plyr: divide and conquer

Hadley Wickham

September 16, 2008

plyr is a set of tools for a common set of problems: you need to break a big problem down into manageable pieces, operate on each piece and then put all the pieces back together. I call this strategy "split-apply-combine" and the three components form the basis for this paper and the `plyr` package.

The paper concludes with two case studies using `plyr` to operate on a 3d array of spatio-temporal dataset, and a large data frame of repeated observations.

1 Introduction

The `plyr` package provides tools for solving a common class of problems, where you break apart a big complicated data structure into small simple pieces, operate on each piece independently and then put all the pieces back together (possibly in a different format to the original). This paper introduces the `ply` family of functions which generalise the `apply` family found in the base package, and include all combinations of input and output of lists, data frames and arrays.

This paper describes version 0.1 of `plyr`, which requires R 2.7.0 or later and has no run-time dependencies. To install it from within R, run `install.packages("plyr")`. Information about the latest version of the package can be found online at `http://had.co.nz/plyr`.

In general, plyr provides a replacement for for loops for a large set of practical problems. The major assumption that the `ply` functions make is that each piece can be operated on independently of the other pieces, so if there is any dependence between the pieces then you will need to use other tools. The need to provide an alternative to loops does not come about because loops are slow (they are not!), but because they do not clearly express the intent of the algorithm, as important details are mixed in with unimportant book-keeping code. The tools of `plyr` aim to eliminate this extra code and illuminate the key components of your computations.

Section 3 introduces the `plyr` family of tools and how to use them. The `plyr` package also provides a number of helper functions for error recovery, splatting, column-wise processing, and reporting progress, described in Section 4. Section 5 discusses the general strategy that these functions support, including cases studies that explore the performance of veteran baseball players and ozone measured over space and time. Finally, Section 6 maps existing R functions to their plyr counterparts and lists related packages. Section 7 describes future plans.

2 Motivation

Why use plyr? Why not use for loops or the built-in apply functions? This section compares plyr code to base R code for an example that is explained in more detail in Section 5.2.

In this example we are going to remove seasonal affects from satellite measurements of ozone. The ozone was measured on a 24×24 grid, each month for six years, and is stored in a $24 \times 24 \times 72$ 3d array. A single location (`ozone[x, y, ]`) is a vector of 72 values, and we can crudely deasonalise it by looking at the residuals of a robust linear model:

```
one <- ozone[1, 1, ]

month <- factor(rep(1:12, length = 72))
model <- rlm(one ~ month - 1)
deseasf <- resid(model)

deseasf <- function(value) rlm(value ~ month - 1)
```

The challenge is now to apply this function to each location, assembling the results back into the same form as the input. We also want to keep the intermediate models in a 2d array, so we can reference a local model (`model[1, 1]`) in a similar way to referencing a local time series (`ozone[1, 1, ]`). In base R, we can tackle this problem with for loops, or with the apply family of functions:

For loops

```
models <- as.list(rep(NA, 24 * 24))
dim(models) <- c(24, 24)

deseas <- array(NA, c(24, 24, 74))

for (i in seq_len(24)) {
  for(j in seq_len(24)) {
    mod <- deseasf(ozone[i, j, ])

    models[[i, j]] <- mod
    deseas[i, j, ] <- resid(mod)
  }
}
```

Apply functions

```
models <- apply(ozone, 1:2, deseasf)

resids <- unlist(lapply(models, resid))
dim(resids) <- c(72, 24, 24)
deseas <- aperm(resids, c(2, 3, 1))
dimnames(deseas) <- dimnames(ozone)
```

The main disadvantage of the for loop is that there is lot of book-keeping code in there. We need to create the output structure before filling them up, and the size of the array is hard coded in multiple places. The apply functions (`apply()` and `lapply()`) simplify this task, but there's no straightforward way to go from the 2d array of models to a 3d array of residuals. In plyr, the code is much shorter because these details are taken care of:

```
models <- aaply(ozone, 1:2, deseasf)
deseas <- aaply(models, 1:2, resid)
```

You may be wondering what those function names mean. All **plyr** functions have a concise but informative naming scheme: the first two characters describe input and output data types. Both of these functions input and output an **a**rray. Other data types are **l**ists and **d**ata frames. Because plyr caters for every combination of input and output data types in a consistent way, it is easy to use the data structure that feels most natural for a given problem.

For example, instead of storing the ozone data in a 3d array, we could also store it in a data frame. This type of format is more common if the data is ragged, irregular or incomplete, where we don't have measurements at every possible location for every possible time point. Imagine the data frame is called ozonedf and has columns lat, long, time, month, and value. To repeat the deasonalisation task with this new data format, we first need to tweak our workhorse method:

```
deseasf_df <- function(df) {
  rlm(value ~ month - 1, data = df)
}
```

Because the data could be ragged, it's much harder to use for loops here and we'll use split(), lapply() and mapply() to complete the task. Here the split-apply-combine strategy maps closely to built-in R functions: we split with split(), apply with lapply() and then combine the pieces into a single result with rbind().

```
pieces <- split(ozonedf, list(ozonedf$lat, ozonedf$long))
models <- lapply(pieces, deseasf)

results <- mapply(function(model, df) {
  cbind(df[rep(1, 72), c("lat", "long")], resid(model))
}, models, pieces)
deseasdf <- do.call("rbind", results)
```

Much of the complication here is the labelling - we only needed to use mapply() so we could match the original data up with the models. plyr takes care of all the tricky labelling stuff for you, so it only takes two lines:

```
models <- dlply(ozone, .(lat, long), deseas_df)
deseas <- ldply(models, resid)
```

The following section describes the plyr functions in more detail. If your interest has been whetted by this example, you might want to skip ahead to page 14 to learn more about this data.

3 Usage

Table 1 lists the basic set of plyr functions. Each function is named according to the type of input it accepts and the type of output it produces. The input type determines how the big data structure can be broken down into small pieces, and the output type determines how the pieces are joined back together again. Breaking down input is described in Section 3.1 and piecing together output is described in Section 3.2.

NB: In this paper, the term **array** includes the special cases of vectors (1d arrays) and matrices (2d arrays) as well, and the term **list-array** refers to an list with dimensions (as opposed to an atomic vector, as is more usual). The common atomic vectors

to \ from	array	data frame	list
array	aaply	daply	laply
data frame	adply	ddply	ldply
list	alply	dlply	llply
nothing	a_ply	d_ply	l_ply

Table 1: The 12 key functions of plyr. Arrays include matrices and vectors as special cases.

are logical, character, integer, and numeric. Dimension labels refer to the output of dimnames(), or for 2d structures, the special cases of rownames() and colnames().

The effects of the input and outputs types are orthogonal, so instead of having to learn all 12 functions individually, it is sufficient to learn the three types of input and four types of input. For this reason, it's useful to refer to a complete row (common output type) or column (common input type) of Table 1. The notation we use for this is d*ply to refer an entire row (same input) and *dply for an entire column (same output).

The **ply functions have either two or three main arguments, depending on the type of input:

- a*ply(data., margins., fun., ..., progress. = "none")
- d*ply(data., variables., fun., ..., progress. = "none")
- l*ply(data., fun., ..., progress. = "none")

The first argument is the data. which will be split up, processed and recombined. The second argument, variables. or margins., describes how to split up the input into pieces. The third argument, fun., is the processing function, and is applied to each piece in turn. All further arguments are passed on to the processing function. If you omit fun. the individual pieces will not be modified, but the entire data structure will be converted from one type to another. The progress. argument controls displaying of a progress bar, and is described at the end of Section 4.

Note that arguments to the **ply functions end in ".". This prevents name clashes with the argument of fun..

3.1 Input

Each type of input has different rules for how to split it up, and are described in detail in the following section. In short:

- a*ply(): Arrays are sliced by dimension in to lower-d pieces.
- d*ply(): Data frames are subsetted by combinations of variables.
- l*ply(): Each element in a list is a piece.

Technical note: The way the input can be split up is not actually determined by the type of the data structure, but the methods that it responds to. An object split up by a*ply() must respond to dim() and accept multidimensonal indexing; by d*ply(), must work with `split()`, and must be coercible to an environment; by list, must work with `length()` and `[[`.

The most important result of that is that data frames can also be passed to `a*ply()`, where they are treated like 2d matrices, and to `l*ply()` which will operate column wise on the data frame.

3.1.1 Input: array (`a*ply`)

The `margins` argument of `a*ply` describes which dimensions to slice along (in the same way that `apply()` does). There are four possible ways to do this for the 2d case is simple, as illustrated Figure 1:

- `margins = 1`: split into rows
- `margins = 2`: split into columns
- `margins = c(1,2)`: split into individual cells

The last way is to not split up the matrix at all, and corresponds to `margins = c()`. (However, there's not much point in using `plyr` to do this!)

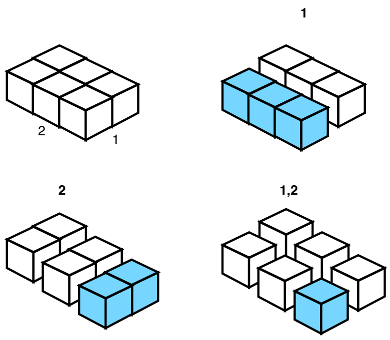

Figure 1: The four ways to split up a 2d matrix, labelled above by the dimensions that they slice up. Original matrix shown at top left, with dimensions labelled. Blue indicates a single piece of the output.

The 3d case is a little more complicated. We have three 2d slices, three 1d slices, and one 0d slice. These are shown in Figure 2. Note how the pieces for the 1d slices correspond to the intersection of the 2d slices. The margins argument works correspondingly for higher dimensions, with a combinatorial explosion in the number of possible ways to slice up the array, choose(slice-d, array-d), to be exact.

These default to working on the first dimension (i.e. row-wise) and automatically splat the function so that function is called not with a single list as input, but each column is passed as a separate argument to the function. Compared to using `mapply`, for the `m*ply` functions you will need to `cbind` the columns together first. This will ensure that each argument has the same length, and allows the `m*ply` functions to have the same argument order as all the other

Special case: `m*ply` A special case of operating on arrays corresponds to the `mapply` function of base R. The plyr equivalents are named `maply`, `mdply`, `mlply` and `m_ply`. `mapply()` takes multiple lists of parameters as input, and calls the processing function with a piece from each list as its parameters. The input to `m*ply()` is a little different: it is a list-array.

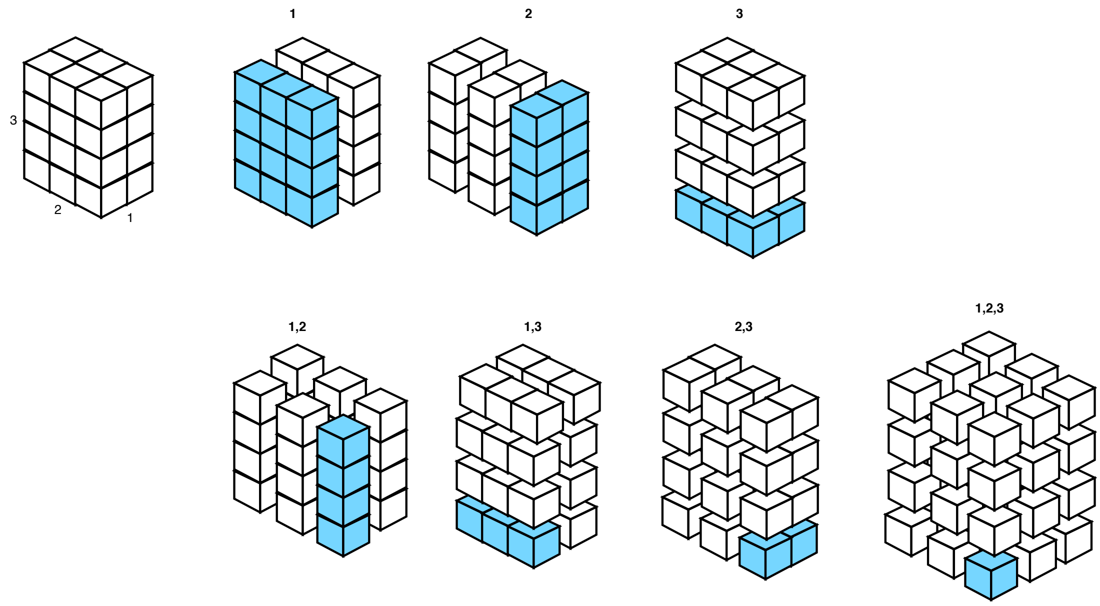

Figure 2: The eight ways to split up a 3d array, labelled above by the dimensions that they slice up. Original array shown at top left, with dimensions labelled. Blue indicates a single piece of the output.

3.1.2 Input: data frame (`d*ply`)

When operating on a data frame, you usually want to split it up into groups based on combinations variables in the data set. For `d*ply` you specify which variables (or functions of variables) to use. These variables are specified in a special way to highlight that they are computed first from the data frame, then the global environment (in which case it's your responsibility to ensure that their length is equal to the number of rows in the data frame).

- `.(var1)` will split the data frame into groups defined by the value of the `var1` variable. If you use multiple variables, `.(a, b, c)`, the groups will be formed by the interaction of the variables, and output will be labelled with all three variables.
- You can also use functions of variables: `.(round(a))`, `.(a * b)`. If you are outputting to a data frame, these will get ugly names (produced by `make.names()`), but you can override them by specifying names in the call: `.(product = a * b)`
- By default, `plyr` will look in the data frame first, and then in the global environment `.(anothervar)`. However, you are encouraged to keep all related variables in the same data frame: this makes things much easier in the long run.

Figure 3 shows two examples of splitting up up a simple data frame. Splitting up data frames is easier to understand (and to draw!) than splitting up arrays, because they're only 2 dimensional.

3.1.3 Input: list (`l*ply`)

Lists are the simplest type of input to deal with because they are already naturally divided into pieces: the elements of the list. For this reason, the `l*ply` functions don't

name	age	sex
John	13	Male
Mary	15	Female
Alice	14	Female
Peter	13	Male
Roger	14	Male
Phyllis	13	Female

.(sex)

name	age	sex
John	13	*Male*
Peter	13	*Male*
Roger	14	*Male*

name	age	sex
Mary	15	*Female*
Alice	14	*Female*
Phyllis	13	*Female*

.(age)

name	age	sex
John	*13*	Male
Peter	*13*	Male
Phyllis	*13*	Female

name	age	sex
Alice	*14*	Female
Roger	*14*	Male

name	age	sex
Mary	*15*	Female

Figure 3: Two examples of splitting up a data frame by variables. If the data frame was split up by both sex and age, there would only be one subset with more than one row: 13-year-old males.

need an argument that describes how to break up the data structure.

Special case: `r*ply` A special case of operating on lists corresponds to `replicate()` in base R, and is useful for drawing distributions of random numbers. This is a little bit different to the other plyr methods. Instead of the `data.` argument, it has `n.` the number of replications to run, and instead of a function it accepts a expression.

3.2 Output

The output type defines how the pieces will be joined back together again, and how they will be labelled. The labels are particularly important to allow you to match up the input with the output.

The input and output types are the same, except there is an additional output option, which discard the output. This is useful for functions with side effects that make changes outside of R

The output type also places some restrictions on what type of results the processing function should return. Generally, the processing function should return the same type of data as the eventual output, (i.e. vectors, matrices and arrays for `*aply` and data frames for `*dply`) but some other formats are accepted for convenience and are described in Table 2. These are explained in more detail in the individual output type sections.

3.2.1 Output: array (`*aply`)

With array output the shape of the output array is determined by the input splits and the dimensionality of each individual result. Figures 4 and 5 illustrate this pictorially for simple 1d and 2d cases. For arrays, the pieces contribute to the output in the expected way; lists are related like a 1d array; and data frames get a dimension for each variable in the split. The dimnames of the array will be the same as the input, if an array; or extracted from the subsets, if a data frame.

Output	Processing function restrictions	Null output
`*aply`	atomic array, or list	`logical()`
`*dply` frame	data frame, or atomic vector	`data.frame()`
`*lply`	none	`list()`
`*_ply`	none	

Table 2: Summary of processing function restrictions and null output values for all output types. Explained in more detail in each output section.

The processing function should return an atomic (i.e. logical, character, numeric or integer) array of fixed size/shape, or a list. If atomic, the extra dimensions will added perpendicular to the original dimensions. If a list, the output will be a list-array. If there are no results, `*aply` will return a logical vector of length 0.

All `*aply` functions have a `drop.` argument. When this is true, the default, any dimensions of length one will be dropped. This is useful because in R, a vector of length three is not equivalent to a 3×1 matrix or a $3 \times 1 \times 1$ array.

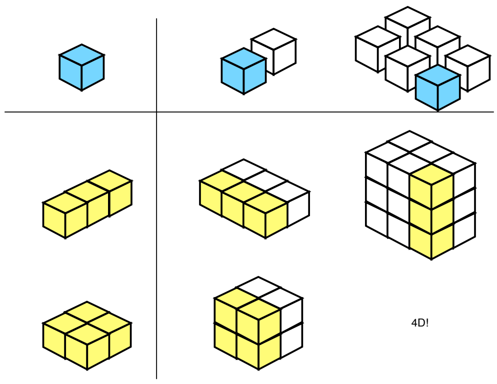

Figure 4: Results from outputs of various dimensionalty from a **single** value, shown top left. Columns indicate input: (left) a vector of length two, and (right) a 2×2 matrix. Rows indicate the shape of a single processed piece: (top) a vector of length 3, (bottom) a 2×2 matrix. Extra dimensions are added perpendicular to existing ones. The array in the bottom-right cell is 4d and so is not shown.

3.2.2 Output: data frames (`*dply`)

When the output is a data frame, it will contain the results and additional columns that identify where in the original data each row came from. These columns make it possible to merge the old and new data if you need to. If the input was a data frame, there will be a column for variables used to split up the original data; if it was a list, a column for the names of the list; if an array, a column for the names of each splitting dimension. Figure 6 illustrates this for data frame input.

The processing functions should either return a data.frame, or a (named) atomic vector of fixed length, which will form the columns of the output. If there are no results, `*dply` will return an empty data frame. `plyr` provides an `as.data.frame` method for functions which can be handy: `as.data.frame(mean)` will create a new function which outputs a data frame.

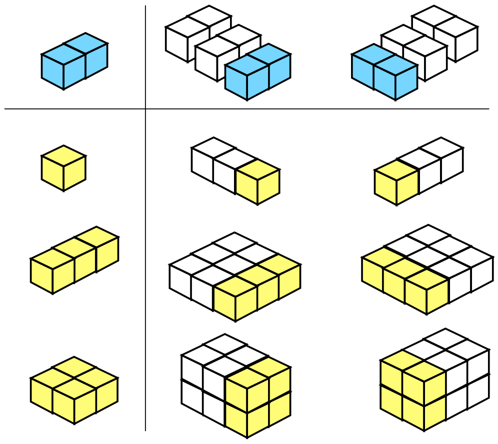

Figure 5: Results from outputs of various dimensionalty from a **1d vector**, shown top left. Columns indicate input: (left) a 2×3 matrix split by rows and (right) and 3×2 matrix split by columns. Rows indicate the shape of a single processed piece: (top) a single value, (middle) a vector of length 3, and (bottom) a 2×2 matrix.

.(sex)

sex	value
Male	3
Female	3

.(age)

age	value
13	3
14	2
15	2

.(sex, age)

sex	age	value
Male	13	2
Male	14	1
Female	13	1
Female	14	1
Female	15	1

Figure 6: Illustrating the output from using `ddply()` on the example from Figure 3 with `nrow()`. Splitting variables shown above each example. Note how the extra labelling columns are added so that you can identify to which subset the results apply.

3.2.3 Output: list (*lply)

This is the simplest output format, where each processed piece is joined together in a list. The list also stores the labels associated with each pieces, so that if you use `ldply` or `laply` to further process the list the labels will appear as if you had used `aaply`, `adply`, `daply` or `ddply` directly. `llply` is convenient for calculating complex objects once (e.g. models), from which pieces of interest are later extracted into arrays and data frames.

There are no restrictions on the output of the processing function. If there are no results, `*lply` will return a list of length 0.

3.3 Output: nothing (*_ply)

Sometimes you are operating on a list purely for the side effects (e.g. plots, caching, output to screen/file). This is a little more efficient than abandoning the output of `*lply` because it doesn't store the intermediate results.

4 Helpers

The `plyr` package also provides a number of helper function which take a function (or functions) as input and return a modified function as output.

- `splat()` converts a function to use. This is useful when you want to pass a function a row of data frame or array, and don't want to manually pull it apart in your function. For example:

  ```
  hp_per_cyl <- function(hp, cyl, ...) hp / cyl
  splat(hp_per_cyl)(mtcars[1,])
  splat(hp_per_cyl)(mtcars)
  ```

  Generally, splatted functions should have `...` as an argument, so you only need to specify the variables that you are interested in. For more information on how splat works, see `do.call`.

  `splat()` is applied to functions used in `m*ply` by default.

- `each()` takes a list of functions and produces a function that runs each function on the inputs and returns a named vector of outputs. For example, `each(min, max)` is short hand for `function(x) c(min = min(x), max = max(x))`. Using each with a single function is useful if you want a named vector as output.

- `colwise()` converts a function that works on vectors, to one that operates column-wise of data frame, returning a data fram. For example, `colwise(median)` is a function that computes the median of each column of a data.frame.

  The optional `.if` argument specialises the function to only run on certain types of vector, e.g. `.if = is.factor` or `.if = is.numeric`. These two restrictions are provided in the premade `calcolwise` and `numcolwise`.

- `failwith()` sets a default value to return if the function throws an error. For example, `failwith(NA, f)` will return an `NA` whenever `f` throws an error.

  The optional `quiet` argument suppresses any notice of the error when it is `TRUE`.

- Given a function, `as.data.frame.function()` creates a new function which coerces the output of the input function to a data frame. This is useful when you are using `*dply()` and the default column-wise output is not what you want.

Each plyr function also has a `progress.` argument which allows you to monitor the progress of long running operations. There are four difference progress bars:

- `"none"`, the default. No progress bar is displayed.
- `"text"` provides a textual progress bar which.
- `"win"` and `"tk"` provide graphical progress bars for Windows and systems with the tcl/tk package loaded.

The progress bars assume that processing each piece takes the same amount of time, so will not be 100% accurate.

5 Strategy

1. Extract a subset of the data for which it is easy to solve the problem
2. Solve the problem by hand, checking as you go
3. Write a function that encapsulates the solution
4. Use the appropriate ply function to split up the original data, apply the function and join the pieces back together.

The following two case studies illustrate these techniques for a range of problems related to a data frame storing the batting records for long-term baseball players, and a 3d array representing space and time values of ozone.

The code shown focuses on data manipulation with `plyr` functions - much of the code to produce the graphics is omitted to save space. To see the full code look at ...

5.1 Case study: baseball

The `baseball` data set contains the batting records for all professional US players with 15 or more years of data. The complete list of variables are described fully `?baseball`, but for this example we will focus on just four: `id`, which identifies the player, `year` the year of the record and `rbi` the number of runs that the player made in the season, and at bat, the number of times the player had an opportunity to hit the ball.

(This is a rather crude analysis, as it doesn't take into account the people that might already be on the other plates)

What we'll explore is the performance of a batter over his career. To get started, we need to calculate the careeryear, i.e. the number of years since the player started playing. This is easy to do if we have a single player:

```
baberuth <- subset(baseball, id == "ruthba01")
baberuth$cyear <- baberuth$year - min(baberuth$year) + 1
```

To do this for all players, we first make a function:

```
calculate_cyear <- function(df) {
  transform(df,
    cyear = year - min(year),
    cpercent = (year - min(year)) / (max(year) - min(year))
  )
}
```

and then split up the whole data frame into people, run the function on each piece and join them back together into a data frame:

```
baseball <- ddply(baseball, .(id), calculate_cyear)
baseball <- subset(baseball, ab >= 25)
```

To summarise the pattern across all players, we first need to figure out what the common patterns are. A time series plot of rbi/ab, runs per bat, is a good place to start. We do this for Babe Ruth, as shown in Figure 7, then write a function to do it for any player (taking care to ensure common scale limits) and then use `d_ply` to save a plot for every player to a pdf. We use two tricks here: `reorder` to sort the players in order of average rbi / ab, and `failwith` to ensure that even if a single plot doesn't work we will still get output for the others.

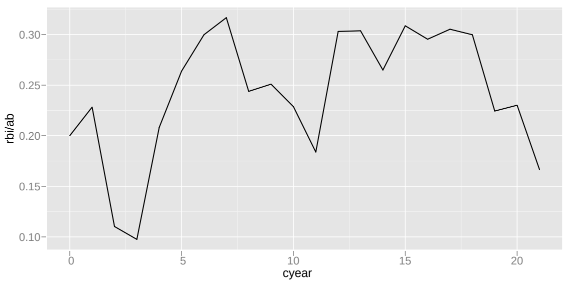

Figure 7: Runs per bat for Babe Ruth.

```
qplot(cyear, rbi / ab, data=baberuth, geom="line")

xlim <- range(baseball$cyear, na.rm=TRUE)
ylim <- range(baseball$rbi / baseball$ab, na.rm=TRUE)
plotpattern <- function(df) {
  print(qplot(cyear, rbi / ab, data = df, geom="line", xlim = xlim, ylim = ylim ))
}

pdf("paths.pdf", width=8, height=4)
d_ply(baseball, .(reorder(id, rbi / ab)), failwith(NA, plotpattern))
dev.off()
```

Flicking through the 1145 plots reveals that there doesn't seem to be much of a common pattern, although many players do seem to have a roughly linear trend with quite a bit of noise. We'll start by fitting a linear model to each player and then exploring the results. This time we'll skip doing it by hand and go directly to the function.

```
model <- function(df) {
  lm(rbi / ab ~ cyear, data=df)
}
model(baberuth)
models <- dlply(baseball, .(id), model)
```

Now we have a list of 1145 models, one for each player. To do something interesting with these, we need to extract some summary statistics. We'll extract the coefficients of the model (the slope and intercept), and a measure of model fit so we can ensure we're not drawing conclusions based on models that fit the data very poorly, the R-squared. The first few rows of `coef` are shown in Table 3.

```
rsq <- function(x) summary(x)$r.squared
coef <- ldply(models, function(x) c(coef(x), rsq(x)))
names(coef) <- c("id", "intercept", "slope", "rsquare")
```

id	intercept	slope	rsquare
aaronha01	0.18	0.00	0.00
abernte02	0.00		0.00
adairje01	0.09	−0.00	0.01
adamsba01	0.06	0.00	0.03
adamsbo03	0.09	−0.00	0.11
adcocjo01	0.15	0.00	0.23

Table 3: The first few rows of the `coef` data frame. Note that the player ids from the original data have been preserved

Figure 8 displays the distribution of r-squared across the models. The models generally do a very bad job of fitting the data! Figure 9 summarises these bad models. These plots show a negative correlation between slope and intercept, and the particularly bad models have estimates for both values close to 0. Reassuringly, there are no players in the bottom left quadrant with both negative slope and intercept.

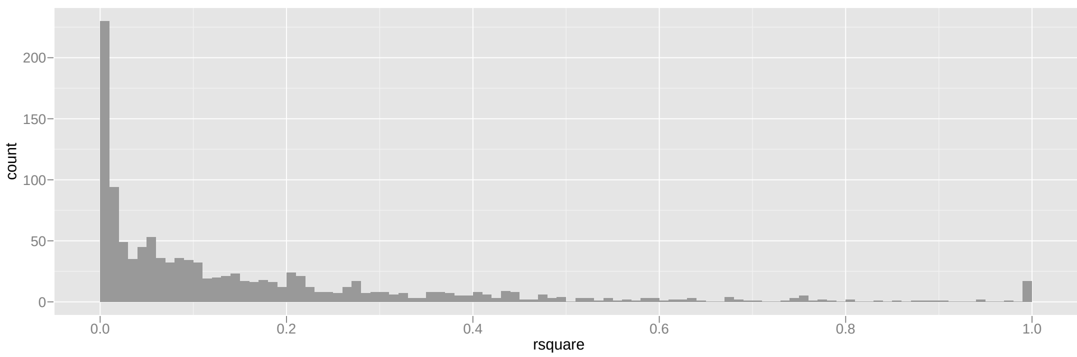

Figure 8: Histogram of model r-squared with bin width of 0.05. Most models fit very poorly! The spike of models with a r-squared of 1 are players with only two data points, found by inspecting `ldply(models[coef$rsquare == 1], "[[", "model")`

This concludes the baseball player case study, which used used `ddply`, `d_ply`, `dlply` and `ldply`. Our statistical analysis was not very sophisticated, but the tools of `plyr`

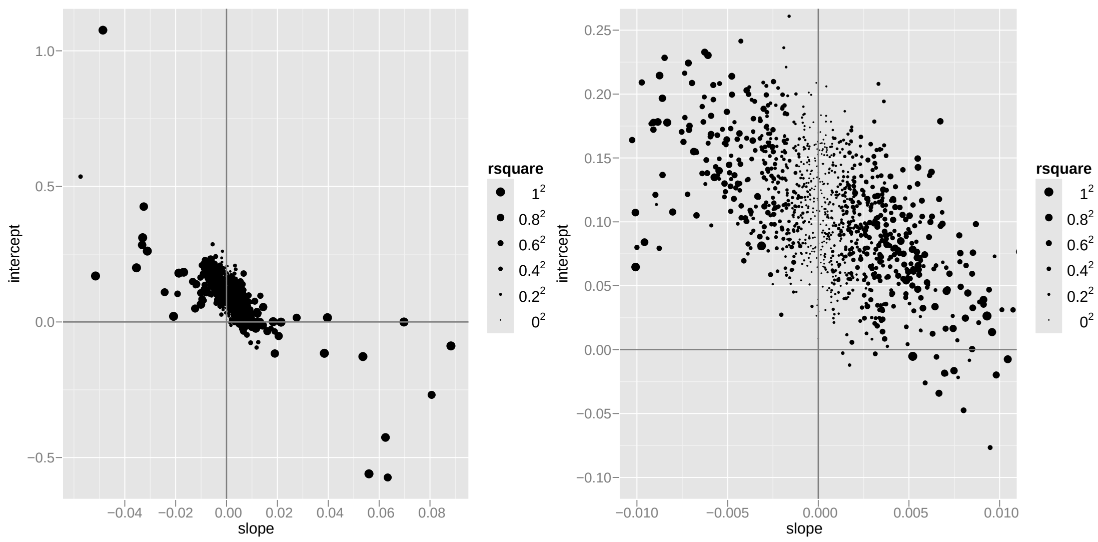

Figure 9: A scatterplot of model intercept and slope, with one point for each model (player). The size of the points is proportion to the R-square of the model. Vertical and horizontal lines emphasis the x and y origins.

made it very easy to work at the player level, and then combine results into a single summary.

5.2 Case study: ozone

In this case study we will analyse a 3d array that records ozone levels over a 24×24 spatial grid at 72 time points (Hobbs et al., To appear). This produces a $24 \times 24 times\, 72$ 3d array, containing a total of 41 472 data points. Figure 5.2 is one way of displaying this data. Conditional on spatial location, each glyph shows the evolution of ozone levels for each of the 72 months (6 years). The striking seasonal patterns make it difficult to see if there are any long-term changes. In this case study, we will explore how to separate out and visualise the seasonal effects.

Again we will start with the simplest case: a single time point, from location (1, 1). Figure 5.2 displays this in two ways: as a single line over time, or a line for each year over the months. This second plot illustrates the striking seasonal variation at this time point. The following code sets up some useful variables.

```
> value <- ozone[1, 1, ]
> time <- 1:72 / 12
> month.abbr <- c("Jan", "Feb", "Mar", "Apr", "May",
+  "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
> month <- factor(rep(month.abbr, length = 72), levels = month.abbr)
> year <- rep(1:6, each = 12)
```

We are going to use a quick and dirty method to remove the seasonal variation: residuals from a robust linear model, predicting amount of ozone by month. We could use a regular linear model, but then our seasonal estimates might be thrown off by an very unusual month. Figure **??** shows the deseasonalised trend from location (1, 1)

```
> library(MASS)
```

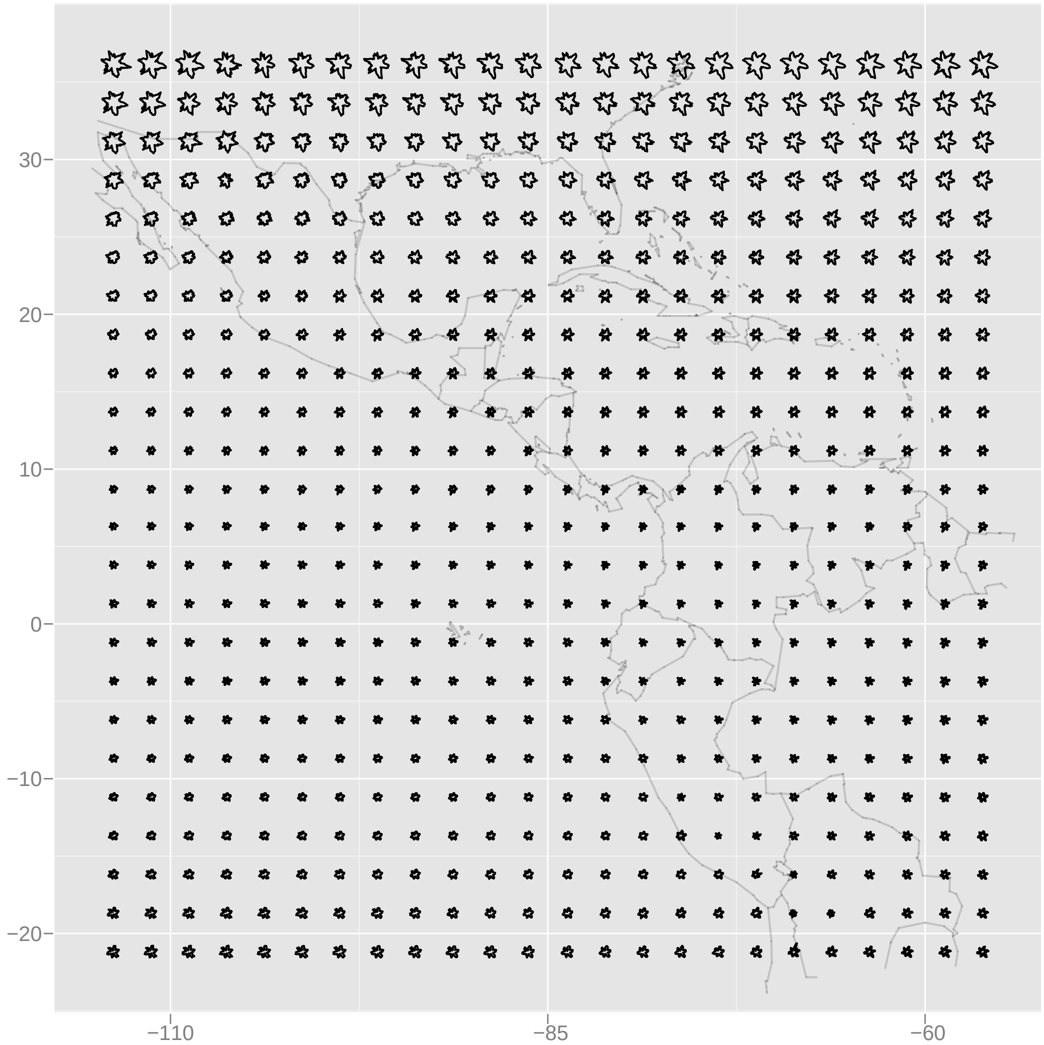

Figure 10: Star glyphs showing variation in ozone over time at each spatial location.

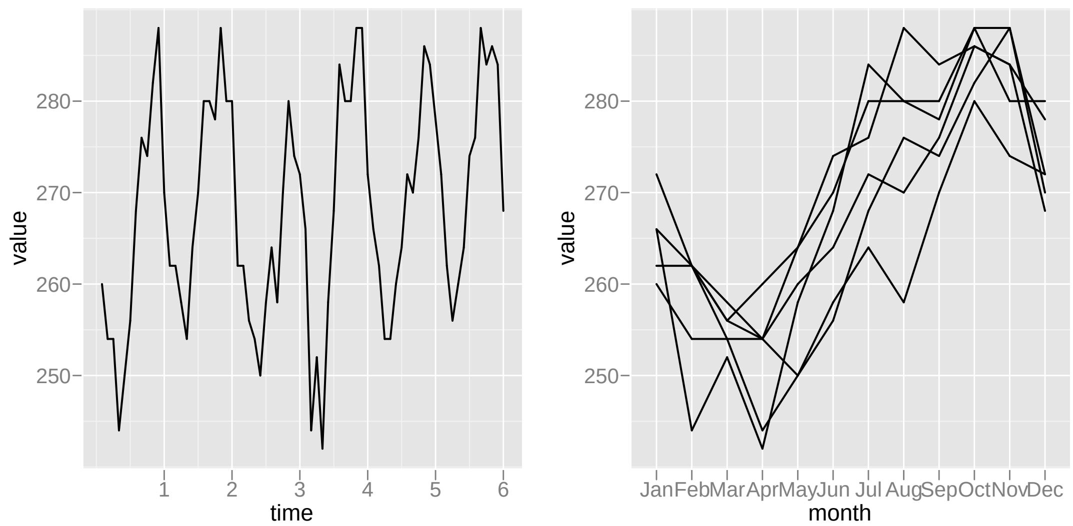

Figure 11: Two ways of displaying the seasonal changes. (Left) A single time series over all six years and (right) a line for each year.

```
+
> deseas1 <- rlm(value ~ month - 1)
> summary(deseas1)

Call: rlm(formula = value ~ month - 1)
Residuals:
   Min     1Q Median     3Q    Max
 -18.7   -3.3    1.0    3.0   11.3

Coefficients:
         Value  Std. Error t value
monthJan 264.40   2.75      96.19
monthFeb 259.20   2.75      94.30
monthMar 255.00   2.75      92.77
monthApr 252.00   2.75      91.68
monthMay 258.51   2.75      94.05
monthJun 265.34   2.75      96.53
monthJul 274.00   2.75      99.68
monthAug 276.67   2.75     100.66
monthSep 277.00   2.75     100.78
monthOct 285.00   2.75     103.69
monthNov 283.60   2.75     103.18
monthDec 273.20   2.75      99.39

Residual standard error: 4.45 on 60 degrees of freedom
> coef(deseas1)
monthJan monthFeb monthMar monthApr monthMay monthJun monthJul monthAug
     264      259      255      252      259      265      274      277
monthSep monthOct monthNov monthDec
```

```
   277       285       284       273
```

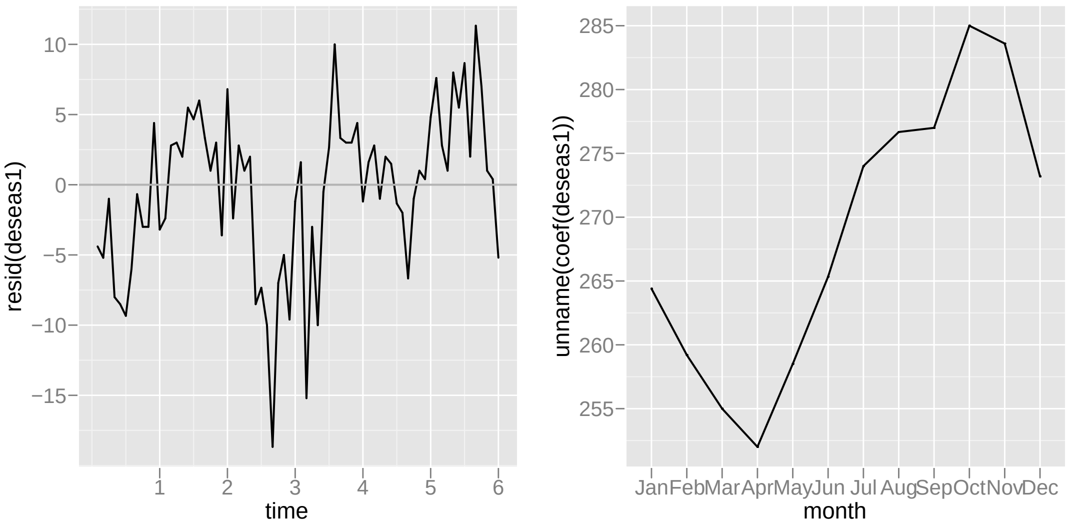

Figure 12: Deasonalised ozone trends. (Left) deasonalised trend over six years. (Right) Estimates of seasonal effects. Compare to Figure 5.2

We next turn this into a function and fit the model to each spatial location. This does take a little while, but we are fitting 576 models! From those models we extract the deasonalised values (the residuals) and the seasonal coefficients. Looking at the dimensionality we see that they're in the same format as the original data. We also carefully label the new dimensions. This is important: just as data frames should have descriptive variable names, arrays should always have descriptive dimension labels.

```
> deseasf <- function(value) rlm(value ~ month - 1)
> models <- alply(ozone, 1:2, deseasf)
+
ERROR: attempt to apply non-function
> coefs <- laply(models, coef)
> dimnames(coefs)[[3]] <- month.abbr
> names(dimnames(coefs))[3] <- "month"
+
> deseas <- laply(models, resid)
> dimnames(deseas)[[3]] <- 1:72
> names(dimnames(deseas))[3] <- "time"
+
> dim(coefs)
[1] 24 24 12
> dim(deseas)
[1] 24 24 72
```

We now have a lot of data to try and understand: for each of the 576 locations we have 12 estimates of monthly effects, and 72 residuals. There are many different ways we could visualise this data. Figures 5.2 and 5.2 visualise these results with star glyph plots. For plotting, it's more convenient to have the data in data frames. There are few different ways to do this: we can convert from the 3d array to a data frame with `melt()` from the reshape package, or use `ldply()` instead of `laply()`. For this example, we'll use a combination of these techniques. We'll convert the original array to a data frame, add on some useful columns, and then perform the same steps as above with this new format. Notice how our effect labelling the dimensions pays off with useful columns in the data frame.

```
> coefs_df <- melt(coefs)
> coefs_df <- ddply(coefs_df, .(lat, long), transform,
+  avg = mean(value),
+  std = value / max(value)
+ )
> levels(coefs_df$month) <- month.abbr
> head(coefs_df)
+
    lat long month value avg   std
1 -21.2 -114   May   264 269 0.928
2 -21.2 -114   Apr   259 269 0.909
3 -21.2 -114   Aug   255 269 0.895
4 -21.2 -114   Jan   252 269 0.884
5 -21.2 -114   Sep   259 269 0.907
6 -21.2 -114   Jul   265 269 0.931
> deseas_df <- melt(deseas)
> head(deseas_df)
+
    lat long time value
1 -21.2 -114    1 -4.40
2 -18.7 -114    1 -3.33
3 -16.2 -114    1 -2.96
4 -13.7 -114    1 -5.00
5 -11.2 -114    1 -4.00
6  -8.7 -114    1 -3.00
```

The star glyphs show temporal patterns conditioned on location. We can also look at spatial pattern conditional on time. One way to do this is to draw tile plots where each cell of the 24×24 grid is coloured according to its value. The following code sets up a function with constant scales to do that. Figure 5.2 shows the spatial variation of seasonal coefficients for January and July. We could do the same thing for the values themselves, but we'd probably want to make an animation rather than looking at all 72 plots individually.

```
> coef_limits <- range(coefs_df$value)
> coef_mid <- mean(coefs_df$value)
> monthsurface <- function(mon) {
+   df <- subset(coefs_df, month == mon)
+   qplot(long, lat, data = df, fill = value, geom="tile") +
+   scale_fill_gradient(limits = coef_limits,
+     low="white", high=muted("red")) + map
+ }
```

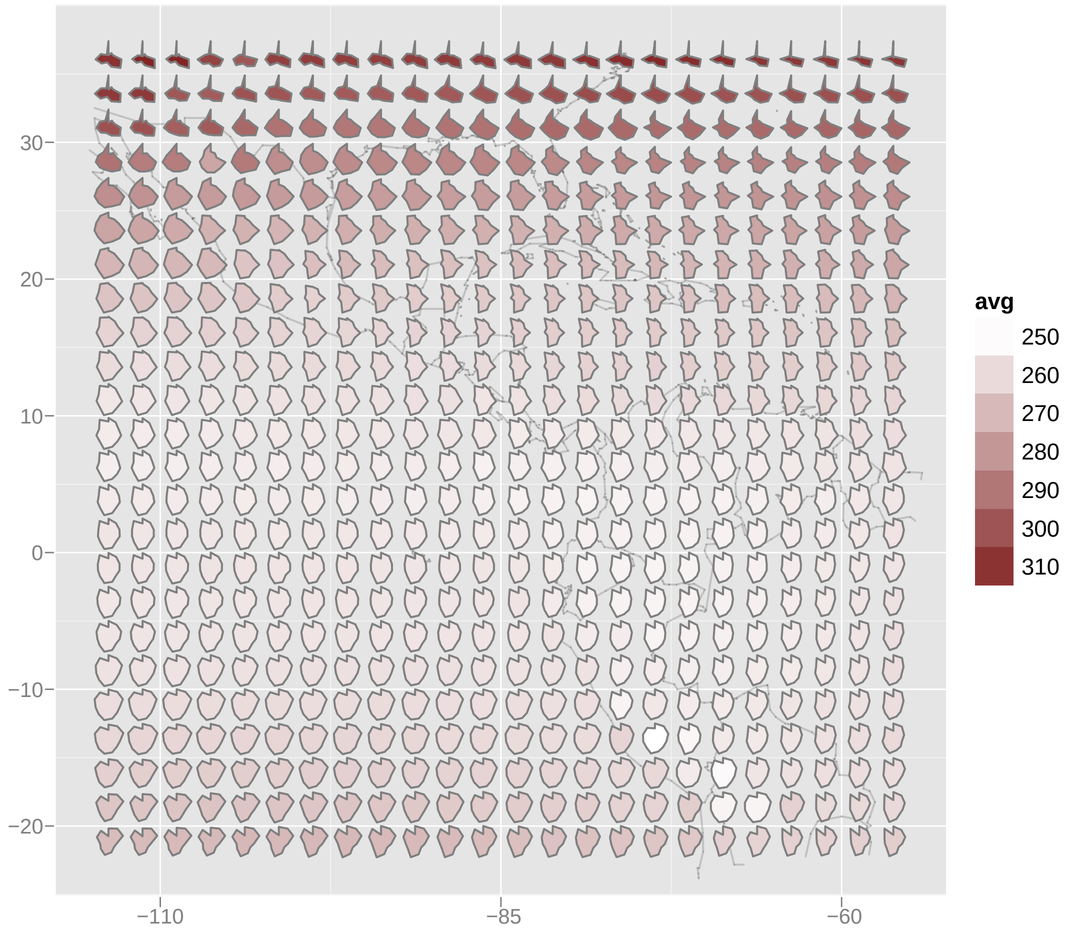

Figure 13: Star glyphs showing seasonal variation. Estimates of seasonal effects are standardised to have the same maximum at each location to make it easier to compare the general pattern. The glyph colours give the overall average ozone measurement. Note the strong spatial correlation: nearby glyphs have similar shapes.

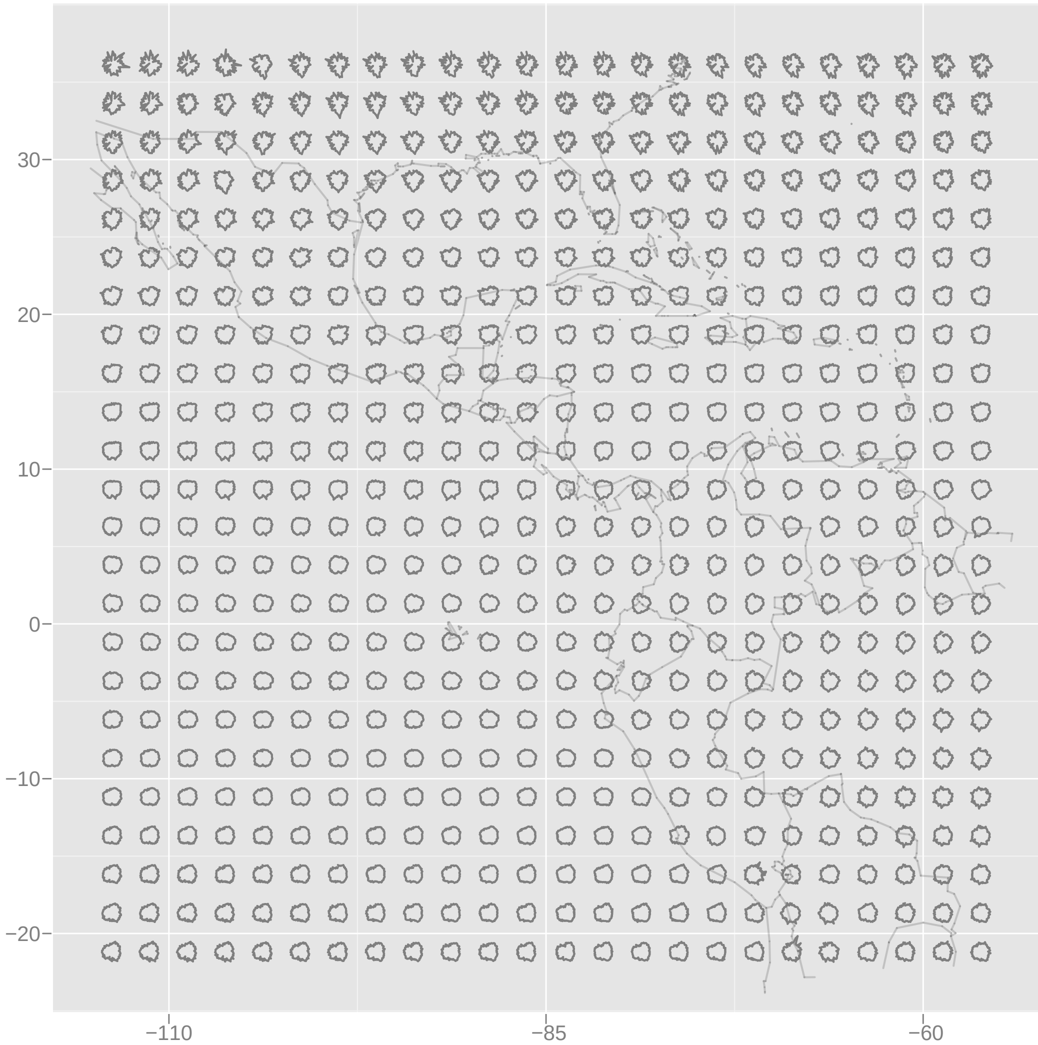

Figure 14: Star glyphs showing deasonalised trends.

```
pdf("ozone-animation.pdf", width=8, height=8)
l_ply(month.abbr, monthsurface, print. = TRUE)
dev.off()
```

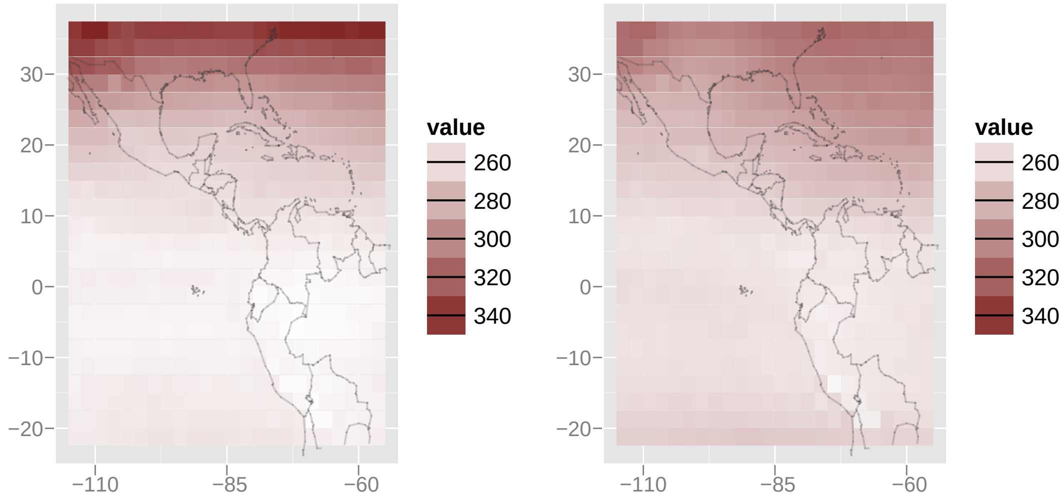

Figure 15: Tile plots of coefficients for January (left) and July (right).

5.3 Other uses

With plyr, randomisation within groups and simulation becomes trivial. For example to

```
ddply(mtcars, .(cyl), transform, mpg = sample(mpg))
```

For simulations, mdply() can be very useful, because it's easy to generate a grid of parameter values and then evaluate them.

```
mdply(expand.grid(mean = 1:5, sd = 1:5), as.data.frame(rnorm), n = 10)
```

6 Equivalence to existing R functions

Table 4 describes the equivalent between functions in base R and the functions provided by plyr. The built in R functions focus mainly on arrays and lists, not data frames, and most provide an argument to determine whether an array or list should be returned if possible (usually called simplify). The ambiguity of the output type can make programming with these functions tricky. The syntax is also less consistent than plyr, for example, mapply takes a function as the first argument rather than the input data. Compared to aaply, aaply returns the new dimensions first rather than last, which means it is not idempotent. For apply, aaply(x, a, identity) == aperm(x, a) regardless of the value of a.

Related functions tapply, ave and sweep have no corresponding function in plyr, and still remain useful. merge is also for combining summaries with the original data. The cast function in the reshape package (Wickham, 2005) is closely related to aaply.

Base function	Input	Output	`plyr` function
`aggregate`	d	d	`ddply` + `colwise`
`apply`	a	a	`aaply`
`by`	d	l	`dlply`
`lapply`	l	l	`llply`
`mapply`	a	a/l	`maply` / `mlply`
`replicate`	r	a/l	`raply` / `rlply`
`sapply`	l	a	`laply`

Table 4: Mapping between apply functions and plyr functions.

There are a number of other packages that also attempt to simplify this class of problems:

- The `doBy` (Højsgaard, 2008) package provides versions of `order`, `sample`, `split`, `subset`, `summary` and `transform` that make it easy to perform each of these operations on subsets of data frames, joining the results back into a data frame. These functions are rather like specialised version of `ddply` but they use a formula based interface, which particularly for `summary` makes it easy to only operate on selected columns.
- The `gdata` (Warnes and Gorjanc., 2008) package which contains a bundle of helpful data manipulation functions including `frameApply` which works like `ddply` or `dlply` depending on function arguments.
- The `scope` (Bergsma, 2007) package which provides `scope`, `scoop`, `skim`, `score` and `probe` which provide a composable set of functions for operating symbolically on subsets of data frames.

You might also find Spector (2008) to be useful. It is an excellent introduction to many of the thorny problems encountered when manipulating data in R.

7 Future plans

The current major shortcoming of `plyr` is speed and memory usage. While theoretically it should be possible to do most of the operations without duplicating the input data set, currently `plyr` makes at least one copy. It is my aim to eventually implement these functions in C for maximum speed and memory efficiency, so that they are competitive with the built in operations.

I am also interested in connecting with `papply` (Currie, 2005) and related packages to make it easy to split up large tasks across multiple cores and multiple computers.

References

T. Bergsma. *scope: Data Manipulation Using Arbitrary Row and Column Criteria*, 2007. R package version 2.2.

D. Currie. *papply: Parallel apply function using MPI*, 2005. URL http://ace.acadiau.ca/math/ACMMaC/software/papply/. R package version 0.1.

J. Hobbs, H. Wickham, H. Hofmann, and D. Cook. Glaciers melt as mountains warm: A graphical case study. *Computational Statistics*, To appear. Special issue for ASA Statistical Computing and Graphics Data Expo 2007.

S. Højsgaard. *doBy: Groupwise computations of summary statistics and other utilities*, 2008. R package version 3.1.

P. Spector. *Data Manipulation with R*. Springer, 2008.

G. R. Warnes and G. Gorjanc. *gdata: Various R programming tools for data manipulation*, 2008. R package version 2.4.2. Includes R source code and/or documentation contributed by Ben Bolker and Thomas Lumley.

H. Wickham. *reshape: Flexibly reshape data.*, 2005. URL http://had.co.nz/reshape/. R package version 0.7.1.