Nonstatistical Things that Linguists Can Do with R: A Review with a Case Study on Chinese Handwriting[1]

James Myers
National Chung Cheng University

R is well known among linguists as a great tool for statistical analysis. However, less discussed in the many excellent R textbooks and tutorials out there is that its power goes far beyond statistics. Among many other things, it can also do text processing, compile and organize experimental data, process and generate images, and interface with other software for speech analysis and artificial intelligence modeling. It is thus not difficult to extend one's R skills to carry out all sorts of new projects, even before (or without) learning an additional computer language like Python. I demonstrate this with a study in progress that explores patterns in handwritten Chinese.

1. Our best friend R

By now most linguists who are not already using R, the free software tool for statistical analysis, have that colleague or teacher or student who is constantly trying to pressure them into trying it out. This makes sense; linguists work with a lot of numbers, in phonetics, sociolinguistics, corpus linguistics, psycholinguistics, and implicitly even in theoretical linguistics (e.g., more people like such-and-such a sentence than that other one). Yes, SPSS (https://www.ibm.com/products/spss-statistics), which was the first statistical program designed for non-statisticians (Wellman, 1998), is still widely used, and even Excel (https://office.microsoft.com/excel) has some built-in statistical tools, but R has a big advantage over both of these: it is free.

[1] This paper combines both of my presentations at NACCL-34 (a regular talk and a tutorial at the preconference workshop on data and methods). Support came from grant 109-2410-H-194-096-MY3 (Taiwan, Ministry of Science and Technology, as it was then known) and much help from my lab assistants and my presentation attendees.

However, it is another advantage of R that is the focus of this paper: it is also a full-fledged computer language. Of course there are many other computer languages useful in linguistics, in particular Python (Van Rossum & Drake, 2009; https://www.python.org/), and in fact, if you are familiar with Python but want to do some statistics, there are Python tools that run R for you (rpy2; https://rpy2.github.io/). Other computer languages are also growing in popularity among data scientists, in particular Julia (Bezanson et al., 2017; https://julialang.org/), which can do a lot of what R can do, but much faster. Of course, I will not tell linguists not to learn as many languages as they want, but still, my sense is that far more linguists already use R than any other computer language. This suggests that it might be useful to review the ways in which linguists who already know (or are being pressured into learning) R for statistics can keep using R to go beyond statistics.

The rest of this introductory section will review the basics: how to get R and learn more about it, how R works, and how it can be extended. In section 2 we will take R beyond statistics in a wide variety of ways. Section 3 focuses on a case study related to Myers (2022a), explaining how I used R to compile, visualize, and analyze data from a pilot experiment on handwriting Chinese characters. Section 4 sums things up.

1.1 R resources

R (R Core Team, 2023) was invented by people with R-initial names (Ross Ihaka and Robert Gentleman) as a free and open-source imitation of S, a statistical programming language developed by Bell Labs (Ihaka, 2011); note that the letter R also happens to come just before S in the alphabet. To download R, visit https://www.r-project.org/, look for the "Download" section, click "CRAN" (for "Comprehensive R Archive Network"), choose a local source (if you are patriotic) or else the first link (sponsored by RStudio, about which more shortly). Then follow the instructions for your

computer's operating system; by default, menus and certain R messages (but not R itself) will be given in your system's language.

Hopefully your installation went well, but if not, well, one advantage of using R is that there are so many people using it that the odds are excellent that you will find somebody describing exactly the same problem you have, and somebody else explaining how to solve it. There are literally millions of sites, in every conceivable language, explaining how to do this and that in R, from the most basic to the most advanced.

There are also hundreds of textbooks on R, including a growing number specifically aimed at linguists, including Baayen (2008), Johnson (2008), Gries (2013), Levshina (2015), and Winter (2019). The R book market has even made room for linguistic subspecialties, including second-language research (Larson-Hall, 2015) and corpus linguistics (Gries, 2017). Other worthwhile and quite readable R books include Good (2013) (on resampling methods, which estimate probabilities when standard statistical tests are lacking) and Kruschke (2015) and McElreath (2016) (on Bayesian statistics, an approach that overcomes the inferential limitations of the so-called frequentist statistics that the rest of the above books focus on). Free online R textbooks for linguists include my own basic Myers (2022b) and the more advanced and authoritative Sonderegger (2022) and Vasishth, Schad, Bürki, and Kliegl (2022).

Being a language rather than a point-and-click tool, R makes it easy to explore data before finalizing the statistical analysis. This is helpful because, often, it takes a bit of creative wandering before it is clear what hypotheses the statistics should be testing anyway. However, free exploration affects the probabilities of what you end up finding, so it should be clearly separated from

confirmatory analyses that test predetermined hypotheses (Nicenboim et al., 2018). R gives researchers a lot of power, but as we all know, with great power comes great responsibility.

By default, downloading R will give you not just the language itself but also a basic GUI (graphical user interface) with a menu bar at the top, with options for changing your working directory, installing and updating extensions, and even changing the size and color of the GUI and its text. There is also a menu option for viewing and editing your data in a somewhat Excel-like way. The heart of R, though, is not the menus but the text window for computer commands ("code"), which you can type in directly, copy/paste from a text file, or load in; R will then do stuff, including perhaps popping out another window for the graphics.

If the default R GUI is not enough GUI for you, you can download RStudio (RStudio Team, 2023; https://posit.co/ – the company is now called Posit). You can pay them for a fancier version, but the free version already has plenty of extra bells and whistles that make it a full IDE (integrated development environment). RStudio divides the screen into four windows: one for the R code that is currently being run (like R default GUI), one for the full R code as you edit it, one for the software objects being created by your code, and one for everything else (file directory, graphics, and help). The main menu and window-specific menus include tools for formatting and debugging the code, among many other things. I personally prefer the simpler default R GUI (four windows are three too many for me), but do what you like.

1.2 How R works

As a language for doing statistics, the mathematics of variable data values, R assumes that the user wants to work with a whole bunch of numbers all at the same time, instead of one by one as in most computer languages. For example, the mean (the most common meaning of "average") of a

sample is calculated by summing up the sample values and then dividing by the sample size, as in the formula in (1), where Σ is the sum operator across all *n* values of the sample *x*.

(1) $\frac{\sum_{i=1}^{n} x_i}{n}$

In R this formula can be written out as in (2), with **x** representing an ordered sequence of numbers (technically known as a vector), **sum()** being the summing function and **length()** giving the vector length. If you type this into R, you will get the results in (3), where 3 is the answer, since that is the mean of the ordered sequence (1,2,3,4,5); [1] indicates that this value is in the first position of a vector of length one (irrelevant here, but R is just expressing its love of vectors).

(2) Code:
```
x = 1:5 # That is, 1, 2, 3, 4, 5
n = length(x)
sum(x)/n # Pretty much the same as in formula (1)
```

(3) Result:
```
[1] 3
```

Note also four other things in the code in (2). First, you can add comments for yourself and other readers by typing # and then anything you want (these can also be in Chinese or any other language). Second, functions always have the syntax **function(arguments, ...)**, which is why it is helpful to use an IDE that can keep track of the parentheses when you put functions inside other functions. Third, a command like **x = y** means that you take the value of **y** and put it into the variable **x**, not that the two values are equal. Because this syntax may be confusing, R also lets you write this as **x <- y**, with **<-** representing a leftward pointing arrow, and in fact some R experts will look down on you if you do not write it this way (even though **x = y** uses fewer keystrokes than **x**

<- y). R itself is more charitable, since it already uses distinct syntax to express equivalence, as in **x == y** (the claim that **x** and **y** are identical); see (4) for an example.

(4) Code: x == 3 # Assuming that we have already run the above code
Result: [1] FALSE FALSE TRUE FALSE FALSE

Of course R has a built-in function for computing the mean, namely **mean()**, though you have to remember that this function only operates on vectors, not on individual values; see the code and results in (5a,b). By contrast, the function **sum()** can handle either vectors or individual values, as in (5c); this is the kind of hard-to-remember detail that keeps R websites in business.

(5) a. Code: mean(c(2,0,2,3)) # c() creates a vector
Result: [1] 1.75
b. Code: mean(2,0,2,3) # Ignores all but the first value
Result: [1] 2
c. Code: sum(c(2,0,2,3)) == sum(2,0,2,3)
Result: [1] TRUE

Before searching the internet for help, though, you might first want to search R's own built-in help, most simply by typing **?** before the function name. For example, typing **?sum** and **?mean** will bring up help pages that show that they assume different types of arguments, though I have to say that these particular pages do not make this very obvious on first glance, and indeed, I find that R's built-in help pages are not always as helpful as I wish they were.

Like all computer languages, R has a function for creating functions: **function()**. As an illustration, we can use it to create our own mean function in (6), where {} clarifies how the code

groups together and **return()** outputs the result. Note that the variable name **v** could be anything, since it is just a placeholder to tell R what to do when actual values are put in.

(6) Code:
```
mymean = function(v) {
  return(sum(v)/length(v))
} # End of function definition

mymean(x) == mean(x) # Trying out the function
```
Result:
```
[1] TRUE
```

Even though R is built for vectors, you can still manipulate values one by one via a loop, though if the vector is very long, this can be much slower than doing it in a "vectorized" manner. Try out the code in (7) and (8) and you will get a sense of this difference in speed.

(7) Code:
```
myslowmean = function(v) {
  n = length(v)
  s = 0; t = 0 # These will contain the incremental sum and total count
  for (i in 1:n) { # Looping as many times as v has values
    s = s + v[i] # Add the ith value of v to the incremental sum s
    t = t + 1 # Count how many values so far
  }
  return(s/t) # Same as sum(v)/length(v)
}
myslowmean(x) == mymean(x)
```
Result:
```
[1] TRUE
```

(8)
```
# From now on, you should try the code to see the results yourself
huge = 1:1e8 # That is, this vector's length is 1 x 10^8 = 100,000,000
huge # See? It is huge....
mymean(huge) # Fast, because it uses vectors directly
myslowmean(huge) # Slow, because it uses a loop
```

Also like any other computer language, R can be used to write logical decision rules, like the example in (9); here there is a doubled logical symbol (**&&**) so that it works on just one value.

But R's vector bias means that logic can also be used to select a subset of values from a vector, as in (10), using the simpler logical symbol (**&**).

```
(9)     if (mean(x) < 3 && mymean(x) > 4) { # Must be false, so n stays 5
          n = 999
        }
(10)    x[x > 2 & x < 4] # 3
```

In vectors, all the elements are the same type of thing (here, all numbers). If you want to create a sequence of different types of things, you use **list()** instead, as illustrated in (11), where the list contains two vectors rather than simply being a single vector of numbers.

```
(11)    y = c(2,c(0,2,3)); y # Vector (2,0,2,3) (";" separates commands on one line)
        y[2] # 0
        z = list(2,c(0,2,3)); z # List made of the vectors (2) and (0,2,3)
        z[[2]] # (0,2,3); note the double brackets to select an element from a list
```

Real-life data usually involves a matrix (a rectangle of values), rather than just a vector. R generally refers to rows before columns, as illustrated in (12).

```
(12)    w = matrix(0, nrow=3, ncol=4); w[1,2] = 99; w # Note where the 99 is placed
        w[2,1] == 99 # False, of course
        w[1,] # The first row
        w[,2] # The second column
```

R is particularly interested in an enhanced matrix called a data frame, in which each individual observation appears on a separate row and each column represents something about that observation. For example, experimental results might have columns identifying participants, stimulus items, the properties of each item and/or participant that are expected to affect the responses, and the responses themselves. We will see examples of such things later.

Like most modern computer languages, R is what is known as an object-oriented language: the commands always make a thing that can be named and manipulated by other commands. This is particularly convenient when running additional analyses on statistical models that are slow to create (perhaps due to looping). So instead of just typing in a command like **DoTheAnalysis(MyData)**, it would be wiser to type in **MyResults = DoTheAnalysis(MyData)**; if necessary you can even apply further commands like **MakeAFancyPlot(MyResults)** or **save(MyResults, file = "MyResults.R")** (this last is a real function, where, as we will further discuss below, we have to use the straight quotes **""** rather than the curly quotes **“”**).

Finally, no statistics software would be complete without plotting, which is not just for the final report but also for the researcher, as part of the data exploration process. The simplest R plotting function is **plot()**, which by default makes a scatter plot, since that is the most general way to see the relation between two variables. For example, if **x** is still **c(1,2,3,4,5)**, then **plot(x,x^2)** will show a rising sequence of dots (i.e., dots long the curve $y = x^2$). The actual curve will be hard to see, since by default R also makes all plots square, so the *x*-axis and *y*-axis will have different scales. Even plotting commands create objects. For example, **hist(x)** creates a histogram for **x**, so if we type **myhist = hist(x)** and then **myhist**, we can see all the values that went into creating this graph (in the results, note that **X$Y** indicates that **Y** is part of **X**; try typing **myhist$mids**).

1.3 Extending R

The base version of R has thousands of built-in functions, including those that do the most famous statistical things (means, scatter plots, histograms, standard deviations, *t* tests, ANOVA, logistic regression, and so on) and lots of others that do pure math (e.g., fans of linear algebra and calculus

should check out the functions **eigen()** and **deriv()**), but there are also hundreds of thousands of additional functions you can get access to by installing extra packages.

For example, I have only recently discovered the package **installr** (Galili, 2022), which helps install R things in Windows, including an entire updated version of R itself. To do this, first install the package itself, using the command **install.packages("installr")** or the menus in the default GUI or in RStudio. To turn on the package (which you only need to do once per R session), type **library(installr)**. One of the functions that this will load is **updateR()**, so now if you just type that in (no argument needed), it will guide you through the update process.

One of the most widely used R packages is **ggplot2** (Wickham, 2016), so named because it was its creators' second attempt to adapt the "grammar of graphics" proposed by Wilkinson (1999). This is essentially a framework for expressing plots in a modular way; for example, the shape of plotting elements, like dots versus lines versus bars, is conceptually distinct from how they are linked to particular variables in a data frame, and so there are different classes of functions and parameters for those different types of concepts. However, while **ggplot2** allows one to fine-tune fancy graphs (albeit with controversial defaults, like a gray background), its grammar is not only different from that of base R (e.g., its commands must be combined with the + sign, instead of simply listing them in sequence), but it also seems to be intrinsically difficult to learn, as admitted by Winter (2019) (otherwise a fan), and even its own creators say "It's hard to succinctly describe how ggplot2 works" (https://ggplot2.tidyverse.org/). I doubt that I am the only **ggplot2** user who usually builds on working examples from the internet, instead of trying to create new plots from scratch. One resource that has helped me is Chang (2013), available both as a printed book and in a frequently updated free online version (https://r-graphics.org/).

No matter how you make your R graphs, you might also want to ensure that you pick colors that look nice on screen and yet remain clear when printed in grayscale or viewed by colorblind readers. Essentially the trick is to use colors that correlate hue (e.g., red vs. green, which look identical to most colorblind viewers) with brightness (e.g., by making the red brighter than the green). To get a sense of what this looks like, visit https://colorbrewer2.org/ (Harrower & Brewer, 2003), and try playing around with the settings. These settings are available in **ggplot2** (https://ggplot2.tidyverse.org/reference/scale_brewer.html), via functions in which the word "color" can also be spelled as "colour." For non-**ggplot2**-users, packages that incorporate print-friendly colors include **sp** (Bivand et al., 2013) and **rje** (Evans, 2022). For example, the code in (13) shows one way to choose a three-way color contrast. Figure 1 shows how the code colors bars in a bar plot (hopefully still distinguishable if you are reading this paper in black and white).

(13)
```
library(sp) # Needed for the function bpy.colors()
allcols = bpy.colors(); ncols = length(allcols); allcols # 100 colors in HEX code
# Illustrate the full range (pch & cex define shape & size of points in base R graphics)
plot(1:ncols, pch=19, cex=2, col=allcols)
# Choose three colors along this continuum, avoiding the endpoints
mycolvals = seq(from=ncols/4, to=ncols-ncols/4, by=ncols/4) # 25, 50, 75
mycols = allcols[mycolvals] # Note the vector logic here!
# Use colors in a bar plot (for other options, type ?barplot)
barplot(1:3, beside=T, names.arg=c("A","B","C"), ylab="Bar height", col=mycols,
  border=mycols, main="Distinct bar colors")
```

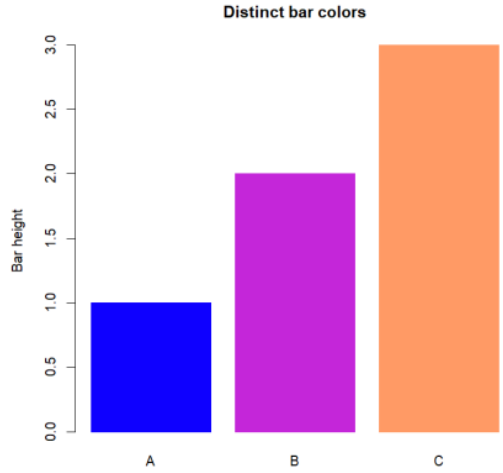

Figure 1. Printer-friendly bar plot colors (bluish, purplish, orange-ish)

The influential team behind **ggplot2** has since combined it with a slew of other packages into the influential **tidyverse** (Wickham et al., 2019). As the name suggests, the creators think the packages make data analyses easier and/or neater, but they do not expect everybody to agree; as they put it (see https://www.tidyverse.org/), it is an "opinionated collection of R packages." I personally do not like it, but Winter (2019) does, as do a growing number of websites, so R users are currently being forced to learn two dialects in order to understand web tips.

More humble R users may enjoy the package **Rcmdr** (for "R Commander"; Fox, 2005), which is designed to make R look and feel more like a point-and-click program like SPSS, though it also offers windows for entering actual R code. Installing and loading its library in base R magically causes a new GUI to appear, with menus filled with statistical tests. An alternative tool to do the same thing and more, without having to get your hands dirty with R at all, is JASP (https://jasp-stats.org/), which is a standalone program (written in C++ and the GUI-creating language QML) that runs R in the background, so all you see are the menus, data, and results (though, again, there is a window for you to write actual R code if you like). A unique strength of

JASP is that it also includes tools for Bayesian statistics, yet another sign that statisticians are serious about making this powerful approach more accessible to ordinary researchers. For a list of several other R interface tools, see https://www.ubuntupit.com/best-free-graphical-user-interfaces-for-r/.

Just as other programs can run R, so too can R run other programs (a point we will return to in a later section). For example, the packages **rstan** (Stan Development Team, 2023) and **brms** (Bürkner, 2017) allow the user to do much fancier Bayesian analyses than are possible in JASP, by interfacing with the full-fledged Bayesian software Stan (https://mc-stan.org/). Even the power of the rival data analysis language Julia can be harnessed via R's **JuliaCall** package (Li, 2019).

Not all of the R packages available online are hosted in the CRAN repository. If you just need one R code file, you can type **source("RcodeFile")** (where **RcodeFile** is a full web address). For example, if you enter **source("https://ablejec.nib.si/R/quincunx.R")** into R, something educational and amusing will happen; to see the code itself, just type the web address into your browser (Blejec, 2009). People who want to install a full R package that is still under development, for example a project that currently only exists as a bunch of separate files on GitHub (https://github.com/), can use the separate program Rtools (also available at CRAN), then install the R **devtools** package (Wickham et al., 2022), which offers the **install_github()** function. Below we will see yet another example of how to install packages missing from CRAN.

2. R beyond statistics

So far, I have only partially kept my promise to focus on non-statistical things, given the occasional allusion to things like means, Bayes, and bar plots. From here on out I will be more careful. In this

section I discuss various R functions and packages for working with text, processing corpora, and interfacing with software of particular interest to linguists.

2.1 Working with text

While R is designed for working with numbers, linguists also want to work with text. Of course, like all computer languages, R has functions for character strings, and since the release of version 4.2.0 in April 2022 (surprisingly late!), it finally plays nicely with Unicode characters, removing the need to handle them with special packages (Kalibera et al., 2022). In other words, Chinese and the IPA no longer baffle it, whether they appear inside files or in filenames, are copy/pasted or typed directly into the GUI, are analyzed or manipulated, or are printed in plots.

The code in (14)-(15) illustrates a variety of useful character functions applied to a mix of Chinese and non-Chinese symbols. Try it out, one line at a time, to see what happens. In particular, note the use of **grep()** for searching regular expressions, a flexible way to do pattern matching in strings (but which is tricky to get right without internet help plus trial and error), and **adist()** for calculating the so-called edit distance between strings, that is, the number of insertions, deletions, and replacements needed to make the strings identical.

(14)
```
# Some basics
x = "大家來學 R"; x # Let's all learn R
nchar(x) # Number of characters (5)
y = "大家來學 R" # Crash! Gotta use straight quotes in R commands
z = ""大家來學 R""; z # That's OK
密碼 = 3; 密碼 # This is also OK!
"""; "" # Quoting a quote mark by using the other type of quote mark; "\"" = literal ""
plot(0,0,type="n"); text(0,0,z) # Make empty plot centered at 0, then add our z text
```

(15)
```
# Manipulating text strings
x1 = paste(x, "吧", sep=""); x1 # Concatenation with nothing ("") as separator
substring(x,4,5) # Characters from position 4 to position 5 from left: "學 R"
```

```
substring(x, nchar(x), nchar(x)) # Just the final character: "R"
strsplit(x, split="來") # Creates a list = sequence of anything of any type
y = unlist(strsplit(x,"")); y # Split by nothing, then coerce list into a vector
grep("R",y) # 5: Finds matches with a “regular expression”: ?regexp or search web
z = "How do you do"; zv = unlist(strsplit(z, " ")); zv # ("How","do","you","do")
grep("d",zv) # Finds it in words 2 & 4
grep("o.",zv) # Finds it only in words 1 and 3, since “.” = at least one character
gsub("do", "don’t", z) # Global substitution: “How don’t you don’t”
adist(x, x1) # “Edit distance” = how many characters must change to match
adist("abc","abz"); adist("abc","abzc"); adist("abc","cba") # Get it?
adist("do",zv); adist(zv, zv) # Vector and matrix logic again
```

For fancier string processing, additional packages are helpful. The **tidyverse** has one called **stringr** (Wickham, 2022a), which basically tries to neaten up the syntax of functions like those above, as well as some of the string operations in the more powerful package **stringi** (Gagolewski, 2022). The cheat sheet for **stringr** (https://github.com/rstudio/cheatsheets/blob/main/strings.pdf) is updated regularly and contains a variety of easy-to-follow examples (there are also cheat sheets for all of the other tidyverse components).

More obscure packages can also be useful for special purposes. For example, if you want to display specific fonts in a plot, you might take a look at the **showtext** package (Qiu, 2023). By default, R recognizes only a very small set of basic fonts; for example, in Windows, typing **windowsFonts()** may show only three: a serif (print-like) font, a sans-serif (no-frills) font, and a monospaced (typewriter-style) font, which in my Windows 11 system are Times New Roman, Arial, and Courier New, respectively. But my computer also has more interesting fonts, like Old English Text MT (similar to the font in the logo for *The New York Times*) and LiSu (隶书 *lìshū* ‘clerical script’, based on simplified character Unicode). Digging into my computer’s fonts folder, I learned that the actual font files are called OLDENGL.TTF and SIMLI.TTF, respectively. This then let me run the code in (16), generating the result in Figure 2.

```
(16)  library(showtext)
      font_add(family="oldeng", regular= "OLDENGL.TTF")
      font_add(family="clerical", regular="SIMLI.TTF")
      showtext_auto() # Enables the above new font names
      # The rest just uses base R graphics functions
      par(mai=c(0,0,0,0)) # Suppresses margins; see ?par for a lot more options
      plot(1:3, 1:3, type="n", axes=F) # Empty plot with no axes; the numbers help set the size
      text(2, 2.2, "大家來學", family = "clerical", cex = 12)
      text(2, 1.8, "R", family = "oldeng", cex = 15)
```

Figure 2. Fancy fonts in a plot (manually cropped in a picture editor to save space here).

Going the other direction, if you have a text on paper and want R to read it, you can try the **tesseract** package (Ooms, 2022), which incorporates the open-source neural-network-based optical character recognition (OCR) algorithm called Tesseract (https://github.com/tesseract-ocr). For example, to read an image file with horizontally written traditional Chinese, you could build on the code in (17), which reads in Wikipedia's traditional Chinese logo; the result is somewhat garbled at the start (the Wikipedia globe itself has text in it), but otherwise works quite well. To read in text from a PDF file rather than an image file, you can use the additional package **pdftools** (Ooms, 2023a), which converts the PDF file to an image file using the function **pdf_convert()**, and then your code can feed the result into **ocr()**.

```
(17)  library(tesseract)
      tesseract_download("chi_tra") # Only needs to be done once
      chin.trad = tesseract("chi_tra") # Give OCR model a name for this session
      # To use the next command, make sure the web address is all on one line
```

```
mytext = ocr("https://upload.wikimedia.org/wikipedia/commons/thumb/a/a0/
Wikipedia-logo-v2-zh.svg/135px-Wikipedia-logo-v2-zh.svg.png", engine= chin.trad)
mytext # I got "AA 人 革\n 維基白科\n 自由的百科全書\n" (\n = line break)
```

Of course, for a one-time job, it may be more convenient to do each step manually with special purpose software (e.g., Adobe Acrobat [https://www.adobe.com/acrobat.html] for OCR, then copy/paste to a text file, then read into R). At the start of a project, there is always the question of which will take more time: doing it by hand, or figuring out how to automate it. Nevertheless, adopting what computer people call a full "workflow" within R can be useful if you have a lot of files to deal with, need to be able to explain exactly what you did to somebody else (collaborators, advisors, or reviewers), or want to replicate the same steps yourself on a future set of data.

2.2 Processing corpora

As noted earlier, there are whole textbooks about using R for corpus linguistics (e.g., Gries, 2017). Here I will focus on simple tricks for doing just two non-quantitative corpus-related things: extracting data from web sites, and tagging text elements.

Despite all the technological advances of the past thirty years, the vast majority of web pages remain what they have been since the birth of the modern internet: HTML files (HyperText Markup Language, for formatting files with links to other pages). HTML files are just text files with hidden annotations that your browser uses to help make the page look pretty.

For example, if you choose the menu option in your browser to view the page "source," you will see that Wikipedia article on Yuen Ren Chao (趙元任) in traditional Chinese for Taiwan (https://zh.wikipedia.org/zh-tw/趙元任) actually starts as in (18). Note that the title is surrounded by the tags <title> and </title>; on the full page, all of the other tags shown here also have

corresponding closing tags (aside from <meta> and the first line, which is a comment). The actual content of the article starts after the <head> section is closed with </head> (not shown here).

(18)
```
<!DOCTYPE html>
<html class="client-nojs ..." lang="zh-Hant-TW" dir="ltr">
<head>
<meta charset="UTF-8">
<title>趙元任 - 維基百科，自由的百科全書</title>)
```

Actually, if you search for "</head>" in your browser's source display, you will see that the content does not start immediately after the head finishes, so let us search for the name "趙元任" after "</head>". As of the time of this writing (Wikipedia is often updated), its first appearance after the HTML head section is in line 377 of the HTML file, as shown in (19). This is the article title, so it gets the highest-level header tag: <h1>. Note also that the "class" parameter given for the tag is "mw-page-title-main", so it seems we are on the right track.

(19)
```
<h1 id="firstHeading" class="firstHeading mw-first-heading" lang="zh-Hant-TW"
dir="ltr"><span class="mw-page-title-main">趙元任</span></h1>
```

If we use R to download this page for us, what we will get is the raw text file that the source display shows, not the pretty version of the page public face. This is illustrated by the code in (20), which downloads the page and then displays the first line after "</head>" that contains "趙元任" (note the "clever" but confusing way in which I decided to do this).

(20)
```
ChaoPage = readLines("https://zh.wikipedia.org/zh-tw/趙元任") # Ignore the warning
ChaoPage.HeadEnd = grep("</head>",ChaoPage) # Line number with this closing tag
ChaoPage.Body = ChaoPage[(ChaoPage.HeadEnd+1):length(ChaoPage)]
ChaoPage.Body[grep("趙元任",ChaoPage.Body)][1] # "\t" = tab
```

No matter what else we might want R to do with the web page data, it is important to know that this is what R is actually seeing, not the formatted version normally shown by the browser. Moreover, the hidden HTML tags themselves can be useful for restricting our search, as we just did (e.g., "趙元任" appears a few times in the HTML head as well, but those cannot be seen by users so we should ignore them in any linguistic analysis).

To make this sort of process more user-friendly, the **tidyverse** includes the package **rvest** (Wickham, 2022b; the name puns on "harvest", as in harvesting or "scraping" web pages). This is illustrated in the R code in (21) (based on examples at https://rvest.tidyverse.org/), which is able to give us just the target string "趙元任" itself, with all of the HTML tags filtered out. Crucially, however, before we know what we can tell R to filter out, we first have to know how the tags are actually used in the HTML file, and whether they are used consistently (often not the case), and as far as I can tell, to do this you just have to poke around the HTML with your own eyes.

(21)
```
library(rvest)
ChaoPage = read_html("https://zh.wikipedia.org/zh-tw/趙元任")
ChaoPage.Title = html_elements(ChaoPage,"h1")
# Now extract the formatted text inside <span> inside the sole <h1>
html_text2(html_element(ChaoPage.Title,"span"))
```

It should be noted, by the way, that when automatically scraping large numbers of files, the **rvest** creators recommend also using the **polite** package (Perepolkinm, 2023), which helps R avoid burdening internet servers with too many requests.

To illustrate another corpus analysis technique, namely, tagging, for simplicity we will set aside Prof. Chao and his HTML-riddled article and build on our extremely short "corpus" from earlier, expanded to "來來來，大家來學 R！". Suppose we have a dictionary that classifies

orthographic characters as being Chinese, non-Chinese, or punctuation, and we want to tag our "corpus" so we can easily see which is which. In the R code in (22), this dictionary has one entry per character in the form of a data frame, and then it is combined with our corpus to yield a new data frame with corpus items plus their tags.

(22)
```
Corpus.raw = "來來來，大家來學 R ！"
Corpus.chars = as.character(unlist(strsplit(Corpus.raw, ""))) # Split and vectorize corpus
Char = sort(unique(Corpus.chars)) # Drop repeats and sort for dictionary
Char # For human inspection
# [1] " ！" ",  " "R"  "來" "大" "學" "家" # Note that ",  " is a Chinese    string here
Class = c("Punc","Punc","Latin","Chin","Chin","Chin","Chin")
Dictionary = data.frame(Char, Class); Dictionary # Take a look!
Corpus = data.frame(Token = 1:length(Corpus.chars), Char = Corpus.chars)
Corpus = merge(Corpus, Dictionary) # Key step, but sorts by Char, which we don't want
Corpus = Corpus[order(Corpus$Token),] # Sorts by corpus token order: [row,col]
Corpus = Corpus[,c("Token","Char","Class")] # Token number back to first column
Corpus # Take a look!
```

In the above code, the extra steps needed to re-sort by token order is the fault of base R's **merge()** function, whose option **sort=F** does not mean "keep original order" as any rational person would expect. This is the sort of awkwardness that motivates the **tidyverse**, which in this case includes **dplyr** (Wickham et al., 2023; the name is meant to imply "data pliers"). This package offers the function **join()** with (hopefully) more intuitive defaults and options.

2.3 Interfacing with software

Earlier I noted that R can be used to talk to special-purpose software for Bayesian analysis and even its rival data analysis language Julia. Here I'll very briefly mention two more cases specifically related to linguistics.

Anybody who studies acoustic phonetics is likely to have some experience using Praat (Boersma, 2001; https://www.fon.hum.uva.nl/praat/), given its great power, small size, ease of use,

and low price (free). While Praat can be used manually via menus and point-and-click tools, processes can also be automated with the help its own built-in coding language. If you already know R, however, you can make it do Praat tasks, without having to learn a new language, with the package **rPraat** (Bořil & Radek, 2016). Rather than attempting a summary here, I will just direct readers to explore the rich set of demos at https://fu.ff.cuni.cz/praat/rDemo.html.

I will be similarly brief with artificial intelligence modeling, also known more prosaically as machine learning. Today this is generally synonymous with deep learning, so-called because of its reliance on many-layered artificial neural networks. Deep learning is what powers modern translation software, speech recognition and generation systems, and those scarily human-like chatbots. While Python is probably the most useful and flexible language to use when building and training such things (Lane et al., 2019, provides a reader-friendly tutorial on its linguistic applications), there are also R-based options. The simplest are packages like **neuralnet** (Fritsch et al., 2019) and **rnn** (Quast, 2023); they only handle pre-millennial shallow-learning models, but such models underlie still-influential classics like Elman (1993). Far more powerful, in fact quite cutting-edge, is the **tensorflow** package (Allaire & Tang, 2023), which interfaces with Python and TensorFlow itself, the most widely used open-source library of deep learning tools (https://www.tensorflow.org/). Tutorials and other resources are available at the project website (https://tensorflow.rstudio.com/), and if you want a book, there is a book too (Chollet et al., 2022).

3. Case study: Exploring handwritten Chinese

In the rest of this paper, I will discuss a single case study, a pilot experiment on Chinese handwriting that built on the perceptual experiment reported in Myers (2022a), in order to expand on several of the points made above. I first give a mini-report on the experimental methods and

results, and then describe how R was used to compile the data, to create the non-statistical graphics, and to implement novel analytical ideas. For readers who want to try everything themselves can find all of the necessary files at https://osf.io/3fzhw/ (at the Open Science Framework).

3.1 The experiment

As with Myers (2022a), the purpose of the experiment was to test an arcane hypothesis about reader/writer knowledge of Chinese character form. Namely, Wang (1983) observed that the leftmost vertical stroke must be curved (i.e., 豎撇 *shùpiē* 'vertical throw-away') in narrow components (as in 丹 *dān* 'cinnabar, red') but in wide ones there is some lexical variation (e.g., it is curved in 周 *zhōu* 'cycle', but straight in 同 *tóng* 'same'); see Wang (1983, pp. 203-206), and Myers (2019, section 3.4.5). Myers (2019) analyzes this correlation in terms of an orthographic analog of prosody: the left side is prosodically weak (due to left-to-right writing order), curving favors prosodically weak positions, wide components have two prosodic constituents, each with an obligatory head (like feet), and thus the leftmost stroke falls in a (strong) head position, where curving is disfavored.

The results of the categorical perception experiment in Myers (2022a) suggested that readers have internalized the curving/width correlation, based on their different patterns in stroke curving judgments in narrow versus wide "arched" stimuli; these were based on the Unicode characters 冂 and 冂, but modified to vary gradiently in the degree of curving along a 10-point continuum while keeping width constant within each set (narrow vs. wide).

In order to test if this tacit knowledge can also be observed in handwriting, and to explore possible articulatory implications of the prosodic hypothesis, a new experiment was run on 39

right-handed native Chinese-speaking traditional character writers. Four arched stimuli were chosen, as shown in Figure 3. These crossed width (narrow vs. wide) and curving (curved vs. straight), with the first contrast as large as in the perceptual experiment but the second relatively subtle, falling just below and just above the threshold dividing curved from straight judgments, as determined in the perceptual experiment.

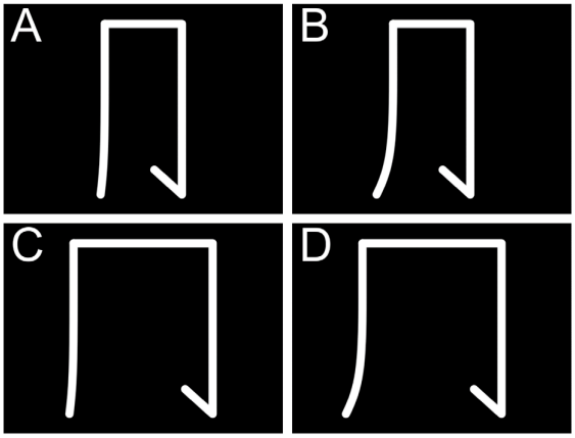

Figure 3. Stimuli used in the experiment crossing width (top vs. bottom rows) with curving (left vs. right columns).

A practice session with eight stimuli of intermediate width was followed by 40 experimental trials with each of the four experimental items appearing in random order ten times each. The participants' task was simply to handwrite each item after it appeared. The display of the stimuli and the recording of the handwritten responses occurred in the same 5 cm by 5 cm square screen area using a Wacom One tablet and pen (https://www.wacom.com).

In this methodology-focused paper I restrict myself to discussing just the five pilot participants (i.e., lab assistants). After processing the raw results files, a single data file was produced, with trial-level data on the participant ID, item type (wide vs. narrow, curved vs. straight), the number of strokes actually written (in order to drop trials with characters written with fewer or more than two strokes), the degree of curving in the first stroke, and the length, duration,

and speed (length/duration) of each stroke. Figure 4 shows all of the correctly written characters, arranged the same way as in Figure 3.

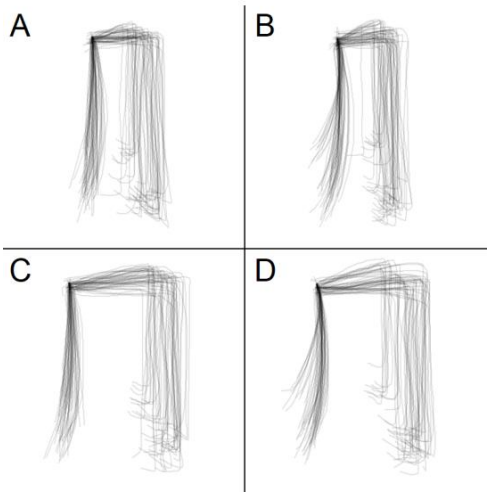

Figure 4. All two-stroke written productions corresponding to the stimuli shown in Figure 3.

The pilot data set is too small for meaningful statistics, but for completeness, the above OSF link also includes R code for statistical analysis, including some **ggplot2** plots (see the file PilotAnalysis.R). Data from the full experiment seems to suggest that there is a greater difference in curving degree in wide characters, consistent with this being the context in which curving can be lexically contrastive. The full data set also provides some support for the prosodic hypothesis: in wide characters, the two strokes are written at more similar speeds than in narrow characters, as if writers treat wide characters as containing two prosodic timing units rather than just one.

Since the focus of this paper is methodology, I should also note the non-R software tools that I used. The Chinese dictionary software Wenlin (文林; https://wenlin.com/; Bishop & Cook, 2007) was used to edit entries in its text-based Character Description Language (CDL), which were then converted into SVG (scalable vector graphics) image files (the colors were later inverted as described below). The experiment was run in the Python-based PsychoPy (Peirce et al., 2022;

https://www.psychopy.org/), which makes pen tracking surprisingly easy: its Mouse component treats pen position as mouse position and pen contacts as left mouse button presses (extra work would have been needed to read in the Wacom pen pressure). Even if you do not have a tablet, you can use a mouse to try out the PsychoPy experiment at the OSF link.

3.2 Using R to compile pen tablet data

The files output by PsychoPy are in CSV (comma-separated values) format, which arranges a table of data into rows with commas between each cell in the row, including empty cells (e.g., "A", "", "B" becomes "A,,B"). This allowed me to make use of the base R function **read.csv()**, which creates a data frame filled with the loaded-in data. Homemade functions then combined the files and converted raw pen (mouse) data into stroke-level descriptions; see the file GetPilotResults.R.

My R code starts by loading these homemade functions with **source("WritingTools.R")**. It then creates an empty data frame with column names for all the variables that I thought I might need. Instead of writing separate **read.csv()** commands for each data file, I got the names of all of the files via the **dir()** function, which lists file names in a folder (directory) as character strings. A **for** loop then loaded in each, one at a time. Some columns I do not need, so I throw them out; for example, to keep just the columns A and B, we can write **mydata = mydata[,c("A","B")]** (remember that in X[Y,Z], the Y and Z represent the rows and columns of X, respectively, so X[,Z] refers to all rows). The code also throws out all rows containing NA (not available), which is how R interprets empty cells, via **na.omit(mydata)**. Since the columns preserved in the data frame do not include the practice trials, those rows are filled with empty cells and thus get dropped.

Each file is then processed via another loop that goes row by row, which is where the functions from WritingTools.R come into play. The most fundamental of these is **vectR()**, which

converts PsychoPy cells containing pen status information (position, contact, and current time, which R reads in as strings) into R vectors. For example, pen contact (left mouse button) status is loaded as strings like "[0, 0, ... , 0, 1, 1, 1, ..., 1, 0, ...]" (where 0 = no pen contact and 1 = pen contact); **vectR()** turns these into numerical vectors like c(0,0,...0,1,1,1,...,1,0,...). These vectors can then be searched to find the starting and ending points of strokes (i.e., where the contact vector changes from 0 to 1 and from 1 to 0, respectively), which in turn allows strokes to be identified and counted. By combining this information is combined with mouse position and time points, we can then create functions that measure response latency (time from start of pen measurements to first contact), duration (time from first to last contact), stroke length (in both space and time), and even, if so desired, inter-stroke durations. For handwriting experiments on more complex characters than tested here, my function **draw.char()** is particularly useful, since it outputs image files showing what the participants actually wrote (a simpler function was used to create Figure 4, as explained below). The **draw.char()** function color-codes stroke order (gradiently from first = black to last = red), and, if desired, also prints little numbers on them (at the start of each stroke, so stroke direction is also made clear); see the images in the folder HandwritingImages_jm at the above OSF link. I encourage anyone who has thought about running handwriting experiments to try out, and adapt, the PsychoPy experimental control files and the functions at that link.

3.3 Creating graphics

The two figures shown in section 3.1 were produced with the help of R's **magick** package (Ooms, 2023b), which makes use of the free and incredibly powerful image processing library called ImageMagick (https://imagemagick.org/). The R implementation receives a good though

incomplete overview at https://cloud.r-project.org/web/packages/magick/vignettes/intro.html. The R code creating these figures are given at the above OSF link in the file MakePix.R.

As noted earlier, the stimuli in Figure 3 above were originally created using Wenlin, but R was used to invert the colors to white-on-black with the **magick** function **image_negate()**. In Figure 3, I also used **magick** to combine the four stimulus images into one by applying the functions **image_read()** to read in each image in a loop, **image_border()** to add white margins around each black image, **image_annotate()** to add the letter labels, **image_montage()** to actually combine the images, and finally **image_write()** to save the result.

Although Figure 4 is structured the same way, the arrangement was actually handled by the base R graphics command **par(mfrow=c(2,2))**, which sets the parameter for a matrix of figures in a two-by-two layout, with each individual figure created using a special-purpose function in WritingTools.R called **overlap.chars()**. This just uses R's base **lines()** function with the color set by the base function **rgb()**: translucent black is created via **rgb(red=0, green=0, blue=0, alpha=0.1)**. Because the color is translucent, the more lines are drawn through some point, the darker that point gets, which makes visible both the general pattern and the variation (a sort of statistical distribution without using any statistics). The function **overlap.chars()** is itself called by code in MakePix.R that makes sure the first pen contact is placed in the same spot for all overlapped characters (i.e., at the upper left of each of the images in Figure 4).

3.4 Automatic analysis of stroke properties

In the overview of the experimental results in section 3.1, I mentioned an interesting finding involving stroke speed, which is simply the length of a stroke divided by its duration. Stroke duration is easy to calculate from the times associated with the start and end of a stroke (using the

get.stroke.dur() function). Calculating stroke length is somewhat harder, since strokes are never perfectly straight; the function **get.stroke.len()** works by summing up the tiny distances between each pair of points in a stroke by applying the Pythagorean theorem to their *x* and *y* coordinates.

Quantifying the degree of stroke curving went beyond my mathematical abilities, however, so I searched the internet until I came across the R package **EBImage** (Pau et al., 2010; https://www.bioconductor.org/packages/release/bioc/html/EBImage.html), which is part of something called the Bioconductor project (for open-source bioinformatics software); quantifying curves is apparently useful for classifying microscopic images. For some reason, this package is not in CRAN, so, to install it, the creators suggest using the R code in (23), which first installs their project's **BiocManager** package (Morgan, 2023), and this in turn installs EBImage.

(23)
```
if (!require("BiocManager", quietly = TRUE)) # Installs BiocManager if not present
  install.packages("BiocManager")
BiocManager::install("EBImage") # Uses BiocManager to install EBImage
```

All I ended up using in this package was the function **localCurvature()**, which is itself written entirely in R (see https://rdrr.io/bioc/EBImage/src/R/localCurvature.R). This fits a parabola to each point of your wiggly line; the more curved your line is at that point, the narrower the parabola is, and so, by definition, the larger the curvature value. For enclosed areas (like a cell under a microscope), convex and concave curves receive negative and positive values, respectively. Presumably due to how it loops around enclosed areas, **localCurvature()** treated my character strokes as if they were the right edge of an enclosed area, thus yielding curving negative values for each point. To provide a single positive score for each stroke, my function **get.curve()** first takes the mean of all point-specific curving values and then negates them.

The point of this story is that even if you want to do weird analyses that nobody may have ever done before, you can still create or find R functions that can help. Do not feel constrained by what you find in R statistics textbooks, even those written specifically for linguists.

4. Take-home message

The previous sentence already does a pretty good job in summarizing this paper, but by convention I need a concluding section, so I will put it another way. Linguists who go to the trouble of learning R for statistical analysis deserve a reward, and one of the greatest is that R is a full-fledged computer language, with a huge number of special packages for doing a wide variety of things, so if you learn R, you can do all of those things too. As shown by the overview in this paper, R can organize data files, manipulate text, handle fonts, process sounds, build AI models, and create, format, and analyze images. Not only is all of this power available for free, but so are most of the tutorials that you need to learn it. Just FYI.

REFERENCES

ALLAIRE, JOSEPH J.; YUAN TANG. 2023. tensorflow: R interface to TensorFlow. R package.

BAAYEN, R. HARALD. 2008. *Analyzing linguistic data: A practical introduction to statistics using R*. Cambridge, UK: Cambridge University Press.

BEZANSON, JEFF; ALAN EDELMAN; STEFAN KARPINSKI; and VIRAL B. SHAH. 2017. Julia: A fresh approach to numerical computing. *SIAM Review* 59.1: 65-98.

BISHOP, TOM; and RICHARD COOK. 2007. A character description language for CJK. Multilingual 18.7: 62-68.

BIVAND, ROGER; EDZER PEBESMA; and VIRGILIO GOMEZ-RUBIO. 2013. *Applied spatial data analysis with R*, second edition. Berlin: Springer.

BLEJEC, ANDREJ. 2009. Quincunx function. https://ablejec.nib.si/.

BOERSMA, PAUL. 2001. Praat, a system for doing phonetics by computer. *Glot International* 5.9/10:341-345.

BOŘIL, TOMÁŠ; and RADEK SKARNITZL. 2016. Tools rPraat and mPraat. In Sojka et al., 367-374.

BÜRKNER, PAUL-CHRISTIAN. 2017. brms: An R package for Bayesian multilevel models using Stan. *Journal of Statistical Software* 80.1:1-28.

CHANG, WINSTON. 2013. *R graphics cookbook*. Sebastopol, CA: O'Reilly.

CHOLLET, FRANÇOIS; TOMASZ KALINOWSKI; and J. J. ALLAIRE. 2022. *Deep learning with R* (2nd edition). Shelter Island, NY: Manning.

CLAWSON, DAN. 1998. *Required reading: Sociology's most influential books*. Amherst: University of Massachusetts Press.

ELMAN, JEFFREY L. 1993. Learning and development in neural networks: The importance of starting small. *Cognition* 48.1: 71-99.

EVANS, ROBIN. 2022. rje: Miscellaneous useful functions for statistics. R package.

FOX, JOHN. 2005. Getting started with the R commander: A basic-statistics graphical user interface to R. *Journal of Statistical Software* 14.9:1-42.

FRITSCH, STEFAN; FRAUKE GUENTHER; and MARVIN N. WRIGHT. 2019. neuralnet: Training of neural networks. R package.

GAGOLEWSKI, MAREK. 2022. stringi: Fast and portable character string processing in R. *Journal of Statistical Software* 103.2.1-59.

GALILI, TAL. 2022. installr: Using R to install stuff on windows OS. R package.

GOOD, PHILLIP I. 2013. *Introduction to statistics through resampling methods and R*, second edition. Hoboken, NJ: John Wiley & Sons, Inc.

GRIES, STEFAN TH. 2013. *Statistics for linguistics with R: A practical introduction* (2nd edition). Berlin: De Gruyter.

GRIES, STEFAN TH. 2017. *Quantitative corpus linguistics with R: A practical introduction*. Abingdon, UK: Routledge.

HARROWER, MARK; and CYNTHIA A. BREWER. 2003. ColorBrewer. org: an online tool for selecting colour schemes for maps. *The Cartographic Journal* 40.1: 27-37.

IHAKA, ROSS. 2011. The R Project: A brief history and thoughts about the future. Talk presented at the University of Otago (April 20). https://www.stat.auckland.ac.nz/~ihaka/?Papers_and_Talks.

JOHNSON, KEITH. 2008. *Quantitative methods in linguistics*. Hoboken, NY: Wiley.

KALIBERA, TOMAS; SEBASTIAN MEYER; and KURT HORNIK. 2022. Changes in R. *The R Journal* 14.2: 336-338. https://journal.r-project.org/news/RJ-2022-2-rcore/.

KRUSCHKE, JOHN. 2015. *Doing Bayesian data analysis: A tutorial with R, JAGS, and Stan, second edition*. Amsterdam: Academic Press.

LANE, HOBSON; COLE HOWARD; and HANNES MAX HAPKE. 2019. *Natural language processing in action: Understanding, analyzing, and generating text with Python*. Shelter Island, NY: Manning.

LARSON-HALL, JENIFER. 2015. *A guide to doing statistics in second language research using SPSS and R* (second edition). Abingdon, UK: Routledge.

LEVSHINA, NATALIA. 2015. *How to do linguistics with R: Data exploration and statistical analysis*. London: John Benjamins.

LI, CHANGCHENG. 2019. JuliaCall: An R package for seamless integration between R and Julia. *The Journal of Open Source Software* 4.35:1284.

MCELREATH, RICHARD. 2016. Statistical rethinking: A Bayesian course with examples in R and Stan. Boca Raton, FL: CRC Press.

MORGAN, MARTIN. 2023. BiocManager: Access the Bioconductor Project package repository. R package.

MYERS, JAMES. 2019. *The grammar of Chinese characters: Productive knowledge of formal patterns in an orthographic system*. Abingdon, UK: Routledge.

MYERS, JAMES. 2022a. Thin & curvy: Unconscious knowledge of a subtle Chinese character stroke pattern. Presented at NACCL-34, September 23-25.

MYERS, JAMES. 2022b. *Yet another statistics-for-linguistics book* https://lngmyers.ccu.edu.tw/var/file/64/1064/img/YASFLB.htm.

NICENBOIM, BRUNO; SHRAVAN VASISHTH; FELIX ENGELMANN; and KATJA SUCKOW. 2018. Exploratory and confirmatory analyses in sentence processing: A case study of number interference in German. *Cognitive Science* 42:1075-1100.

OOMS, JEROEN. 2022. tesseract: Open source OCR engine. R package.

OOMS, JEROEN. 2023a. pdftools: Text extraction, rendering and converting of PDF documents. R package.

OOMS, JEROEN. 2023b. magick: Advanced graphics and image-processing in R. R package.

PAU, GRÉGOIRE; FLORIAN FUCHS; OLEG SKLYAR; MICHAEL BOUTROS; and WOLFGANG HUBER. 2010. EBImage: An R package for image processing with applications to cellular phenotypes. *Bioinformatics* 26.7: 979-981.

PEIRCE, JONATHAN; REBECCA HIRST; and MICHAEL MACASKILL. 2022. *Building experiments in PsychoPy*, second edition. London: Sage.

PEREPOLKIN, DMYTRO. 2023. polite: Be nice on the web. R package

QIU, YIXUAN. 2023. showtext: Using fonts more easily in R graphs. R package.

QUAST, BASTIAAN. 2023. rnn: Recurrent neural network. R package.

R CORE TEAM. 2023. R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/.

RSTUDIO TEAM. 2023. RStudio: Integrated development environment for R. https://posit.co/.

SOJKA, PETR; ALEŠ HORÁK; IVAN KOPEČEK; and KAREL PALA (eds.) 2016. *Text, speech, and dialogue*. Heidelberg: Springer International Publishing.

SONDEREGGER, MORGAN. 2022. Regression modeling for linguistic data. https://github.com/msonderegger/rmld-v1.1.

STAN DEVELOPMENT TEAM. 2023. RStan: The R interface to Stan. R package.

VAN ROSSUM, GUIDO; and FRED L. DRAKE. 2009. *Python 3 reference manual*. Scotts Valley, CA: CreateSpace.

VASISHTH, SHRAVAN; DANIEL SCHAD; AUDREY BÜRKI; and REINHOLD KLIEGL. 2022. Linear mixed models in linguistics and psychology: A comprehensive introduction. https://vasishth.github.io/Freq_CogSci/.

WANG, JASON CHIA-SHENG. 1983. *Toward a generative grammar of Chinese character structure and stroke order*. Ph.D. dissertation, University of Wisconsin-Madison.

WELLMAN, BARRY. 1998. Doing it ourselves: The SPSS manual as sociology's most influential recent book. In Clawson (ed.), 71-78.

WICKHAM, HADLEY. 2016. *ggplot2: Elegant graphics for data analysis*. Berlin: Springer.

WICKHAM, HADLEY et al. 2019. Welcome to the tidyverse. *Journal of Open Source Software* 4.43: 1686.
WICKHAM, HADLEY. 2022a. stringr: Simple, consistent wrappers for common string operations. R package.
WICKHAM, HADLEY. 2022b. rvest: Easily Harvest (Scrape) Web Pages. R package.
WICKHAM, HADLEY; ROMAIN FRANÇOIS; LIONEL HENRY; KIRILL MÜLLER; and DAVIS VAUGHAN. 2023. dplyr: A grammar of data manipulation. R package.
WICKHAM, HADLEY; JIM HESTER; WINSTON CHANG; and JENNIFER BRYAN. 2022. devtools: Tools to make developing R packages easier. R package.
WILKINSON, LELAND. 1999. *The grammar of graphics*. Berlin: Springer.
WINTER, BODO. 2019. *Statistics for linguists: An introduction using R*. Abingdon, UK: Routledge.