Blindfire Plotter Explained

Jimmy Oh

Department of Statistics
University of Auckland

December 17, 2012

Contents

1 Introduction

This document attempts to explain some of the core coding ideas that allow Blindfire Plotter to function. It also goes into some of the more interesting technical details of the code and how it works.

It does not cover general usage of Blindfire Plotter nor is it a comprehensive explanation of all the code. For the most part, this document is dedicated to the explanation of `codefrags`.

The document is a work in progress and will be updated sporadically.

2 Codefrags

Central to what gives Blindfire Plotter its generality and ability to be extended in a modular fashion are the `codefrags` (short for Code Fragments). The code that generates the `codefrags` are deceptively short.

```
## Construct code fragments
codefrags = list()
for(i in 1:length(respnum))
  codefrags[[i]] = substitute(eval(respnum[[i]]$expr,
             lapply(respnum[[i]]$els, function(x)
                    datagrab(datdf, eval(x, datdf), facs))
             ), list(i = i))
```

We will explain how this works and what it does, but first we must understand the inputs.

2.1 Respnum

The first input we must understand is `respnum`, which is the parsed version of `respform` and `scaleresp`. How `BFPparseFormula` does the parsing is not covered here, what is important at this stage is only what it produces.

```
> BFPparseFormula(respform = a ~ b + c, scaleresp = NULL)
$a
$a$expr
a

$a$els
$a$els$a
a



$`b + c`
$`b + c`$expr
b + c

$`b + c`$els
$`b + c`$els$b
b

$`b + c`$els$c
c
```

The parsing is intended to expose the formula in a format that is convenient to use. An unfortunate side-effect is that the output is rather bloated and inelegant. Still, the casual user will never be exposed to it and it will never be very large.

The formula is first split into a list by '~', then inside each element of this list, we keep the original expression in `expr` and explicitly state all the elements used in a sub-list `els`.

If a `scaleresp` had been specified, this simply adds a '`/scaleresp`' to each `expr`, like so:

```
> BFPparseFormula(~ a + b, ~ c)
$`(a + b)/c`
$`(a + b)/c`$expr
(a + b)/c

$`(a + b)/c`$els
$`(a + b)/c`$els$a
a

$`(a + b)/c`$els$b
b

$`(a + b)/c`$els$c
c
```

Thus `scaleresp` is merely a shortcut argument for convenient scaling of multiple responses split by '~'.

It should also be mentioned that the formula accepted is quite robust, able to handle interesting arithmetic.

```
> BFPparseFormula(~ (a + 1)^1 - 1, ~ c)
$`((a + 1)^1 - 1)/c`
$`((a + 1)^1 - 1)/c`$expr
((a + 1)^1 - 1)/c

$`((a + 1)^1 - 1)/c`$els
$`((a + 1)^1 - 1)/c`$els$a
a

$`((a + 1)^1 - 1)/c`$els$c
c
```

2.2 Datagrab and Facs

The other inputs we must understand are `datagrab` and `facs`.

`datagrab` is a wrapper function for `by`, and is mainly a relic of an older version of Blindfire Plotter when it did more interesting things. It is likely to be phased out completely to be replaced with calls to `by`.

`facs` is a character vector containing the column names of the factors Blindfire Plotter is currently examining. This vector is produced by the various *modules* and will not be examined in depth here.

As a demonstration, consider the `CO2` dataset (included with the default R distribution).

```
> CO2[1:5,]
  Plant   Type  Treatment conc uptake
1   Qn1 Quebec nonchilled   95   16.0
2   Qn1 Quebec nonchilled  175   30.4
3   Qn1 Quebec nonchilled  250   34.8
4   Qn1 Quebec nonchilled  350   37.2
5   Qn1 Quebec nonchilled  500   35.3

## Using facs = "Type"
> datagrab(CO2, CO2$uptake, "Type")
     Quebec Mississippi
     1408.8       877.1

## Using facs = c("Type", "Treatment")
> datagrab(CO2, CO2$uptake, c("Type", "Treatment"))
             Treatment
Type          nonchilled chilled
  Quebec             742   666.8
  Mississippi        545   332.1
```

2.3 How It Works

To generate `codefrags` we first loop through each `respnum`, that is, each part of the formula as split by '~'.

```
for(i in 1:length(respnum))
```

For each such part, we further loop through the `els` sub-list.

```
lapply(respnum[[i]]$els, function(x)
```

Thus, we are taking each element needed to evaluate the expression. We then determine what this element is by a call to `eval`, but we do this using our data (`datdf`) as our environment. This has an interesting implication, the element can be a column inside our data (`datdf`), or it could lie somewhere in our `.GlobalEnv`. It can potentially even be a combination of the two, though this is not recommended as it can lead to confusion and mistakes.

```
> expr
a + b
> a = 1
> b = 2
> datdf = data.frame(b = 3)
## eval without specifying an environment
> eval(expr) ## 1 + 2
[1] 3
## We now eval using datdf as our environment
> eval(expr, datdf) ## 1 + 3 as b from datdf takes precedence
[1] 4
```

This "interesting implication" generally has no practical impact on how Blindfire Plotter functions, beyond the fact that it is possible, and that the user may unwittingly use a variable in the `.GlobalEnv` and not one from their data[1].

Once we have retrieved the actual values of our elements (which should be vectors for Blindfire Plotter to work), we pass this to `datagrab` to obtain the restructured element.

```
datagrab(datdf, eval(x, datdf), facs)
```

Note then, that each element is restructured separately, then stored in a list. This list of restructured elements is then used as the environment in which our `expr` is evaluated.

```
eval(respnum[[i]]$expr,
     lapply(respnum[[i]]$els, function(x)
            datagrab(datdf, eval(x, datdf), facs)))
## Or more simply, using pseudo-code
eval(respnum[[i]]$expr, List of Elements)
## Where the Elements are the actual data vectors which have been
##   restructured with respect to the facs we are currently examining.
```

Finally, we use `substitute` to replace the `i`'s with the current `i` of our loop of `respnum`. Thus `codefrag[[1]]` will have `respnum[[1]]`, etc.

[1] Something that is often the case with R, so is not a 'feature' specific to Blindfire Plotter.

2.4 How It Is Used

In the `BFPinnerfunc` which handles much of the heavy lifting for every *module* of Blindfire Plotter, we evaluate each element of `codefrags`[2] using `list(facs = facs)` as our environment. Thus our final evaluation to obtain the restructured data used for plotting involves an `eval` nested inside a `datagrab` nested inside a `lapply` nested inside an `eval`[3] which is further nested inside another `eval`. The level of nesting could be reduced, but to do so would come at the cost of making the code even more difficult to understand.

```
eval(codefrags[[1]], list(facs = facs))
```

Once again, `facs` is a character vector containing the column names of the factors Blindfire Plotter is currently examining. This vector is produced by the various *modules* and will not be examined in depth here.

2.5 For What End?

So we now understand that `codefrags` does all that. So what? Well, what `codefrags` enables is the evaluation of non-trivial expressions (including arithmetic but also potentially other R functions like `sort`[4]) on a generalised level, and with respect to any combination of factors that can be specified as a simple character vector.

Thus the benefit is two-fold:

- Accept non-trivial expressions and hence allow for various permutations of the response variables at the argument level, e.g. `a ~ b`, `~ a + b`, `~ a - b + 1`, `~ a * b`, `a/(a + b) ~ b/(a + b)`, etc.
- Make it simple to add new *modules* to examine combinations of variables not currently covered by Blindfire Plotter, as the module only needs to be complex enough to create character vectors of the factors, and does not have to worry at all about how to go about evaluating this with respect to whatever formula has been specified in `respform`.

[2] Remembering that `codefrags` will be a list, each element corresponding to an expression of `respform` split by '~'.

[3] Which had in turn been nested inside a `substitute`, which has thankfully already been evaluated by this stage.

[4] Conceptually. To actually use `sort` would require extending the `sort` Methods to handle arrays of at least three dimensions, ideally more for generality.