note that there are no spaces in the column names. It is somewhat safer (in the sense that you will have fewer compatibility problems) to not use spaces.

You can then save this as a .CSV file (in my version of Excel by going to "File->Save As" and then choose the CSV file type that fits your operating system, although that shouldn't really matter). CSV files are just text, so you can open them with your favorite text editor as well. You can also make them by hand if you feel so inclined.

R can also write files back to disk using the following command:

```
write.csv( yourTable, file.choose( new=TRUE ))
```

where of course you should put the name of the table/list you want to save instead of `yourTable`.

Note that R reads in the column names. You can now refer to the columns using these names. Try:

```
yourTable$Type1
yourTable$Type2
```

Selecting data

Suppose you have a table with both categorical and numerical data in it, such as:

Gender	**Adult**	**F1**	**F2**
female	yes	351	967
male	yes	247	854
male	yes	268	862
female	yes	328	972
female	no	379	1023
male	no	368	1031

And you have read this table to a variable called `Formants`. As you have seen, you can select all elements from column `F1` using

```
Formants$F1
```

But can you select all female formants? Yes you can!

```
Formants[Formants$Gender=="female",]$F1
```

Note the double == (this means equal to in R, a single = would not work!) Also note the lonely comma before the closing square bracket. This indicates you are selecting rows.
If you leave out the last `$F1`, you create a new table that contains only the female entries.

```
FemaleFormants<-Formants[Formants$Gender=="female",]
```

If you then want all the second formants of all adult females you can make a new selection:

```
FemaleFormants[FemaleFormants$Adult=="yes",]$F2
```

You can achieve the same result with one command by using:

```
Formants [Formants $Gender=="female"& Formants $Adult=="yes",]$F2
```

I would recommend using the step-by-step selection method, as you never know when you might need the intermediate results again.
You can use any condition (that makes sense) as a selection criterion. For example, if for some odd reason, you are interested in the F1 that are lower than 300 or F2 that are higher than 1000, you can use:

```
Formants[Formants $F1<300| Formants $F2>1000,]
```

And if you are interested in the row with the minimal value for F1, you can use the function `which.min(...)`:

```
table[which.min(table$F1),]
```

I am sure you can guess which command to use for the maximum.

Descriptive stats

You can now calculate descriptive statistics about the columns in the data you read from your file, using for example:

```
mean( yourTable$Type1 )
median( yourTable$Type1 )
sd( yourTable$Type1 )
```

```
        quantile( yourTable$Type1, 0.25 )
```

for the mean, the median, the standard deviation and the first quartile, respectively.
Calculating *t*-scores for the elements of the table is an interesting exercise in how R works. You can use the following sequence of commands:

```
        Mean.Type1 <- mean(yourTable$Type1 )
        SD.Type1 <- sd(yourTable$Type1 )
        T.Type1 <- (yourTable$Type1-Mean.Type1)/SD.Type1
```

Note that you created variables called `Mean.Type1, SD.Type1 and T.Type1`. If you want to see the *t*-values themselves, you can type:

```
        T.Type1
```

You could have achieved the same result by directly doing all calculations in one command:

```
        yourTable$Type1-mean(yourTable$Type1)/sd(yourTable$Type1)
```

but as a rule of thumb, you make fewer errors and it is easier to correct them if you split up a calculation.
Cohen's *d* (a measure of effect size of differences of means) can be computed in a completely analogous manner by calculating the difference between two means and dividing it by the standard deviation. Calculating the pooled standard deviation:

$$\bar{\sigma} = \sqrt{\frac{(n_1 - 1)\bar{\sigma}_1^2 + (n_2 - 1)\bar{\sigma}_2^2}{n_1 + n_2}}$$

from the standard deviations of the groups and the group sizes is a good exercise in using R as a calculator, especially if you know that you can calculate the variance (σ^2) of a list `x` with `var(x)` or calculate the squared value of a variable `y` with `y^2`, and that you use `*` for multiplication.

Inferential stats

If you want to test whether the mean of a given (normally distributed) data set is significantly different from zero you can use the command `t.test` :

```
        t.test(yourTable$Type1 )
```

R then gives you the following output:

```
                One Sample t-test


        data:  yourTable$Type1
        t = 9.794, df = 3, p-value = 0.002262
        alternative hypothesis: true mean is not equal to 0
        95 percent confidence interval:
         2.312091 4.537909
        sample estimates:
        mean of x
            3.425
```

This gives you quite a bit of information about the data set (assuming it is normally distributed!) It gives you the values you would report for a significance test: a t-value, the number of degrees of freedom (df) and the *p*-value. You would report this as: *t*(3) = 9.79, *p* = 0.0023 (note that I round off at 2 significant digits). It also gives you the 95% confidence interval of the mean: in 95 out of 100 samples, the mean is expected to be betyouen 2.31 and

4.54 Finally it gives you the mean of the sample, calculated in the usual way. The confidence interval can be reported in a paper as: $\mu = 3.43 \pm 1.11$
If you want to test whether the mean is different from another number, you can give that as a second parameter. Try:

```
t.test(yourTable$Type1, mu=3 )
```

Is the difference significant?
If you have paired data (that is two measurements per participant) and this data is stored in the table such that each line contains the data for each participant (so in our example table, participant one would have values 3.8 and 1.6, participant two 4.1 and 1.9 etc.), you can do a paired t-test as follows:

```
t.test(yourTable$Type1, yourTable$Type2, paired=TRUE )
```

if, on the other hand, you have unpaired data (your columns are derived from a different list of participants, for example) you can do an unpaired t-test as follows:

```
t.test(yourTable$Type1,yourTable$Type2, paired=FALSE )
```

In this case, you may want to investigate whether the variance of your two groups of difference. For this you should do an F-test, and this can be done as follows:

```
var.test(yourTable$Type1,yourTable$Type2 )
```

This gives you an output that is formatted similarly to that of the t-test:

```
        F test to compare two variances

data:  yourTable$Type1 and yourTable$Type2
F = 6.4505, num df = 3, denom df = 3, p-value = 0.1601
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
  0.4178038 99.5912094
sample estimates:
ratio of variances
          6.450549
```

which could be reported as: $F(3,3) = 6.45$, p = 0.16 F is the ratio of variances (variance 1 divided by variance 2), and you expect it to be 1 if the variances are equal. As you can see, for an insignificant difference, the confidence interval contains 1.

Non-parametric tests

If you cannot assume a normal distribution of your data, you should use non-parametric tests. You can test for normality using the Shapiro-Wilk test. Try:

```
shapiro.test( yourTable$Type1 )
```

and you will get the output:

```
        Shapiro-Wilk normality test

data:  yourTable$Type1
W = 0.9533, p-value = 0.7369
```

Where W is the "test statistic" (the value you would have had to calculate by hand and look up in a table 30 years ago). You would report that the null-hypothesis that the data is normally distributed cannot be rejected using the Shapiro-Wilk test, with $W = 0.95$, $p = 0.74$

This is to be expected, as the data are indeed normally distributed (I know, because I generated them myself).
If data sets are not distributed normally you need to use parameter free tests, for example the Wilcoxon tests. By the way, these are safe to use for normally distributed data, so if you are unsure about your data, it is better to err on the side of caution and use a parameter-free test (although this will generally be less powerful). In R you can use them completely analogously to the t-test:

```
wilcox.test( yourTable$Type1 )
wilcox.test( yourTable$Type1, mu=3 )
wilcox.test( yourTable$Type1, yourTable$Type2, paired=TRUE )
wilcox.test( yourTable$Type1, yourTable$Type2, paired=FALSE )
```

Simple plots

In order to appreciate the plot functions, it is useful to make a few lists:

```
t <-seq(-1,1,by=0.01)
d1<-sin(pi*t)
d2<-t^2
```

You can now make a simple plot with:

```
plot( d1 )
```

or if you want to plot one variable against another:

```
plot( t, d1 )
```

If you prefer lines you can use:

```
plot(t,d1, type="l")
```

If you like red lines, use :

```
plot(t,d1, type="l", col="red")
```

You can see that each time you use plot, a new plot is created. If you want to add a line to an existing plot, use:

```
lines( t, d2 )
```

if you want to plot points instead of lines, you can use:

```
plot(t,d1, type="p")
```

and the somewhat counterintuitive :

```
lines( t, d2, type="p" )
```

Boxplots can be made using the `boxplot` command. Suppose you want to plot data from variables men, women and children, then you would give the command:

```
boxplot( men, women, children)
```

and the men, women and children are plotted from left to right. Try intitializing these variables with:

```
men<-43.5+ 1.5*rnorm(100)
women<-36.5+1.5*rnorm(100)
children<-30+4.5*rnorm(100)
```

to have a pseudo-shoesize example.
You can make your boxplots a bit more usable by setting labels and colors. Try:

```
boxplot( men, women, children, notch = TRUE, names = c("men",
"women", "children"), col=c("light blue", "pink", "yellow"),
ylab="shoe size" )
```

You can make a histogram by using:

```
hist(men, freq=FALSE )
```

although for the given data set, that is not going to result in a very interesting plot. This function has many parameters (see the help function) but breaks is an especially relevant one: with it you specify the borders (breaks) between the bins. If you wanted to plot a histogram of shoe sizes, for example you could use:

```
hist(men, freq=FALSE, breaks=seq(36.5,48.5,by=1 ))
```

(where you can adjust the start and endpoints of the bins, but note that like this, a bin for any shoe size, say 44, will go from 43.5 to 44.5).

Areas of a graph can be filled using the polygon-function:

```
polygon(xcoord, ycoord, col='red')
```

if you want to plot a red square, you can use

```
plot.new()
xcoord = c( 0,0,1,1 )
ycoord = c( 0,1,1,0 )
polygon(xcoord, ycoord, col='red')
```

Unless you have already opened a window with a previous plot command, you need to use `plot.new()` otherwise the polygon function doesn't have a window in which to draw. Also note that the function `c` concatenates a series of numbers you specify.

There are many things that you can change about the appearance of a plot. The use of the parameter `col` to change the color is an example. Other examples are: `xlab` and `ylab` for the x-axis label and y-axis label, respectively and `xlim` and `ylim` for the x-limits and y-limits for example. Try:

```
plot(t,d1, xlab='x', ylab='sin(x)', xlim=c(-4,4),ylim=c(-2,2))
```

More about parameters can be found in the help function for `plot` and in the help item for general graphics parameters: `par`.

If you want to print out your plot and hang it above your bed (or more usefully, save it to a file so that you can use it in a document you are writing) things work slightly differently on my PC and on my Mac. On the PC, I need to right-click on the plotting window and choose the option "Print…" or "Save as postscript…". On my Mac, I need to select the plotting window and then choose "Print…" or "Save" in the file menu. When I select "Save" the plot will be saved as a .pdf file.
Alternatively, you can use the `pdf` command:

```
pdf( file=file.choose( new=TRUE ), title="Your Title", paper="a4" )
```

But that doesn't appear to work on both my Mac and my PC.

Basic analysis of variance

If you have your data in a table that has as columns the factors (independent variables) and the measurements (dependent variables) doing an analysis of variance is straightforward, but the R way of doing it illustrates their view of analysis of variance as fitting a model to the data. Suppose you have a table like the one you have seen before, but now you have multiple measurements per speaker. A fragment of this table could look like this:

Speaker	**Gender**	**Adult**	**F1**	**F2**
a	female	yes	351	967
a	female	yes	328	972
b	male	yes	247	854
b	male	yes	268	862

Lets assume you have loaded this table in a variable called `Formant`. Note that you can't do much with a table this small, you need a larger table. Also note that for an analysis of variance to make sense, you need more than two values (levels) per factor (this would be the case for the speakers in this table) or if you want to look at interactions between factors. Finally, note that your levels should be text strings, not numbers (there is probably a way around this last requirement, but I don't know it).

You could use a basic one-way analysis of variance (ANOVA) to investigate whether there may be significant differences between speakers. In R you can use the `aov` function.

```
MyLM<-aov(F1~Speaker, data=Formant )
```

(for the mathematically curious: LM stands for Linear Model, as R considers analysis of variance as a special case of linear model). Note that this function does not print output, yet. It only creates a variable that contains information about the analysis.

You can now create a standard ANOVA-summary table using the command summary:

```
summary(MyLM)
```

and you get something like:

```
            Df Sum Sq Mean Sq F value Pr(>F)
Speaker     75 486755    6490   9.193 <2e-16 ***
Residuals   76  53656     706
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Which is in fact what you get if you do an analysis of variance of the data for the vowel [i] from Peterson & Barney (1952) of the null-hypothesis that the mean value of F1 does not depend on speaker. It seems that F1 is different for different speakers. This is not very surprising, as you have heaped all speakers (adult, child, male and female) together. However, it is straightforward to investigate a more complex model (a two-way ANOVA).

If you want to investigate the effect of gender, you can do this as an ANOVA:

```
MyLM<-aov(F1~ Gender+Speaker, data=Formant )
```

And you get:

```
            Df Sum Sq Mean Sq F value   Pr(>F)
Gender       1 108323  108323 153.433  < 2e-16 ***
Speaker     74 378432    5114   7.244 6.19e-16 ***
Residuals   76  53656     706
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Note that the order of the factors matters in the `aov` command:

```
MyLM<-aov(F1~ Speaker+Gender, data=Formant )
```

Does not work, as each speaker has only one gender, whereas each gender has multiple speakers! This also means that it is not meaningful to investigate the interaction between these two factors.

If you want do a post-hoc test of differences in mean after an analysis of variance, you can do a Tukey Honest test of Significant Differences. In the Peterson and Barney data set, there is a factor Type, with levels "female", "male" and "child" if you do a one-way ANOVA on this factor with the following command:

```
    MyLM=aov( F1~Type,data=pbi)
```

You can see from the summary (not shown) that the difference is significant. But it doesn't tell you between which groups (that is not ANOVA's job). Therefore you use:

```
    TukeyHSD(MyLM, "Type",ordered=TRUE)
```

Which gives the following output:

```
     Tukey multiple comparisons of means
       95% family-wise confidence level
       factor levels have been ordered

  Fit: aov(formula = F1 ~ Type, data = pbi)

  $Type
           diff      lwr       upr    p adj
  w-m 43.69859 22.77155  64.62563 6.10e-06
  c-m 93.48788 68.12510 118.85066 0.00e+00
  c-w 49.78929 23.72844  75.85013 3.67e-05
```

from which you learn that all differences are significant (and besides, you learn the confidence intervals for the differences).

A parameter-free alternative to the one-way analysis of variance is the Kruskal-Wallis rank sum test. The R-command for investigating the difference in F1 between speakers is:

```
    Kruskal.test( F1~Speaker, data=Formant )
```

Which for the Peterson and Barney data set would give you: the following output (note that this test does not require you to calculate a model first).

```
  Kruskal-Wallis rank sum test

data:  F1 by Speaker by Type
Kruskal-Wallis chi-squared = 141.595, df = 75, p-value = 5.349e-06
```

Note that I have been using ANOVA like there is no tomorrow. However, if you are serious you should first check for normality (using e.g. the Shapiro-test) and then check for equality of variance (using the F-test – called `var.test` in R). You will probably find that none of the assumptions of ANOVA are met in the Peterson and Barney data set. Fortunately ANOVA is pretty robust to violations of its assumptions, but the lack of real data meeting ANOVA's assumptions is one of the reasons I prefer simple designs with parameter-free tests.

Basic correlation and regression

To play around with the correlation functions, you can generate the following test data vectors:

```
    x<-rnorm(100)
```

```
y<-x+rnorm(100)
```

A correlation between these two vectors is calculated with the function `cor`:

```
cor(x,y)
```

Note that correlation assumes that data are paired – that is, each data point has a value for x and for y), so vectors `x` and `y` should have the same length.
The explained variance is given by:

```
cor(x,y)^2
```

and the unexplained variance (do I even have to tell you this?) by:

```
1-cor(x,y)^2
```

If you prefer to use Spearman's rho or Kendall's tau (after all, they are more robust to departures from linear trends) you do:

```
cor(x,y, method="kendall")
cor(x,y, method="spearman")
```

You can test for significance on these correlations with the `cor.test` function:

```
cor.test(x,y)
```

With which you get an output that looks like:

```
        Pearson's product-moment correlation

data:  x1 and y1
t = 10.3676, df = 98, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.6140623 0.8052698
sample estimates:
      cor
0.7232474
```

If you prefer to use Kendall's tau or Spearman's rho as the basis of your test, knock yourself out:

```
cor.test(x,y, method="kendall")
cor.test(x,y, method="spearman")
```

R automatically reports the relevant statistics.
If you want to do a regression analysis of correlated data, you could of course calculate the relevant parameters of your regression line using R as a calculator. There is a more user-friendly way that uses R's ability to fit a linear model to the data using the function lm.

```
yourLM<-lm(y~x)
```

Typing `yourLM` and pressing Enter gives you the information you need. You get something like:

```
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)          x
   -0.04709    0.96313
```

Where (intercept) is the predicted value if x is zero, and the value below x is the slope of the line. The regression line for this data set would then be:

$$y' = 0.96x - 0.05$$

which is pretty close to the correct $y' = x$ (which you knew because you generated the data yourself).

If you have plotted your data points, you can add the trend line to your plot using:

```
abline( yourLM )
```

References

Peterson, G. E., & Barney, H. L. (1952). Control methods used in a study of the vowels. *The Journal of the Acoustical Society of America*, **24(2)**, 175–184.