We will be using R for the computing in this course. It is freely available for all platforms at www.r-project.org. We will also use the packages lattice, latticeExtra, nlme, reshape and possibly a few others. I'll try to cover everything you might need but if you've never used it before be prepared to take some extra time to get up to speed.

Workflow recommendations: My preferred workflow is to have a R source code file for each project I work on, and open it in R in another window; I can then record which commands I want to remember or reuse and easily run them using a keyboard shortcut (differs by platform). I never save my "workspace" when closing R; this refers to everything that is in R's memory at the time. I have found that it is easy to depend on elements that have been saved from previous sessions without realizing it or remembering how they were created. By storing all the commands I use to accomplish a task in the separate R source code file, I know I can always recreate my workspace when I later return to the project.

Loading packages: Most of the packages we will use are installed by default; however, there may be packages we will need that are not, such as the latticeExtra package. On some platforms you can install packages through menus; on all platforms you can do it using the install.packages command.

```
> install.packages("latticeExtra")
```

Getting help: Getting help is easy *if* you know the name of the command you want help for. For example, for help with the lm command, type one of these.

```
> ?lm
> help("lm")
```

If you don't know the name of the command, here are two ways to start your search. First, you canbrowse the help pages with help.start.

```
> help.start()
```

Or, to do a more general web search, start at rseek.org. This is a custom Google search which limits itself to pages with information about R.

Basic R Overview

Arithmetic, storing results as an object

```
> 1 + 1

[1] 2

> a <- 1 + 1
> b <- 3
> a + b

[1] 5
```

Basic R Object Types: boolean, character, factor, numeric, list (any can be extended); also, R is vector-Based: `c`, `[]`, `$`

```
> a <- c(FALSE, TRUE, FALSE)
> a

[1] FALSE  TRUE FALSE

> b <- c("aa", "bb", "cc")
> b

[1] "aa" "bb" "cc"

> c <- 1:3
> c

[1] 1 2 3

> d <- c(3.5, 4.6, 5.7)
> d[1]

[1] 3.5

> d[c(1, 3)]

[1] 3.5 5.7

> c + d

[1] 4.5 6.6 8.7

> my.list <- list(a = a, b = b, c = c, cd = c + d)
> my.list

$a
[1] FALSE  TRUE FALSE

$b
[1] "aa" "bb" "cc"

$c
[1] 1 2 3

$cd
[1] 4.5 6.6 8.7

> names(my.list)
```

```
[1] "a"  "b"  "c"  "cd"
```

```
> my.list$a
```

```
[1] FALSE  TRUE FALSE
```

```
> my.list["a"]
```

```
$a
[1] FALSE  TRUE FALSE
```

```
> my.list[["a"]]
```

```
[1] FALSE  TRUE FALSE
```

```
> my.list[c(1, 4)]
```

```
$a
[1] FALSE  TRUE FALSE

$cd
[1] 4.5 6.6 8.7
```

Factors

```
> aa <- c("a1", "a2", "a10")
> aaf <- factor(aa)
> aaf
```

```
[1] a1  a2  a10
Levels: a1 a10 a2
```

```
> levels(aaf)
```

```
[1] "a1"  "a10" "a2"
```

```
> levels(aaf) <- c(1, 10, 2)
> as.numeric(aaf)
```

```
[1] 1 3 2
```

```
> as.numeric(as.character(aaf))
```

```
[1]  1  2 10
```

```
> factor(aa, levels = c("a1", "a2", "a10"))
```

```
[1] a1  a2  a10
Levels: a1 a2 a10
```

```
> factor(aa, levels = c("a1", "a2", "a10"), labels = c(1, 2, 10))
```

```
[1] 1  2  10
Levels: 1 2 10
```

Data Frames lists, with each object required to have the same length

```
> df <- data.frame(a = a, b = b, c = c, cd = c + d)
> str(df)

'data.frame':	3 obs. of  4 variables:
 $ a : logi  FALSE TRUE FALSE
 $ b : Factor w/ 3 levels "aa","bb","cc": 1 2 3
 $ c : int  1 2 3
 $ cd: num  4.5 6.6 8.7

> df <- data.frame(a = a, b = b, c = c, cd = c + d, stringsAsFactors = FALSE)
> str(df)

'data.frame':	3 obs. of  4 variables:
 $ a : logi  FALSE TRUE FALSE
 $ b : chr  "aa" "bb" "cc"
 $ c : int  1 2 3
 $ cd: num  4.5 6.6 8.7

> df$a

[1] FALSE  TRUE FALSE

> df[1:2, 1:2]

      a  b
1 FALSE aa
2  TRUE bb
```

Functions R functions are also objects

```
> my.function <- function(a, b) {
+     (a + b)/2
+ }
> my.function

function (a, b)
{
    (a + b)/2
}

> my.function(3, 4)

[1] 3.5

> my.function(c(3, 13), c(4, 14))

[1]  3.5 13.5
```

Reading in data and looking at it

The most common way to read in data is with `read.table`; there are also a variants with defaults suitable for `csv` files (`read.csv`) and tab-delimited files (`read.delim`). Files can be read either locally or from a url. Here I read in the Big Mice data set referenced in our text, and check the structure, the dimensions, the first few rows, and the last few rows. This is always a good idea to make sure you've read it in correctly. Links to other data sets from this book can be found at `http://rem.ph.ucla.edu/rob/mld/data.html`.

```
> d <- read.delim("http://rem.ph.ucla.edu/rob/mld/data/tabdelimiteddata/bigmice.txt")
> str(d)

'data.frame':        735 obs. of  6 variables:
 $ group : int  1 1 1 1 1 1 1 1 1 1 ...
 $ id    : int  1 1 1 1 1 1 1 1 1 1 ...
 $ weight: int  120 NA NA 138 NA NA 258 NA NA 408 ...
 $ day   : int  0 1 2 3 4 5 6 7 8 9 ...
 $ dday  : int  0 1 2 3 4 5 6 7 8 9 ...
 $ cday  : int  -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 ...

> dim(d)

[1] 735   6

> head(d)

  group id weight day dday cday
1     1  1    120   0    0  -10
2     1  1     NA   1    1   -9
3     1  1     NA   2    2   -8
4     1  1    138   3    3   -7
5     1  1     NA   4    4   -6
6     1  1     NA   5    5   -5

> tail(d)

    group id weight day dday cday
730     4 35    913  15   15    5
731     4 35    959  16   16    6
732     4 35   1001  17   17    7
733     4 35   1002  18   18    8
734     4 35   1082  19   19    9
735     4 35   1105  20   20   10
```

Data read in using these commands are stored in data frames, which are matrices where each column represents a variable and each row an observation of those variables. Rows and columns can be accessed using the `[ ]` or `$` operators.

```
> d[1:3, 1:3]

  group id weight
1     1  1    120
2     1  1     NA
3     1  1     NA

> head(d[, 3])

[1] 120  NA  NA 138  NA  NA

> head(d$weight)

[1] 120  NA  NA 138  NA  NA

> d[2:4, ]

  group id weight day dday cday
2     1  1     NA   1    1   -9
3     1  1     NA   2    2   -8
4     1  1    138   3    3   -7
```

Note the use of the colon : to specify a range; specific values can also be specified using the `c` command (which is short for combine).

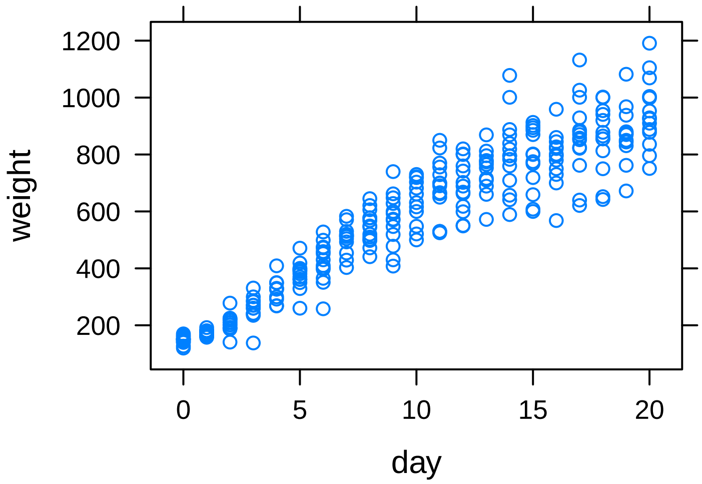
```
> d[c(2, 4, 6), ]

  group id weight day dday cday
2     1  1     NA   1    1   -9
4     1  1    138   3    3   -7
6     1  1     NA   5    5   -5
```

Basic Plotting Commands

There are at least three distinct ways to make plots in `R`; with the built-in graphics, with the `lattice` library, and with the `ggplot` library. I am most familiar with the built-in graphics and the `lattice` library, and will try to demonstrate just using the `lattice` library to keep things consistent.

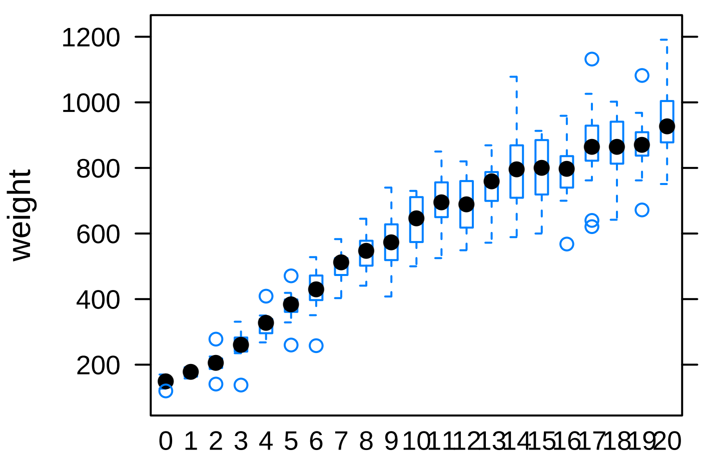
The basic plotting command is `xyplot`; it requires a formula describing which variables to plot and the name of the data frame containing those variables.

```
> library(lattice)
> p1 <- xyplot(weight ~ day, data = d)
> plot(p1)
```



To make a box and whiskers plot, use `bwplot`; we first make `day` into a categorical variable using the `factor` command.

```
> p2 <- bwplot(weight ~ factor(day), data = d)
> plot(p2)
```



Modifying plots

Labels can be easily be changed or added to plots:

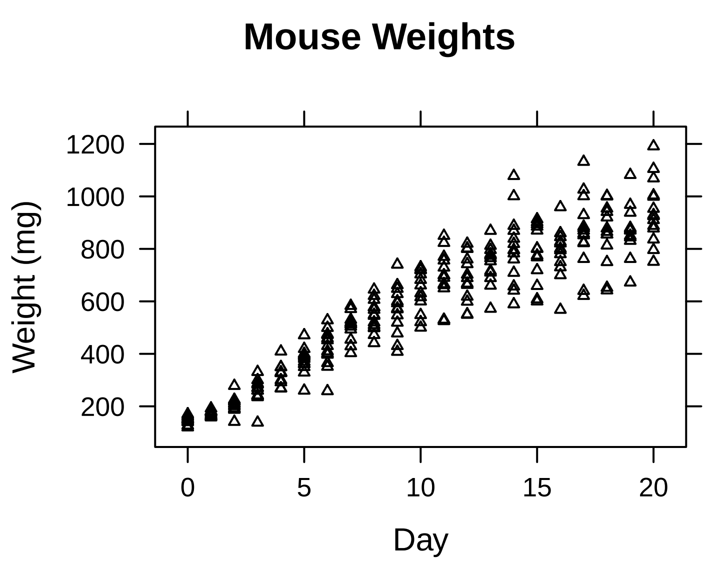
```
> p3 <- xyplot(weight ~ day, data = d, xlab = "Day", ylab = "Weight (mg)",
+     main = "Mouse Weights")
> plot(p3)
```

To change other elements of the plot, like point color, type, and size, it's best to use the theme mechanism. The following code saves the standard theme, modifies the plotting symbol to be black triangles of size 0.5.

```
> ltheme <- standard.theme("pdf")
> ltheme$plot.symbol$col <- "black"
> ltheme$plot.symbol$pch <- 2
> ltheme$plot.symbol$cex <- 0.5
```

The plot can then be updated before plotting.

```
> p3b <- update(p3, par.settings = ltheme)
> plot(p3b)
```



To see elements of the theme, you can delve into it by using the `names` function and the `$` operator.

```
> stheme <- standard.theme("pdf")
> names(stheme)
```

```
 [1] "grid.pars"         "fontsize"          "background"
 [4] "panel.background"  "clip"              "add.line"
 [7] "add.text"          "plot.polygon"      "box.dot"
[10] "box.rectangle"     "box.umbrella"      "dot.line"
[13] "dot.symbol"        "plot.line"         "plot.symbol"
[16] "reference.line"    "strip.background"  "strip.shingle"
[19] "strip.border"      "superpose.line"    "superpose.symbol"
[22] "superpose.polygon" "regions"           "shade.colors"
[25] "axis.line"         "axis.text"         "axis.components"
[28] "layout.heights"    "layout.widths"     "box.3d"
[31] "par.xlab.text"     "par.ylab.text"     "par.zlab.text"
[34] "par.main.text"     "par.sub.text"
```

```
> str(stheme$plot.symbol)
```

```
List of 6
 $ alpha: num 1
 $ cex  : num 0.8
 $ col  : chr "#0080ff"
 $ font : num 1
 $ pch  : num 1
 $ fill : chr "transparent"
```

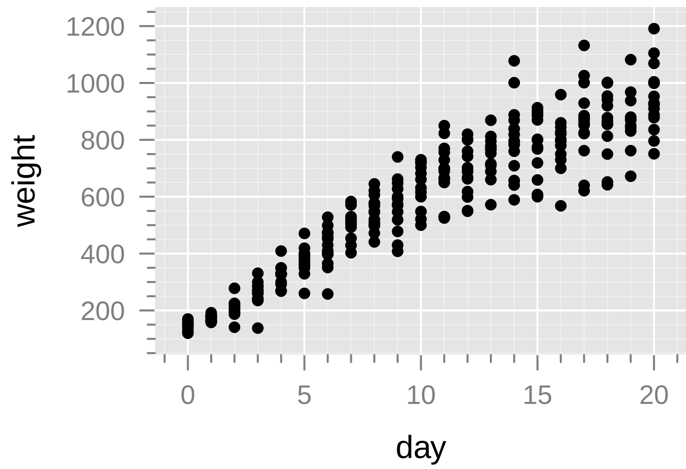
Also try the `show.settings()` function to view current settings.

A theme can also be set as the default. Then NEW plots will be made using this theme, that is, you'll need to close your currently open plots first.

```
> lattice.options(default.theme = ltheme)
```

The `latticeExtra` has several themes prepared that you may prefer, including one that is similar to the `ggplot` defaults. The theme is created with the `ggplot2like` function; some additional options are also needed and are set with `ggplot2like.opts`.

```
> library(latticeExtra)
> lattice.options(default.theme = ggplot2like())
> lattice.options(ggplot2like.opts())

> plot(xyplot(weight ~ day, d))
```



Reshaping data

Sometimes our data will not be in the format we need in order to plot and analyze it; it may be in *wide* format, instead of *long* format. The `reshape` package is a nice way to convert between the two. Here's a data set I made up:

```
> d <- read.csv("http://www.stat.umn.edu/~arendahl/Teaching/EPSY8282/class/class02.csv
> d

  subject gender     time0     time1     time2
1       1      M  9.659145  9.353645  8.439919
2       2      F 12.384359 12.082321 12.258120
3       3      M 10.244508 10.458735  9.886974
4       4      F 12.070143 12.874918 13.402598
5       5      M 14.211441 16.272405 16.508425
6       6      F 12.397092 12.595313 11.411346
```

To get the data in long format we "melt" it by distinguishing between identifier and measured variables. (The first 8 rows of the result are shown.)

```
> library(reshape)
> dm <- melt(d, id.vars = 1:2, measure.vars = 3:5)
> head(dm, 8)

  subject gender variable     value
1       1      M    time0  9.659145
2       2      F    time0 12.384359
3       3      M    time0 10.244508
4       4      F    time0 12.070143
```

```
5       5      M    time0 14.211441
6       6      F    time0 12.397092
7       1      M    time1  9.353645
8       2      F    time1 12.082321
```

We can get it back into wide format by "casting" it. This uses a formula with the variables to use as rows on the left and the variables to use as columns on the right. By default, it expects the response variable to be named `value`.

```
> cast(dm, subject + gender ~ variable)

  subject gender     time0     time1     time2
1       1      M  9.659145  9.353645  8.439919
2       2      F 12.384359 12.082321 12.258120
3       3      M 10.244508 10.458735  9.886974
4       4      F 12.070143 12.874918 13.402598
5       5      M 14.211441 16.272405 16.508425
6       6      F 12.397092 12.595313 11.411346

> cast(dm, gender + subject ~ variable)

  gender subject     time0     time1     time2
1      F       2 12.384359 12.082321 12.258120
2      F       4 12.070143 12.874918 13.402598
3      F       6 12.397092 12.595313 11.411346
4      M       1  9.659145  9.353645  8.439919
5      M       3 10.244508 10.458735  9.886974
6      M       5 14.211441 16.272405 16.508425

> cast(dm, variable ~ gender + subject)

  variable      F_2      F_4      F_6      M_1       M_3      M_5
1    time0 12.38436 12.07014 12.39709 9.659145 10.244508 14.21144
2    time1 12.08232 12.87492 12.59531 9.353645 10.458735 16.27240
3    time2 12.25812 13.40260 11.41135 8.439919  9.886974 16.50842
```

It's easy to aggregate over certain variables by simply not including them in the formula. A function must be specified to use for the aggregation.

```
> cast(dm, gender ~ variable, fun.aggregate = mean)

  gender    time0    time1    time2
1      F 12.28386 12.51752 12.35735
2      M 11.37170 12.02826 11.61177

> cast(dm, variable ~ gender, fun.aggregate = mean)
```

```
  variable        F        M
1    time0 12.28386 11.37170
2    time1 12.51752 12.02826
3    time2 12.35735 11.61177
```

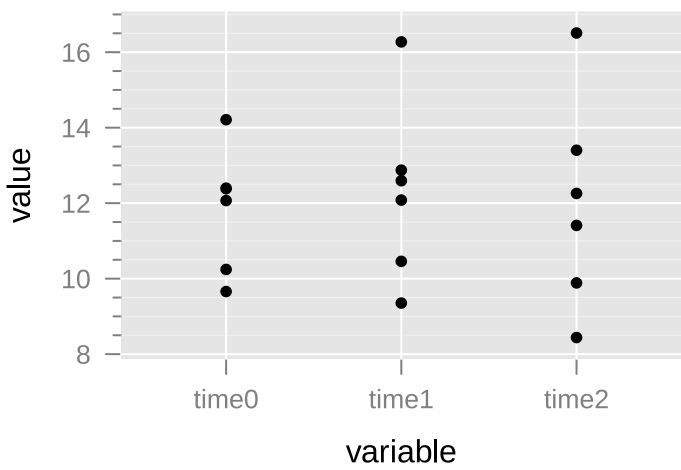
```
> cast(dm, subject + gender ~ ., fun.aggregate = mean)
```

```
  subject gender     (all)
1       1      M  9.150903
2       2      F 12.241600
3       3      M 10.196739
4       4      F 12.782553
5       5      M 15.664090
6       6      F 12.134583
```

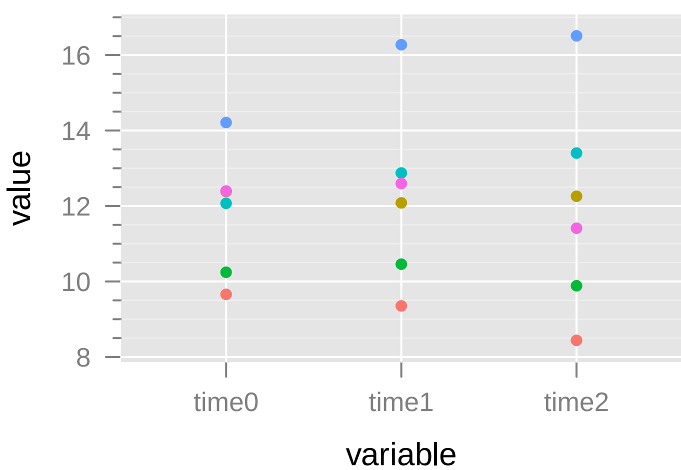
More on plotting

I'll use this made-up data set to briefly outline three of the things the `lattice` library makes easy; adding lines to a plot, dividing a plot by a given variable, and displaying points similarly within a plot based on a given variable.

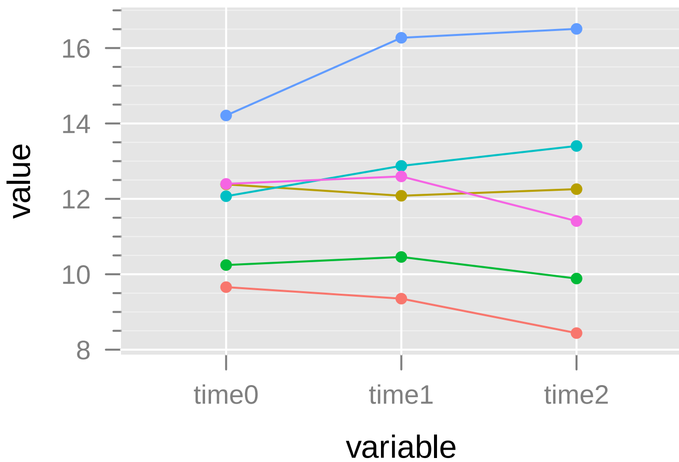
```
> dm$subject <- factor(dm$subject)
> plot(xyplot(value ~ variable, data = dm))
```



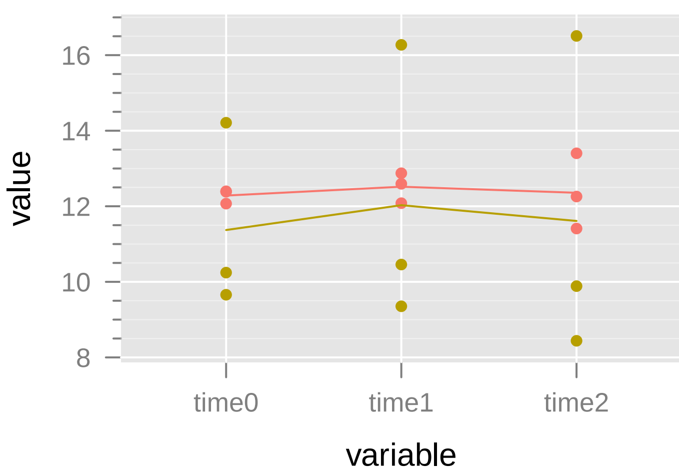

```
> plot(xyplot(value ~ variable, group = subject, data = dm))
```



```
> plot(xyplot(value ~ variable, group = subject, data = dm, type = c("p",
+     "l")))
```

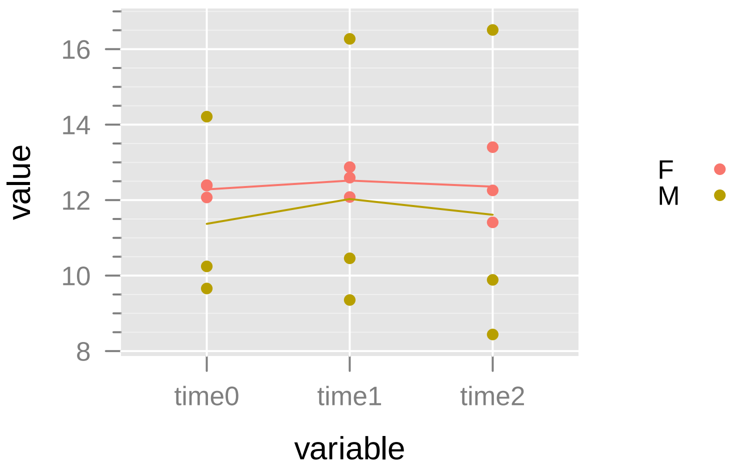



```
> plot(xyplot(value ~ variable, group = gender, data = dm, type = c("p",
+     "a")))
```

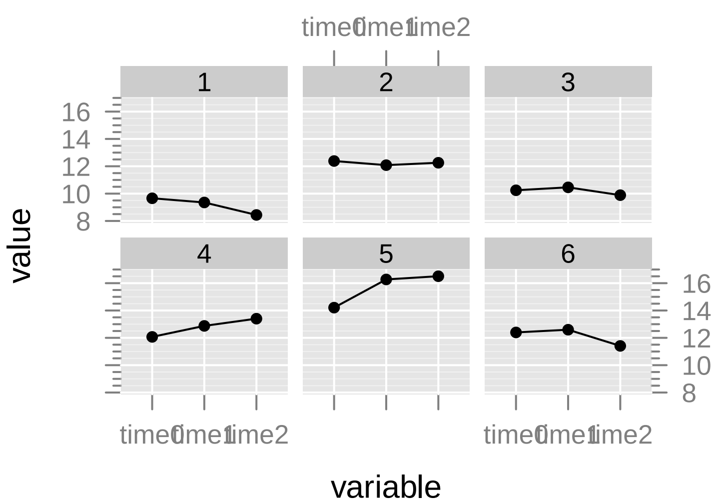



```
> plot(xyplot(value ~ variable, group = gender, data = dm, type = c("p",
+     "smooth")))
```

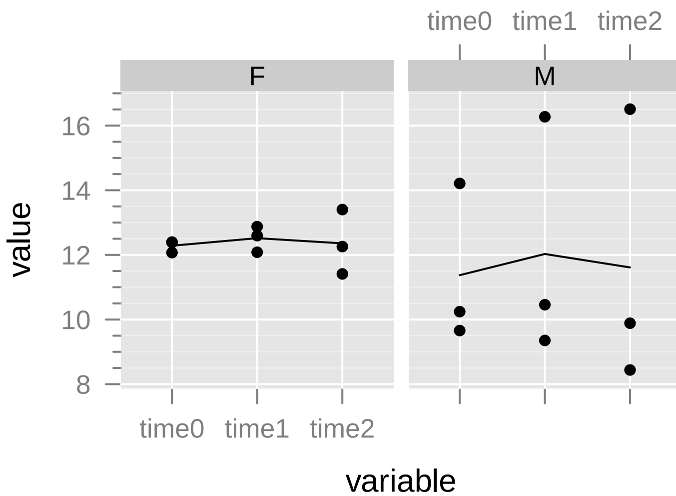
```
> px <- xyplot(value ~ variable, group = gender, data = dm, type = c("p",
+     "a"))
> px1 <- update(px, auto.key = TRUE)

> px1 <- update(px, auto.key = list(space = "right"))
> plot(px1)
```




```
> px3 <- xyplot(value ~ variable | subject, dm, type = c("p", "l"))

> px3 <- xyplot(value ~ variable | subject, dm, type = c("p", "l"),
+     as.table = TRUE)
> plot(px3)
```



```
> px4 <- xyplot(value ~ variable | gender, type = c("p", "a"),
+     dm)
> plot(px4)
```




```
> px5 <- xyplot(value ~ variable | gender, group = subject, dm,
+     type = c("p", "l"))
> plot(px5)
```