Displaying model fits in Lattice plots

Deepayan Sarkar

The `lattice` add-on package for R is an implementation of Trellis graphics (originally developed for S and S-PLUS). It is a powerful and elegant high-level data visualization system, with an emphasis on multivariate data, that is sufficient for typical graphics needs, and is also flexible enough to handle most nonstandard requirements.

This article discusses a situation many of us often find ourselves in, where we want to augment a raw data plot with a model fit. We restrict our attention to regression models, that is, models where the response variable is continuous. We assume that the reader has a basic familiarity with model-fitting in R (including the formula-based modeling language) and the use of `summary()`, `fitted()`, `predict()`, and related methods. We also assume basic familiarity with `lattice`.

We use two datasets for our examples. The first one is the `Oxboys` dataset from the `nlme` package, which records the growth (height) over time of 26 boys from Oxford. Each boy had his height measured on 9 occasions.

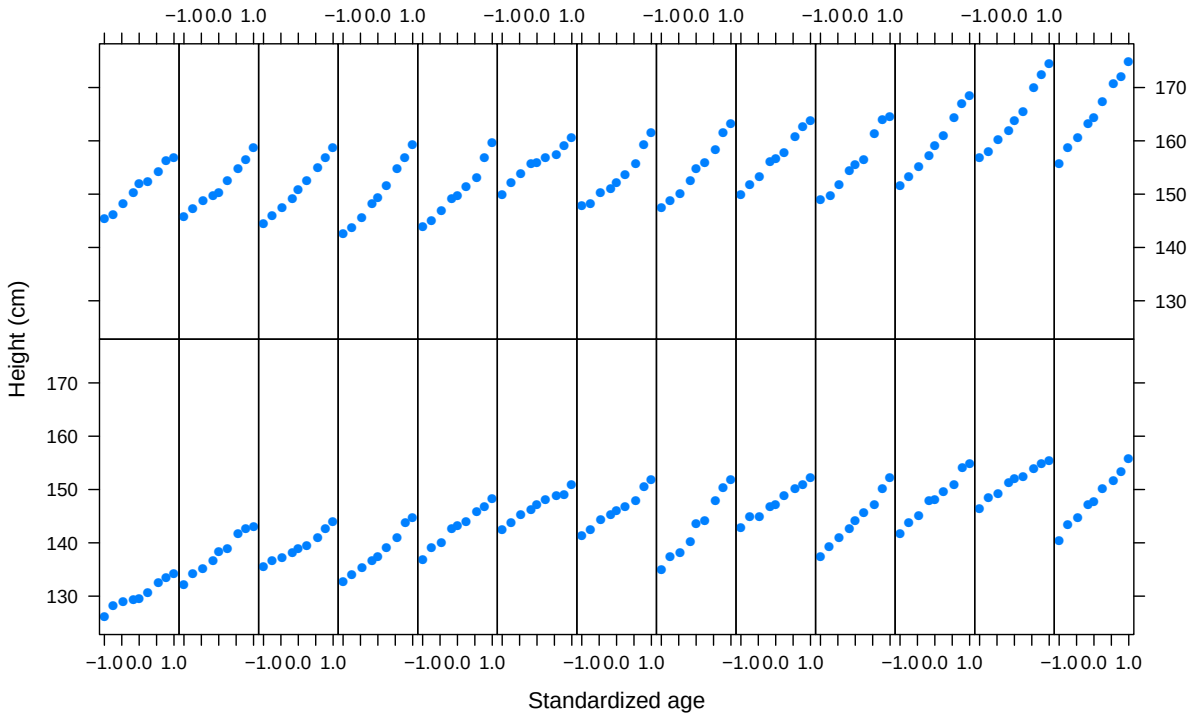
```
> data(Oxboys, package = "nlme")
> head(Oxboys, 20)
```

```
   Subject     age height Occasion
1        1 -1.0000  140.5        1
2        1 -0.7479  143.4        2
3        1 -0.4630  144.8        3
4        1 -0.1643  147.1        4
5        1 -0.0027  147.7        5
6        1  0.2466  150.2        6
7        1  0.5562  151.7        7
8        1  0.7781  153.3        8
9        1  0.9945  155.8        9
10       2 -1.0000  136.9        1
11       2 -0.7479  139.1        2
12       2 -0.4630  140.1        3
13       2 -0.1643  142.6        4
14       2 -0.0027  143.2        5
15       2  0.2466  144.0        6
16       2  0.5562  145.8        7
17       2  0.7781  146.8        8
18       2  0.9945  148.3        9
19       3 -1.0000  150.0        1
20       3 -0.7479  152.1        2
```

A simple Trellis plot of the data can be obtained by

```
> xyplot(height ~ age | Subject, data = Oxboys,
         strip = FALSE, aspect = "xy", pch = 16,
         xlab = "Standardized age", ylab = "Height (cm)")
```

Each panel represents one subject. As the subject identifiers are uninformative, the usual strips on top of each panel has been omitted. Our objective will be to fit growth curves (possibly nonlinear) to the data and superpose them on the plot.



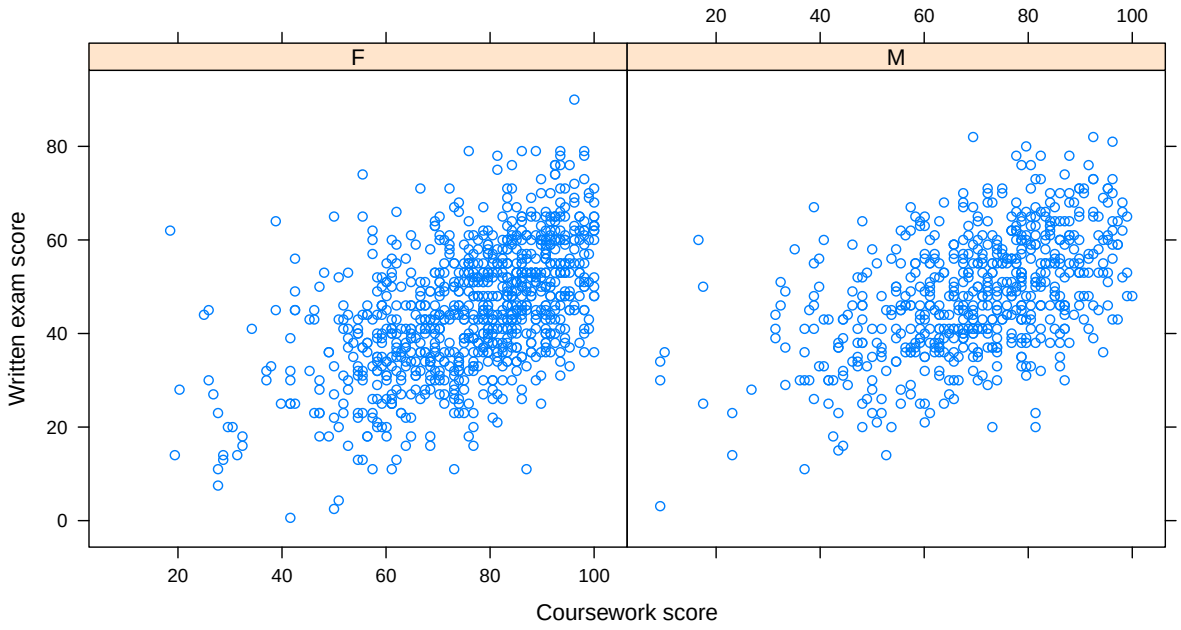
The second example is the `Gcsemv` dataset from the `mlmRev` package. This dataset records the GCSE exam scores in a science subject for 1905 students. The marks in two components (course work and written paper) are recorded separately. We are interested in modeling the expected written score based on course work score, taking into account the gender of the student.

```
> data(Gcsemv, package = "mlmRev")
> head(Gcsemv)
```

```
  school student gender written course
1  20920      16      M      23     NA
2  20920      25      F      NA   71.2
3  20920      27      F      39   76.8
4  20920      31      F      36   87.9
5  20920      42      M      16   44.4
6  20920      62      F      36     NA
```

A plot of the raw data is produced by

```
> xyplot(written ~ course | gender, data = Gcsemv,
         xlab = "Coursework score",
         ylab = "Written exam score")
```



Adding to a Lattice display

In both examples, we wish to add to the basic plot of the raw data. In traditional R graphics, one usually adds components to a plot incrementally, using "low-level" functions such as `lines()`. The analogue in Lattice graphics is to write *panel functions*.

Understanding panel functions

The idea of panel functions often intimidate beginners, but it is actually quite simple. As the name suggests, panel functions are simply R functions (!). They play a central role in `lattice` because they are responsible for actually drawing the graphical content of panels. Each `lattice` plot has a panel function. This function gets executed every time a panel is drawn, with the data specific to that panel supplied as arguments. As the input data is different for each panel, so is the result.

Every high level function has a default panel function, e.g., `xyplot()` has default panel function `panel.xyplot()`. When creating a `lattice` plot, one can replace this default by one's own choice by specifying a `panel` argument. The default panel function is of course a valid (though uninteresting) choice, and thus, the code that produced the GCSE score plot above is equivalent to

```
> xyplot(written ~ course | gender, data = Gcsemv,
         xlab = "Coursework score",
         ylab = "Written exam score",
         panel = panel.xyplot)
```

Here "`panel.xyplot`" is a predefined function. But creating new functions is not at all difficult in R. Here we explicitly define a new inline function, which simply calls `panel.xyplot()` with exactly the arguments given to it.[1]

```
> xyplot(written ~ course | gender, data = Gcsemv,
         xlab = "Coursework score",
         ylab = "Written exam score",
         panel = function(...) {
             panel.xyplot(...)
         })
```

Even if it is not completely clear what is going on here, it should not be difficult to believe (or to check) that while things look a little more complicated, the results remain unchanged.

Now, as we remarked earlier, panels look different because the data that goes into the panel function each time it is executed is different. To do anything interesting, we need to get access to this data. The specific arguments that are used to pass data to the panel function depends on the specific high-level `lattice` function being used, which in this case is `xyplot()`. A look at the help page for `panel.xyplot()` tells us that these arguments are `x` and `y`. Thus, another equivalent way to create the plot above is

```
> xyplot(written ~ course | gender, data = Gcsemv,
         xlab = "Coursework score",
         ylab = "Written exam score",
         panel = function(x, y, ...) {
             panel.xyplot(x, y, ...)
         })
```

Here, we finally have access to the panel-specific data, although we still have not done anything interesting with it. Although the sequence of calls so far produce identical results, it is important to understand the concepts that have been introduced to appreciate what follows.

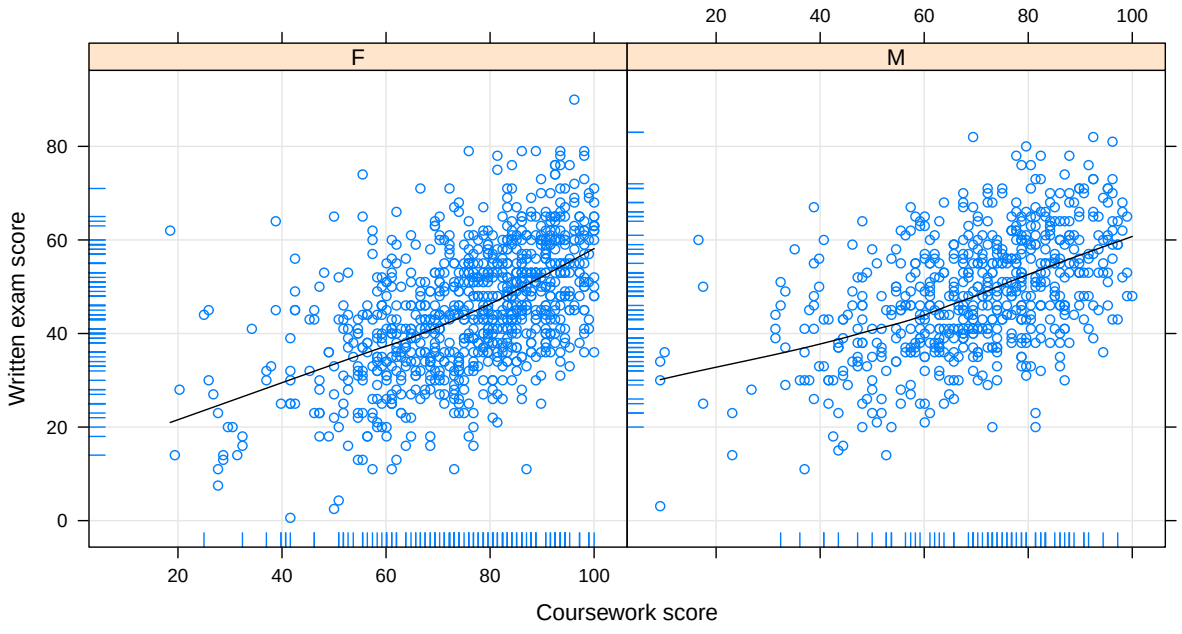
[1] The mysterious `...` requires some explanation, which we will get to soon.

A nontrivial panel function

Now that we have access to the data inside the inline panel function we are defining, we can use it to add further elements to the plot. In the following example, we add several elements: a reference grid (which does not really depend on the data), a loess fit, and marginal "rugs" indicating cases where one (but not both) of the score components are missing.

```
> xyplot(written ~ course | gender, data = Gcsemv,
         xlab = "Coursework score",
         ylab = "Written exam score",
         panel = function(x, y, ...) {
             panel.grid(h = -1, v = -1)
             panel.xyplot(x, y, ...)
             panel.loess(x, y, ..., col = "black")
             panel.rug(x = x[is.na(y)],
                       y = y[is.na(x)])
         })
```

Here the individual elements are added by the component functions `panel.grid()`, `panel.loess()`, `panel.rug()`, etc., which all produce some form of graphical output, based on arguments supplied to it. Together they define a *procedure* for plotting the data in a panel, encapsulated in the "panel function". Notice that we also needed to include a call to `panel.xyplot()`, as without it the raw data would not have been plotted. Also notice that `panel.grid()` is called *before* it, but `panel.loess()` *after*, so that the grid is rendered *below* the points but the loess fit *above*. Having the panel function completely control the display allows this kind of fine control.



Passing arguments through the ... argument

We have used a `...` construct in all the panel functions defined above; it is now time to understand how it works. The idea is fairly intuitive. Functions normally have zero, one, or more named arguments. It can also optionally have a special `...` argument. When such a function is called, they can be supplied further arguments not matching the named arguments. These arguments can then be passed on to other functions called by it, where it may match a named argument. The first explicit use of an inline function above provides an example of this:

```
panel = function(...) {
    panel.xyplot(...)
}
```

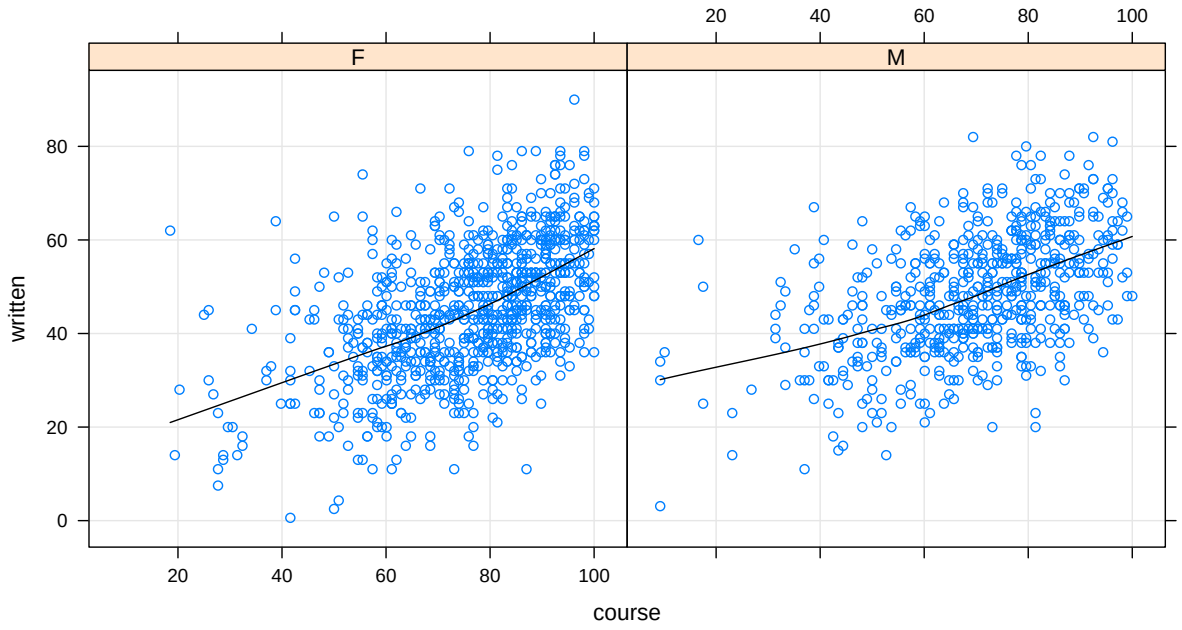
The `panel` function is called with arguments named `x` and `y`. Although `panel` itself does not recognize these names, it will dutifully pass them on to `panel.xyplot()`, which *does* recognize them.

Using optional features of predefined panel functions

`panel.xyplot()` has only two compulsory arguments (`x` and `y`), but it has quite a few optional arguments (with appropriate default values) which can modify its behaviour in various ways. In particular, two of its arguments, `grid` and `type`, can be used to make it include a background grid and a fitted loess smooth respectively (see `help(panel.xyplot)` for details). For example, we may write

```
> xyplot(written ~ course | gender, data = Gcsemv,
         panel = function(x, y) {
             panel.xyplot(x, y, grid = TRUE,
                          type = c("p", "smooth"),
                          col.line = "black")
         })
```

to produce the plot below.



Simplifying the call

A very useful fact is that the previous call is equivalent to

```
> xyplot(written ~ course | gender, data = Gcsemv,
         grid = TRUE, type = c("p", "smooth"), col.line = "black",
         panel = function(x, y, ...) {
             panel.xyplot(x, y, ...)
         })
```

This is a consequence of how `xyplot()` itself is designed. It has a `...` argument, which allows arbitrary additional arguments to be supplied to it. Arguments that are not recognized by `xyplot()` are passed on to the panel function.

Now notice how the panel function in the last plot is similar to our initial panel function examples that did nothing special. Following our earlier steps in reverse, we now see that the above is equivalent to

```
> xyplot(written ~ course | gender, data = Gcsemv, grid = TRUE,
         type = c("p", "smooth"), col.line = "black",
         panel = panel.xyplot)
```

and hence also equivalent to

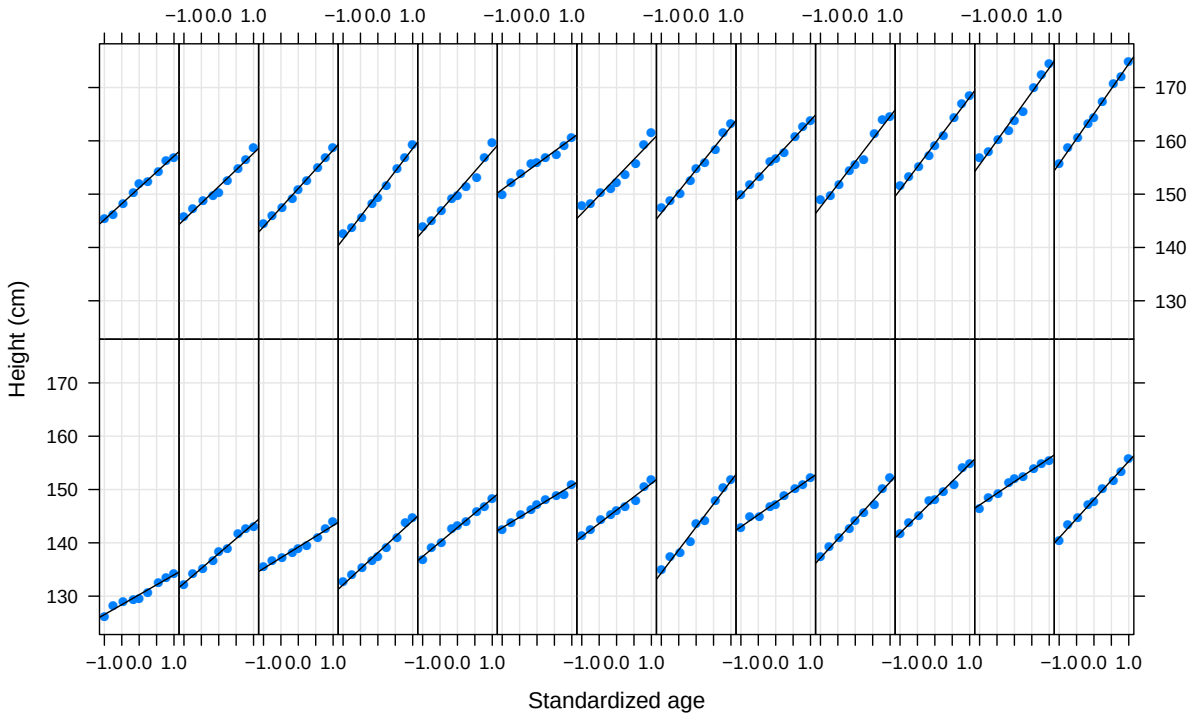
```
> xyplot(written ~ course | gender, data = Gcsemv, grid = TRUE,
         type = c("p", "smooth"), col.line = "black")
```

The end result is thus produced by a call that looks quite simple, and is quite close to the plot produced using the complicated panel function above (except for the rugs). Of course, this approach only works for features already supported by the default panel function, and requires knowledge of what features are available. For example, rugs are not supported by `panel.xyplot()`, and thus require an explicit panel function. Still, most of the default panel functions (named as "`panel.`" followed by the high-level function name) *do* have optional arguments that implement the most common variants, making this a quite useful approach.

Back to regression lines

Returning to our original question of how to add model fits, let us now consider the `Oxboys` dataset. We can of course add a loess smooth to each panel as before, but we wish to stick to parametric models for the remainder of this discussion. The help page for `panel.xyplot()` tells us that `type="r"` will add a linear regression line, so we can do

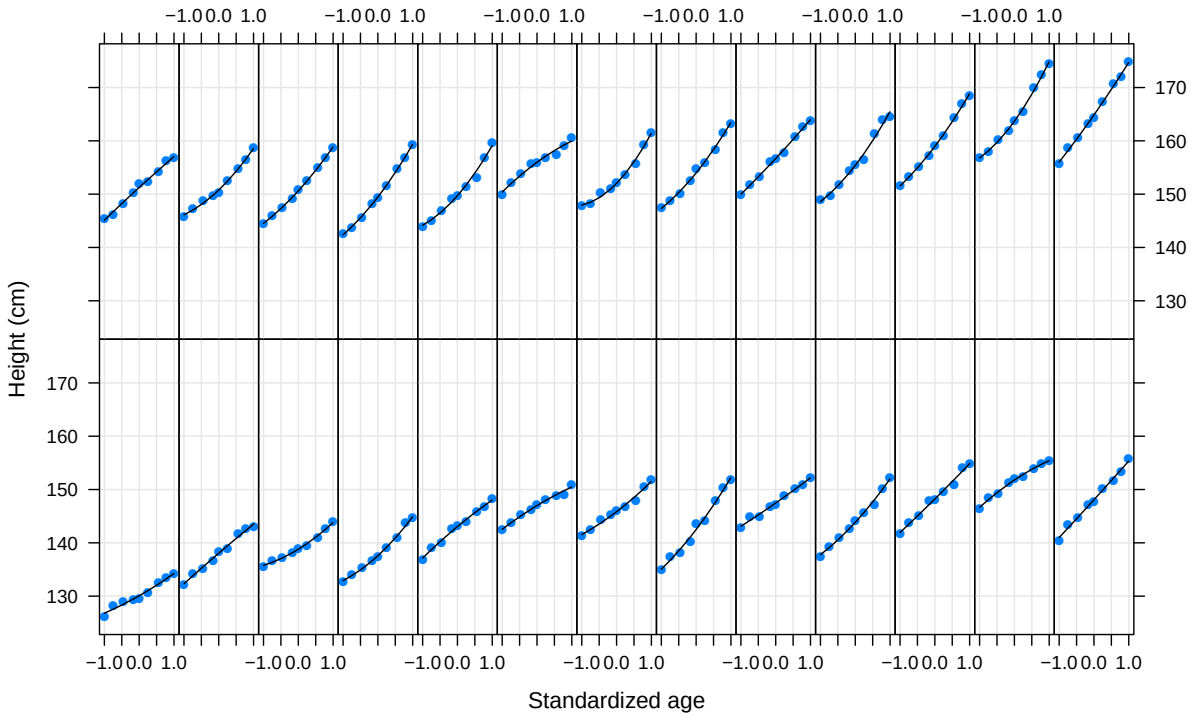
```
> xyplot(height ~ age | Subject, data = Oxboys, strip = FALSE,
         aspect = "xy", pch = 16, col.line = "black",
         grid = TRUE, type = c("p", "r"),
         xlab = "Standardized age", ylab = "Height (cm)")
```



This does not seem entirely appropriate though, as we expect growth curves to be nonlinear. Without thinking too much about *what* kind of nonlinearity would be appropriate, let us start out with a simple quadratic model. We can no longer get away without a panel function, but we can use what we have already learned to come up with

```
> xyplot(height ~ age | Subject, data = Oxboys, strip = FALSE,
         aspect = "xy", pch = 16, grid = TRUE,
         panel = function(x, y, ...) {
             panel.xyplot(x, y, ...)
             fm <- lm(y ~ poly(x, 2))
             panel.lines(x, fitted(fm), col.line = "black")
         },
         xlab = "Standardized age", ylab = "Height (cm)")
```

The specific model used here is not important, and we can replace the call to `lm()` inside the panel function with any other suitable modeling function, provided of course that the model fit uses only the data for that panel. The `panel.lines()` call draws lines joining its arguments (in the order provided), producing a reasonable approximation of the quadratic curve that we actually want to represent.



Moving from panel-specific to more general models

The approach taken so far is fundamentally limited in the sense that it can only handle models that are fitted using the within-panel data alone, because that is the only data available to the panel function. In practice, we are often interested in more complex approaches that model the full data. To use such models in a `lattice` plot, it is simplest to fit the model separately, *before* attempting to create the plot.

Again, the particular model used is not important for our purposes. For illustration, we will use a mixed effect model that generalizes the quadratic model used above, with common linear and quadratic coefficients, and a subject-specific random intercept. Formally, the model is given by

$$y_{ij} = \alpha_0 + b_i + \alpha_1 x_{ij} + \alpha_2 x_{ij}^2 + \varepsilon_{ij}$$

where i indexes subjects, j indexes repeated measurements (occasion) of a subject, x_{ij} denotes (standardized) age, and y_{ij} denotes height. The error terms b_i and ε_{ij} are assumed to be independent, with $b_i \sim N(0, \tau^2)$ and $\varepsilon_{ij} \sim N(0, \sigma^2)$. The parameters in the model are the coefficients α_i and the variance terms τ^2 and σ^2. The model can be fit using the `lme4` package as follows.

```
> library(lme4)
> fm.mixed <- lmer(height ~ age + I(age^2) + (1 | Subject), data = Oxboys)
```

We might now proceed to view the estimates of the parameters and related numerical quantities; for example, using

```
> summary(fm.mixed)

Linear mixed model fit by REML
Formula: height ~ age + I(age^2) + (1 | Subject)
   Data: Oxboys
   AIC   BIC logLik deviance REMLdev
 940.7 957.9 -465.3      930   930.7
Random effects:
 Groups   Name        Variance Std.Dev.
 Subject  (Intercept) 65.570   8.0975
 Residual              1.641   1.2810
Number of obs: 234, groups: Subject, 26

Fixed effects:
            Estimate Std. Error t value
(Intercept) 149.0617     1.5930   93.57
age           6.5152     0.1295   50.29
I(age^2)      0.7412     0.2264    3.27

Correlation of Fixed Effects:
         (Intr) age
age      -0.001
I(age^2) -0.059 -0.020
```

However, our goal here is simply to visually incorporate the fitted model in the plot, and all we need for that are the (closely related) `fitted()` and `predict()` methods for the particular modeling function used. The `fitted()` function returns the fitted values for the *same* data that have been used to fit the model. The `predict()` function can be used to obtain the predicted values for a *new* set of inputs. If no new data is provided, `predict()` essentially behaves like `fitted()`.

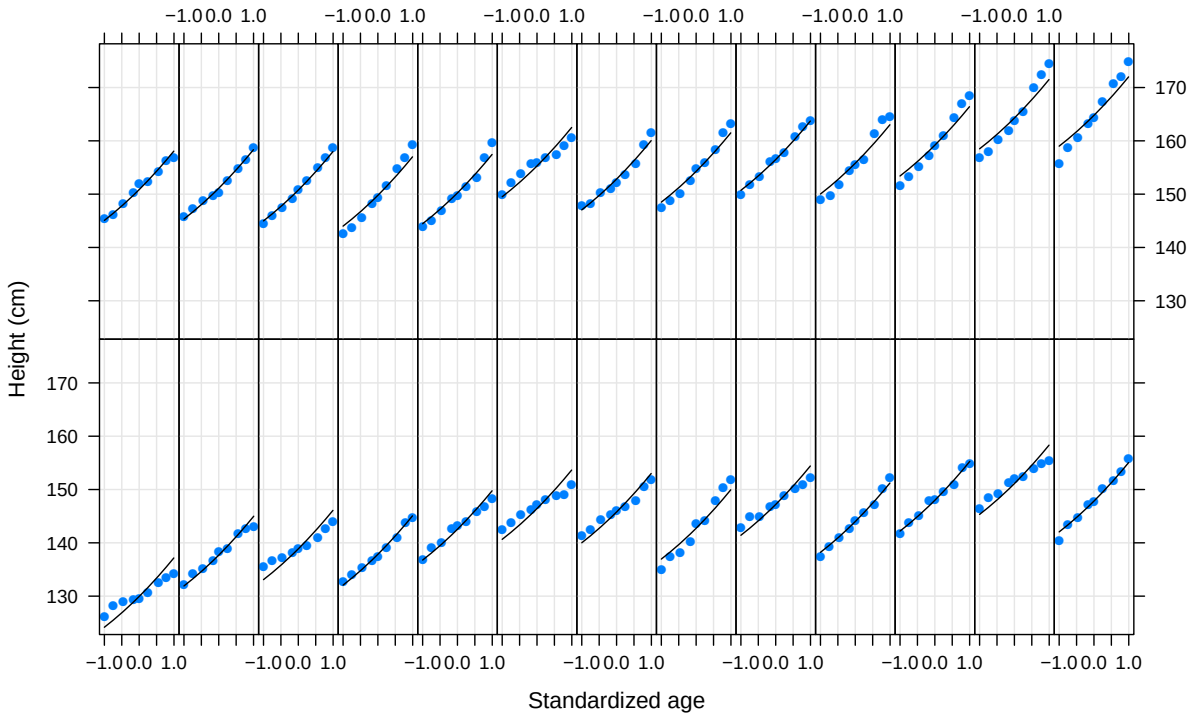
We now have the fitted model object `fm.mixed`, and we wish to plot the fitted regression curve in each panel. Two approaches are possible. We can make the fitted model available to the panel function, and then use it as needed. Alternatively, we can augment our dataset with the necessary information *before* plotting anything. We start with the first approach.

Fitted models in panel functions

We already know how to make the fitted model available to the panel function: pass it in as an argument not recognized by `xyplot()`, and it will be passed on to the panel function. Thus, we have

```
> xyplot(height ~ age | Subject, data = Oxboys, fit = fm.mixed,
         strip = FALSE, aspect = "xy", pch = 16, grid = TRUE,
         panel = function(x, y, ..., fit) {
             panel.xyplot(x, y, ...)
             subj.coef <- coef(fit)$Subject[packet.number(), ]
             ypred <- with(subj.coef,
                           `(Intercept)` + `age` * x + `I(age^2)` * x^2)
             panel.lines(x, ypred, col = "black")
         },
         xlab = "Standardized age", ylab = "Height (cm)")
```

However, this seems unnecessarily complicated. Partly this is due to the use of the `lmer()` function; the fitted model does not have a `predict()` method (for legitimate reasons), and so we had to manually construct the predictions using the result of `coef()`. In general, for multipanel plots the current panel will represent some subset of the full dataset, and we would need to somehow figure out the corresponding part of the model. Here, for example, we needed to figure out (inside the panel function) which `Subject` was represented in the panel.



One simpler but fairly general approach works well in this and many other situations. Consider the following example, which produces identical output.

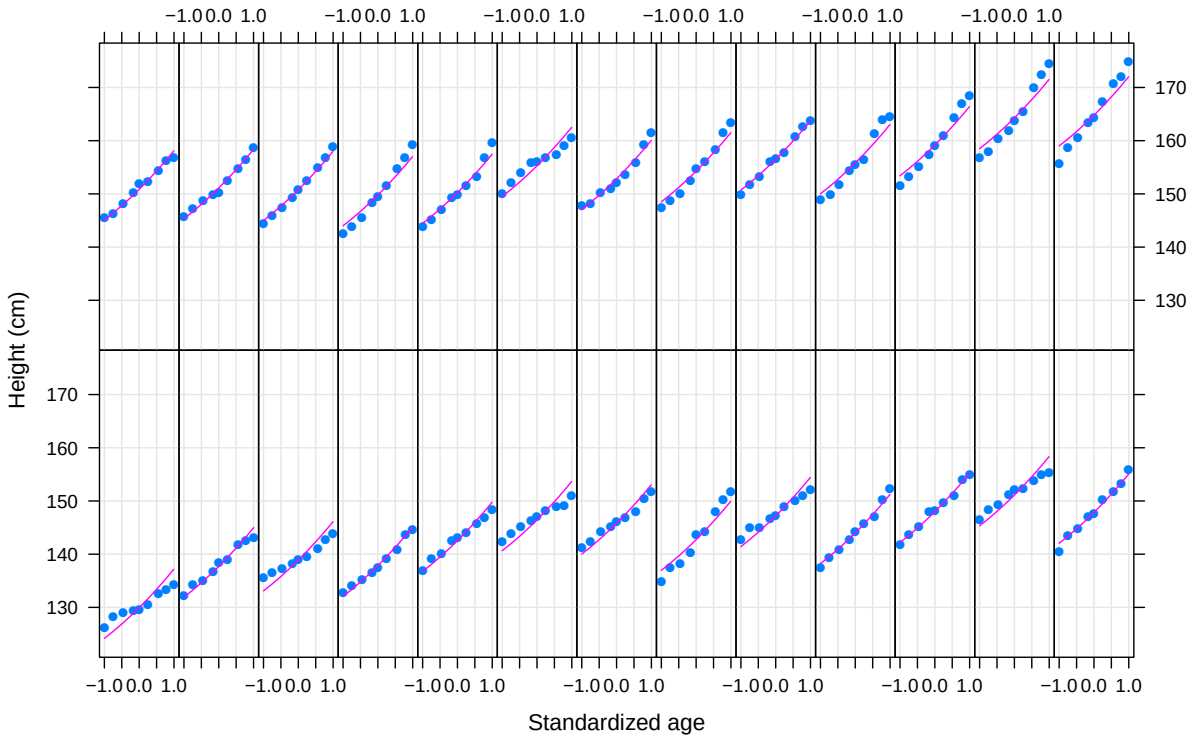
```
> xyplot(height ~ age | Subject, data = Oxboys, fit = fm.mixed,
         strip = FALSE, aspect = "xy", pch = 16, grid = TRUE,
         panel = function(x, y, ..., fit, subscripts) {
             panel.xyplot(x, y, ...)
             ypred <- fitted(fit)[subscripts]
             panel.lines(x, ypred, col = "black")
         },
         xlab = "Standardized age", ylab = "Height (cm)")
```

This makes use of two features. First, the `fitted()` method extracts the fitted values from the model object. This of course corresponds to the full dataset, and is not specific to the panel; it will be a vector with the same length as the number of observation used to fit the model, and will be identical for each panel. The second piece is the `subscripts` argument, which `lattice` passes to the panel function, containing the indices of the observations in the original dataset that end up in the panel. This is used to extract the corresponding elements of `fitted(fit)`.

An interesting corollary of this approach is the following trick.

```
> xyplot(height + fitted(fm.mixed) ~ age | Subject, data = Oxboys,
         strip = FALSE, aspect = "xy", pch = 16, grid = TRUE,
         type = c("p", "l"), distribute.type = TRUE,
         xlab = "Standardized age", ylab = "Height (cm)")
```

Except for the color of the line, the output is essentially identical. This treats the fitted values as an additional set of `y` values, and plots them with lines rather than points. See `help(panel.superpose)` for an explanation of the `distribute.type` argument.

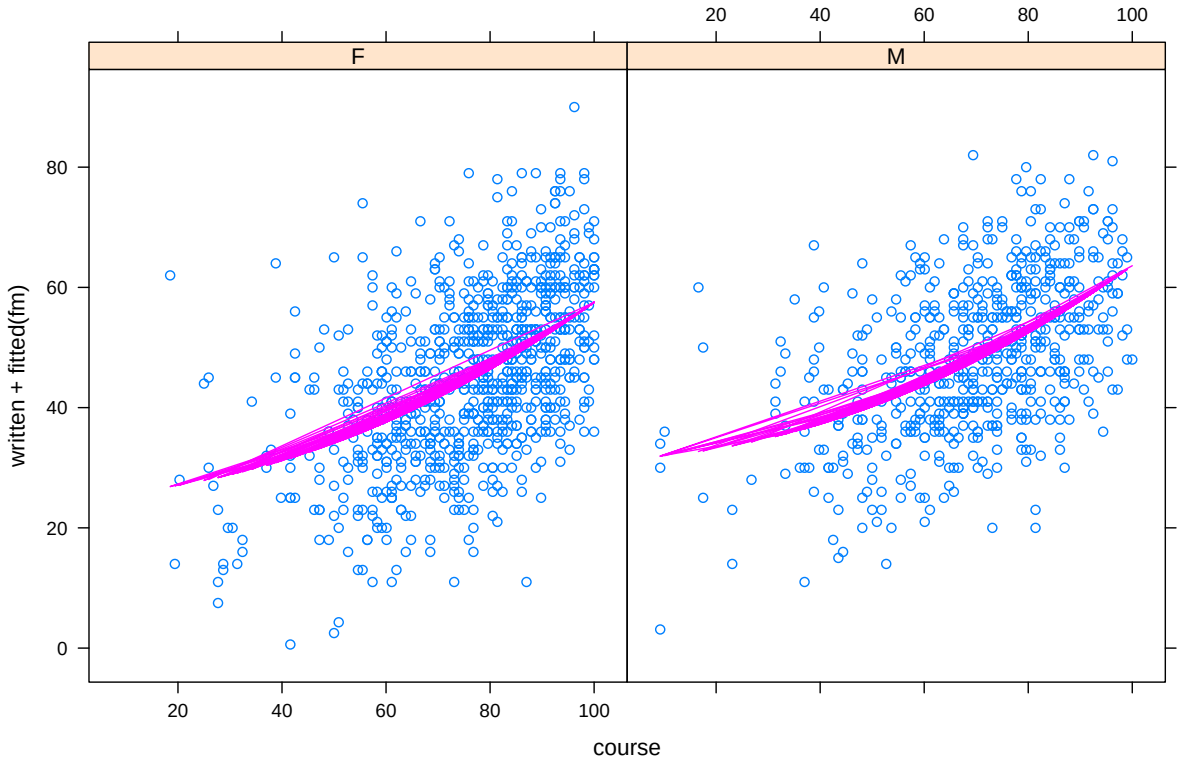


Irregular data

Unfortunately, this approach does not work in all situations. For example, in the `Gcsemv` example, we might try the following.

```
> fm <- lm(written ~ course + I(course^2) + gender, Gcsemv, na.action = na.exclude)
> xyplot(written + fitted(fm) ~ course | gender, data = Gcsemv,
         type = c("p", "l"), distribute.type = TRUE)
```

The `na.exclude` argument is needed, as otherwise missing values are omitted from `fitted(fm)` making its length different from that of the other variables. However, as we can see below, the result is a mess because the x values are not ordered (and there are too many of them as well).



We can of course always stick to the built-in solutions; for example,

```
> xyplot(written ~ course | gender, Gcsemv,
         type = c("p", "r"), col.line = "black")
```

However, more complex models need more work. Consider the three models

```
> fm0 <- lm(written ~ poly(course, 2),
            data = subset(Gcsemv, !(is.na(written) | is.na(course))))
> fm1 <- update(fm0, written ~ poly(course, 2) + gender)
> fm2 <- update(fm0, written ~ poly(course, 2) * gender)
```

The first model is a quadratic regression model that ignores gender. The second model incorporates gender as an additive term, and the third model further includes interaction terms. Here we have used the "non-missing" subset of `Gcsemv` in `fm0` to avoid further grief related to missing values, and used the `update()` function to simplify the subsequent model fitting calls. Our goal is to compare the fits from `fm2` and `fm1` with that from `fm0`.

All we really need are the fitted values at a sufficiently dense grid covering the range of the predictors. Our approach will be to compute these values separately and then combine them suitably before plotting. This approach generalizes to more models, as well as other types of models. We first define a suitable evaluation grid.

```
> course.rng <- range(Gcsemv$course, finite = TRUE)
> grid <-
      expand.grid(course = do.breaks(course.rng, 30),
                  gender = unique(Gcsemv$gender))
```

Next we evaluate the corresponding fitted values for each of the three models.

```
> fm0.pred <- cbind(grid, written = predict(fm0, newdata = grid))
> fm1.pred <- cbind(grid, written = predict(fm1, newdata = grid))
> fm2.pred <- cbind(grid, written = predict(fm2, newdata = grid))
```

Notice that we have done this in a way that each set of predictions comes in the form of a data frame with columns `course`, `gender`, and `written`.

```
> str(fm0.pred)
```

```
'data.frame':        62 obs. of  3 variables:
 $ course : num  9.25 12.28 15.3 18.32 ...
 $ gender : Factor w/ 2 levels "F","M": 2 2 2 2 2 2 2 2 ...
 $ written: num  29.4 29.7 30.1 30.5 ...
```

We will now transform our original dataset to have the same form.

```
> orig <- Gcsemv[c("course", "gender", "written")]
> str(orig)
```

```
'data.frame':        1905 obs. of  3 variables:
 $ course : num  NA 71.2 76.8 87.9 44.4 NA 89.8 17.5 ...
 $ gender : Factor w/ 2 levels "F","M": 2 1 1 1 2 1 1 2 ...
 $ written: num  23 NA 39 36 16 36 49 25 ...
```

and then combine it with the predictions from the models we wish to compare.

```
> combined <- make.groups(original = orig, fm0 = fm0.pred, fm2 = fm2.pred)
```

The `make.groups()` function combines similarly structured datasets (or simple atomic vectors) with possibly different number of observations into a single dataset, adding a new variable to indicate the origin of each observation in the new dataset.

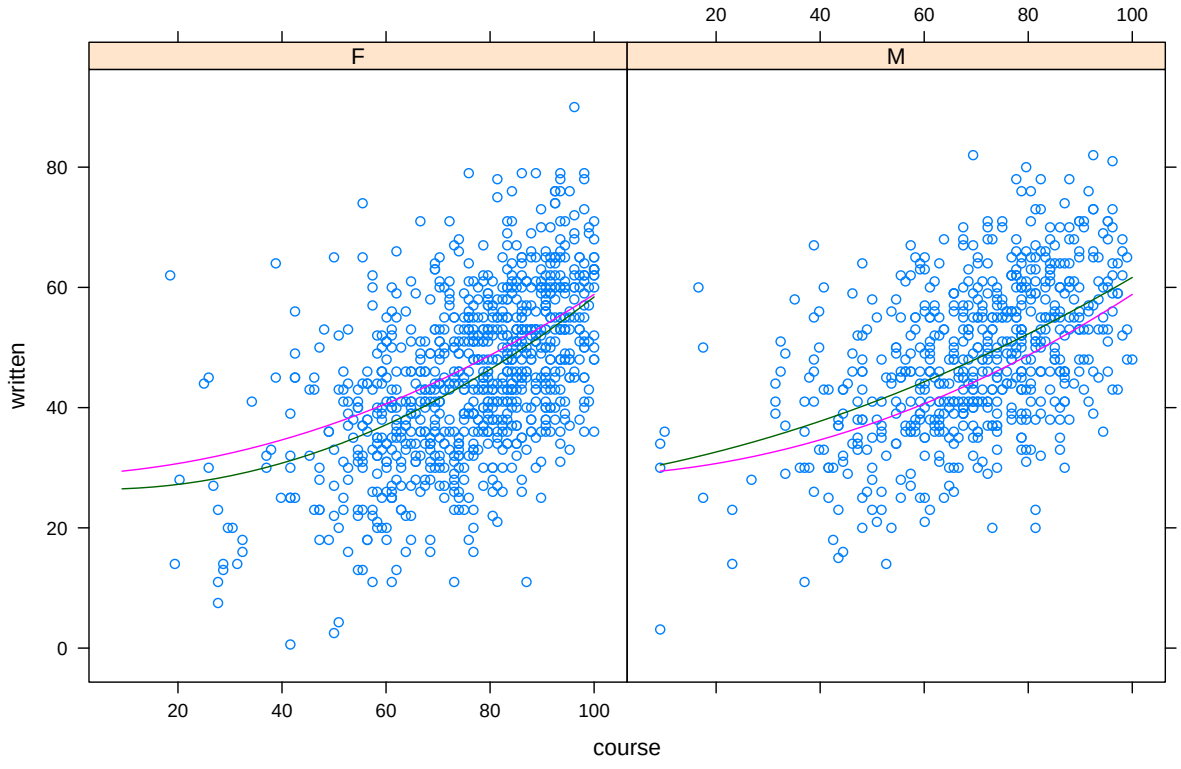
```
> str(combined)

'data.frame':        2029 obs. of  4 variables:
 $ course : num  NA 71.2 76.8 87.9 44.4 NA 89.8 17.5 ...
 $ gender : Factor w/ 2 levels "F","M": 2 1 1 1 2 1 1 2 ...
 $ written: num  23 NA 39 36 16 36 49 25 ...
 $ which  : Factor w/ 3 levels "original","fm0",..: 1 1 1 1 1 1 1 1 ..
```

The `combined` dataset thus consists of the original raw observations (`orig`), and evaluations of the fitted models `fm0` and `fm2`. All this manipulation finally leads to the call that produces the plot we want.

```
> xyplot(written ~ course | gender,
         data = combined, groups = which,
         type = c("p", "l", "l"), distribute.type = TRUE)
```

The raw data are plotted as points, whereas the model fit evaluations are joined by lines to produce a representation of the fitted curves.

A similar comparison of `fm0` and `fm1` is left a an exercise.



Summary

The take-home message from all this is that custom panel functions provide finest level of control, but built-in panel functions are also powerful because we can take specify their arguments using argument passing. However, this requires knowledge of arguments (so read documentation!).

For adding regression lines from complex models, it is usually best to obtain a suitable representation of the relevant data *before* the plotting is done. The special function `panel.superpose()` is useful for grouping, and in particular the `distribute.type` argument is useful when adding model fits.

Session information

- R Under development (unstable) (2011-07-26 r56509), `x86_64-unknown-linux-gnu`
- Locale: `LC_CTYPE=en_IN`, `LC_NUMERIC=C`, `LC_TIME=en_IN`, `LC_COLLATE=en_IN`, `LC_MONETARY=en_IN`, `LC_MESSAGES=en_IN`, `LC_PAPER=C`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_IN`, `LC_IDENTIFICATION=C`
- Base packages: base, datasets, graphics, grDevices, methods, stats, tools, utils
- Other packages: lattice 0.19-33, lme4 0.999375-39, Matrix 0.999375-50
- Loaded via a namespace (and not attached): grid 2.14.0, nlme 3.1-101, stats4 2.14.0