Basic plots in R

27 June 2018

Quantitative Methods in Historical Linguistics
Henri Kauhanen

Contents

1 Data 1

2 Barplots 2

3 Scatterplots 7

4 Lineplots 12

5 A note on formulae 15

6 Plotting a logistic function 16

7 A complete example 20

8 Exporting graphics 24

9 A note on graphics packages 26

10 Help? 26

1 Data

In this tutorial, I will guide you through the steps of making some of the most basic **plots** (i.e. graphs, diagrams) in R. Plots are incredibly useful for many purposes: for exploratory data analysis, for presenting information in a paper/publication/portfolio, and so on — so it is important to know how to get started.

I will build this tutorial mostly around some of the data we have seen in the lectures, namely Glaser's[1] data on the loss of final fortition in Southern German. Download this from ILIAS (`german_fortition.csv`) and load it into R:

[1] Glaser (1985). Also see Fruehwald, Gress-Wright & Wallenberg (2013).

```
gf <- read.csv("german_fortition.csv")
```

To see that the data loaded succesfully, simply type `gf`:

```
gf

##   date  p  b  t  d  k  g
## 1 1276 18  0 29  0 54 19
## 2 1373 10  8 24  5 17 59
## 3 1483  2 16  2 22  0 78
## 4 1523  2 14  3  6  0 73
```

I am going to add columns for the relative frequencies of final fortition in the different contexts, just as we did in the lectures:

```
gf$p_freq <- gf$p/(gf$p + gf$b)
gf$t_freq <- gf$t/(gf$t + gf$d)
gf$k_freq <- gf$k/(gf$k + gf$g)
```

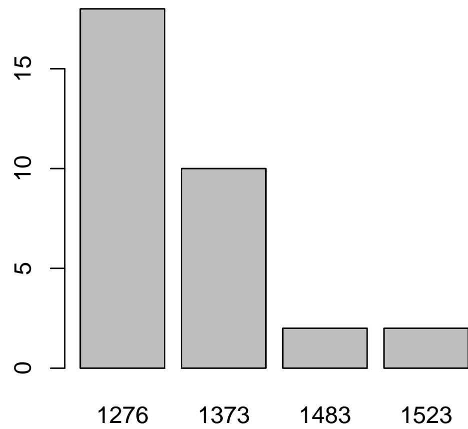
Here's how the data frame looks now:

```
gf

##   date  p  b  t  d  k  g    p_freq     t_freq    k_freq
## 1 1276 18  0 29  0 54 19 1.0000000 1.00000000 0.7397260
## 2 1373 10  8 24  5 17 59 0.5555556 0.82758621 0.2236842
## 3 1483  2 16  2 22  0 78 0.1111111 0.08333333 0.0000000
## 4 1523  2 14  3  6  0 73 0.1250000 0.33333333 0.0000000
```

2 Barplots

Barplots are created using the `barplot` command. You need to specify which column(s) of the data frame need to be plotted, plus the information that is to go under the barplot. For example, to plot the absolute frequencies of [p] (i.e. fortition of /b/) in the Glaser data against year as a barplot, use:

```
barplot(gf$p, names.arg=gf$date)
```

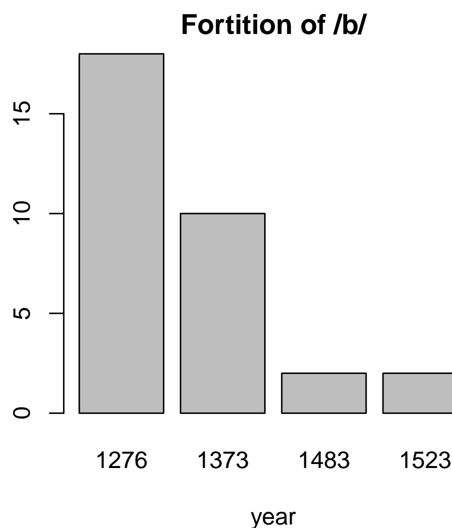


It is easy to add a title:

```
barplot(gf$p, names.arg=gf$date, main="Fortition of /b/")
```

Or a label for the x-axis:

```
barplot(gf$p, names.arg=gf$date, main="Fortition of /b/", xlab="year")
```



You can also rotate the barplot into a horizontal configuration:

```
barplot(gf$p, names.arg=gf$date, main="Fortition of /b/", ylab="year",
        horiz=TRUE)
```

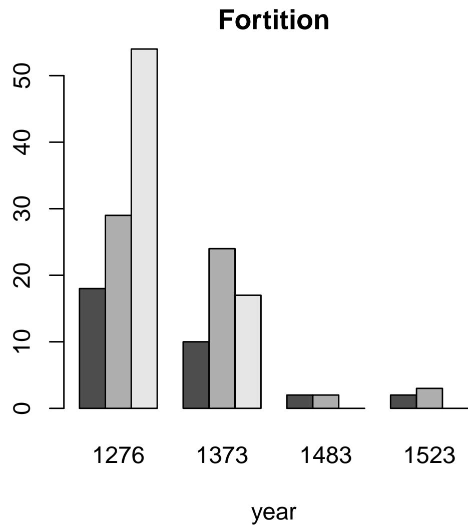
(Note that here I've changed the `xlab` argument into `ylab`, since the label now needs to go on the y-axis.)

What if we want to display fortition in all the phonemes in one and the same barplot? We will first need to `rbind` the three relevant columns together:

```
phonemes <- rbind(gf$p, gf$t, gf$k)
phonemes

##      [,1] [,2] [,3] [,4]
## [1,]   18   10    2    2
## [2,]   29   24    2    3
## [3,]   54   17    0    0
```

Then:

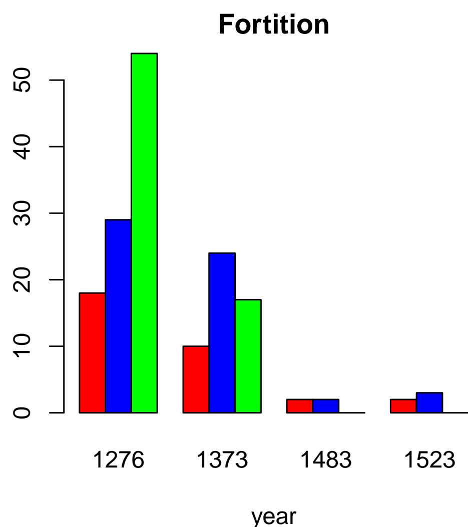
```
barplot(phonemes, names.arg=gf$date, main="Fortition", xlab="year",
        beside=TRUE)
```



Note the new argument, `beside=TRUE`. This is needed so that the three columns in each year settle side by side. (Try for yourself what happens if you omit the argument!)

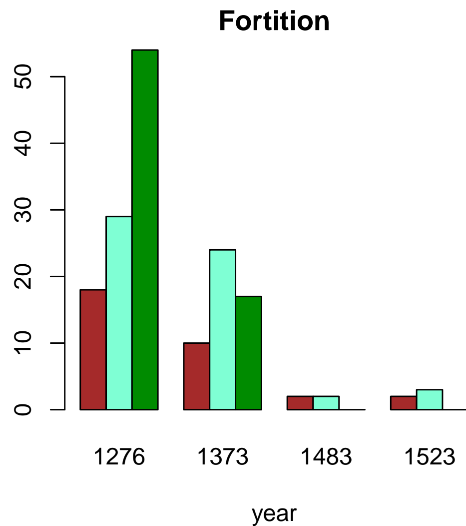
To apply some colours to make the plot look nicer (and easier to read!), we can specify a `col` argument:

```
barplot(phonemes, names.arg=gf$date, main="Fortition", xlab="year",
        beside=TRUE, col=c("red", "blue", "green"))
```



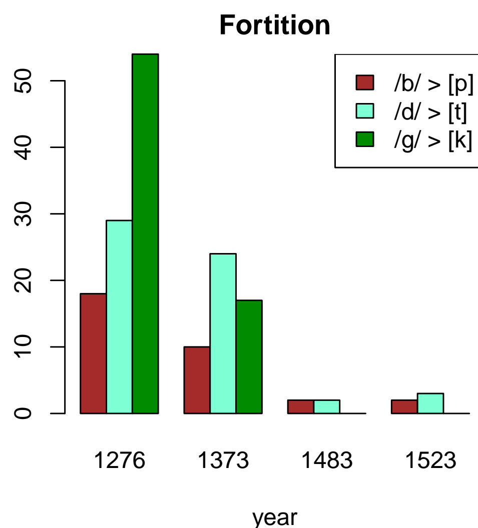
Note that what we specify as the colour argument is a vector (constructed with the `c` command) of strings, each string giving the name of a colour. The list of possible colours is almost endless (and it is also possible to define your own custom colours—but that is beyond this tutorial): see `http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf`. Here's another selection of colours for good measure:

```
barplot(phonemes, names.arg=gf$date, main="Fortition", xlab="year",
        beside=TRUE, col=c("brown", "aquamarine", "green4"))
```



Okay—but how will the viewer of our barplot know what each colour means? To specify that information, we need to add something known as a plot **legend**. This is added as a separate command after the `barplot` command:

```
barplot(phonemes, names.arg=gf$date, main="Fortition", xlab="year",
        beside=TRUE, col=c("brown", "aquamarine", "green4"))
legend("topright", fill=c("brown", "aquamarine", "green4"),
       legend=c("/b/ > [p]", "/d/ > [t]", "/g/ > [k]"))
```



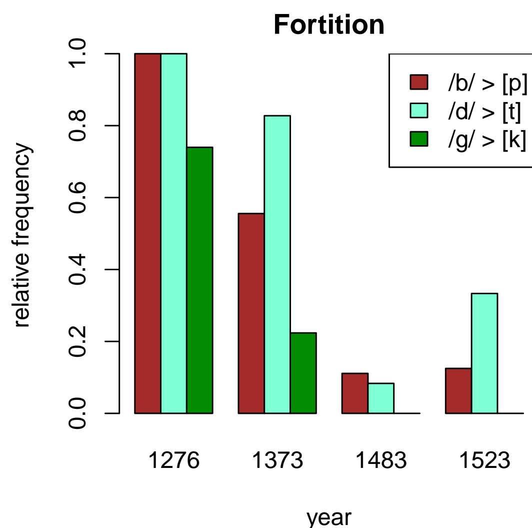
Let's unpack that a bit. The first argument to the `legend` function specifies the position of the legend in the plot. (Other options are "right", "bottomright", "topleft", and so on...) The second argument specifies with which colour the little legend boxes are to be filled. The final argument—confusingly named the same as the command itself—specifies the legend text.

Let's also try barplotting the relative frequencies. For this, we again need to `rbind` the relevant columns:

```
phonemes_freq <- rbind(gf$p_freq, gf$t_freq, gf$k_freq)
phonemes_freq

##          [,1]      [,2]       [,3]      [,4]
## [1,] 1.000000 0.5555556 0.11111111 0.1250000
## [2,] 1.000000 0.8275862 0.08333333 0.3333333
## [3,] 0.739726 0.2236842 0.00000000 0.0000000
```

And voilà:

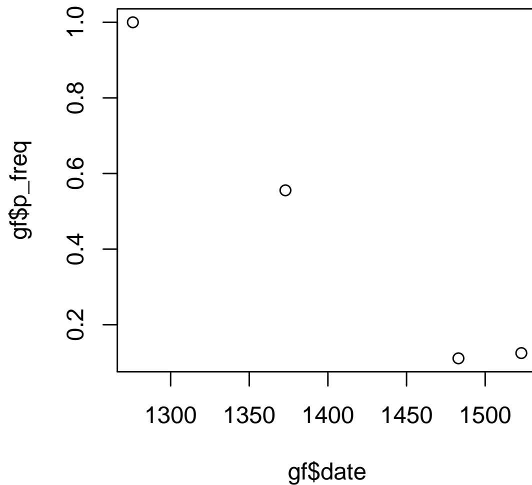
```
barplot(phonemes_freq, names.arg=gf$date, main="Fortition", xlab="year",
        beside=TRUE, col=c("brown", "aquamarine", "green4"),
        ylab="relative frequency")
legend("topright", fill=c("brown", "aquamarine", "green4"),
       legend=c("/b/ > [p]", "/d/ > [t]", "/g/ > [k]"))
```



3 Scatterplots

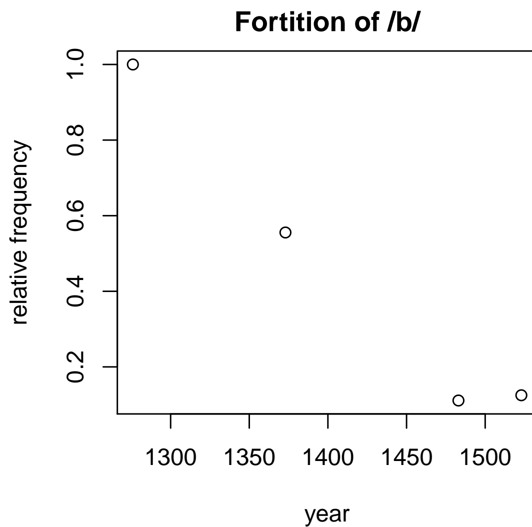
It actually makes more sense to plot Glaser's data in a different way, as a **scatterplot**. This is a simple plot of points in a two-dimensional plane, the two dimensions being given (in this case) by frequency and time (year). In R, scatterplots are given by the `plot` command:

```
plot(gf$date, gf$p_freq)
```



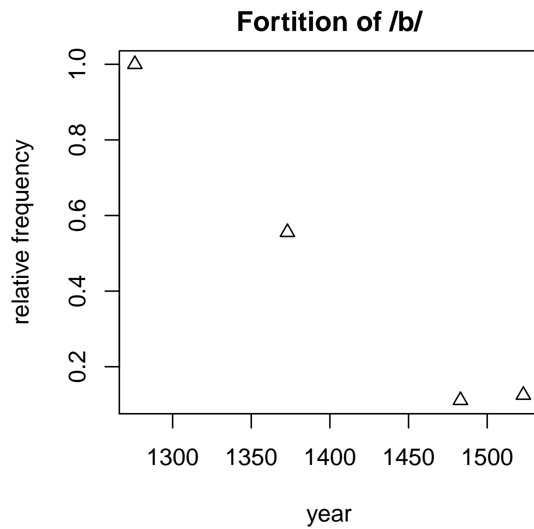
Note that the x-axis is specified as the first argument and the y-axis as the second argument. The result is four points in a plane (since our data frame has four rows). We can again use optional plotting arguments to prettify the plot a bit:

```
plot(gf$date, gf$p_freq, xlab="year", ylab="relative frequency",
     main="Fortition of /b/")
```

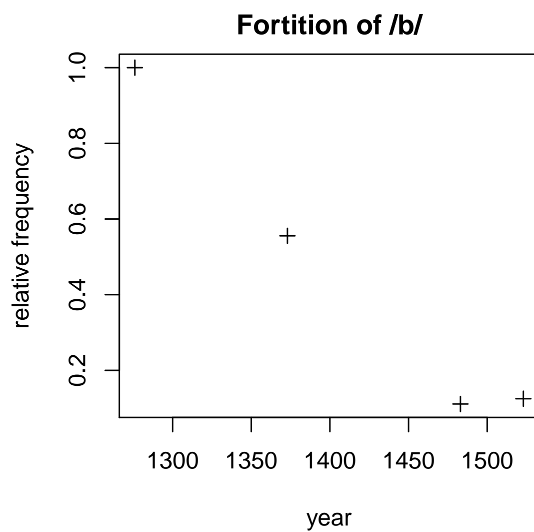


We can also change the representation of the points. This is done using the optional `pch` (**plotting character**) argument, which can take either an integer or a string as its value:

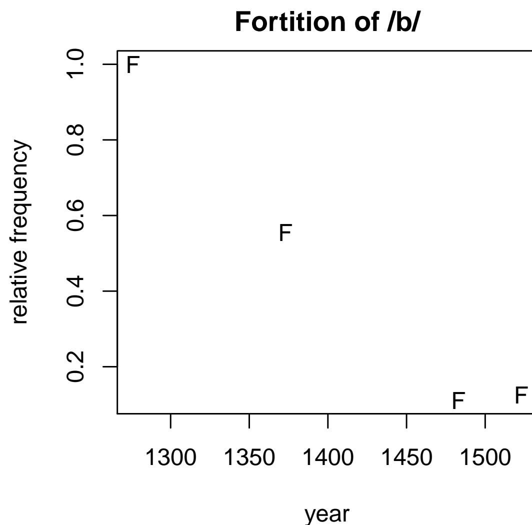
```
plot(gf$date, gf$p_freq, xlab="year", ylab="relative frequency",
     main="Fortition of /b/", pch=2)
```




```r
plot(gf$date, gf$p_freq, xlab="year", ylab="relative frequency",
     main="Fortition of /b/", pch=3)
```



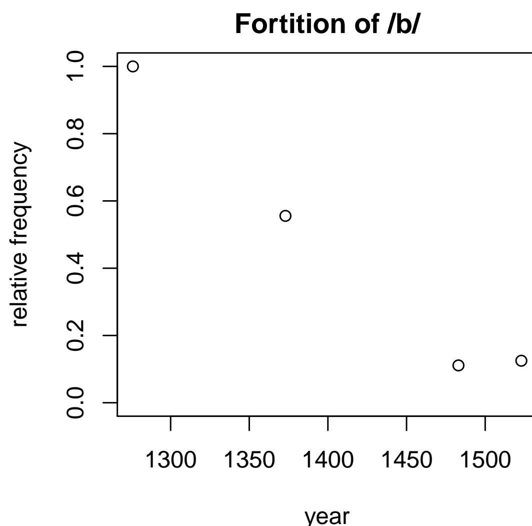

```r
plot(gf$date, gf$p_freq, xlab="year", ylab="relative frequency",
     main="Fortition of /b/", pch="F")
```



To see a list of the plotting characters that the different integers correspond to, visit http://www.sthda.com/english/wiki/r-plot-pch-symbols-the-different-point-shapes-available-in-r.

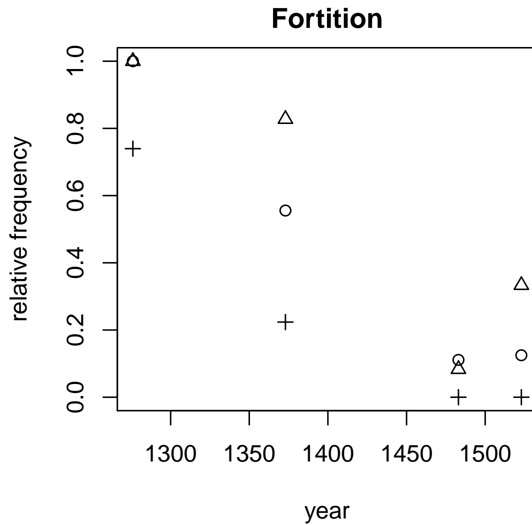
Another small aesthetic improvement we can make to the plot is to make the y-axis extend from 0 to 1 (the mathematically possible range for a relative frequency). Notice that in the previous plots, the y-axis begins from something like 0.1 rather than 0 (this is because the lowest relative frequency in the data is 0.125, and R automatically figures out the y-axis limits according to that). We can correct this with the optional `ylim` argument. It takes as a value a vector of two elements; the first element becomes the lower bound of the axis and the second one the upper bound:

```
plot(gf$date, gf$p_freq, xlab="year", ylab="relative frequency",
     main="Fortition of /b/", pch=1, ylim=c(0,1))
```



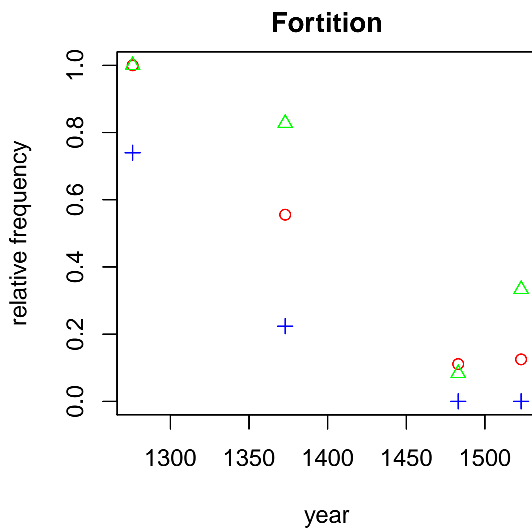
Adding the rest of the phonemes is a bit different from what we did with the barplots. To add more points to a scatterplot, we call the `points` command after the initial `plot` command. We'll do this twice to add the two remaining phonemes/contexts:

```
plot(gf$date, gf$p_freq, xlab="year", ylab="relative frequency",
     main="Fortition", pch=1, ylim=c(0,1))
points(gf$date, gf$t_freq, pch=2)
points(gf$date, gf$k_freq, pch=3)
```



The plot is still not very easy to read. Let's add some colors to make it more legible:

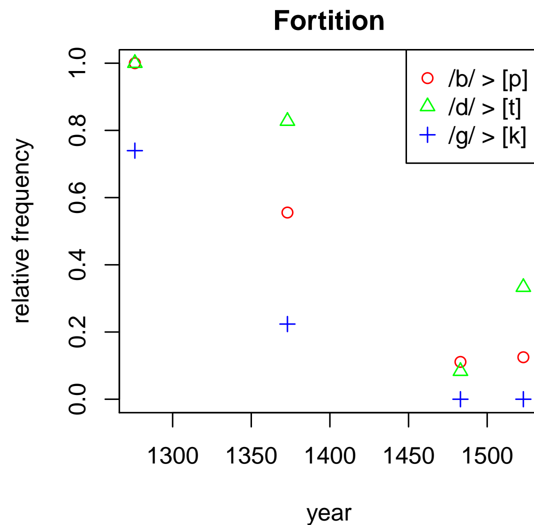
```
plot(gf$date, gf$p_freq, xlab="year", ylab="relative frequency",
     main="Fortition", pch=1, ylim=c(0,1), col="red")
points(gf$date, gf$t_freq, pch=2, col="green")
points(gf$date, gf$k_freq, pch=3, col="blue")
```



If we now add a legend, we have a pretty reasonable plot representing the loss of final fortition in the three phonemes.

```
plot(gf$date, gf$p_freq, xlab="year", ylab="relative frequency",
     main="Fortition", pch=1, ylim=c(0,1), col="red")
```

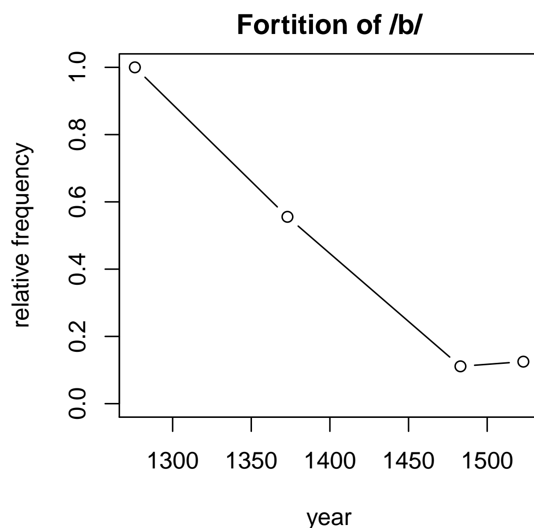
```
points(gf$date, gf$t_freq, pch=2, col="green")
points(gf$date, gf$k_freq, pch=3, col="blue")
legend("topright", pch=c(1,2,3), col=c("red", "green", "blue"),
       legend=c("/b/ > [p]", "/d/ > [t]", "/g/ > [k]"))
```



4 Lineplots

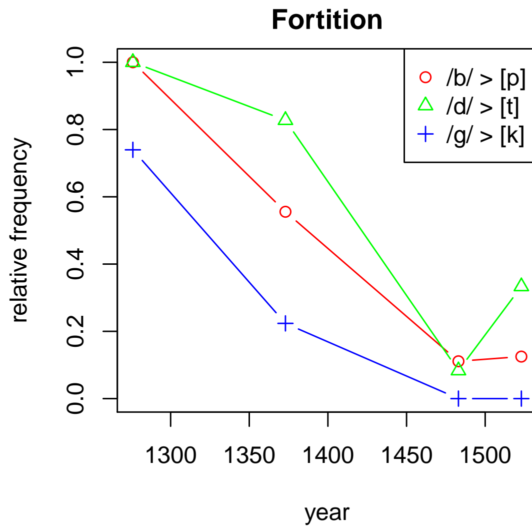
We can also add some lines to a scatterplot to "connect the dots". This is often useful for visualizing trends in data. The relevant argument is `type="b"` (b here stands for "both", meaning we want both points and lines):

```
plot(gf$date, gf$p_freq, xlab="year", ylab="relative frequency",
     main="Fortition of /b/", pch=1, ylim=c(0,1), type="b")
```



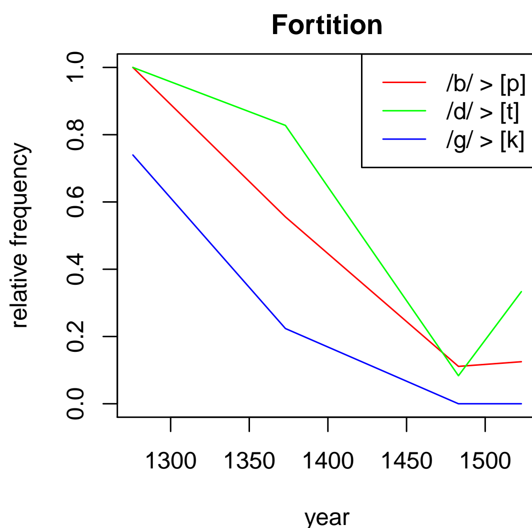
Here's all three phonemes:

```
plot(gf$date, gf$p_freq, xlab="year", ylab="relative frequency",
     main="Fortition", pch=1, ylim=c(0,1), col="red", type="b")
points(gf$date, gf$t_freq, pch=2, col="green", type="b")
points(gf$date, gf$k_freq, pch=3, col="blue", type="b")
legend("topright", pch=c(1,2,3), col=c("red", "green", "blue"),
       legend=c("/b/ > [p]", "/d/ > [t]", "/g/ > [k]"))
```



To get just the lines, and no points, specify `type="l"`:

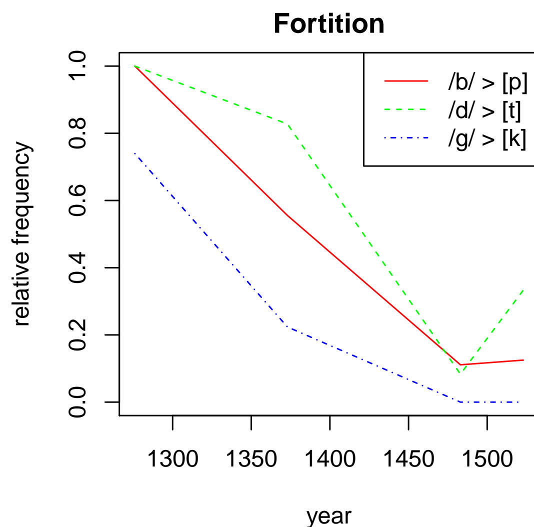
```
plot(gf$date, gf$p_freq, xlab="year", ylab="relative frequency",
     main="Fortition", ylim=c(0,1), col="red", type="l")
points(gf$date, gf$t_freq, col="green", type="l")
points(gf$date, gf$k_freq, col="blue", type="l")
legend("topright", lty=c(1,1,1), col=c("red", "green", "blue"),
       legend=c("/b/ > [p]", "/d/ > [t]", "/g/ > [k]"))
```



Notice that here I've changed the `legend`, too. Since we have no points now, there is no point (hehe) in specifying a `pch` (plotting character) argument to the `legend`

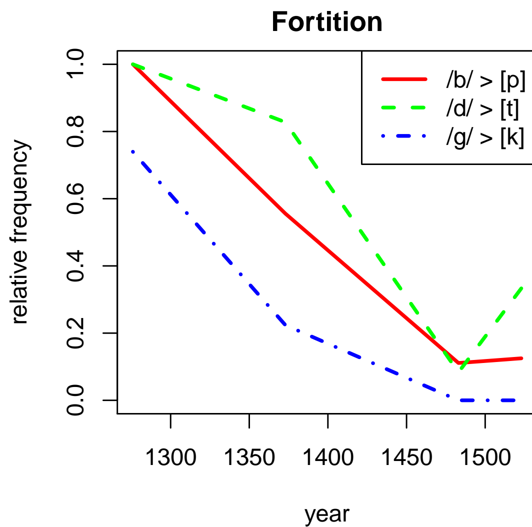
function. Instead, we have `lty`, specifying the **line type**. I've set it to the value of 1, which is the default and gives a solid line. But we can change this to a different integer to specify a different line type. See http://www.sthda.com/english/wiki/line-types-in-r-lty for the different options.

```
plot(gf$date, gf$p_freq, xlab="year", ylab="relative frequency",
     main="Fortition", ylim=c(0,1), col="red", type="l", lty=1)
points(gf$date, gf$t_freq, col="green", type="l", lty=2)
points(gf$date, gf$k_freq, col="blue", type="l", lty=4)
legend("topright", lty=c(1,2,4), col=c("red", "green", "blue"),
       legend=c("/b/ > [p]", "/d/ > [t]", "/g/ > [k]"))
```



Another useful argument for lineplots is `lwd`. This controls the **line width**. Try:

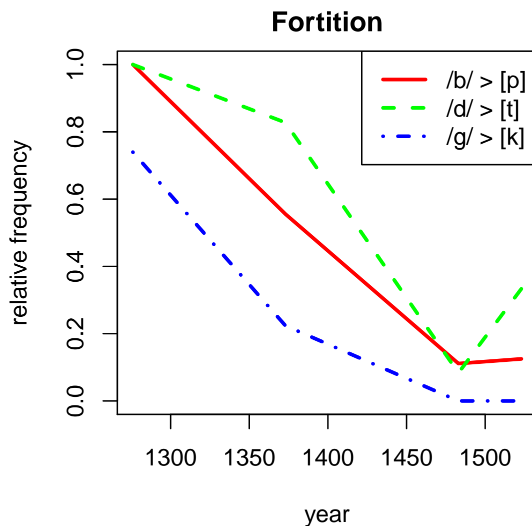
```
plot(gf$date, gf$p_freq, xlab="year", ylab="relative frequency",
     main="Fortition", ylim=c(0,1), col="red", type="l", lty=1, lwd=2.5)
points(gf$date, gf$t_freq, col="green", type="l", lty=2, lwd=2.5)
points(gf$date, gf$k_freq, col="blue", type="l", lty=4, lwd=2.5)
legend("topright", lty=c(1,2,4), col=c("red", "green", "blue"),
       legend=c("/b/ > [p]", "/d/ > [t]", "/g/ > [k]"), lwd=2.5)
```



5 A note on formulae

Before moving on, I want to mention something about formulae. Remember that a formula in R is an expression involving the tilde (~)? Formulae are used when we need to specify a mathematical function for curve-fitting purposes, for example, as we have seen with `nls`. But they can also be used in plotting. Using formulae, the code for the previous plot becomes:

```
plot(p_freq~date, gf, xlab="year", ylab="relative frequency",
     main="Fortition", ylim=c(0,1), col="red", type="l", lty=1, lwd=2.5)
points(t_freq~date, gf, col="green", type="l", lty=2, lwd=2.5)
points(k_freq~date, gf, col="blue", type="l", lty=4, lwd=2.5)
legend("topright", lty=c(1,2,4), col=c("red", "green", "blue"),
       legend=c("/b/ > [p]", "/d/ > [t]", "/g/ > [k]"), lwd=2.5)
```



The end result is exactly the same. But note that when using formulae, the y-axis comes before the x-axis in the code (and third comes the data frame itself).

Whether you want to use formulae in plotting is entirely up to your own tastes.

6 Plotting a logistic function

We've been working a lot with logistic functions, so it is important to know how to plot one. (Actually, even though I here only talk about logistic functions, the same methods can be applied to plot any function you like.)

Recall that the logistic function is the mathematical function

$$f(x) = \frac{1}{1 + \exp(s \times (k - x))}.$$

You can also write y instead of $f(x)$ to make it clearer that this function is representing a curve in the two-dimensional xy-plane:

$$y = \frac{1}{1 + \exp(s \times (k - x))}.$$

So, for any value of x,

$$\frac{1}{1 + \exp(s \times (k - x))}$$

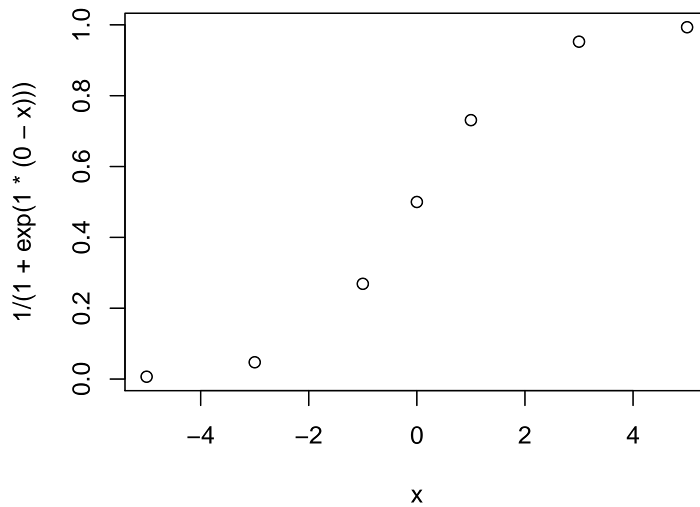
is the corresponding value of y. You can of course use a pocket calculator and draw the function by hand using this method: pick a few x values, calculate the corresponding y values, draw the points, and connect the points. But that is very, very time consuming. Better let R do it for us.

The basic idea is still the very same. We pick a number of values of x, let R calculate the corresponding y values, and plot the points. Let's now take a concrete example. Suppose we want to plot the logistic function with $s = 1$ and $k = 0$. We then know that the midpoint of the function/change is at $x = 0$ (because of the value of k), and we know that the function is increasing rather than decreasing (because of the value of s). Let's first make a vector of x values:

```
x <- c(-5, -3, -1, 0, 1, 3, 5)
```

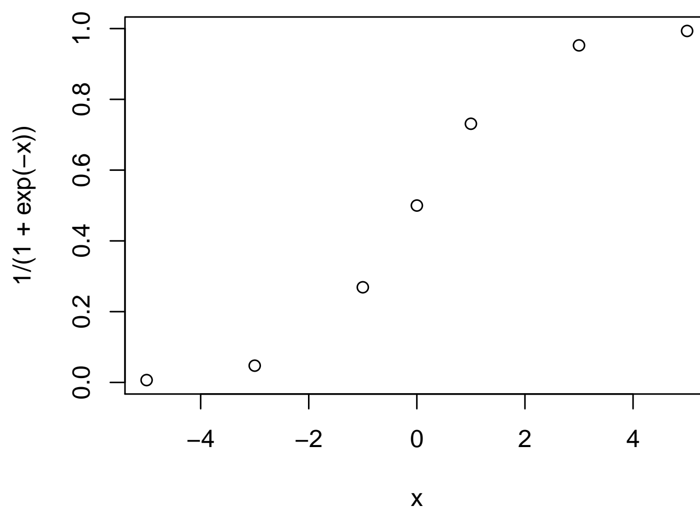
Plotting the function is now as simple as:

```
plot(x, 1/(1 + exp(1*(0-x))))
```



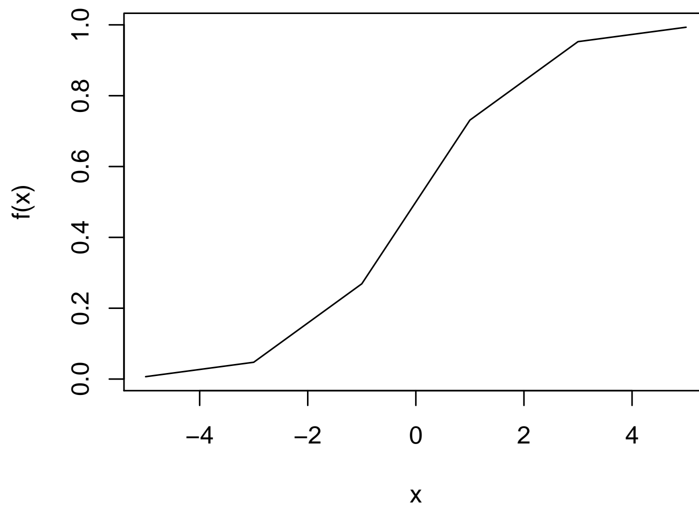
In fact, since $1 \times (0 - x) = -x$, in this case we can simply write:

```
plot(x, 1/(1 + exp(-x)))
```



Now let's make the plot a bit nicer. I'm going to modify the y-axis label and turn the plot into a line plot:

```
plot(x, 1/(1 + exp(-x)), ylab="f(x)", type="l")
```



Okay that's fine, but the function looks a bit jagged. Why is that? Well, because we have taken only 7 values of x. So we have 7 points in the plane representing the function. When we connect these with lines, we obviously get only a very approximate representation of the function. To improve our plot, we'll have to give R more values of x.

Here's the easiest way of doing that. Remember the `seq` command from our first R tutorial? We can use this command to generate sequences of numbers from some given starting point to some given end point, for a given length. Like so:

```
x <- seq(from=-5, to=5, length.out=100)
```

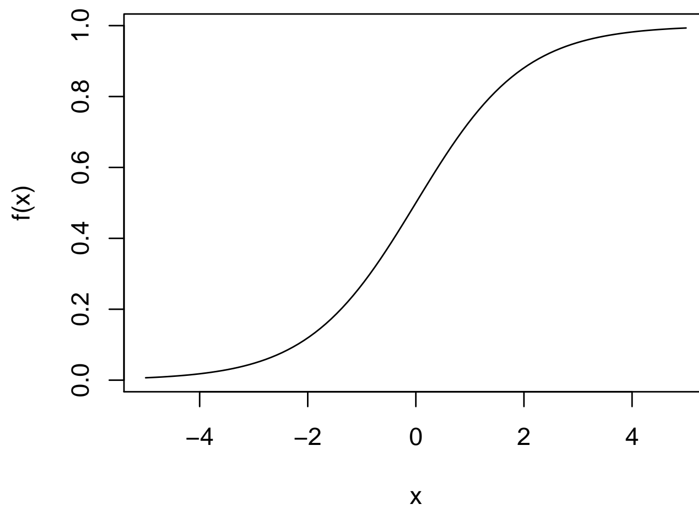
The variable x should now contain 100 equally-spaced numbers between -5 and 5, and you can check this easily by just typing x:

```
x

##   [1] -5.00000000 -4.89898990 -4.79797980 -4.69696970 -4.59595960
##   [6] -4.49494949 -4.39393939 -4.29292929 -4.19191919 -4.09090909
##  [11] -3.98989899 -3.88888889 -3.78787879 -3.68686869 -3.58585859
##  [16] -3.48484848 -3.38383838 -3.28282828 -3.18181818 -3.08080808
##  [21] -2.97979798 -2.87878788 -2.77777778 -2.67676768 -2.57575758
##  [26] -2.47474747 -2.37373737 -2.27272727 -2.17171717 -2.07070707
##  [31] -1.96969697 -1.86868687 -1.76767677 -1.66666667 -1.56565657
##  [36] -1.46464646 -1.36363636 -1.26262626 -1.16161616 -1.06060606
##  [41] -0.95959596 -0.85858586 -0.75757576 -0.65656566 -0.55555556
##  [46] -0.45454545 -0.35353535 -0.25252525 -0.15151515 -0.05050505
##  [51]  0.05050505  0.15151515  0.25252525  0.35353535  0.45454545
##  [56]  0.55555556  0.65656566  0.75757576  0.85858586  0.95959596
##  [61]  1.06060606  1.16161616  1.26262626  1.36363636  1.46464646
##  [66]  1.56565657  1.66666667  1.76767677  1.86868687  1.96969697
##  [71]  2.07070707  2.17171717  2.27272727  2.37373737  2.47474747
##  [76]  2.57575758  2.67676768  2.77777778  2.87878788  2.97979798
##  [81]  3.08080808  3.18181818  3.28282828  3.38383838  3.48484848
```

```
## [86] 3.58585859 3.68686869 3.78787879 3.88888889 3.98989899
## [91] 4.09090909 4.19191919 4.29292929 4.39393939 4.49494949
## [96] 4.59595960 4.69696970 4.79797980 4.89898990 5.00000000
```

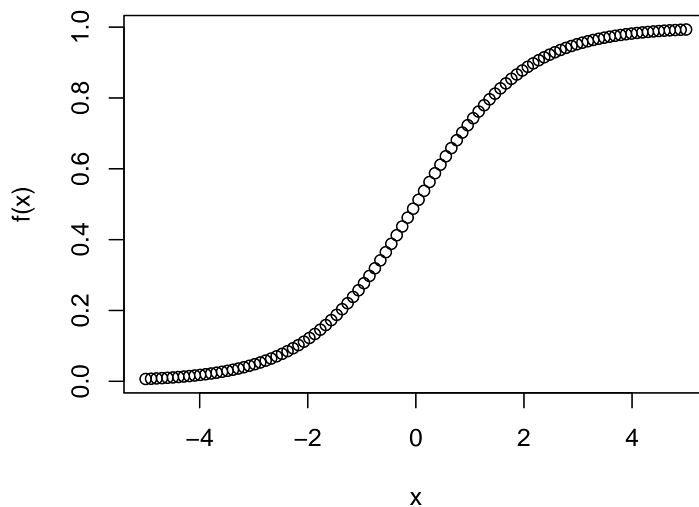
If we now plot the function, we will get a much smoother curve:

```
plot(x, 1/(1 + exp(-x)), ylab="f(x)", type="l")
```



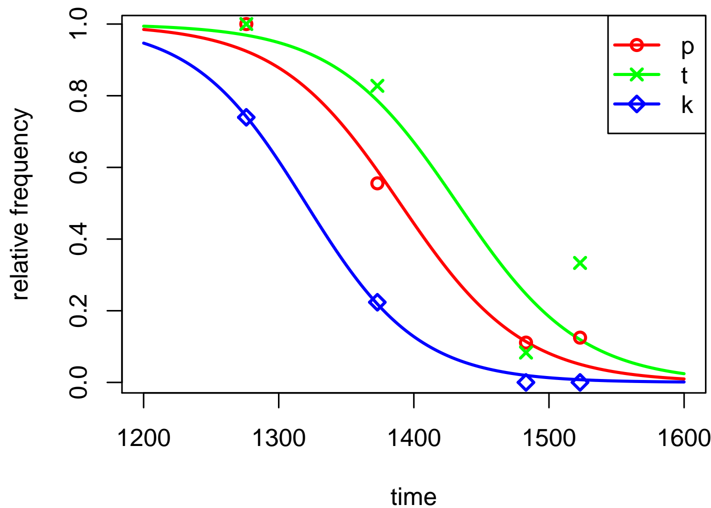
The curve is still constructed by R as a series of line segments between points in the *xy*-plane. But now it is smoother as there are more of those line segments and each of them is very short (⇒ better approximation of the function). Of course, you can plot points too:

```
plot(x, 1/(1 + exp(-x)), ylab="f(x)", type="p")
```



7 A complete example

In the lectures, I showed you this plot on the loss of final fortition in Early New High German, with both Glaser's data points and logistic fits to each context:



With the above information, you should be able to reproduce it now. Let's walk through the steps.

1. First, we will define some variables which we will need to use more than once in constructing the plot. These are the colours, the line width for the logistic curves, and the plotting characters used to plot the data points:

```
colours <- c("red", "green", "blue")
line_width <- 2
characters <- c(1,4,5)
```

2. To plot the logistic curves, we will also need a sequence of time values:

```
t <- seq(from=1200, to=1600, length.out=1000)
```

3. The Glaser data is already in the data frame `gf` (see above), but so far we don't have the logistic fits. So let's carry out those next, just as we have done in the lectures:

```
p_model <- nls(p_freq~1/(1 + exp(s*(k-date))), gf,
               start=list(s=-0.01, k=1400))
t_model <- nls(t_freq~1/(1 + exp(s*(k-date))), gf,
               start=list(s=-0.01, k=1400))
k_model <- nls(k_freq~1/(1 + exp(s*(k-date))), gf,
               start=list(s=-0.01, k=1400))
```

4. We can use `coef` to see the best-fitting parameter values that the `nls` (Gauss–Newton) algorithm found:

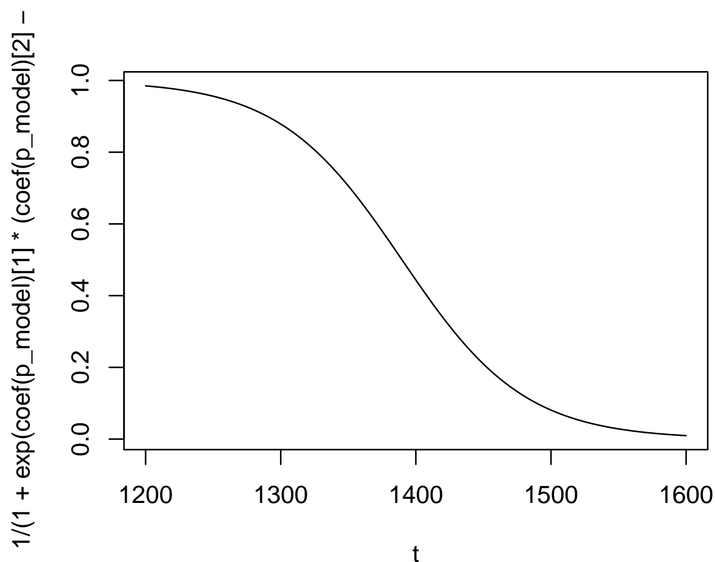
```r
coef(p_model)

##           s           k
##   -0.02204335 1389.71780632

coef(t_model)

##           s           k
##   -0.02223679 1432.22617617

coef(k_model)

##           s           k
##   -0.02406459 1319.97726598
```

We can also extract the s and k values individually. This will be of use in the next step when we plot the logistic curves. For example:

```r
coef(p_model)[1]

##          s
## -0.02204335

coef(p_model)[2]

##        k
## 1389.718
```

5. Let's start by plotting one of the logistic curves:

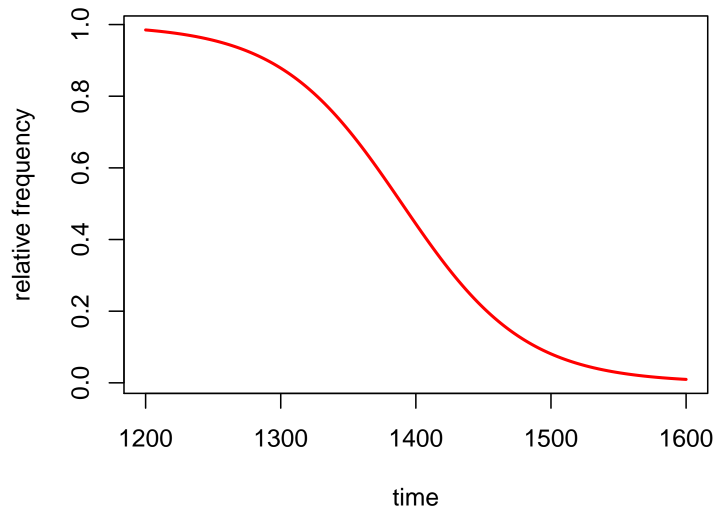
```r
plot(t, 1/(1 + exp(coef(p_model)[1]*(coef(p_model)[2] - t))), type="l")
```



Notice that the formula here is that of the logistic function, with the s parameter given by `coef(p_model)[1]` and the k parameter given by `coef(p_model)[2]`.

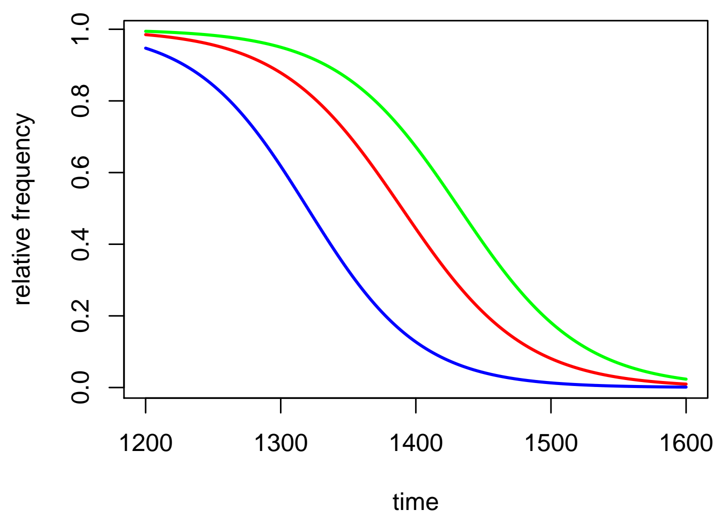
6. Add some colour and axis labels:

```
plot(t, 1/(1 + exp(coef(p_model)[1]*(coef(p_model)[2] - t))), type="l",
    xlab="time", ylab="relative frequency", lwd=line_width, col=colours[1])
```



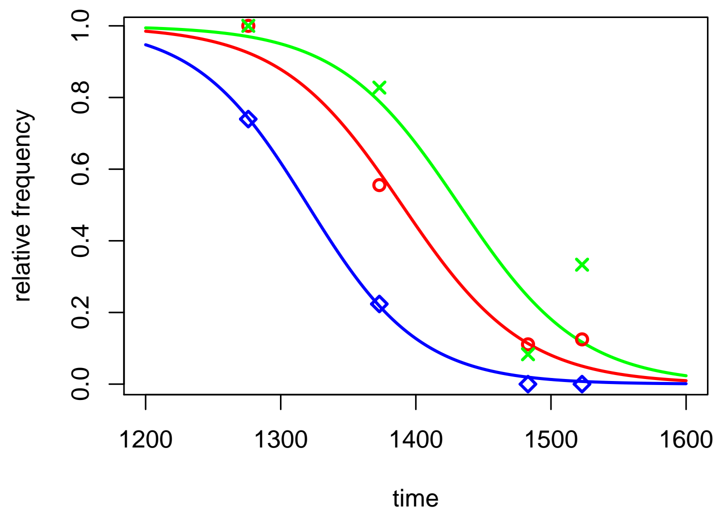
7. Now let's add the other two curves:

```
points(t, 1/(1 + exp(coef(t_model)[1]*(coef(t_model)[2] - t))), type="l",
       lwd=line_width, col=colours[2])
points(t, 1/(1 + exp(coef(k_model)[1]*(coef(k_model)[2] - t))), type="l",
       lwd=line_width, col=colours[3])
```



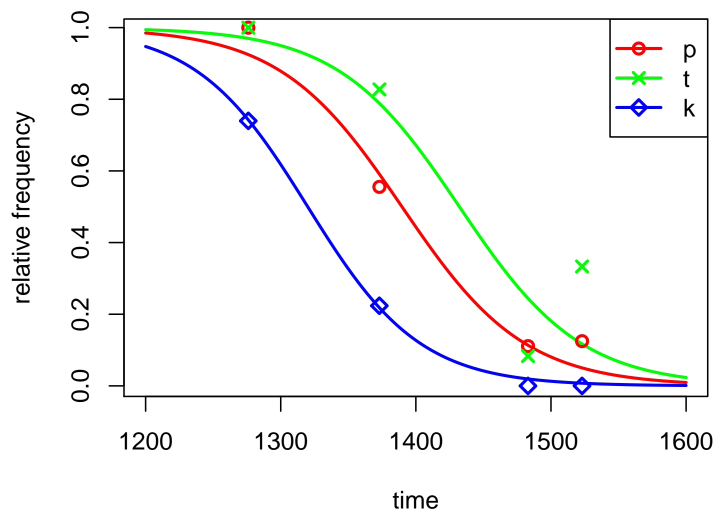
8. Next, we need to add the data points:

```
points(p_freq~date, gf, pch=characters[1], lwd=line_width, col=colours[1])
points(t_freq~date, gf, pch=characters[2], lwd=line_width, col=colours[2])
points(k_freq~date, gf, pch=characters[3], lwd=line_width, col=colours[3])
```



9. The only thing that's missing is the legend:

```
legend("topright", lwd=line_width, col=colours, pch=characters,
       legend=c("p", "t", "k"))
```



10. All in all, we used the following code:

```
colours <- c("red", "green", "blue")
line_width <- 2
characters <- c(1,4,5)
t <- seq(from=1200, to=1600, length.out=1000)
p_model <- nls(p_freq~1/(1 + exp(s*(k-date))), gf,
               start=list(s=-0.01, k=1400))
t_model <- nls(t_freq~1/(1 + exp(s*(k-date))), gf,
               start=list(s=-0.01, k=1400))
k_model <- nls(k_freq~1/(1 + exp(s*(k-date))), gf,
```

```
                start=list(s=-0.01, k=1400))
plot(t, 1/(1 + exp(coef(p_model)[1]*(coef(p_model)[2] - t))), type="l",
    xlab="time", ylab="relative frequency", lwd=line_width, col=colours[1])
points(t, 1/(1 + exp(coef(t_model)[1]*(coef(t_model)[2] - t))), type="l",
       lwd=line_width, col=colours[2])
points(t, 1/(1 + exp(coef(k_model)[1]*(coef(k_model)[2] - t))), type="l",
       lwd=line_width, col=colours[3])
points(p_freq~date, gf, pch=characters[1], lwd=line_width, col=colours[1])
points(t_freq~date, gf, pch=characters[2], lwd=line_width, col=colours[2])
points(k_freq~date, gf, pch=characters[3], lwd=line_width, col=colours[3])
legend("topright", lwd=line_width, col=colours, pch=characters,
       legend=c("p", "t", "k"))
```

Since this is a *lot* to type, it makes sense to open a text editor, write the above piece of code as a function,[2] and save the file, and then `source` the code in R. This carries two benefits: (1) it will be much easier to print the plot in R, and (2) it will be much easier to modify the details of the plot if you need to do so (by simply editing the text file that contains the code and rerunning `source` on it).

8 Exporting graphics

Now we know how to produce a variety of pretty plots. How to get them out of R, so that you can include them in your paper or portfolio, for example?

There are many ways of doing this, depending mostly on what type of file you want to have. The basic logic, however, is always the same:

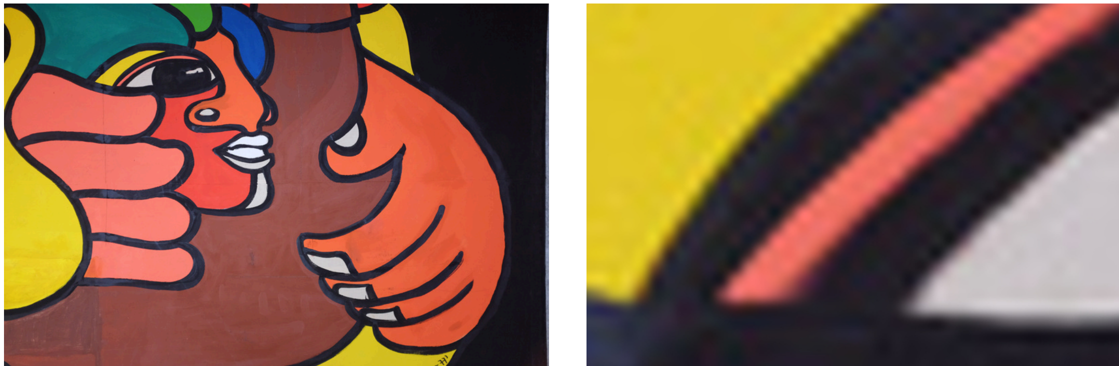
1. First, we need to tell R which filetype we want. In technical R jargon, this means "opening a **graphics device**".
2. Then we plot.
3. Finally, we need to tell R that we aren't going to plot anymore. Technically, we need to close the graphics device we opened in step 1. Note that the plot is written to file only at this stage. So if you forget to close your graphics device, you will end up with an empty image file!

Computer graphics can be roughly divided into two categories: bitmaps and vector graphics. A bitmap is, essentially, a matrix of pixels (i.e. picture elements, those little[3] square things on your screen). A technical consequence of this is that a bitmap cannot be resized losslessly. If you enlarge a bitmap, what you get is larger pixels and consequently poor image quality. If you shrink a bitmap, the information from more than one pixel

[2] Note that there are two distinct senses of 'function' here. A mathematical function is, basically, a rule that associates a value $y = f(x)$ with every value of x. A function in a programming language such as in R is a part of programme (also known as a "subroutine") that does something. A function in this sense can have input in the form of **arguments** (but it need not), and it may **return** something, for example a number or a string (but it need not), or it could for example produce a plot. We will (have) talk(ed) more about R functions in the lectures.

[3] If you are old enough to have played video games in the 1980s or early 1990s, you will know that pixels used to be a bit bigger back in those days...

will be squeezed into a single pixel, and again you get poorer image quality. Here's an example of a bitmap, together with a detail 20× magnified:



With some exceptions (e.g. plotting maps—but that will not be a concern in this course), the sorts of plots we produce with R are better suited to be saved as vector images. A vector image is not an array of pixels, but rather a list of information about what the image should look like. The practical consequence is that a vector image can be resized at will without loss of information.

R supports both bitmap and vector graphics, so you can export your plot in either format. Starting with bitmaps, BMP, JPEG and PNG are some of the most common forms of these. R has a command for each. For example, to create a BMP file with name `myimage.bmp` and of size 600 × 400 (600 pixels wide and 400 pixels high), type:

```
bmp("myimage.bmp", width=600, height=400)
```

Having opened the graphics device, you can now plot your stuff. For example:

```
barplot(gf$p, names.arg=gf$date, main="Fortition of /b/", ylab="year",
        horiz=TRUE)
```

Finally, you must close the graphics device as discussed above. For this, R has the special command `dev.off` which you can call without any arguments:

```
dev.off()
```

You should now have a barplot in the file `myimage.bmp`.

The procedure for creating a JPEG or a PNG file is the same, except you need to call either the `jpeg` or the `png` function instead of `bmp`.

If you are writing an image file of large dimensions, you may find that you need to increase the font size of your plot so that it is legible. The default font size (that is, if you don't specify any value for the optional argument) is 12 points. You can increase (or decrease) this:

```
bmp(filename="myimage.bmp", width=2000, height=1500, pointsize=48)
```

The same holds for `jpeg` and `png`.

Vector graphics formats include PDF, PS and SVG. To produce a PDF file, for example, you would type:

```
pdf("myimage.pdf", width=7, height=5)
barplot(gf$p, names.arg=gf$date, main="Fortition of /b/", ylab="year",
        horiz=TRUE)
dev.off()
```

With PDF, the `width` and `height` arguments are no longer specified in pixels, but rather in inches. Values between 5 and 10 are usually fine. Since a vector image can be rescaled at will, these arguments really only serve to specify the aspect ratio of the image (i.e. width to height ratio).

To get an SVG, use the `svg` command instead of `pdf`. To get a PS (PostScript) image, use `postscript`. But there is rarely any need to use these more exotic file formats. Usually a PDF (vector) or a PNG (bitmap) file will do.

9 A note on graphics packages

The above has been based on so-called **R base graphics**. This means that we've done all the plotting using only commands which are part of the basic R installation.

In addition to this, dedicated R packages exist for producing more complicated or prettier graphics. The two most famous are `lattice` and `ggplot2`. You can google these to find out more. (It is unlikely you will need either of them in this course, but it's always good to know that they exist.)

Some other packages also introduce new plotting capabilities. We have already seen the `igraph` and `phangorn` packages, both of which allow you to plot trees.

10 Help?

As always, your first port of call should be R's own documentation. For example, if you have troubles with the `plot` command, access its help page as follows:

```
?plot
```

And similarly for all other commands.

If that doesn't help, try googling your problem.

Finally, you are always welcome to email me, and I'll be happy to help.

References

Fruehwald, J., Gress-Wright, J. & Wallenberg, J. 2013. Phonological rule change: the constant rate effect. In S. Kan, C. Moore-Cantwell & R. Staubs (Eds.), *NELS 40: Proceedings of the 40th Annual Meeting of the North East Linguistic Society* (pp. 219–230). GLSA Publications. http://www.research.ed.ac.uk/portal/files/14416788/Fruewald_Gress_Wright_Wallenberg_Phonological_Rule_Change.pdf.

Glaser, E. 1985. *Graphische Studien zum Schreibsprachwandel vom 13. bis 16. Jahrhundert.* Heidelberg: Carl Winter Universitätsverlag.