The `coefplot2` package

Ben Bolker

April 17, 2012

1 Introduction

The purpose of the `coefplot2` is to make the following tasks easy, or at least easier:

- quickly visualize the point estimates and measures of uncertainty of fitted statistical models;
- compare fits of models with the same modeling approach but different sets of predictors;
- compare fits of the same models fitted with different algorithms or implementations;
- produce beautiful, flexible, publication-quality plots of coefficient estimates

The starting point is the `coefplot` function in the `arm` package, which allows the first (see also the `coefplot` package, `http://www.r-statistics.com/2010/07/visualization-of-regression-coefficients-in-r/`

2 Design goals & issues

minimizing dependencies in order to meet its goal of understanding/being able to extract data from data structures from many packages, `coefplot2` will necessarily (at least) `Suggest:` many packages. However, the chances for various nasty kinds of namespace pollution/collision are high, especially with various combinations of S3 and S4 methods (`nlme`, various flavours of `lme4`, `glmmADMB` ...). Therefore, the dependencies should be made as weak as possible, and methods etc. should be imported only if absolutely necessary.

modularity back-end (`coeftab`) should extract information in a flexible way into a standardized format, while front-end (`coefplot[2]`) should use this standardized format for plotting. Furthermore, the code for merging lists of coefficients from different models (possibly with different subsets of parameters), currently inside `coefplot2.fitList` should be abstracted into a separate function for merging `coeftab` objects into `coeftabList` objects (or something like that)

back-end flexibility different model types have different kinds of parameters. I am particularly interested in mixed models, where we might be interested in picking out (1) fixed-effect parameters; (2) random effects; (3) variance parameters [in different parameterizations – (log)-Cholesky factor, variance/covariance, standard deviation/correlation ...]. Other models may have other kinds of parameters – dispersion parameters (some GLM[M]s and negative binomial models), heteroscedasticity and correlation parameters (e.g. from `lme`), zero-inflation parameters (from `glmmADMB`, `pscl`), ...

alternative plotting front-ends it's a pain, but all three of the existing plotting interfaces (base, `lattice`, and `ggplot`) have different features that make supporting all three of them (potentially) worthwhile

- base graphics are the hardest to make look pretty, and require lots of parameters to be defined, but are also the easiest for users to understand, modify, and augment
- lattice graphics are intermediate: prettier by default (and some prefer the style to ggplot's), but still typically require lots of parameters. Plotting confidence intervals is tricky without writing customized panel functions (or using `Hmisc`'s `xYplot` extension to get the same results). `lattice` is also self-contained, and Recommended, so carries no additional dependencies. For this case, extending the `xyplot` S3 generic function is probably the way to go ...
- ggplot graphics are (perhaps) the prettiest, and allow a good deal of flexibility, but also carry a string of dependencies (although they are tightly coupled and hence less of a problem). More importantly, they require a bit of a paradigm shift on the user's part, relative to base graphics. Here it's probably best to use the `fortify` and `autoplot` mechanisms (see e.g. `fortify.confint.glht` in `ggplot2`).

options for error bars provide a reasonably flexible way to specify the definition of the error bars: could be $\pm$ a specified number of SD, an α level (translated via normal approximation, or ?? t approximation if applicable?, or translated to a credible interval, or a quantile ...)

default aesthetics there are some design tradeoffs; it won't hurt to do things the way we like them, but we should provide flexibility for those who want to be more old-fashioned/please reviewers, supervisors, etc.

- point-range graphs (cleaner? less non-data-ink?) vs. traditional error bars with serifs/end caps
- inner/outer bars (e.g. thick lines for $\pm 1SD$ or 50% credible intervals; thin lines for $\pm 2SD$ or 95% CI) vs traditional error bars
- horizontal presentation (allows more room for labels) vs traditional vertical presentation (more familiar, but often requires staggering/rotating/abbreviating labels)
- allow for violin plots etc. in scenarios that allow them (bootstrap, MCMC CI)?

information to be incorporated in `coeftab` we need to decide what kind of information `coeftab` should carry along. e.g. it will be very useful for it to know the assignments of categorical variables to factors (so these can be grouped in the output). It might be useful for it to know (1) standard deviations of predictor variables (for post-hoc scaling), (2) link functions (for back-transformation). Should the ...?

transformations make it easy, or at least possible, for users to (1) back-transform estimates and CI from a predictor to a response scale; (2) change from unscaled to scaled parameter estimates (this might require access to the original model, or at least the model frame)

grouping want to allow access to grouping variables for parameters, such as factor assignment of parameters, allowing them to be grouped by point/line colour or (?) by background rectangles (see Glycera example)

3 description of `coeftab`

`coeftab` currently inherits from `data.frame`. It has columns for a point estimate, the standard error of the estimate (usually based on local curvature), and some number of quantiles (by default 2.5, 25, 75, 97.5), which could be derived in a variety of different ways (based on SE with normal or t distribution, or on quantiles or HPD intervals of MCMC output, or (??) on bootstrap or parametric bootstrap output). We could make the structure richer, e.g. by creating a list of components for different parameter types (fixed vs random effects/BLUPs vs variances/covariances vs dispersion or zero-inflation parameters ...) — this might have advantages but would have a big disadvantage in terms of overall transparency and letting users hack what they needed out of the results (unless we were extremely careful in designing accessors etc. so that the objects still *looked* like data frames). `coeftab`s may also contain p-value columns. The cheaper/cheesier way to carry along information would be in attributes, which could be hidden in the default plot method. (At present `print.coeftab` is set to `printCoefmat`, so that p-values are formatted nicely if present.)

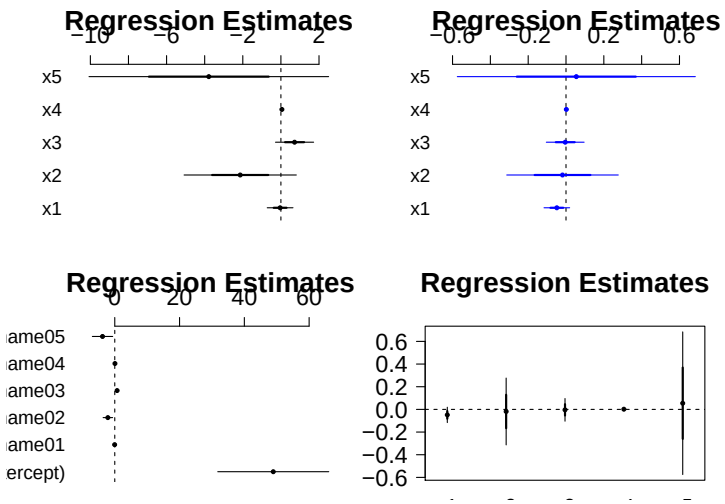
I'm not quite sure how lists of `coeftab`s should be handled: should there be a separate class for them, so the lists can be kept separate until merging is needed for plotting/formatting?

4 description of `coefplot`

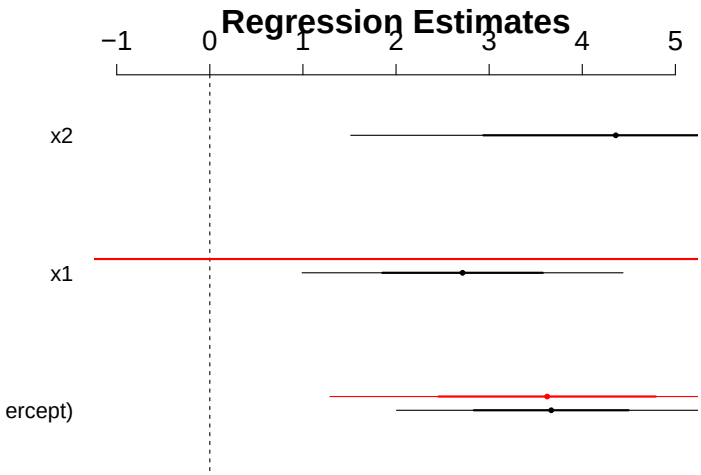
5 `arm::coefplot` examples

```
## plot 1
par(mfrow = c(2, 2))
coefplot(fit1)
coefplot(fit2, col.pts = "blue")
## plot 2
longnames <- c("(Intercept)", longnames)
coefplot(fit1, longnames, intercept = TRUE, CI = 1)
```

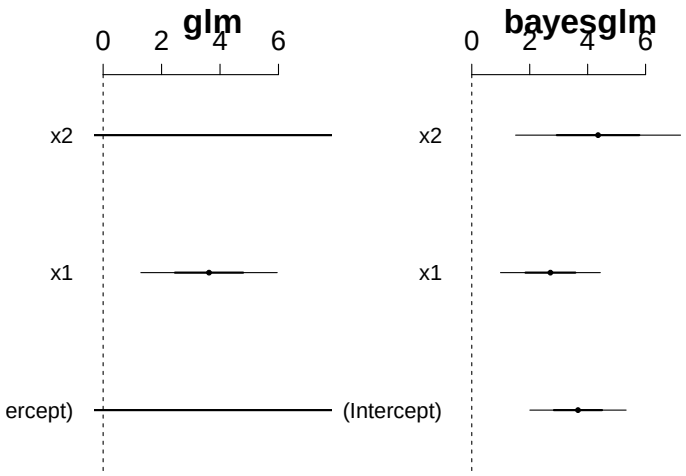
```r
## plot 3
coefplot(fit2, vertical = FALSE, var.las = 1, frame.plot = TRUE)
```



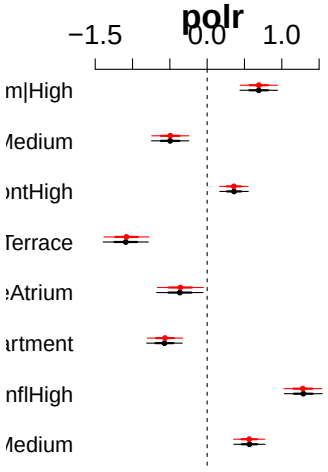

```r
## plot 4, stacked: bayesglm >> glm
coefplot(M2, xlim = c(-1, 5), intercept = TRUE)
coefplot(M1, add = TRUE, col.pts = "red")
```




```r
## ==================== arrayed plot ====================
par(mfrow = c(1, 2))
x.scale <- c(0, 7.5)  ## fix x.scale for comparison
coefplot(M1, xlim = x.scale, main = "glm", intercept = TRUE)
coefplot(M2, xlim = x.scale, main = "bayesglm", intercept = TRUE)
```

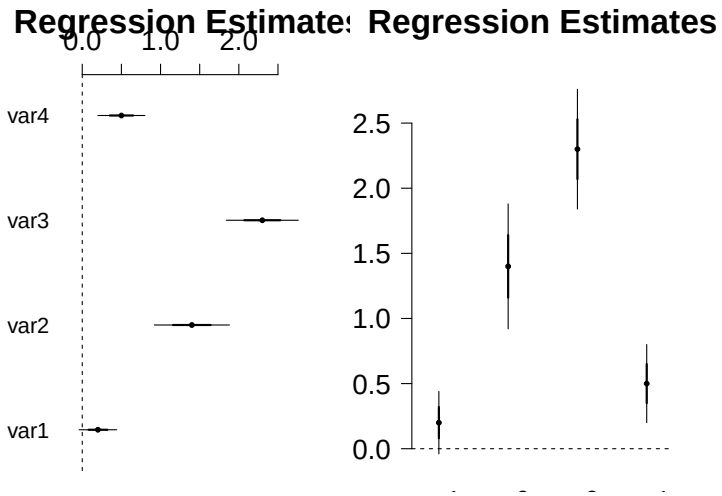



```
## plot 5: the ordered logit model from polr
par(mfrow = c(1, 2))
coefplot(M3, main = "polr")
##
## Re-fitting to get Hessian
##
##
## Re-fitting to get Hessian
##
coefplot(M4, main = "bayespolr", add = TRUE, col.pts = "red")
```



```
#### plot 6: plot bugs & lmer
library(lme4.0)
##
## Attaching package: 'lme4.0'
##
## The following object(s) are masked from 'package:lme4':
##
##     getME, glmer, isREML, lmer, lmList, nlmer, refit, sigma,
##     VarCorr
##
## The following object(s) are masked from 'package:coda':
##
##     HPDinterval
##
## The following object(s) are masked from 'package:stats':
##
##     AIC, BIC
##
M5 <- lmer(Reaction ~ Days + (1 | Subject), sleepstudy)
## M5.sim <- mcsamp(M5) coefplot(M5, var.idx=5:22, CI=1,
ylim=c(18,1),
## main='lmer model')
detach("package:lme4.0")
```

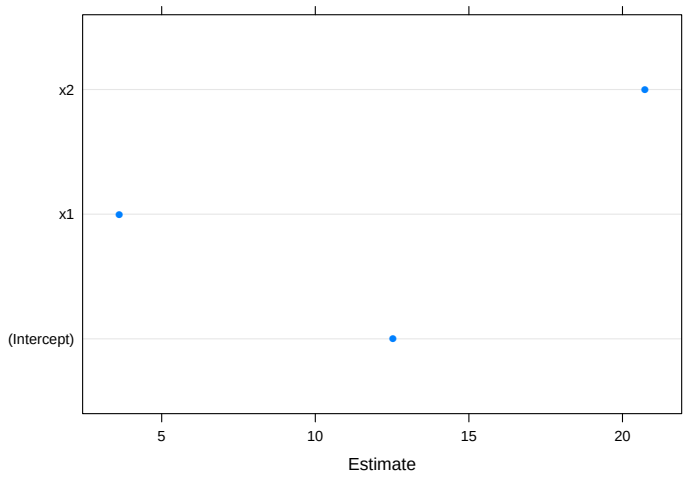
```
## plot 7: plot coefficients & sds vectors
par(mfrow = c(1, 2))
coef.vect <- c(0.2, 1.4, 2.3, 0.5)
sd.vect <- c(0.12, 0.24, 0.23, 0.15)
longnames <- c("var1", "var2", "var3", "var4")
coefplot(coef.vect, sd.vect, varnames = longnames, main =
"Regression Estimates")
coefplot(coef.vect, sd.vect, varnames = longnames, vertical =
FALSE,
    var.las = 1, main = "Regression Estimates")
```



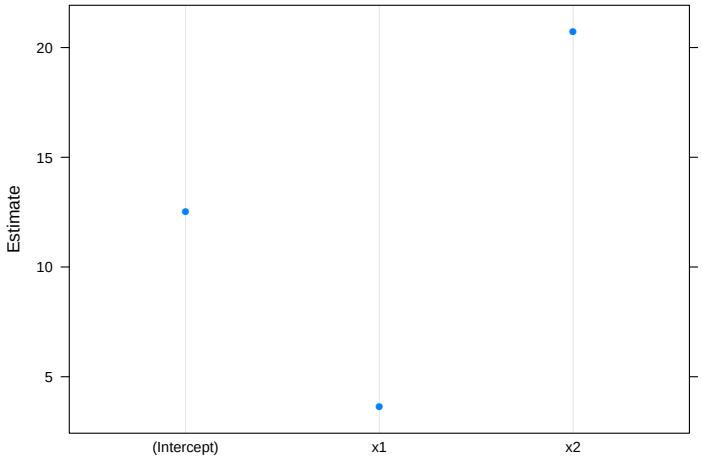

```
detach("package:arm")
detach("package:lme4")
```

6 Prettier graphs/using lattice/ggplot2

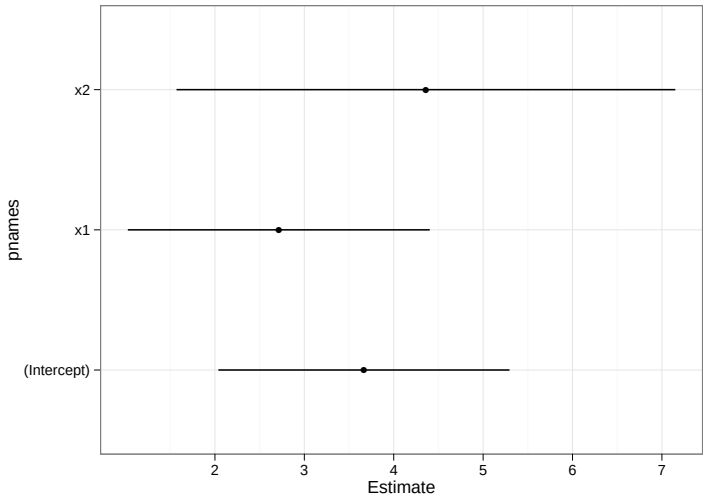
```
dotplot.coeftab <- function(object, horizontal = FALSE, ...) {
    object$pnames <- rownames(object)
    if (!horizontal) {
        dotplot(pnames ~ Estimate, type = "p", data = object,
...)
    } else {
        dotplot(Estimate ~ pnames, type = "p", data = object,
...)
    }
}
dotplot(coeftab(M1))
```




```
dotplot(coeftab(M1), horizontal = TRUE)
```




```
fortify.coeftab <- function(object) {
    object$pnames <- rownames(object)
    object <- plyr::rename(object, c(‘Std. Error‘ = "std_error",
‘2.5%‘ = "lwr",
        ‘97.5%‘ = "upr", ‘25%‘ = "lwr2", ‘75%‘ = "upr2"))
    as.data.frame(object)
}
qplot(pnames, Estimate, ymin = lwr, ymax = upr, data =
fortify(coeftab(M2)),
    geom = "pointrange") + coord_flip()
```


x2
x1
(Intercept)
pnames
2
3
4
5
6
7
Estimate