Solutions to Lab 1

Multivariate Statistics with R

This week's lab will focus on reshaping your data and breking it down using plots for better visualisation. Roll up your sleeves and let's get going!

1 reshape()

Data do not always come in the form that we would like them to. For example, when collecting data for subjects in multiple waves, it is typical to enter the data with a single row for each participant. A separate column is then created for each wave of measurement. This is referred to as wide format. Many functions in `R` however, such as the `lmer()` function (which will be introduced next week) expect data to be in long format, where measures of the same variable only take up one column, there is an additional column representing time, and observations of each subject are spread across multiple rows. Take a look at the vocabulary data discussed in the lecture (download them from LEARN and save into your current `Rproject` folder):

```
# read in the vocab data
Vocab <- read.table("vocabulary.txt", header = T)
```

They are currently in wide format:

```
head(Vocab)
```

```
##   Gender Subject Vocab.8 Vocab.9 Vocab.10 Vocab.11
## 1   Boys       1    1.75    2.60     3.76     3.68
## 2   Boys       2    0.90    2.47     2.44     3.43
## 3   Boys       3    0.80    0.93     0.40     2.27
## 4   Boys       4    2.42    4.15     4.56     4.21
## 5   Boys       5   -1.31   -1.31    -0.66    -2.22
## 6   Boys       6   -1.56    1.67     0.18     2.33
```

Each row shows the vocabulary scores of each subject in grades 8-11, represented in the columns named `Vocab.8`,`Vocab.9`, `Vocab.10`, and `Vocab.11`. When reformatting data, it is helpful if the columns in the wide format are named `variable.time` because the command to reshape data can automatically handle this naming scheme, where variable is the name of what is being measured and time is a number representing time of measurement (this could be time 0, 1, 2. . . , wave 1, 2, 3. . . , year 2003, 2004, 2005. . . , *etc.*). For the vocabulary data, the variable would be `Vocab`.

1.1 Wide to long

```
# reshape data into long format
Vocab.long <- reshape(Vocab, idvar=c('Subject'), varying = names(Vocab)[3:6],
timevar = 'Grade', direction = 'long')
```

The function reshape takes several arguments:

- `idvar` names the column(s) that uniquely identify each subject. Typically this is something like id. For the vocabulary data, this is the `Subject` column.
- `varying` names the columns that represent repeated observations of the same subject. These should use the `variable.time` format and in our example they are the `Vocab.N` columns (columns 3-6 of the data).

- `timevar` specifies what the new column representing time should be called. Here time is specified by grade, so `Grade` would make for a suitable name. The values in the newly created column will be all the numbers seen in the time part of the `variable.time` columns.
- `direction` is either `"long"` or `"wide"` and tells `reshape()` in which direction to transform the data. For the vocabulary data, the direction is `"long"` because we want to transform it from wide format into long format.

So let's see what we have here:

```
head(Vocab.long)
```

```
##     Gender Subject Grade Vocab
## 1.8   Boys       1     8  1.75
## 2.8   Boys       2     8  0.90
## 3.8   Boys       3     8  0.80
## 4.8   Boys       4     8  2.42
## 5.8   Boys       5     8 -1.31
## 6.8   Boys       6     8 -1.56
```

Notice that the *rows* of the dataframe are automatically renamed as `idvar.timevar`.

Task 1: Check that the vocabulary scores of subjects 1 and 2 are the same in the wide and long formats.

```
### ANSWER ###

Vocab[Vocab$Subject %in% c(1,2), ] # only select lines for subjects 1 and 2
```

```
##   Gender Subject Vocab.8 Vocab.9 Vocab.10 Vocab.11
## 1   Boys       1    1.75    2.60     3.76     3.68
## 2   Boys       2    0.90    2.47     2.44     3.43
```

```
# if you're unfamiliar with the very useful operator %in%,
# you can check out the help file by typing ?`%in%` into console
x <- Vocab.long[Vocab.long$Subject %in% c(1,2), ]
x[order(x$Subject), ]
```

```
##      Gender Subject Grade Vocab
## 1.8    Boys       1     8  1.75
## 1.9    Boys       1     9  2.60
## 1.10   Boys       1    10  3.76
## 1.11   Boys       1    11  3.68
## 2.8    Boys       2     8  0.90
## 2.9    Boys       2     9  2.47
## 2.10   Boys       2    10  2.44
## 2.11   Boys       2    11  3.43
```

```
# you can now inspect the scores visually, or if you want a more programatic
# check, you can do something like
y <- unlist(c(Vocab[Vocab$Subject == 1, grep("Vocab", names(Vocab))],
Vocab[Vocab$Subject == 2, grep("Vocab", names(Vocab))]))
all(x[order(x$Subject), "Vocab"] == y)
```

```
## [1] TRUE
```

```
# grep() gives you the indices of the elements of a vector that match the
# specified pattern, in our case indices of names of columns of our Vocab
# dataset that have the string "Vocab" in them
grep("Vocab", names(Vocab))
```

```
## [1] 3 4 5 6
```

```r
# next we select only the row for which Subject == 1 and only the columns
# given to us by the grep() above
Vocab[Vocab$Subject == 1, grep("Vocab", names(Vocab))]
```

```
##   Vocab.8 Vocab.9 Vocab.10 Vocab.11
## 1    1.75     2.6     3.76     3.68
```

```r
# then we put together this row and the row for Subject == 2
c(Vocab[Vocab$Subject == 1, grep("Vocab", names(Vocab))],
Vocab[Vocab$Subject == 2, grep("Vocab", names(Vocab))])
```

```
## $Vocab.8
## [1] 1.75
##
## $Vocab.9
## [1] 2.6
##
## $Vocab.10
## [1] 3.76
##
## $Vocab.11
## [1] 3.68
##
## $Vocab.8
## [1] 0.9
##
## $Vocab.9
## [1] 2.47
##
## $Vocab.10
## [1] 2.44
##
## $Vocab.11
## [1] 3.43
```

```r
# as you can see, this returns a list and so we unlist() it to turn it
# into a vector and store it in y
y <- unlist(c(Vocab[Vocab$Subject == 1, grep("Vocab", names(Vocab))],
Vocab[Vocab$Subject == 2, grep("Vocab", names(Vocab))]))

# finally, we take the x we created above, only extract the "Vocab" column,
# make sure it's sorted by subject number and ask R if these two vectors
# are all the same
all(x[order(x$Subject), "Vocab"] == y)
```

```
## [1] TRUE
```

The following three questions refer to some fertility data so a little background is in order: Countries have been going through a shift to lower fertility, referred to as the demographic transition. This change in fertility has come along with changes in economies and rising standards of living. However, is it possible that differences in climate explain differences in fertility? The data come from EarthTrends (now sadly defunct) and are in wide format. (again, get them from LEARN and put them in your current `Rproject` folder.)

```r
Fert <- read.csv("fert-trends.csv")
names(Fert)
```

```
##  [1] "ISO"            "continent"      "latitude"       "ecozone"       
##  [5] "Country"        "fertility.1995" "log_gdp.1995"   "fertility.1965"
##  [9] "log_gdp.1965"   "fertility.1980" "log_gdp.1980"   "fertility.2000"
## [13] "log_gdp.2000"   "fertility.2005" "log_gdp.2005"   "fertility.1975"
## [17] "log_gdp.1975"   "fertility.1970" "log_gdp.1970"   "fertility.1985"
## [21] "log_gdp.1985"   "fertility.1990" "log_gdp.1990"   "fertility.1960"
## [25] "log_gdp.1960"
```

Each row contains information about a country. The first 5 columns provide information that does not change over time:

- `ISO`, `Country`: ISO code and country name,
- `continent`, `latitude`: location of the country and distance from the equator,
- `ecozone`: one of 6 ecozones that divide the world in terms of ecosystems and climate.

The other columns denote repeated observations over time: `fertility.YYYY` (measured as average number of births per woman) and `log_gdp.YYYY` (natural logarithm of gross domestic product from 1960-2005 in five year intervals).

Task 2: What is the name of a column or columns that uniquely identify each group (country) in the data?

```
### ANSWER ###

# 'ISO' or 'Country' (ISO is shorter so it's more convenient to use)
# you can see it identifies groups uniquely as there is only one
# instance of each value of ISO:
table(Fert$ISO)
```

```
## 
## ARG AUT BEL BFA BRA BWA CAN CHE CHL CZE DEU DNK ESP EST ETH FRA GBR HRV 
##   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1 
## IDN IND IRN ISL ITA JPN KOR KWT LBN MAR MEX MLT MYS NGA NZL PER PHL POL 
##   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1 
## PRI PRT RUS SRB SVK SVN THA TUR UGA USA 
##   1   1   1   1   1   1   1   1   1   1
```

Task 3: What are the indices needed for `names(Fert)` that will return the names of columns that vary over time?

```
### ANSWER ###

# you are looking for variables with .YYYY in their names.
# you can either look at the colkumn names
names(Fert)
```

```
##  [1] "ISO"            "continent"      "latitude"       "ecozone"       
##  [5] "Country"        "fertility.1995" "log_gdp.1995"   "fertility.1965"
##  [9] "log_gdp.1965"   "fertility.1980" "log_gdp.1980"   "fertility.2000"
## [13] "log_gdp.2000"   "fertility.2005" "log_gdp.2005"   "fertility.1975"
## [17] "log_gdp.1975"   "fertility.1970" "log_gdp.1970"   "fertility.1985"
## [21] "log_gdp.1985"   "fertility.1990" "log_gdp.1990"   "fertility.1960"
## [25] "log_gdp.1960"
```

```
# and see that the indices are 6:25
# or, a bit more advanced, you can use grep() again along
# with some pattern matching:
grep("\\d{4}", names(Fert))
```

```
##  [1]  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

```
# \\d stands for digit so \\d{4} means we are looking for
# indices of those names of Fert that include a 4-digit number
```

Task 4: Use `reshape()` to transform the `Fert` data frame into long format; store the result in a new data frame `Fert.long`. The results should look like this:

```
### ANSWER ###

Fert.long <- reshape(Fert, varying = names(Fert)[6:25], timevar = "year",
idvar = c("ISO"), direction = "long")
head(Fert.long)
```

```
##          ISO continent latitude   ecozone      Country year fertility
## ARG.1995 ARG   AMERICA   -34.00  Neotropic    Argentina 1995      2.63
## AUT.1995 AUT    EUROPE    47.33 Palearctic      Austria 1995      1.39
## BEL.1995 BEL    EUROPE    50.83 Palearctic      Belgium 1995      1.60
## BFA.1995 BFA    AFRICA    13.00 Afrotropic Burkina Faso 1995      6.77
## BRA.1995 BRA   AMERICA   -10.00  Neotropic       Brazil 1995      2.45
## BWA.1995 BWA    AFRICA   -22.00 Afrotropic     Botswana 1995      3.70
##            log_gdp
## ARG.1995  8.910181
## AUT.1995 10.301256
## BEL.1995 10.246900
## BFA.1995  5.446737
## BRA.1995  8.467583
## BWA.1995  8.022241
```

1.2 Long to wide

Should you need it, reshape can also be used to convert wide data to long data (for example, if you are working with another piece of software, such as SPSS, that works well with wide data).

```
### ANSWER ###

# Reshape the data back to wide format
reshape(Vocab.long, idvar='Subject', v.names='Vocab', timevar='Grade', direction='wide')
```

- `idvar`: the column name that identifies each group (usually subject ID)
- `v.names`: columns of variables that change over measurement occasion or condition.
- `timevar`: name of column that tracks time or condition.

Each value of the time variable will become a separate column in the wide format.

2 Plotting small multiples

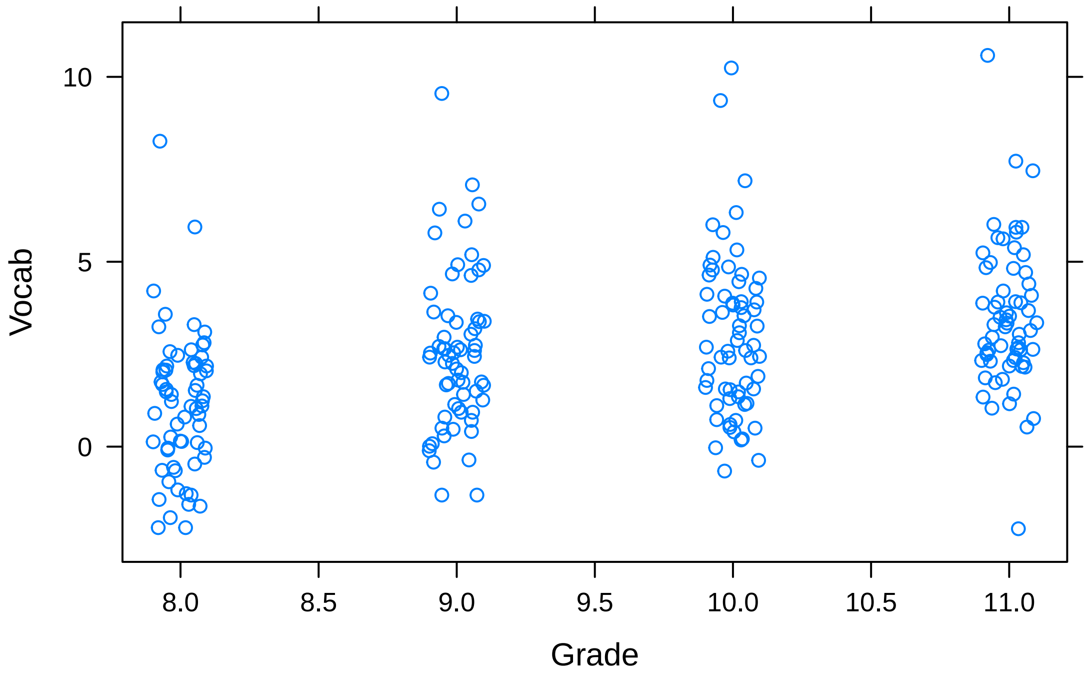
The principle of small multiples in plotting data says that lots of data can be digested if it is split into smaller pieces and then displayed side-by-side. Historically, these have also been referred to as *trellis* graphics. The `lattice` [1] package implements trellis graphics in `R`.

[1] see `?Lattice` in the help system for a high-level overview.

```r
# load lattice (comes with base R, so no need to install it)
library(lattice)
```

At its most basic, lattice works like the most of the generic plotting functions, accepting a formula for the variables to plot, but it also allows you to pass in a data frame.

```r
# Plot vocabulary data
vocab.grade.p <- xyplot(Vocab ~ Grade, data = Vocab.long, jitter.x = TRUE)
print(vocab.grade.p)
```

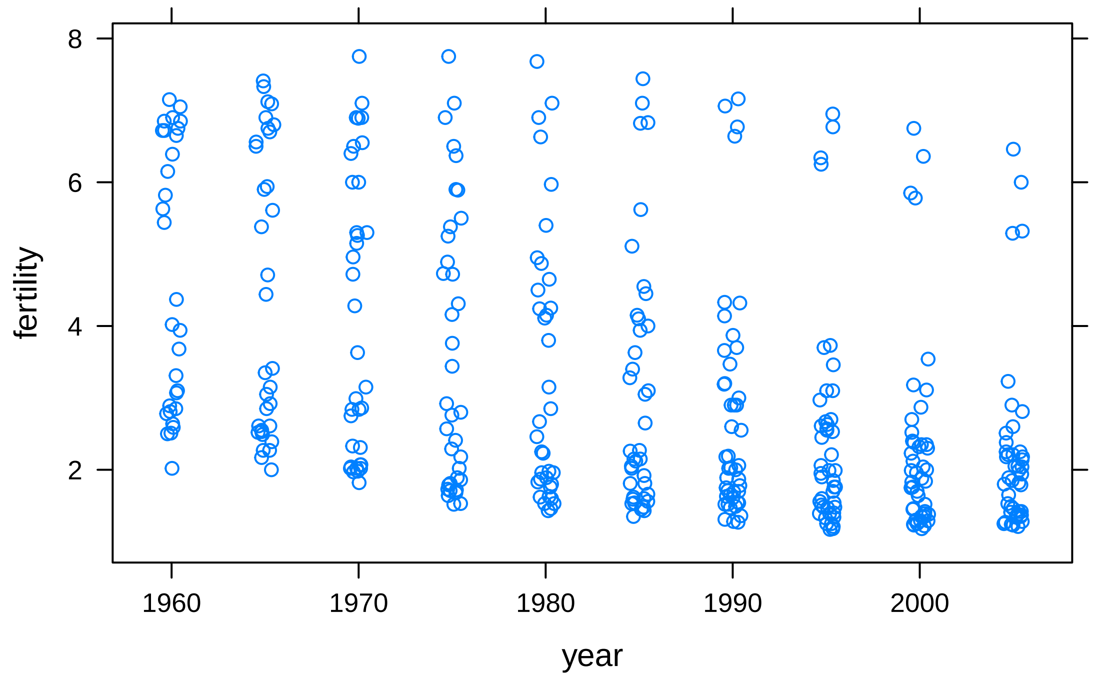


Here, the points are jittered on the x-axis (`jitter.x = TRUE`) so that they do not overlap. The output of `lattice` functions (such as `xyplot()`) can also be saved. The plot of vocabulary versus grade is stored in the `vocab.grade.p` object and then displayed on screen using the `print()` function.

Task 5: Use `xyplot` to plot all of the fertility data over time from the data frame that is in long format.

```r
### ANSWER ###

xyplot(fertility ~ year, data = Fert.long, jitter.x = TRUE)
```



2.1 Plotting data separately for each group

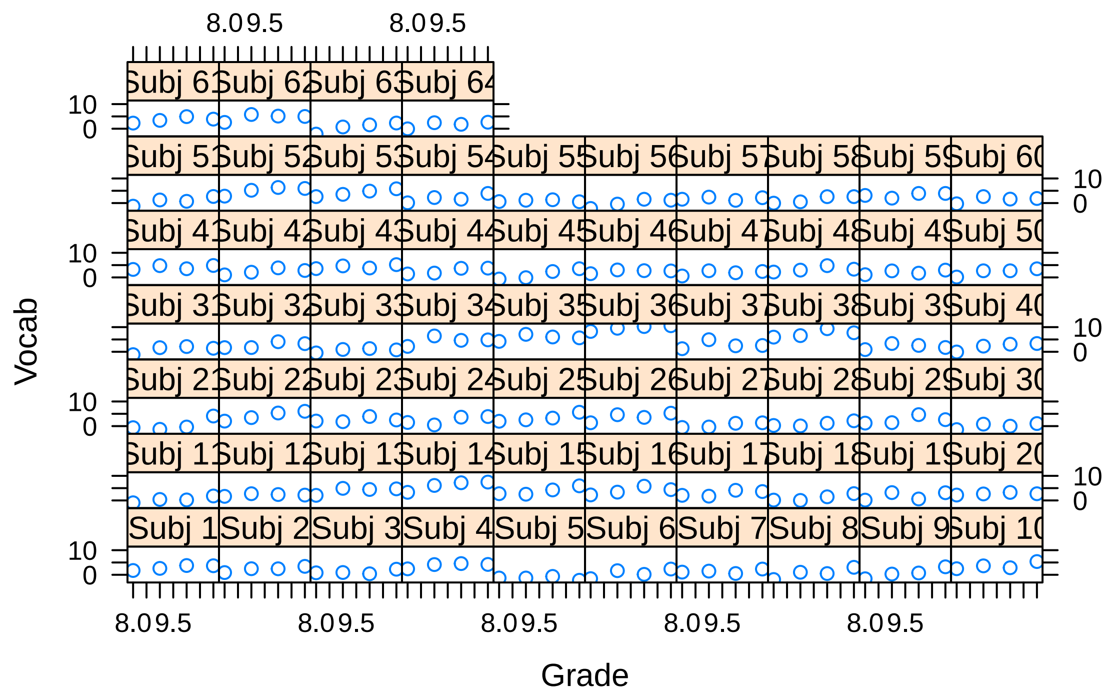
Trellis graphics allow you to print the data for each group (such as subject or country in the vocabulary and fertility data) side-by-side. This is useful for getting an overall impression of the trends in your variables of interest while clustering each observation together in a way appropriate to how the data are organised. In `lattice`, this is achieved by putting a grouping factor at the end of the plot formula, separated by a pipe (`|`) character. For example, to plot vocabulary scores separately for each student, the formula is `Vocab ~ Grade | Subject`.

```
# create meaningful panel labels
subs <- unique(Vocab$Subject)

# retrieve subject numbers
# duplicate column "Subject" and convert to factor
Vocab.long$Subject_plot <- as.factor(Vocab.long$Subject)

# Subject label
levels(Vocab.long$Subject_plot) <- paste("Subj", subs, sep = " ")

# Plot vocab data for each subject
print(xyplot(Vocab ~ Grade | Subject_plot, data = Vocab.long))
```



Note that the graph has its origin in the **lower left corner**. By default, the order of the panels is from left to right across the rows, starting with the bottom row. If the conditioning variable is a factor (`Subject_plot`), strip labels are derived from factor levels. If there are too many groups, such as the 64 students in the vocabulary study, it is useful to only display a subset of the data. `xyplot()` has a `subset = argument` that takes a logical vector (of `TRUE/FALSE` values) that specifies which rows of the data to plot. Recall from last semester that a random sample of participant IDs to plot can be created using the `sample()` function. For example, to get 5 random numbers between 1 and 10 without replacement:

```
sample(1:10, 5)
```

```
## [1] 7 9 8 4 5
```

We can get a sample of 16 random subject IDs with:

```
vocab.subjects.sample <- sample(unique(Vocab$Subject), 16)
vocab.subjects.sample
```

```
##  [1] 61 27 34 59 47  6 30 10 17 50 36 63 33 24 16  8
```

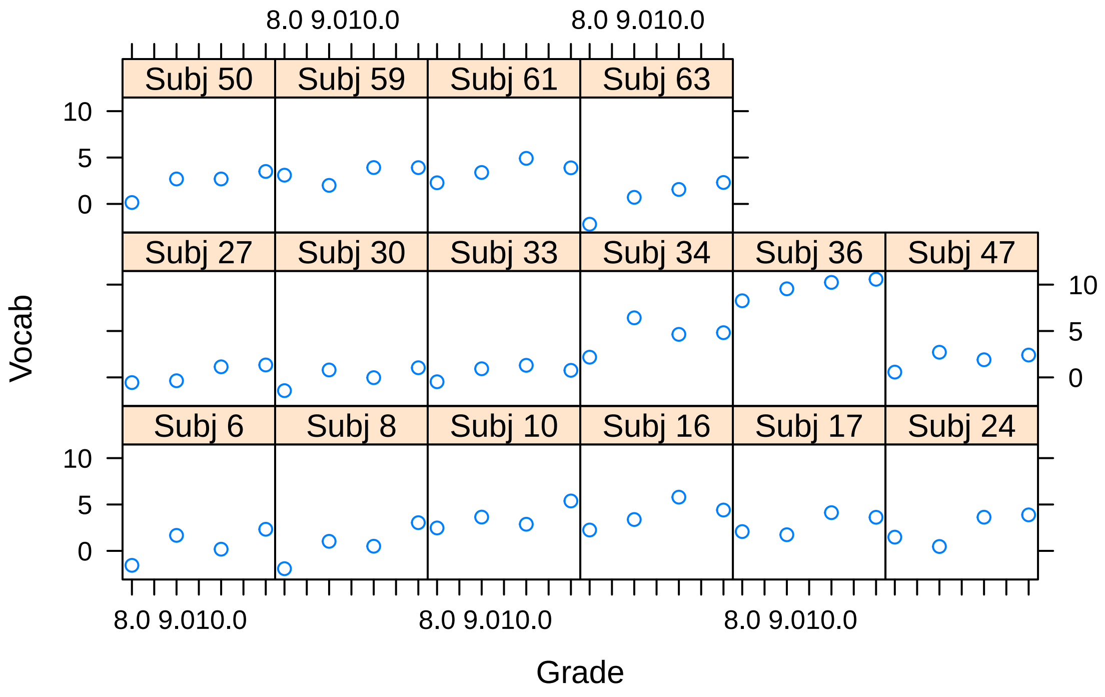
Task 6: What is the purpose of calling `unique(Vocab$Subject)`? Why would you not want to use `sample(Vocab$Subject, 16)`?

```
### ANSWER ###

# Vocab$Subject *might in principle* contain multiple records of each subject ID.
# If this were the case, sampling from it might not give you 16 different IDs because
# the sampling is random. Sampling from unique(Vocab$Subject) gets around the issue.
```

The random sample of subjects vocabulary scores can be created by getting the subset where `Subject %in% vocab.subjects` is `TRUE`:

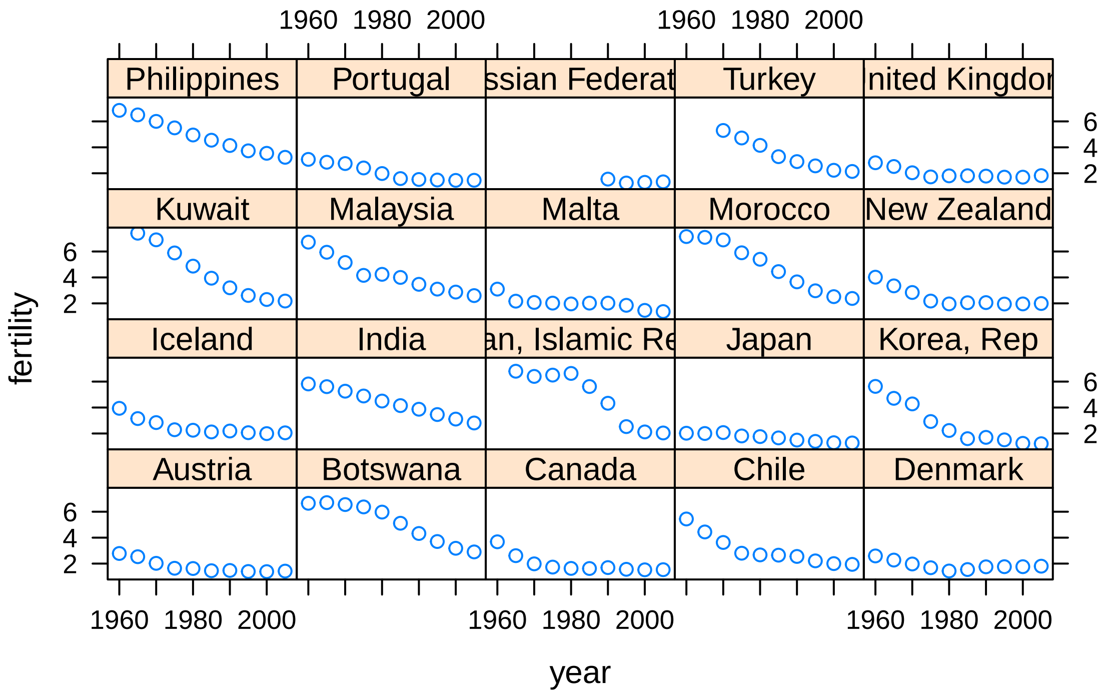
```
# Plot a random selection of 16 subjects
xyplot(Vocab ~ Grade | Subject_plot, data = Vocab.long,
       subset = Subject %in% vocab.subjects.sample)
```



Task 7: Create a plot of fertility trends in a random subset 20 of countries.

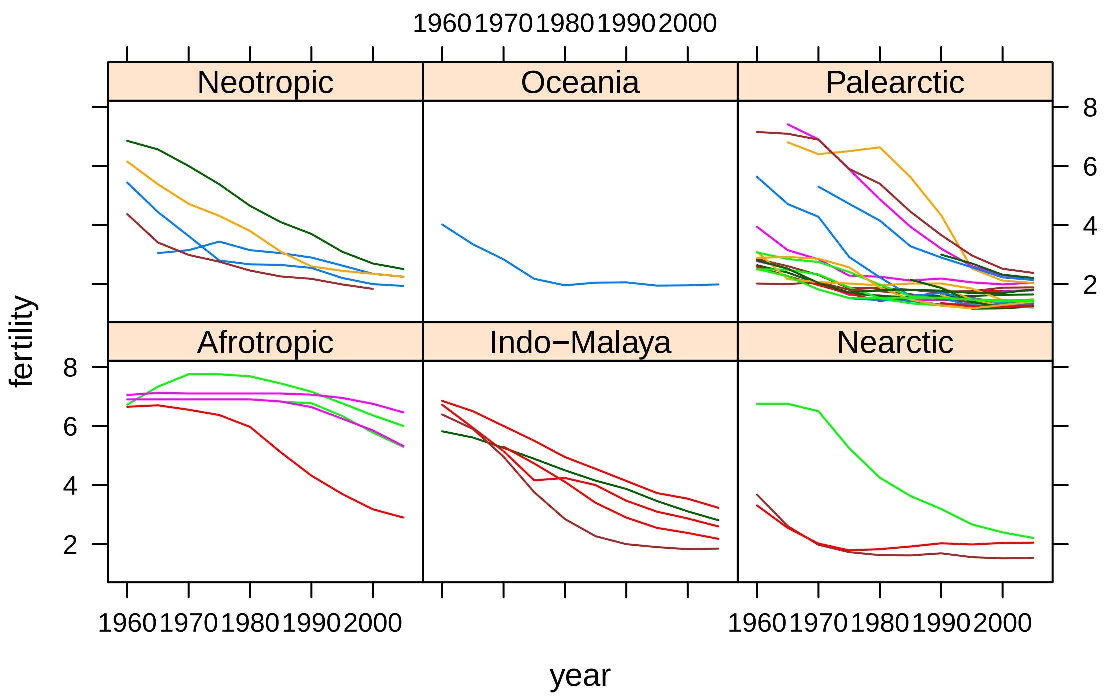
```
### ANSWER ###

xyplot(fertility ~ year | Country, data = Fert.long,
       subset = ISO %in% sample(unique(Fert$ISO), 20))
```



Plotting small multiples is also really powerful when the data can be grouped in multiple ways. For example, we can visualize fertility trends in each ecozone using the formula `fertility ~ year | ecozone`. At the same time, trends for individual countries are created using additional `groups =` and `type =` arguments. `type = 'a'` specifies average trend lines for each group.

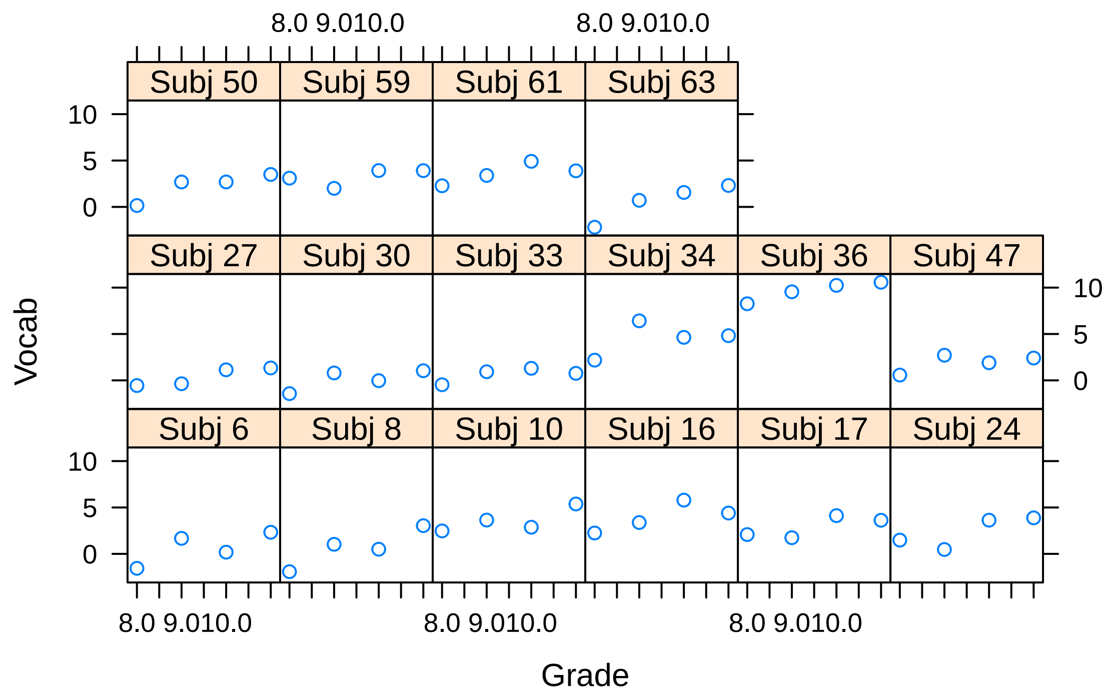
```
# Plot fertility trends by ecozone
xyplot(fertility ~ year | ecozone, groups=Country, data=Fert.long, type='a')
```



2.2 Multiple panels

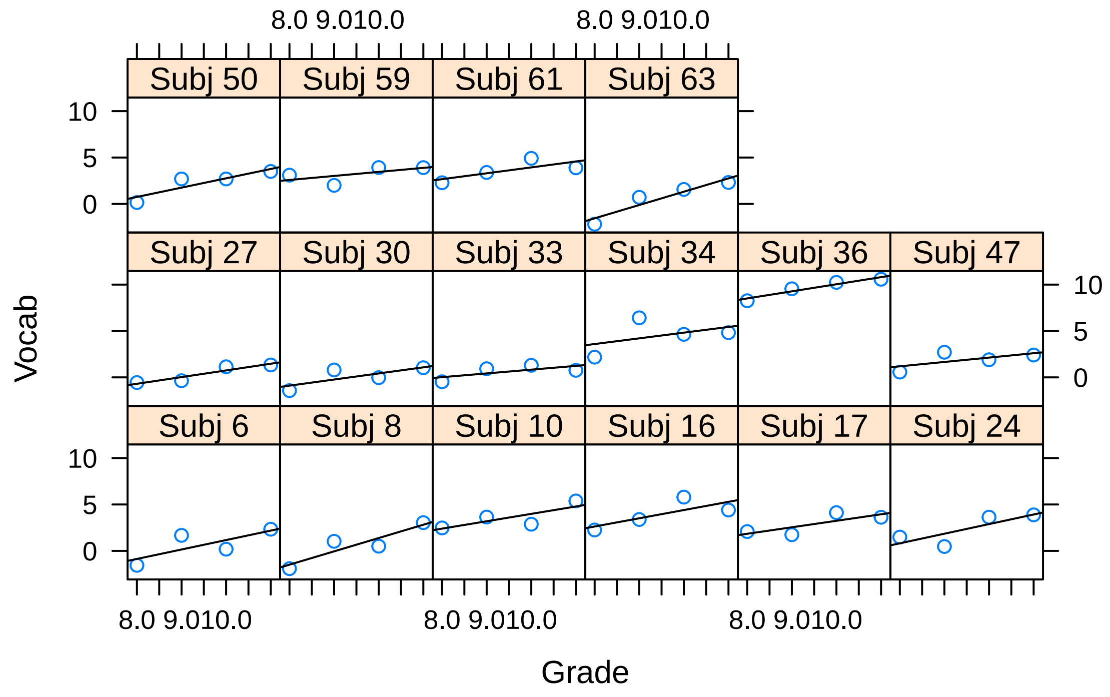
The real power of lattice over the default `plot()` functions is its ability to overlay different aspects of the data on top of each other, such as plotting points and also placing fitted regression lines over them. Each different "view" of the data is referred to as a *panel*. Panels are implemented as a function of the form `function(x, y, ...) {}`, where `x` and `y` take on the values of the variables in the right- and lefthand sides of the formula used when calling `xyplot()`. The body of the function is made up of a series of commands that then actually plot the data. To start, we will make a plot that just recreates the scatterplot that `xyplot()` creates by default from the vocabulary data.

```
# xyplot using panels
xyplot(Vocab ~ Grade | Subject_plot, data = Vocab.long,
       subset = Subject %in% vocab.subjects.sample,
       panel = function(x, y) {panel.xyplot(x, y)})
```



In this plot, the variables in the panel are automatically bound (`x` set equal to `Grade` and `y` takes on the value of `Vocab`). `panel.xyplot(x, y)` creates the scatter plot. Additional panels are added by putting other plotting commands inside the function passed to the `panel` argument. `panel.abline` takes the output from `lm()` (linear model) and plots a regression line for each group in the data.

```
# add regression lines
xyplot(Vocab ~ Grade | Subject_plot, data = Vocab.long,
       subset = Subject %in% vocab.subjects.sample,
       panel = function(x, y) {
         panel.xyplot(x, y)
         panel.abline(reg = lm(y ~ x))
       })
```



3 Exercises

The `reshape()` function can be useful even if you do not have longitudinal data. The way that the function treats time can be generalized to non-numerical measurement occasions, such as experimental condition. The data for the exercises come from a study (Nuthmann, 2013) that investigated from the size of the region around viewers' current point of gaze from which they can take in information when searching for objects in real-world scenes. Visual span size was estimated using the gaze-contingent moving window paradigm. The experiment featured six different window radii (W1 to W6) and a natural viewing control condition (W7). Download data from LEARN (visualspan-wide.csv).

```
##   subject    RT_W1    RT_W2    RT_W3    RT_W4    RT_W5    RT_W6    RT_W7
## 1       1 5916.556 3203.143 2752.900 1934.000 2762.182 2093.143 1317.538
## 2       2 4601.600 4877.800 2778.500 6669.500 2780.083 3138.833 1981.600
## 3       3 4788.667 2760.000 2293.231 1746.143 2221.692 1864.545 1100.200
## 4       4 7885.300 4276.692 2171.286 3401.333 2283.857 2364.583 1830.133
## 5       5 5666.286 3072.750 2994.167 2249.417 2094.818 1884.214 2077.308
## 6       6 7257.714 4207.917 3982.846 3383.364 2821.273 2838.077 1735.786
```

Exercise 1: Use `reshape()` to turn the visual span data into long format, with a column for condition (`window_size`) and a column that gives the reaction time.

Tip 1: If `reshape()` throws an error, check additional arguments you can supply to the function (*e.g.*, `sep`).

Tip 2: Make sure that the column representing window size is a factor after you create the long-format data frame.

```
### ANSWER ###

# reshape
Span.long <- reshape(data = Span, varying = names(Span)[2:8],
                     v.names = "reaction_time", timevar = "window_size",
                     idvar = "subject", direction = "long", sep = "_W")
head(Span.long, 15)
```

```
##      subject window_size reaction_time
## 1.1        1           1      5916.556
## 2.1        2           1      4601.600
## 3.1        3           1      4788.667
## 4.1        4           1      7885.300
## 5.1        5           1      5666.286
## 6.1        6           1      7257.714
## 7.1        7           1      7642.200
## 8.1        8           1      5623.200
## 9.1        9           1      7748.583
## 10.1      10           1      6865.000
## 11.1      11           1      5973.714
## 12.1      12           1      5264.200
## 13.1      13           1      6143.750
## 14.1      14           1      6328.333
## 15.1      15           1      6426.833
```

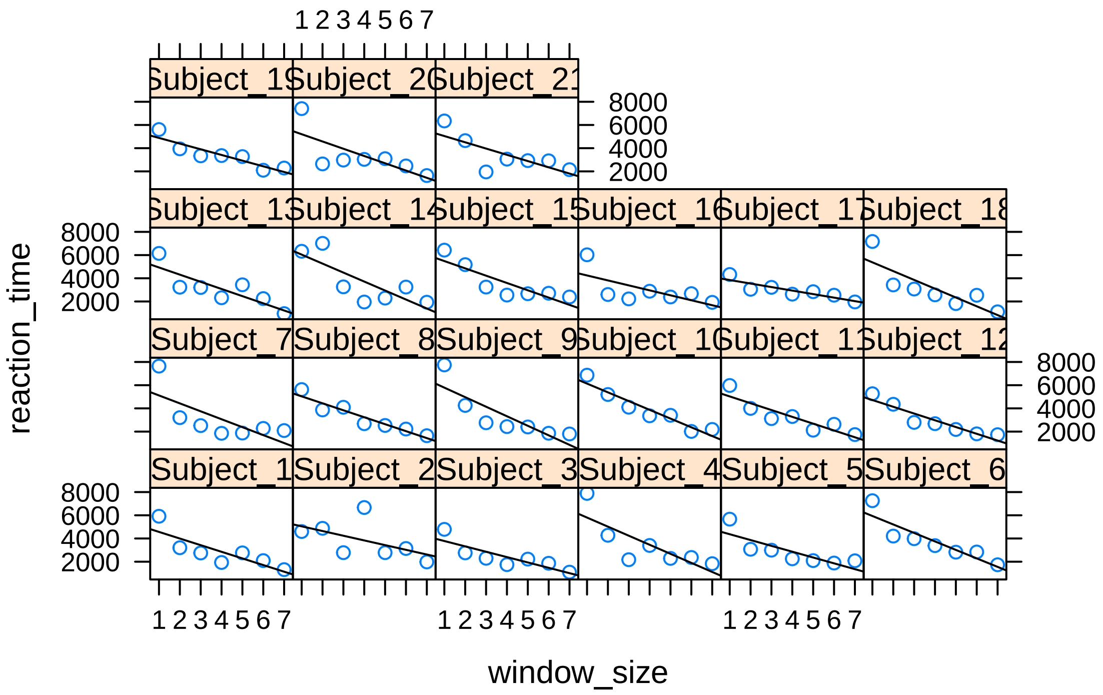
```
# Make windows size a factor
Span.long$window_size_factor <- as.factor(Span.long$window_size)
Span.long$subject <- as.factor(Span.long$subject)

# paste0 [that's a zero] is a lazy way of saying paste(..., sep = "")
levels(Span.long$subject) <- paste0("Subject_", unique(Span.long$subject))
```

Exercise 2: Create a plot that shows reaction times separately by participant.

```
### ANSWER ###

xyplot(reaction_time ~ window_size | subject, data = Span.long,
       panel=function(x, y) {
         panel.xyplot(x, y)
         panel.abline(reg = lm(y ~ x))
       })
```



reaction_time
window_size
Subject_1
Subject_2
Subject_3
Subject_4
Subject_5
Subject_6
Subject_7
Subject_8
Subject_9
Subject_1
Subject_1
Subject_12
Subject_1
Subject_14
Subject_1
Subject_16
Subject_1
Subject_18
Subject_1
Subject_2
Subject_21
2000
4000
6000
8000
1 2 3 4 5 6 7