DOI: http://dx.doi.org/10.1590/1678-992X-2019-0122

Scientia Agricola

ISSN 1678-992X

Biometry, Modeling and Statistics

lattice: Easy construction of professional graphs for agricultural research in R environment

Marcin Kozak*

Warsaw University of Life Sciences – Dept. of Botany, Nowoursynowska, 159 – 02-766 – Warsaw – Poland.
*Corresponding author <nyggus@gmail.com>

Edited by: Paulo Cesar Sentelhas

Received May 06, 2019
Accepted November 04, 2019

ABSTRACT: This paper shows how to apply the `lattice` package of R to create effective scientific graphs. The readers will learn basic notions of the package and ways to work with it in an easy way. The R code the paper provides will help them create various graphs, including a scatter plot, a box plot, a density plot, and a bar plot; with a little work, the code can be changed to make other graphs. The paper emphasizes the trellis display, a useful but still undervalued technique in scientific visualization.

Keywords: visualization, charts, software, statistics

Introduction

Agricultural research calls for a variety of methods for analyzing various types of data. Analysis is one thing, interpretation is another. While most analytical methods in agriculture use statistics, interpretation should not base solely on rigorous statistical analysis. Sometimes exploratory tools can help notice interesting phenomena in data, which can then be analyzed using such rigorous methods. But alone, seldom does statistical analysis suffice to deeply analyze and interpret agricultural data.

To interpret data efficiently, one can graph them. A bunch of graphical methods to visualize scientific data is large, and so is their availability—so what can limit researchers in using them is skills. So many methods to choose from can be dangerous, too: For most data, we can choose from among quite a large set of graphs, and thus making a good graph requires knowledge and skill. This paper, I hope, will help you gain them.

Graphical methods can show much more than raw data and their summaries can—but also more than can rigorous statistical analysis. This does not mean, however, that data visualization should replace statistics, but rather that these two tools can and should cooperate in order to provide as clear a picture of the phenomena studied as possible.

Most statistical methods offer data summaries, and although it is true that we scientists usually look for trends, trends alone do not suffice. What if something is atypical? Summaries not only will not show such anomalies in the data, but they also can draw an incomplete and untrue picture of the phenomena studied. In practically every single case, we should check the data, to protect ourselves from overlooking important phenomena in them. Failing to do so, we might, for instance, fit a linear regression line to analyze a non-linear relationship; or, by analyzing means without checking the raw data, we might miss an outlier observation—and just one such outlier can change the whole ranking of means. That graphical methods are better than statistical tests to test ANOVA assumptions have already been discussed (e.g., Kozak and Piepho, 2018).

Data exploration and interpretation, however, are not the only occasions when graphical methods can help. Data presentation is another. It is already a cliché to say that scientific results are usually best presented in a graphical form—but it's a true cliché. More often than not, visualizing data can help understand them by those who do not know the data's details—the readers of scientific papers in particular.

It takes some craft to make a good graph, however (e.g., Tufte, 1983; Cleveland, 1985 and 1993; Wilkinson, 2006; Wickham, 2009; Kozak, 2010). Whether exploring data or presenting them, we have to do it right. Graphing data in a random way will seldom work—we have to know what we want to graph, we have to know how we can do it effectively, and we have to do it with skill. What also does not have to work is following, without careful consideration, some standard practices, such as using a pie chart only because we have per cent values (do they sum up to 100%?) or using a bar chart for one-way data (maybe the data represent a time trend, usually better shown using a line graph?).

There is one important lesson I learned when teaching data visualization to undergraduate and graduate students of various disciplines. People, even educated, often respond negatively to graphs they have not used or at least met. Most students I teach are used to pie charts and bar plots, and quite a few to line plots. But, however amazing it can sound, most of them consider simple scatterplots difficult. By a simple scatterplot I understand a quantitative variable presented on a vertical axis against another quantitative variable presented on a horizontal axis, one of the most frequent scientific graph types. In fact, seeing such a graph, even with just a handful of points graphed, they often attack it as overly difficult to read and interpret. From my discussions with them it's clear that they consider such graphs difficult because they compare them—usually unconsciously—with pie and bar charts, both of which are simpler and present simpler data structures than does the scatterplot. So, I have to explain them why and when such a scatterplot can be effective—and that, combined with several examples, convinces at least part of them.

If such a simple plot can give educated people the willies, what about much more advanced and complex graphs presenting much more data in more complex structures?

Thus, I do think researchers should spend some time learning basics of data visualization. Otherwise, they may be reluctant to use even quite simple graphs, only a little more advanced than the bar plot.

Graphical data exploration can be a brainstorming and iterative process: You graph whatever comes to your mind, and you look for anything interesting or peculiar in the data; you can try various graphs, looking for the one that best shows the picture. That way, you may draw a whole world of rich conclusions, which you might miss by graphing only what you think at the outset is worth graphing (e.g., in two-way ANOVA, a bar plot of means together with an interaction plot).

Data analysis does not require as fine-tuned graphs as data presentation does. Usually, when making a graph to analyze your data, you need not pay attention to some aspects of the graph, such as axis titles or color—if only you see what you need to see, that's fine. But when you want your graph to be read by others, you should pay attention to every single detail.

This paper offers a simple guide for agricultural researchers to the `lattice` package (Sarkar, 2008) of R (R Core Team, 2019). Focusing on graphs especially valuable for agricultural researchers, I will show how to use this package to make professional—even quite complex—graphs. After reading the paper and going through the examples, you should be able to construct such graphs by your own. I will not focus on the principles of graphing data, which I did in my earlier paper in this journal (Kozak, 2010)—but we will use them, and we will use here the same data sets and make similar graphs.

I make two assumptions here. First, I assume you know these basic principles of graphing data—because without such knowledge it's better not to try to make any graph. That would be like driving a car without even the basic knowledge of how to do so. Many sources are available (my favorite are Tufte [1983] and Cleveland [1995, 1996]), including the above-mentioned paper I published in this journal (Kozak, 2010). I also assume you have a basic knowledge of R, which will enable you to read data and use basic operating symbols and functions.

So, I assume we will work together. If you do not know too much of R, however, don't give up: Reading the paper may help you get a general picture of `lattice's` possibilities and of how it works. But to learn using it, you should switch on your computer, run R (even better, RStudio; http://rstudio.com), load the `lattice` and `latticeExtra` packages, and copy the code from the paper into R's console. Thus, I will not include in this paper all graphs we will work on, just those most important ones—but you can make all of them all simply by copying the code into the R console.

A short introduction to `lattice`

The `lattice` package was developed by Deepayan Sarkar. In 2008, he published a book on this package (Sarkar, 2008). Although `lattice` offers a variety of graphs, its backbone is based on the idea of the trellis display, developed by Cleveland and colleagues (e.g., Cleveland, 1985 and 1993; Becker et al., 1996). A simple idea, the trellis display is powerful: Thanks to its layout, it enables showing and interpreting even complex grouped data on one graph.

Even though `lattice` graphs look very professional, they are quite easy to make if only one knows how to do it. `Lattice` has several built-in functions for standard types of graphs such as

- a bar plot
- a scatter plot, both 2D and 3D
- a strip plot
- a dot plot (Cleveland, 1983)

You might have noticed that the above list misses pie charts. Sarkar himself explains this (Sarkar, 2008): "`lattice` does not contain a function that produces pie charts. This is entirely by choice, as pie charts are a highly undesirable form of graphical representation (see Cleveland (1985) for a discussion), and their use is strongly discouraged."

We could append the list with less known types of graphs, such as

- parallel coordinate plot
- scatterplot matrix (SPLOM)

`lattice` also offers statistical graphs, such as

- histogram
- density plot
- box plot
- violin plot
- quantile-quantile plot

`lattice's` formula interface makes it easy to construct these graphs in a trellis display, often with just one line of code, as I will show later. What's more, with a little skill, you can control most graphical parameters of these graphs. Some of them, however, are more difficult to control, and it can take considerable skill to adjust a graph to all your needs.

While many packages offering graphs need to be used with much care because their default parameters often can be a bad choice (e.g., Su, 2008; Wnuk and Dębski, 2016), it is not so with `lattice` (although not everyone likes the default choice of colors). Its great advantage is that it is based upon in-depth research on data visualization, thanks to which much of its default parameters (such as point symbols and their size, line symbols, rotation of tick mark labels on axes, etc.) can be used without hesitation. This is not to say, however, that when using `lattice`, you don't have to worry about the look of your graphs; you always should. But this is to say that with lattice, you have hi gh chances of ending up with a good graph even if you do not change the default parameters. In fact, if you feel your knowledge about principles of graphing data is limited, you might wish to use the default setting lattice offers. I would not say the same about many other packages for data visualization.

If you are still wondering why you should choose lattice over other packages, consider the following points:

- lattice offers simple and advanced graphs, which can be made with a little programming skill and in a relatively short time
- once you learn the formula interface, you will have no problems with working with a variety of graph types lattice constructs
- more often than not, the default settings of lattice graphs are good

However useful lattice is, I am not claiming everyone should use it. There are other options. Outside of R, the Plotly module of the Python programming language offers rich visualization possibilities. R itself offers other possibilities. You can use R's graphics package, built-in into R as its graphical base. You can use R's ggplot2 (Wickham, 2009). ggplot2 is powerful but less intuitive for non-specialists. It gives, however, much more flexibility than does lattice, an important advantage when you want to construct atypical graphs. This does not mean that either one is better but that the two packages target slightly different audiences: lattice is for those who seek a quick and relatively easy way of constructing professional scientific graphs, but from a limited set of possibilities, while ggplot2 is for those who are looking for flexibility, especially when working on complex visualizations. Note that all graphs you can make in lattice can also be made with ggplot2, but usually you would need more skill and time and work. We cannot say the otherwise, however: Not all graphs constructed with ggplot2 can be constructed with lattice; or, many of them might be, but that would require a lot of skill and time and work. Thus, basically, if you just want to make professional scientific graphs from among those listed above, you can choose lattice, and it will repay your relatively little effort.

lattice basics: A scatterplot

lattice comes with the basic R installation, so it needs not be installed. To use it, you need to load it first:

```
library(lattice)
```

The basic lattice package offers a lot of visualization possibilities. We can broaden them by installing latticeExtra, an additional package which adds some advanced utilities to lattice (Sarkar and Andrews, 2016). We will use it later on, so let's install and load it.

```
install.packages("latticeExtra")

library(latticeExtra)
```

For simplicity, we will use the same data sets I used in my previous paper on graphing (Kozak, 2010), but we will start with one of the most well-known data sets, the iris data by Anderson (1935). After Fisher (1936) had used it, the dataset started its journey towards unintended and quite amazing future: It became famous among statisticians, machine learning specialists, and data analysts, who use it to test their methods and show how they work (Kozak and Łotocka, 2013). It is also often used in data visualization.

In R, this data set is available as dataset iris. We can glance over its first rows using function head:

```
head(iris)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

So, we have here four quantitative traits (sepal length, sepal width, petal length, and petal width) for three iris species (*Iris setosa*, *I. virginica*, and *I. versicolor*), 50 observations per species.

Let's start our lattice adventure by making a simple scatterplot of petal length versus its width (Figure 1):

```
xyplot(Petal.Length ~ Petal.Width, data = iris)
```

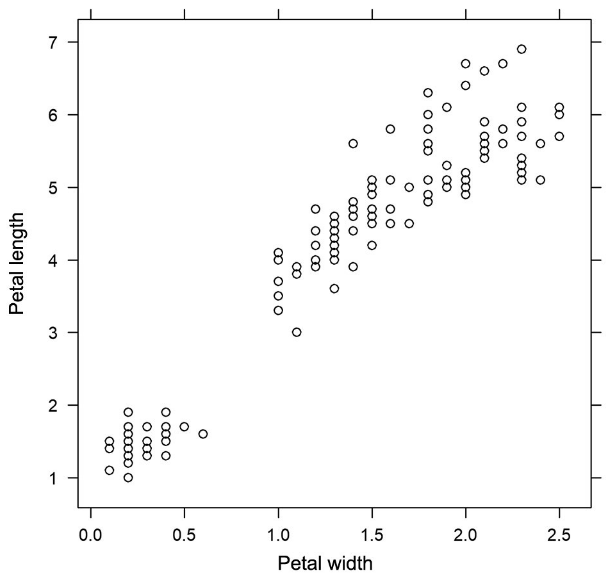

Figure 1 – The simplest version of the scatterplot showing petal length versus petal width. For the moment, the information about the three species is ignored.

As simple as that. Pay attention to the formula, here `Petal.Length ~ Petal.Width`. It's the essence of any call to `lattice`'s graphical functions. What goes before the tilde is the dependent variable; here it's a variable to be represented by the y axis. After the tilde goes a variable to be represented by the x axis. Later we will use more complex formulas, which will help us to add more layers to the graph.

As already mentioned, `lattice`'s important advantage is its default settings (for the numerous aspects of the graph, such as point symbol and size, color, axes, etc.). Usually using them is a good idea. But what one *must always* think about is the graph's layout: Which type of graph should I use? Is color better than other tools (such as different symbols)? What should be on the y and x axes? Do I have a grouping variable I should include? If so, how should I do it? What scale should I use (maybe logarithmic)? Do I have problems with overlapping data points? And so on.

What we *should* change most of the times is axis titles (usually called axis labels), which by default are the same as variable names (here, Petal.Length and Petal.Width). We can do it using `xlab` and `ylab` arguments:

```
xyplot(Petal.Length ~ Petal.Width, data = iris,
       xlab = "Petal width", ylab = "Petal length")
```

Let us now think about the data. The data set has 150 rows, so Figure 1 should show 150 data points. Does it? It's difficult to say from just looking at the graph, and we will not count them one by one—but the graph might suggest we do not see all the 150 points. Such a phenomenon can happen when some points overlap (so some petals have exactly the same lengths and widths). We can check that by counting the number of unique rows (considering only petal length and width, so the third and fourth columns of the `iris` data frame):

```
nrow(unique(iris[, 3:4]))

## [1] 102
```

Since there are 102 unique rows, the graph does not show all the points—only 68% of them. While coping with this problem does not have to be easy with other software, it is easy in R. Suffice to use the function `jitter`, which adds so-called *jitter*, a small amount of random noise added to a quantitative variable (Chambers et al., 1983):

```
xyplot(jitter(Petal.Length) ~ jitter(Petal.Width), data = iris,
       xlab = "Petal width", ylab = "Petal length")
```

And now, all—or at least most of—the points can be seen.

Note that above we have ignored quite an important piece of information about the data, that is, the species. The data has three Iris species, which we can see in R by typing `unique(iris$Species)`, so we should show them on the graph:

```
xyplot(jitter(Petal.Length) ~ jitter(Petal.Width),
       data = iris, groups = Species,
       xlab = "Petal width", ylab = "Petal length")
```

In `lattice`, if we use color for graphs, groups are differentiated using different colors. When we use black and white graphs, they are differentiated using different graphical symbols. When working in R console, the color version is default. Since we will be working with graphs in shades of grey, you should initialize the corresponding so-called *theme*:

```
lattice.options(default.theme = standard.theme(color = FALSE))
```

Later on, however, try the very same graphs in the default (color) theme. To do so, open a new R session and omit this theme change.

Let's move on, but this time we will work on several things at the same time (Figure 2):

```
levels(iris$Species) = paste("I.", levels(iris$Species))
plot1 <- xyplot(jitter(Petal.Length) ~ jitter(Petal.Width),
                data = iris, groups = Species,
                xlab = "Petal width", ylab = "Petal length",
                type = c("p", "r"),
                aspect = "iso",
                auto.key = list(space = "right", lines = T, font = 3))
print(plot1)
```

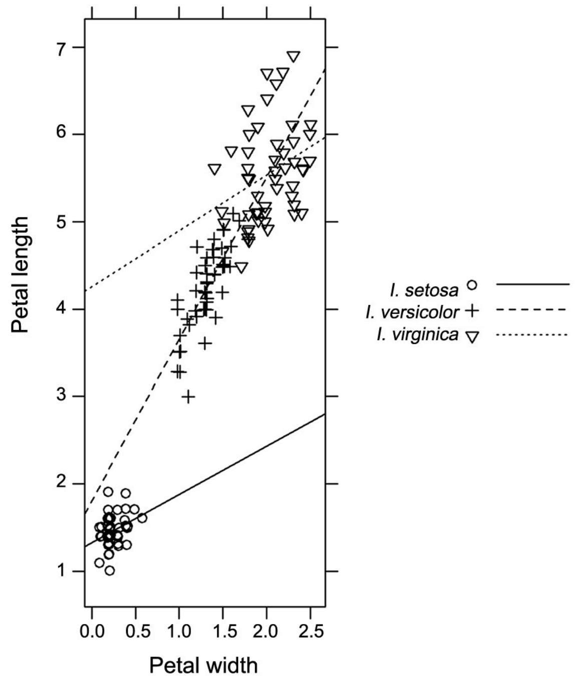

Figure 2 – Petal length versus petal width for three Iris species. Jitter was added to both variables. Different symbols represent the groups, and different lines represent group-wise linear relationships between petal length and width. Isometric scales are used so that 1 cm has the same physical distance at both axes.

Actually, this time we have made quite a big step. In addition to what we already had, we have

- changed the species names in the legend, by adding "I." before them, and used italic type face; the easiest way to do so is by changing the data (as we did above), not the plot itself
- made the graph an object called `plot1`; we can use this object later, for example when saving it to a disk or even to adjust some of its elements (those not related to the data)
- used points (`type = "p"`) for points and added regression lines (`type = "r"`); when two types of points are used, a vector is used to pass this information: `type = c("p", "r")`
- used an isometric property (`aspect = "iso"`) for the graph, thanks to which 1 cm has the same physical distance at both axes, making them directly comparable (Cleveland et al. 1988); here it makes sense because the two variables—petal length and width—are comparable, which is a key condition for using the isometric property
- added the legend, located right to the plot: `auto.key = list(space = "right")`
- used a function `print` to print the plot; we have to do it to print a graph when we assign it to an object
- added italics to the legend, using the `font` argument to the `scales` list

That we made the plot an object (which we called `plot1`) gives us some possibilities to adjust it without rewriting the whole function call once more. Instead, we can easily update some of the graph's elements—those unrelated to the data. For instance, we can change the linear regression lines to local regression lines:

```
plot2 <- update(plot1, type = c("p", "smooth"))
print(plot2)
```

The only thing that was changed to `plot1` was the type argument. After updating a `lattice` object (here, `plot2` is an updated object of object `plot1`), we need to use the `print` function to re-print it. To practice, you can play a little bit with updating `lattice` objects and, at the same time, with the `aspect` argument, for instance,

```
print(update(plot1, aspect = 1))
print(update(plot1, aspect = .5))
print(update(plot1, aspect = 2))
```

For Figure 2, however, I prefer the isometric property, for the reason given above. Note that in the three lines of code above we neither created new objects (as we did to create `plot2`) nor overwrote the existing one (`plot1`)—we simply printed the updated plots.

`lattice` basics: Other types of graph

Above we used a scatterplot, one of the most often used graph in science. We will now consider some other graph types `lattice` enables us to create.

A dotplot is a good start. Proposed by Cleveland (1993), it serves a similar purpose as a bar plot but does not share its disadvantages. In a dot plot, a quantitative variable is plotted against a qualitative variable. Notice the similarities between this graph type (or any other we're going to use) and more typical ones, such as a scatterplot. Their differences, on the other hand, are not that big: While a scatterplot shows a quantitative variable against another quantitative variable, a dot plot shows a qualitative variable against a quantitative variable.

We will work with the same `haynes` dataset of the agricolae package (de Mendiburu, 2019) as I did in Kozak (2010). If you do not have the package installed, do it with command `install.packages("agricolae")`. The data represent mean area under the disease progress curve (AUDPC) for 16 potato clones from eight sites across the USA in 1996. Since we are going to use only the data set, we need not load the whole package, just the data:

```
data(haynes, package = "agricolae")
```

This is what the data look like:

```
head(haynes)

##        clone  FL   MI   ME  MN  ND  NY  PA   WI
## 1   A84118-3 284 1113 1053 997 612 590 484  800
## 2  AO80432-1 254  690 1112 832 761 518 555 1153
## 3  AO84275-3 395 1089 1090 387 633 296 347  850
## 4 AWN86514-2 136  296  374  44 377 101  46   41
## 5    B0692-4  87  653  412  73 334  70  26  141
## 6    B0718-3 130  126  329  72 444 167  61  211
```

Let's make the graph, but for the moment for just one site ("FL"):

```
dotplot(clone ~ FL, haynes, xlab = list("Mean AUDPC", cex = .8))
```

First, note that the dotplot function uses a formula `qualitative variable ~ quantitative variable` and puts the qualitative one at the y axis. We can change that, here by setting `FL ~ clone`, but it seldom would be a good idea (you can check it yourself why).

Note how I decreased the font of the x axis label. In a similar way, we could change the font of other elements. For instance, to change the font of tick mark labels of y axis (so, of the clone names), use the `scales` argument (which is a list as well):

```
dotplot(clone ~ FL, haynes, xlab = list("Mean AUDPC", cex = .8),
        scales = list(y = list(cex = .7)))
```

We could—and in fact should—consider ordering the clones by AUDBC, which we can do using the `reorder` function, as follows:

```
dotplot(reorder(clone, FL) ~ FL, haynes,

        xlab = list("Mean AUDPC", cex = .8))
```

In the next section, we will return to this example and work with the other sites. Now, we will create a graph that shows group means with their standard errors. For this, we will use dataset `InsectSprays`, which gives counts of insects in agricultural experimental units treated with different insecticides. Since we are dealing with counts, we will use a generalized linear model with a Poisson error structure and a (default) log link. Here's how we can derive the standard errors of the estimates:

```
data(InsectSprays)

mod <- glm(count ~ spray - 1, data = InsectSprays,

    family = "quasipoisson")

results <- as.data.frame(summary(mod)$coefficients[, 1:2])

results$spray <- paste("Spray", LETTERS[1:6])

colnames(results)[2] <- "SE"
```

Note that we used quasi-estimation, which worked here better than the original Poisson distribution. If you type `plot(mod)`, you will learn the model looks correct. The dataset `results` contains the transformed data (means and their standard errors), so we need to back-transform them. From among various ways of doing so, let us use function `mutate` from the dplyr package (remember to install it if you haven't done that yet):

```
library(dplyr)

results <- mutate(results,

                  means = exp(Estimate),

                  SEs = exp(SE),

                  L = means - SEs,

                  U = means + SEs

               )
```

We are ready to create a bar plot that shows the means with their standard errors (Figure 3):

```
barchart(reorder(spray, means) ~ means,

         data = results,

         col = "grey", border = "transparent",

         L = results$L, U = results$U,

         xlab = expression("Mean number of insects (" %+-%"SE)"),

         xlim = c(0, 18.5),

         panel = function(x, y, L, U, ..., subscripts) {

           panel.barchart(x, y, ..., subscripts = subscripts)

           panel.arrows(x0 = L[subscripts], y0 = y,

                        x1 = U[subscripts], y1 = y,

                        code = 3, angle = 90, length = 0.025)

         })
```

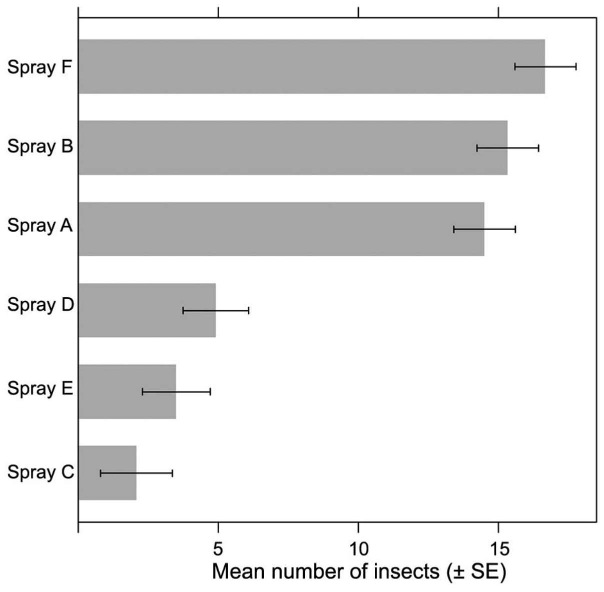

Figure 3 – Mean number of insects for six sprays. Error bars represent the standard errors of the means. Data source: Beall (1942), through the `InsectSprays` data set in R.

Here, we

- ordered the sprays by their respective means
- used a transparent border for grey bars
- used the `expression` function, to write the "±" symbol
- had to "tell" the function how to plot the bars. For this, we used a `subscript` option that `lattice` uses in such instances. We also used panel functions: the `panel.barchart` function (a default panel function for `barchart`) and the `panel.arrors` function (to draw the error bars)

To learn more about these functions, type `?expression`, `?panel.barchart`, or `?panel.arrows`.

Easy to notice, we have entered the area of more complex functionalities of `lattice`. The length of this paper makes it impossible for me to explain all the details of using panel functions. To use them at the basic level, however, it is enough to follow the pattern (from the above command). While explaining how they work is quite difficult, using them by imitation is not that difficult. But remember to define the values the function is to use—`L` and `U`, for the lower and upper limits—and to use them in the panel function. To be able to use all the possibilities the panel functions offer, however, an interested reader should use advanced sources on `lattice` (e.g., Sarkar, 2008).

Knowing the general functioning of `lattice` functions, we can create other plots to present the same data. Here is an example:

```
dotplot(reorder(spray, means) ~ means,

        data = results,

        col = "black",

        L = results$L, U = results$U,

        xlab = expression("Mean number of insects (" %+-%"SE)"),

        xlim = c(0, 18.5),

        panel = function(x, y, L, U, ..., subscripts) {

          panel.dotplot(x, y, ..., subscripts = subscripts)

          panel.arrows(x0 = L[subscripts], y0 = y,

                       x1 = U[subscripts], y1 = y,

                       code = 3, angle = 90, length = 0.025,)

        })
```

Although we have created quite a different graph, note how similar the function calls are, the panel functions in particular. I hope this has convinced you that what I wrote above about imitating the use of panel functions is true.

Now that we are familiar with the main rules governing creating graphs with `lattice`, we can easily apply the other graphing functions `lattice` offers. Below, we will analyze the distributions of variables using two types of graphs: the box plot and the density plot.

Both require more data points than just a few, but the `iris` data set offers samples of sufficient size. Assuming we are making the graphs to analyze the data and not to present the graphs in a publication, we need not worry about all aspects of the graphs' look.

Here's a box plot for petal length:

```
bwplot(Species ~ Petal.Length, data = iris)
```

and here's a density plot:

```
densityplot(~ Petal.Length, data = iris, groups = Species, auto.key = TRUE)
```

For the first time, we have used the legend, which we done using the `auto.key` argument. Doing so is simple—suffice to add the argument `auto.key = TRUE`. To customize a legend, we can use a list of the elements we want to change, like here:

```
densityplot(~ Petal.Length, data = iris, groups = Species,

            auto.key = list(space = "right", points = FALSE, lines = TRUE))
```

or

```
densityplot(~ Petal.Length, data = iris, groups = Species,

            auto.key = list(columns = 3, points = FALSE, lines = TRUE))
```

Although the argument `auto.key` offers quite rich functionalities, it is just a simplified version of the argument `key`, which offers even more.

The essence of `lattice`: Trellis display

It's time to make a bigger step. We are now moving to the essence of `lattice`: the trellis display (Cleveland, 1985, 1993; Becker et al., 1996). The simplest version of the trellis display for the iris data would be as follows:

```
xyplot(jitter(Petal.Length) ~ jitter(Petal.Width) | Species, data = iris)
```

Note how easy making a trellis display is—we just added one element to the formula: `| Species`. To improve the graph, we will use what we have learned above (Figure 4):

```
xyplot(jitter(Petal.Length) ~ jitter(Petal.Width) | Species, data = iris,

       xlab = "Petal width", ylab = "Petal length",

       type = c("p", "r", "g"),

       aspect = "iso",

       strip = strip.custom(factor.levels =

                  c(expression(italic("I. setosa")),

                    expression(italic("I. versicolor")),

                    expression(italic("I. virginica")))))
```

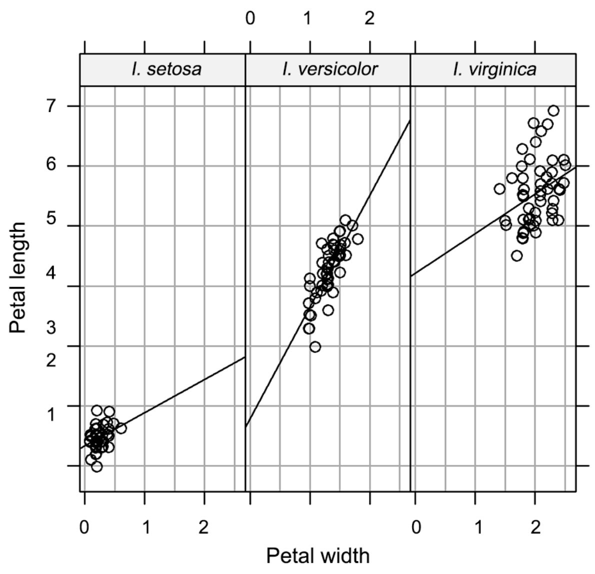

Figure 4 – The trellis display for the `iris` data: petal length versus petal width for the three Iris species. Data source: Anderson (1935), available through the `iris` dataset in R.

Note that

- the formula has a new element, `| Spieces`, which is interpreted as "make separate plots for each `Species`"; these plots are called "panels"
- we have added the **"g"** type, which is responsible for adding the grid lines to the panels
- to add italics to the species names, we have changed the strip font type, with the help of the `strip.custom` function

Trellis displays (Cleveland, 1993) have a number of advantages, the main one being that they make it easy to compare data group-by-group. Each group is presented in a separate panel, but—*which is crucial*—the panels are formatted in the same way, so we can easily compare the data they show. Here, this means that each panel has the same limits of the x and y axes.

In Figure 4, we made a trellis display of scatterplots, but we can work with most other types of plots. We will now make a dot plot for the `haynes` data set, but this time each site will have its own panel (Figure 5). First, we need to transform the data into a long format, with the help of the `melt` function from the reshape2 package (remember to install and load it before using the function):

```
haynes_melted <- melt(haynes, variable.name = "site", value.name = "AUDBC")

dotplot(reorder(clone, AUDBC) ~ AUDBC | reorder(site, AUDBC),
        haynes_melted,
        xlab = list("Mean AUDPC", cex = .8),
        scales = list(cex = .8),
        layout = c(2, 4))
```

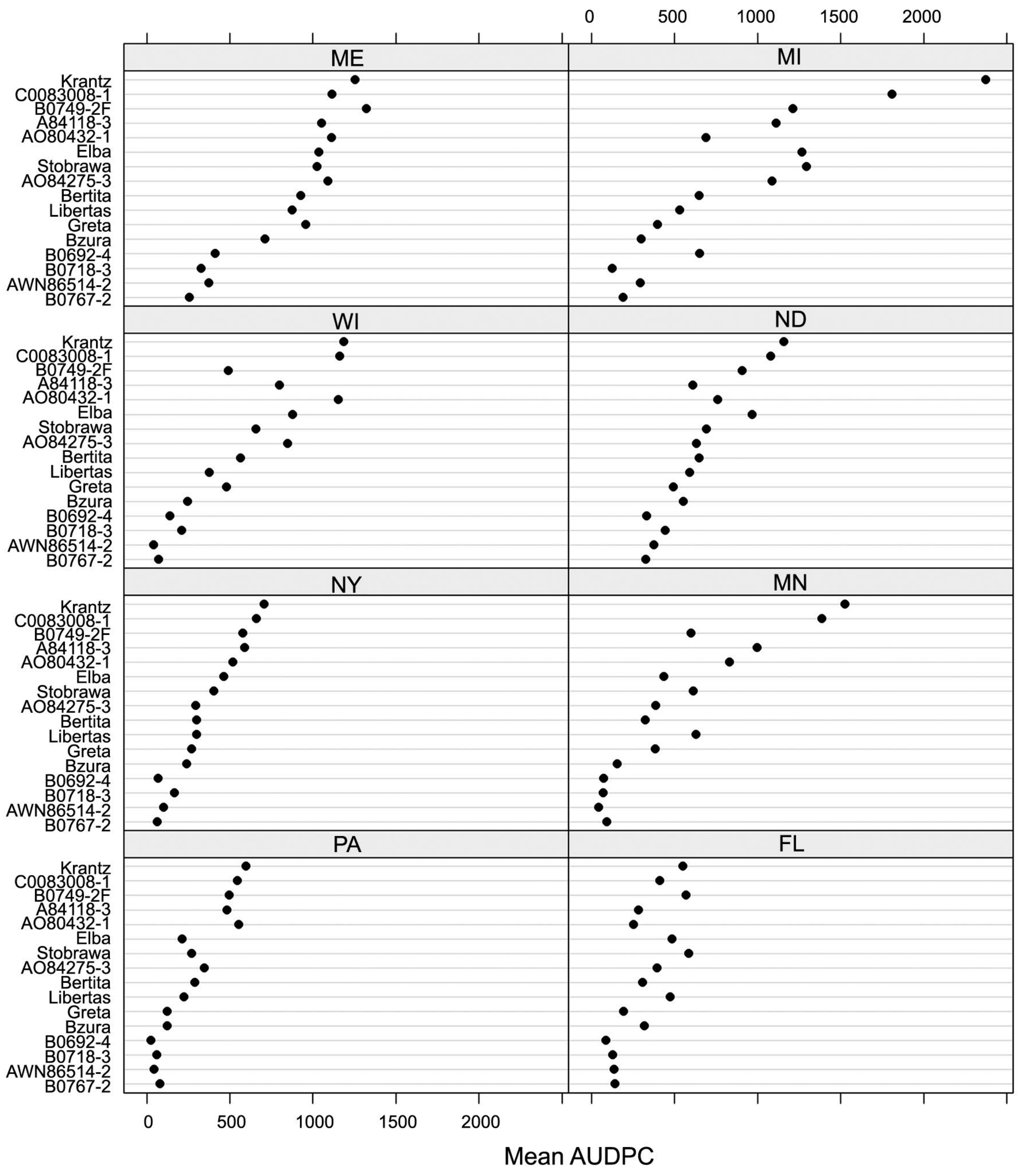

Figure 5 – Mean AUDPC of 16 clones in eight locations. The panels' order matches the increasing mean AUDPC value, and so does the clones' order. Data source: Haynes et al. (1998) through the `haynes` dataset in the agricolae package of R.

We used the function reorder two times: `reorder(clone, AUDBC)` orders clones on the y axis by the mean AUDBC for the clones across all the sites; and `reorder(site, AUDBC)` orders sites on the panels by the mean AUDBC for the sites across all the clones. Thus, the clone at the bottom of the y axis has the lowest while the clone at the top of the y axis has the greatest mean value of AUDBC across the sites; and the site in the left bottom panel has the lowest while that in the right top panel has the highest mean AUDBC across the clones.

Other examples

In this section, I will show you several additional examples representing the rich possibilities of `lattice`. The examples will be simple in terms of the code, but this simplicity will *not* be reflected in the simplicity of the resulting graphs.

The below code creates a strip plot, showing petal lengths of all 150 individuals, 50 per iris species (Figure 6):

```
levels(iris$Species) = c("I. setosa", "I. versicolor", "I. virginica")
stripplot(Species ~ Petal.Length, data = iris,
          jitter = TRUE, xlab = "Petal length",
          scales = list(y = list(font = 3)))
```

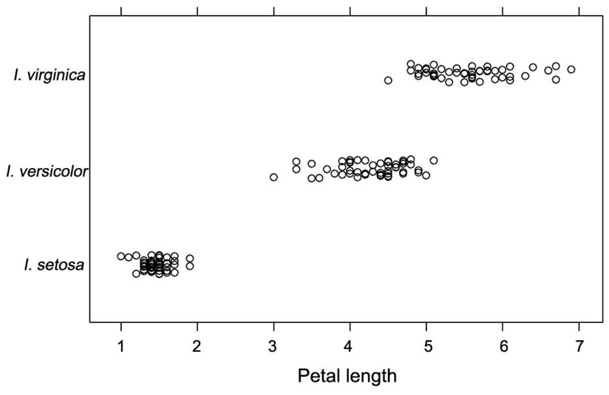

Figure 6 – Petal lengths of the three Iris species (n = 50 per species). Data source: Anderson (1935), available through the `iris` dataset in R.

Now a density plot:

```
iris_melted <- melt(iris)

w = densityplot(~ value | variable*Species, data = iris_melted,
                plot.points = F)
print(w)
```

In `plot.points = F`, `F` stands for `FALSE`; this setting means that we do not want the graph to include points (which would be plotted along the x axis), just the lines representing the density functions.

We can use the `useOuterStrips` function of the `latticeExtra` package to make the graph look a little better:

```
w1 = useOuterStrips(w)
print(w1)
```

And here are two other versions of the graph:

```
densityplot(~ value | variable, data = iris_melted,
            groups = Species, plot.points = F,
            auto.key = list(space = "right"))

densityplot(~ value | Species, data = iris_melted,
            groups = variable, plot.points = F,
            auto.key = list(space = "right"),
            layout = c(3,1), aspect = 1)
```

Here are two versions of a trellis box plot:

```
bwplot(Species ~ value | variable, data = iris_melted, layout = c(3, 1))

bwplot(variable ~ value | Species, data = iris_melted, layout = c(4, 1))
```

Another useful plot is a matrix of scatter plots, easily available via `lattice's splom` function (two versions again):

```
splom(~ iris[, 1:4] | iris$Species, layout = c(3, 1))

splom(~ iris[, 1:4], groups = iris$Species, auto.key = list(space = "right"))
```

Saving graphs in various file formats

These days, journals expect their authors to submit graphs in various file types. For vector graphs, most common are PDF and EPS (encapsulated postscript), and for raster images, TIFF and JPEG. `lattice` makes it possible to construct such files in quite an easy way, with the function `trellis.device`. It applies other functions built into R. Beginning users seldom need to know more than how to use these functions with their several main arguments.

For example, let us create the following graph:

```
levels(iris$Species) = paste("I.", levels(iris$Species))
plot1 <- xyplot(jitter(Petal.Length) ~ jitter(Petal.Width),
                data = iris, groups = Species,
                xlab = "Petal width", ylab = "Petal length",
                type = c("p", "r"), aspect = "iso",
                auto.key = list(space = "right", lines = T, font = 3))
```

To make a PDF file, you can use the following code:

```
trellis.device(device = "pdf", file = "./Figure.pdf")
print(plot1)
dev.off()
```

Remember about closing the connection by `dev.off()`. Failing to do so, you will fail to make the graph: R will be still waiting for you to either change the graph or close the file. The dot in the path represents the working directory, but you can of course use any path and any valid file name (remembering about the .pdf extension). This call will construct a PDF file with the default settings. We can change them, for instance, using arguments

- `width` and `length` (both having default values of 7 inches) to specify the graph's size
- `paper` (e.g., `paper = "a4"`) to specify the sheet's format from among several standards
- `colormodel` ("`srgb`", "`gray`" and "`cmyk`") to specify the color model

See `?pdf` for more such arguments and more information about them. Note that when you choose `colormodel = "gray"`, the graph will be made in shades of grey, so you need not worry about changing the graph's setting: `lattice` will do that for you!

As already mentioned, R enables us to create EPS files, often required by scientific journals for vector graphs. Here is an example:

```
trellis.device("postscript", file = "./Figure.eps",
               width = 6, height = 6, paper = "special", horizontal = FALSE)
print(plot1)
dev.off()
```

In both `pdf` and `postscript` devices, we use inches to set up the device's parameters. In other devices, like `tiff`, we can use also other units:

```
plot2 <- update(plot1, legend = NULL)
plot2 <- update(plot2,
                auto.key = list(space = "top",
                                lines = T, font = 3, cex = .8))
trellis.device("tiff", file = "./Figure.tiff",
               width = 7, height = 10, units = "cm", res = 300)
print(plot1)
dev.off()
```

Note the double use of `update`: since it does not enable one to change the existing legend, the first instance removes the legend while the second adds its new version.

Conclusions

We have discussed simple and more advanced data visualization with the `lattice` package of R. To create simple graphs is easy, and to create more complex ones does not have to be overly difficult, either. It may happen, however, that controlling some of the graph elements may require skills and knowledge—though similar graphs might be much more difficult to create with other software.

I do not claim that `lattice` or even `R` is the only sensible option. It depends on various factors, such as whether you already know other software that enables creating professional graphs, your abilities to learn basic programming, and so on. But if you are starting your adventure with creating professional graphs, you don't know any other professional software, and you are not afraid of spending some time on learning how to write code to create graphs—then R might be a good choice. Its popularity among researchers has been continuously increasing, not only for data visualization, but mainly for data analysis. Thus, R might occur helpful in a variety of research situations. For those who know at least some `R`, `lattice` should not pose too many problems: Its basic syntax should be quite straightforward for those who are familiar with the syntax for creating formulas (for example, in linear models).

I do not want to repeat the arguments I used in the introduction to this paper, so let me just stress that `lattice` enables one to create even advanced graphs with quite simple syntax. The package's default options are usually quite well-chosen, so beginning users do not have to worry about changing various aspects of their graphs.

As I mentioned, `ggplot2` is another option in `R`, and it can even give you more opportunities to control graphs and create atypical, complex ones. It is, however, more difficult than `lattice`, so it might require more knowledge and experience. For those who do not wish to spend too much time on learning data visualization but need to create professional graphs, `lattice` seems a good option.

Even though data visualization's is clearly important in science publishing, such tools as R's `lattice` and `ggplot2` or Python's Plotly are still seldom chosen. Researchers seem to prefer simpler tools (such as Microsoft Excel). I have noticed, however, among agricultural graduate students and young researchers a trend towards R: More and more of them have heard of it, learn it, and consider it a valuable tool for data analysis. It is also high time that also we, agricultural researchers, start considering R a valuable tool for data visualization, a tool that helps create professional scientific graphs with little effort and skill. And `lattice` seems to be one of several good choices for this.

References

Anderson, E. 1935. The irises of the Gaspe Peninsula. Bulletin of the American Iris Society 59: 2-5.

Beall, G. 1942. The transformation of data from entomological field experiments. Biometrika 29: 243–262.

Becker, R.A.; Cleveland, W.S.; Shyu, M.J. 1996. The visual design and control of trellis display. Journal of Computational and Graphical Statistics 5: 123-155.

Chambers, J.M.; Cleveland, W.S.; Kleiner, B.; Tukey, P.A. 1983. Graphical Methods for Data Analysis. Wadsworth, Monterey, CA, USA.

Cleveland, W.S. 1985. The Elements of Graphing Data. Wadsworth, Monterey, CA, USA.

Cleveland, W.S. 1993. Visualizing Data. Hobart Press, Summit, NJ, USA.

Cleveland, W.S.; McGill, M.E.; McGill, R. 1988. The shape parameter of a two-variable graph. Journal of the American Statistical Association 83: 289–300.

Fisher, R.A. 1936. The use of multiple measurements in taxonomic problems. Annals of Eugenics 7: 179-188.

Haynes, K.G.; Lambert, D.H.; Christ, B.J.; Weingartner, D.P.; Douches, D.S.; Backlund, J.E.; Fry, W.; Stevenson, W. 1998. Phenotypic stability of resistance to late blight in potato clones evaluated at eight sites in the United States. American Journal of Potato Research 75: 211-217.

Kozak, M. 2010. Basic principles of graphing data. Scientia Agricola 67: 483-494.

Kozak, M.; Łotocka, B. 2013. What should we know about the famous Iris data? Current Science 104: 579-580.

Kozak, M.; Piepho, H.P. 2018. What's normal anyway? Residual plots are more telling than significance tests when checking ANOVA assumptions. Journal of Agronomy and Crop Science 204: 86-98.

Mendiburu, F. 2017. agricolae: Statistical Procedures for Agricultural Research. R package version 1.2-8. Available at: https://CRAN.R-project.org/package=agricolae [Accessed May 6, 2019]

Mendiburu, F. 2019. agricolae: Statistical Procedures for Agricultural Research. R package version 1.3-1. Available at: https://CRAN.R-project.org/package=agricolae [Accessed May 6, 2019]

R Core Team. 2019. R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. Available at: https://www.R-project.org/ [Accessed May 6, 2019]

Sarkar, D. 2008. Lattice: Multivariate Data Visualization with R. Springer, New York, NY, USA. Sarkar, D.; Andrews, F. 2016. latticeExtra: Extra Graphical Utilities Based on Lattice. R package version 0.6-28. Available at: https://CRAN.R-project.org/package=latticeExtra [Accessed May 6, 2019]

Su, J.S. 2008. It's easy to produce chartjunk using Microsoft® Excel 2007 but hard to make good graphs. Computational Statistics and Data Analysis 52: 4594-4601.

Tufte, E.R. 1983. The Visual Display of Quantitative Information. Graphics Press, Cheshire, CT, USA.

Wickham, H. 2009. ggplot2: Elegant Graphics for Data Analysis. Springer, New York, NY, USA.

Wilkinson, L. 2006. The Grammar of Graphics. Springer Science & Business Media, Berlin, Germany.

Wnuk, A.; Dębski, K.J. 2016. Should we trust graphs' default settings in the R packages? Communications in Biometry & Crop Science 11: 114-126.