# The split-apply-combine strategy for data analysis

Hadley Wickham

May 10, 2009

Many data analysis problems involve the application of a split-apply-combine strategy, where you break up a big problem into manageable pieces, operate on each piece independently and then put all the pieces back together. This insight gives rise to a new R package that allows one to smoothly apply this strategy, without having to worry what sort of structure your data is stored in.

The paper includes two case studies showing how these insights make it easier to work with a number of batting records for veteran baseball players and a large 3d array of spatio-temporal ozone measurements.

All code used in the paper is available in the supplemental materials online.

## 1 Introduction

What do we do when we do data analysis? What are common actions and what are common mistakes? Given the importance of this activity in statistics, there is remarkably little research on how data analysis happens. This paper attempts to remedy a very small part of that lack by describing one common data analysis pattern: split-apply-combine. You see the split-apply-combine strategy whenever you break up a big problem into manageable pieces, operate on each piece independently and then put all the pieces back together. This crops up in all stages of an analysis:

- During data preparation, when performing group-wise ranking, standardisation, or normalisation, or in general when creating new variables that are most easily calculated on a per-group basis.
- When creating summaries for display or analysis, for example, when calculating marginal means, or conditioning a table of counts by dividing out group sums.
- During modelling, when fitting separate models to each panel of panel data. These models may be interesting in their own right, or used to inform the construction of a more sophisticated hierarchical model.

Just recognising the split-apply-combine strategy when it occurs is useful, because it allows you to see the similarly between problems that previously might have appeared unconnected. This helps suggest appropriate tools and frees up mental effort for the aspects of the problem that are truly unique. This strategy can be used with many

existing tools: APL's array operators (Friendly and Fox, 1994), Excel's pivot tables, the SQL's `group by` operator, and the `by` argument to many SAS procedures. However, the strategy is even more useful when used with software specifically developed to support it; matching the conceptual and computational tools reduces cognitive impedance. This paper describes one implementation of the strategy in R (R Development Core Team, 2009), the `plyr` package.

In general, `plyr` provides a replacement for for loops for a large set of practical problems, and abstracts away from the details of the underlying data structure. An alternative to loops is not required because loops are slow (they are not!), but because they do not clearly express intent, 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 the computation.

To motivate the development and use of `plyr`, Section 2 compares code that uses `plyr` functions with code that uses tools available in base R. Section 3 introduces the `plyr` family of tools, describes the three types of input and four types of output, and details the way in which input is split up and output is combined back together. 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 two cases studies that explore the performance of veteran baseball players, and the spatial-temporal variation of ozone. Finally, Section 6 maps existing R functions to their `plyr` counterparts and lists related packages. Section 7 describes future plans for the package.

This paper describes version 0.1.8 of `plyr`, which requires R 2.8.1 or later and has no run-time dependencies. Information about the latest version of the package can be found online at `http://had.co.nz/plyr`. To install it from within R, run `install.packages("plyr")`. The code used in this in this paper is available online in the supplemental materials.

**Notation.** **Array** includes the special cases of vectors (1d arrays) and matrices (2d arrays). Arrays can be made out of any **atomic** vector: logical, character, integer, or numeric. A **list-array** is a non-atomic array (a list with dimensions), which can contain any type of data structure, such as a linear model or 2d kernel density estimate. **Dimension labels** refer to `dimnames()` for arrays; `rownames()` and `colnames()` for matrices and data frames; and `names()` for atomic vectors and lists.

## 2 Motivation

How does the explicit specification of this strategy help? What the advantages of `plyr` over for loops or the built-in apply functions? This section compares `plyr` code to base R code with a teaser from Section 5.2, where we remove seasonal effects from 6 years of monthly satellite measurements, taken ozone on a $24 \times 24$ grid. The 41 472 measurements are stored in a $24 \times 24 \times 72$ array. A single location (`ozone[x, y, ]`) is a vector of 72 ($= 6 \times 12$) values.

We can crudely deseasonalise a location by looking at the residuals from a robust linear model:

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

month <- ordered(rep(1:12, length = 72))
```

```
model <- rlm(one ~ month - 1)
deseas <- resid(model)

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

The challenge is to apply this function to each location, reassembling the output into the same form as the input, a 3d array. It would also be nice to keep the models in a 2d list-array, so we can reference a local model (`model[[1, 1]]`) in a similar way to referencing a local time series (`ozone[1, 1, ]`); keeping data-structures consistent reduces cognitive effort. 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, 72))
dimnames(deseas) <- dimnames(ozone)

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: the size of the array is hard coded in multiple places and we need to create the output structures before filling them with data. The apply functions, `apply()` and `lapply()`, simplify the task, but there isn't a straightforward way to go from the 2d array of models to the 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 these function names mean. All `plyr` functions have a concise but informative naming scheme: the first and second characters describe the input and output data types. The input determines how the data should be split, and the output how it should be combined. Both of the functions used above 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; if we didn'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 deseasonalisation task with this new data format, we first need to tweak our workhorse method to take a data frame as input:

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

Because the data could be ragged, it's difficult to use a for loops 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 data frame with rbind().

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

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

Most of the complication here is the labelling: we only needed to use mapply() to match the data to the models. plyr takes care of all the tricky labelling for you, so it only takes two lines:

```
models <- dlply(ozonedf, .(lat, long), deseasf_df)
deseas <- ldply(models, resid)
```

dlply takes a data frame and returns a list, and ldply does the opposite: it takes a list and returns a data frame. Compare this code to the code needed when the data was stored in an array.

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 16 to learn more about this example and see some plots of the data before and after removing the seasonal effects.

## 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: a = array, d = data frame, l = list, and _ = nothing. The input type determines how the big data structure is broken apart into small pieces, described in Section 3.1; and the output type determines how the pieces are joined back together again, described in Section 3.2.

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 the four types of output. For this reason, it's useful to refer to a complete row (common input type) or column (common output type) of Table 1. The notation we use for this is d*ply to refer an entire row (same type of input) and *dply for an entire column (same type of output).

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

| input \ output | array | data frame | list | discarded |
|---|---|---|---|---|
| array | `aaply` | `adply` | `alply` | `a_ply` |
| data frame | `daply` | `ddply` | `dlply` | `d_ply` |
| list | `laply` | `ldply` | `llply` | `l_ply` |

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

- `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 display of a progress bar, and is described at the end of Section 4.

Note that all arguments start with ".". This should prevent name clashes with the arguments of the processing function, and helps to visually delineate arguments that control the repetition from arguments the control the individual steps.

### 3.1 Input

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

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

**Technical note.** The way the input can be split up is determined not 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 be coercible to a list; by list, must work with `length()` and `[[`. This means that data frames can be passed to `a*ply()`, where they are treated like 2d matrices, and to `l*ply()` where they are treated as a list of vectors (the variables).

#### 3.1.1 Input: array (`a*ply`)

The `.margins` argument of `a*ply` describes which dimensions to slice along. If you are familiar with `apply`, `a*ply` works the same way. There are four possible ways to do this for the 2d case. Figure 1 illustrates three of them:

- `.margins = 1`: slice up into rows

- `.margins = 2`: slice up into columns
- `.margins = c(1,2)`: slice up into individual cells

The fourth 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!

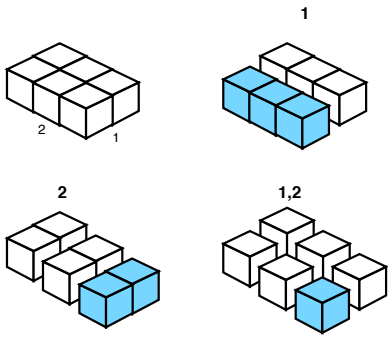


Figure 1: The three ways to split up a 2d matrix, labelled above by the dimensions that they slice. Original matrix shown at top left, with dimensions labelled. A single piece of each of the possible outputs is coloured blue.

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

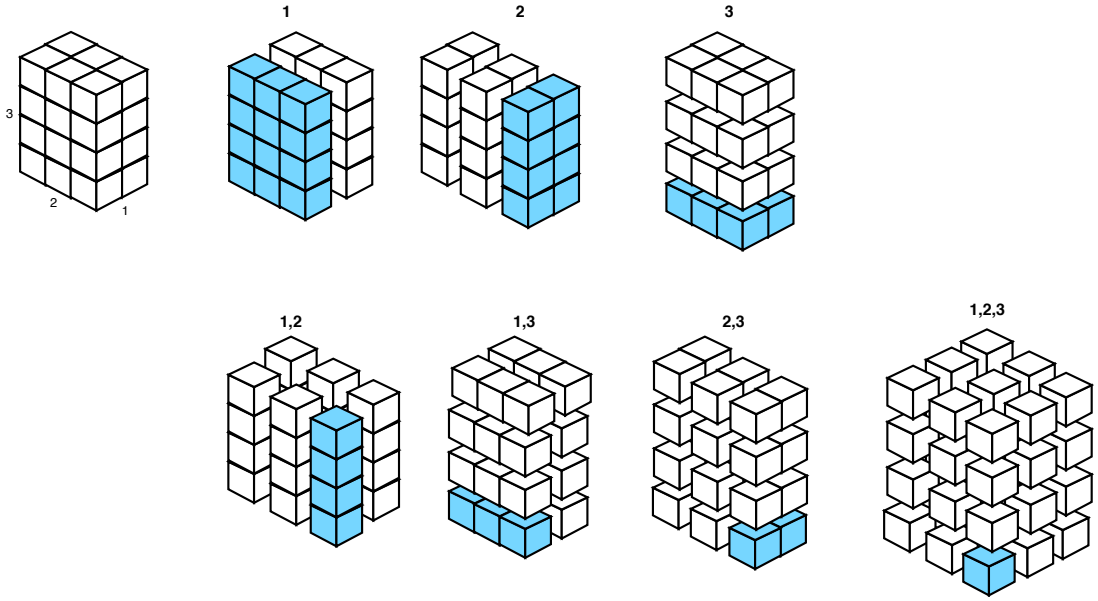


Figure 2: The seven ways to split up a 3d array, labelled above by the dimensions that they slice up. Original array shown at top left, with dimensions labelled. Blue indicates a single piece of the output.

**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`.

`m*ply()` takes a matrix or data frame, splits it up by rows and calls the processing function using each piece as its parameters. Figure 3 shows how you might use this to draw random numbers from normal distributions with varying parameters.

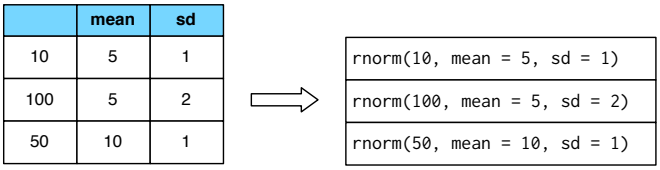


Figure 3: Using `m*ply` with `rnorm()`, `m*ply(data, rnorm)`. The function is called once for each row, with arguments given by the columns. Arguments are matched by position, or name, if present.

### 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.

Alternatively, you can use two more familiar ways of describing the splits:

- As a character vector of column names: `c("var1", "var2")`
- With a (one-sided) formula `~ var1 + var2`

Figure 4 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 need an argument that describes how to break up the data structure. Using `l*ply` is equivalent to using `a*ply` on a 1d array. `l*ply` can also be used with atomic vectors.

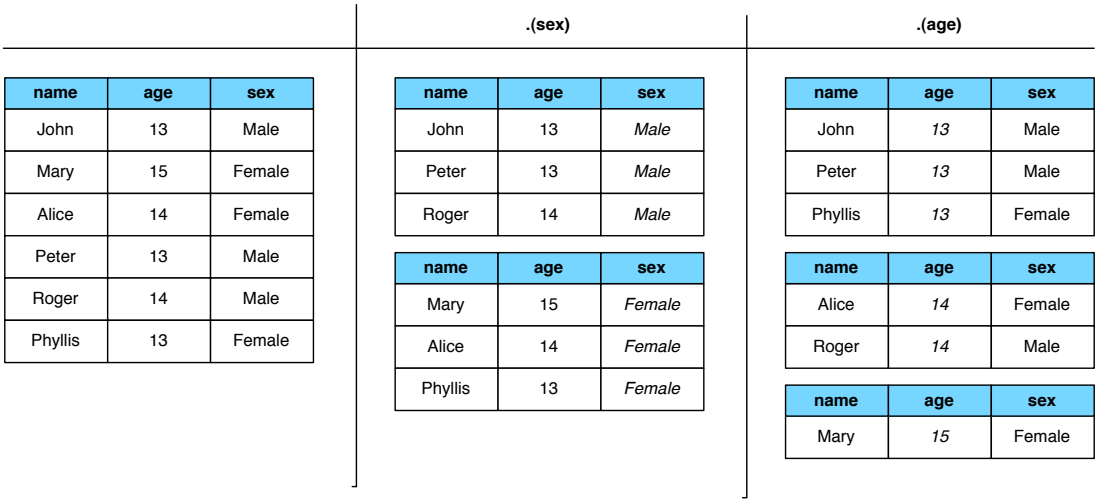


Figure 4: Two examples of splitting up a data frame by variables. If the data frame was split up by both sex and age, there would only be one subset with more than one row: 13-year-old males.

| Output | Processing function restrictions | Null output |
|---|---|---|
| `*aply` | atomic array, or list | `vector()` |
| `*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.

**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, which is evaluated afresh for each replication.

### 3.2 Output

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

The input and output types are the same, except there is an additional output data type, `_`, which discards the output. This is useful for functions like `plot()` and `write.table()` that are called only for their side effects, not their return value .

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 5 and 6 illustrate this pictorially for simple 1d and 2d cases. For arrays, the pieces contribute to the output in the expected way; lists are treated like a 1d array; and data frames get a dimension for each variable in the split. The dimension labels of the array will be the same as the input, if an array; or extracted from the subsets, if a data frame.

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 $3 \times 1$ matrix or a $3 \times 1 \times 1$ array.

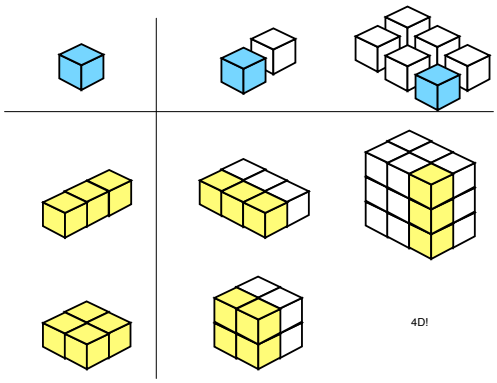


Figure 5: Results from outputs of various dimensionalty from a **single** value, shown top left. Columns indicate input: (left) a vector of length two, and (right) a $3 \times 2$ matrix. Rows indicate the shape of a single processed piece: (top) a vector of length 3, (bottom) a $2 \times 2$ matrix. Extra dimensions are added perpendicular to existing ones. The array in the bottom-right cell is 4d and so is not shown.

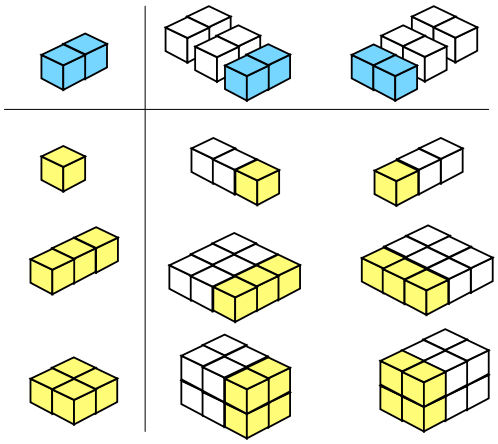

Figure 6: Results from outputs of various dimensionalty from a **1d vector**, shown top left. Columns indicate input: (left) a $2 \times 3$ matrix split by rows and (right) and $3 \times 2$ matrix split by columns. Rows indicate the shape of a single processed piece: (top) a single value, (middle) a vector of length 3, and (bottom) a $2 \times 2$ matrix.

### 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 7 illustrates this for data frame input.

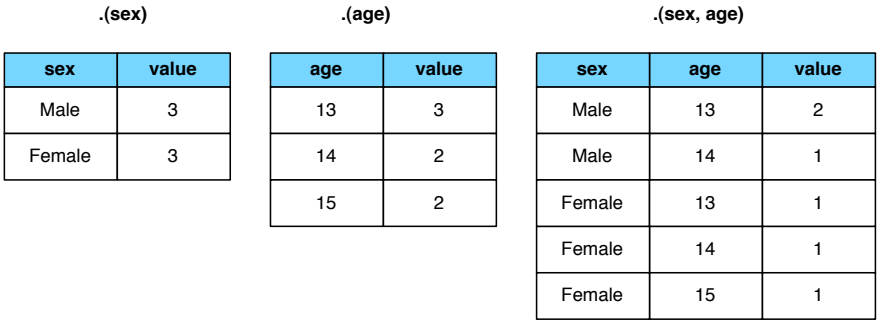


Figure 7: Illustrating the output from using `ddply()` on the example from Figure 4 with `nrow()`. Splitting variables shown above each example. Note how the extra labelling columns are added so that you can identify to which subset the results apply.

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.

### 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 piece, 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 you later extract pieces of interest 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). In this case `*_ply` is a little more efficient than abandoning the output of `*lply` because it doesn't store the intermediate results.

The `*_ply` functions have one additional argument, `.print`, which controls whether or not each result should be printed. This is useful when working with lattice or ggplot2 graphics.

## 4 Helpers

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

- splat() creates a new function that uses do.call(). It converts a function that taking multiple arguments to one that takes a single argument, a list. 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 frame. 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. Alternatively, you can provide a vector of column names, and colwise() only operate on those columns.

- 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 notification of the error when 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 different progress bars:

- "none", the default. No progress bar is displayed.
- "text" provides a textual progress bar.
- "win" and "tk" provide graphical progress bars for Windows and systems with the tcl/tk package (the mac and most linux platforms).

The progress bars assume that processing each piece takes the same amount of time, so will not be 100% accurate. Psychologically, adding a progress bar to a long-running process makes it feel like it takes much less time.

## 5 Strategy

Having learned the basic structure and operation of the `plyr` family of functions, you will now see some examples of using them in practice. The following two case studies explore two data sets: a data frame of batting records from long-term baseball players, and a 3d array recording ozone measurements that vary over space and time. Neither of these data studies do more than scratch the surface of possible analyses, but do show of a number of different ways to use `plyr`.

Both cases follow a similar process:

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

The code shown in this paper is necessarily abbreviated. The data sets are large, and often only small subsets of the data are shown. The code focuses on data manipulation, and much of the graphics code is omitted. You are encouraged to experiment with the full code yourself, available as a supplement to this paper.

### 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 is described fully in `?baseball`, but for this example we will focus on just four: `id`, which identifies the player, `year` the year of the record; `rbi`, runs batted in, the number of runs that the player made in the season; and `ab`, at bat or the number of times the player faced a pitcher.

What we'll explore is the performance of a batter over his career. To get started, we need to calculate the "career year", 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 <- transform(baberuth, cyear = year - min(year) + 1)
```

To do this for all players, we don't need to write our own function, because we can apply `transform()` to each piece:

```
> baseball <- ddply(baseball, .(id), transform,
+   cyear = year - min(year) + 1)
```

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 8, 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. We also restrict the data to focus only on records where `ab` is greater than 25: this prevents problems with a small number on the denominator.

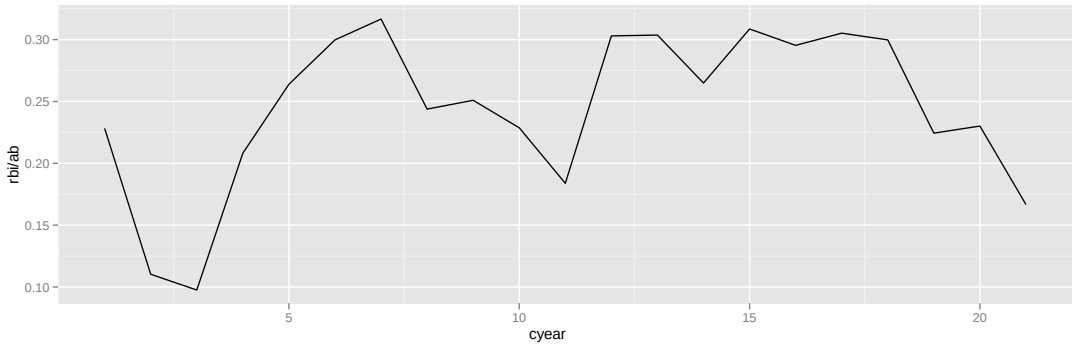


Figure 8: Runs per bat for Babe Ruth.

```
baseball <- subset(baseball, ab >= 25)
xlim <- range(baseball$cyear, na.rm=TRUE)
ylim <- range(baseball$rbi / baseball$ab, na.rm=TRUE)
plotpattern <- function(df) {
  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),
  print. = TRUE)
dev.off()
```

Flicking through the 1145 plots reveals few common patterns, 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. (Not recommended in practice!)

```
> model <- function(df) {
+   lm(rbi / ab ~ cyear, data=df)
+ }
> model(baberuth)

Call:
lm(formula = rbi/ab ~ cyear, data = df)

Coefficients:
(Intercept)        cyear
    0.20797      0.00332

> bmodels <- 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 ($R^2$) so we can ensure we're not drawing conclusions based on models that fit the data very poorly. The first few rows of `coef` are shown in Table 3.

```
> rsq <- function(x) summary(x)$r.squared
> bcoefs <- ldply(bmodels, function(x) c(coef(x), rsquare = rsq(x)))
> names(bcoefs)[2:3] <- c("intercept", "slope")
```

| 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 `bcoefs` data frame. Note that the player ids from the original data have been preserved

Figure 9 displays the distribution of R-squared across the models. The models generally do a very bad job of fitting the data, although there are few with an $R^2$ very close to 1. We can see the data that generated these perfect fits by merging the coefficients with the original data, and then selecting records with an $R^2$ of 1:

```
> baseballcoef <- merge(baseball, bcoefs, by = "id")
> subset(baseballcoef, rsquare == 1)$id
 [1] "bannifl01" "bannifl01" "bedrost01" "bedrost01" "burbada01" "burbada01"
 [7] "carrocl02" "carrocl02" "cookde01"  "cookde01"  "davisma01" "davisma01"
[13] "jacksgr01" "jacksgr01" "lindbpa01" "lindbpa01" "oliveda02" "oliveda02"
[19] "penaal01"  "penaal01"  "powerte01" "powerte01" "splitpa01" "splitpa01"
[25] "violafr01" "violafr01" "wakefti01" "wakefti01" "weathda01" "weathda01"
[31] "woodwi01"  "woodwi01"
```

All the models with a perfect fit only have two data points. Figure 10 is another attempt to summarise the models. These plots show a negative correlation between slope and intercept, and the particularly bad models have estimates for both values close to 0.

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` made it very easy to work at the player level. This is an sensible first step when creating a hierarchical model.

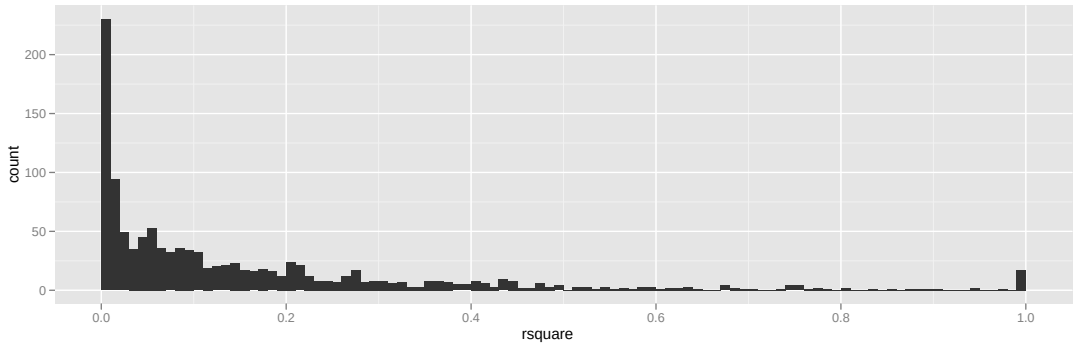


Figure 9: Histogram of model R-squared with bin width of 0.05. Most models fit very poorly!

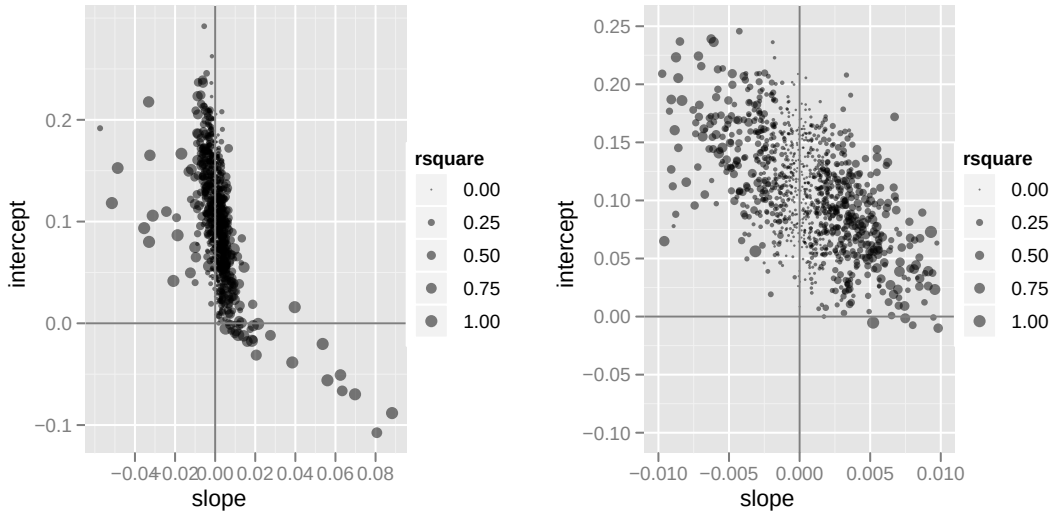


Figure 10: A scatterplot of model intercept and slope, with one point for each model (player). The size of the points is proportional to the R-square of the model. Vertical and horizontal lines emphasise the x and y origins.

### 5.2 Case study: ozone

In this case study we will analyse a 3d array that records ozone levels over a $24 \times 24$ 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 11 shows one way of displaying this data. Conditional on spatial location, each star glyph shows the evolution of ozone levels for each of the 72 months (6 years). The construction of the glyph is described in Figure 12; it is basically a time series in polar coordinates. 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.

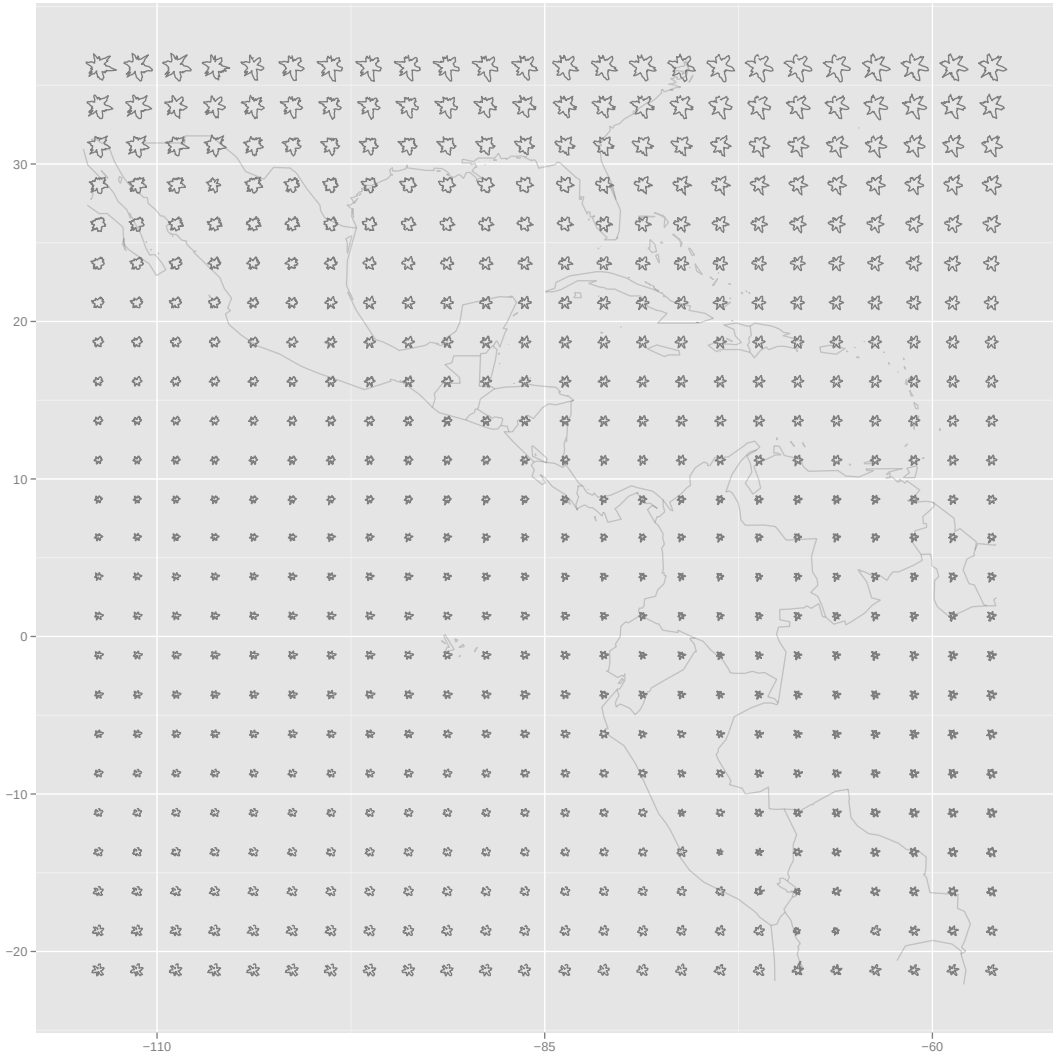


Figure 11: Star glyphs showing variation in ozone over time at each spatial location.

Again we will start with the simplest case: a single time point, from location (1, 1). Figure 13 displays this in two ways: as a single line over time, or a line for each year over the months. This plot illustrates the striking seasonal variation at this time point. The following code sets up some useful variables.

```
> value <- ozone[1, 1, ]
```

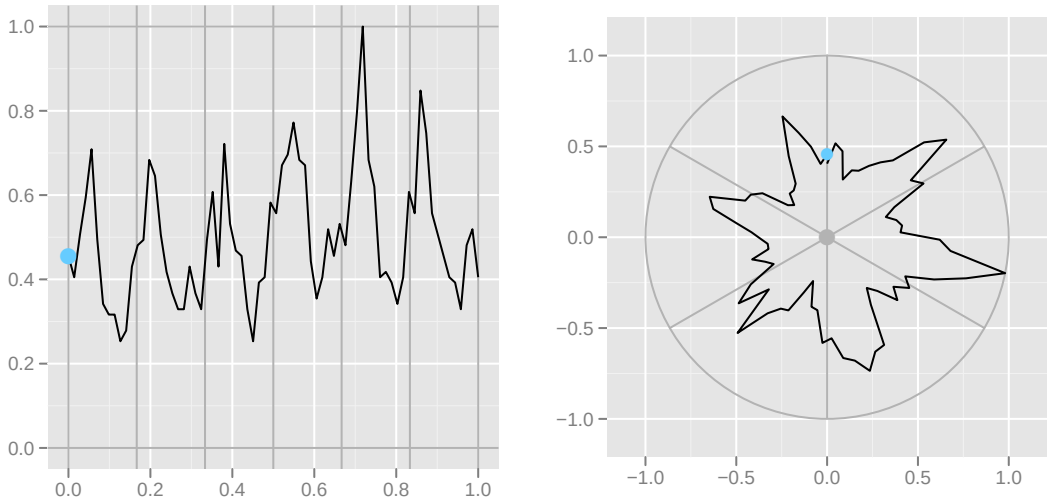


Figure 12: Star glyphs are time-series (left) plotted in polar coordinates (right). Both time and ozone value have been scaled to lie between 0 and 1: the smallest value in the entire dataset will be 0 and the largest will be 1. Grey lines indicate these boundaries, as well as the boundaries between years. A red point shows the position of the first value: it is close to the last value in the glyph. This glyph is the glyph on the top-left of Figure 11.

```
> 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 that predicts the amount of ozone for each month. We could use a regular linear model, but then our seasonal estimates might be thrown off by an unusual month. Figure 14 shows the deseasonalised trend from location (1, 1).

```
> library(MASS)
>
> 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
```

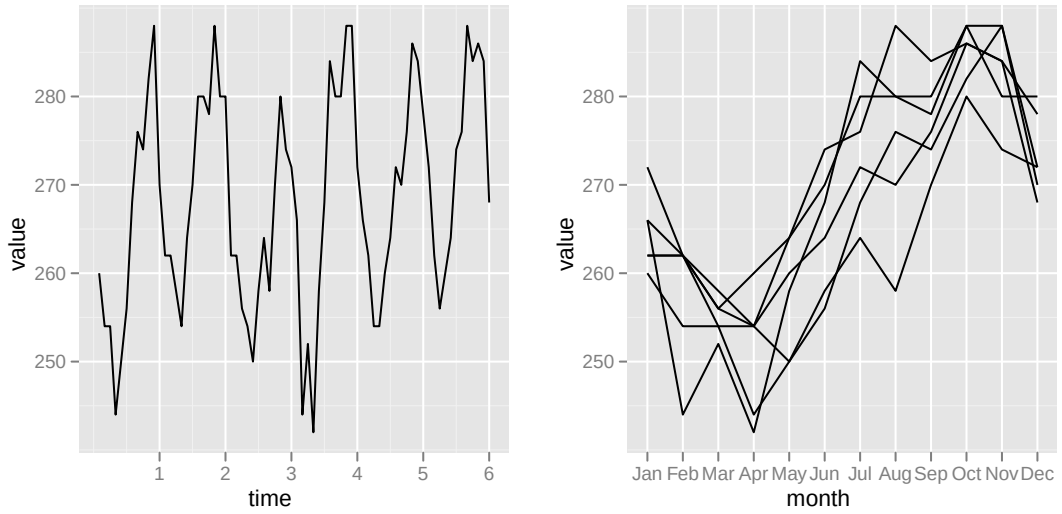


Figure 13: Two ways of displaying the seasonal changes. (Left) A single time series over all six years and (right) a line for each year.

```
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
```

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! As is common when fitting large numbers of models, some of the models don't fit very well, and `rlm()` does not converge. We figure out where these lie by looking at the `converged` attribute for each model. In a real analysis it would be important to figure why these locations are troublesome and deal with them appropriately, but here we'll just ignore them.

```
> deseasf <- function(value) rlm(value ~ month - 1, maxit = 50)
> models <- alply(ozone, 1:2, deseasf)
WARNING: rlm failed to converge in 50 steps
WARNING: rlm failed to converge in 50 steps
WARNING: rlm failed to converge in 50 steps
WARNING: rlm failed to converge in 50 steps
```

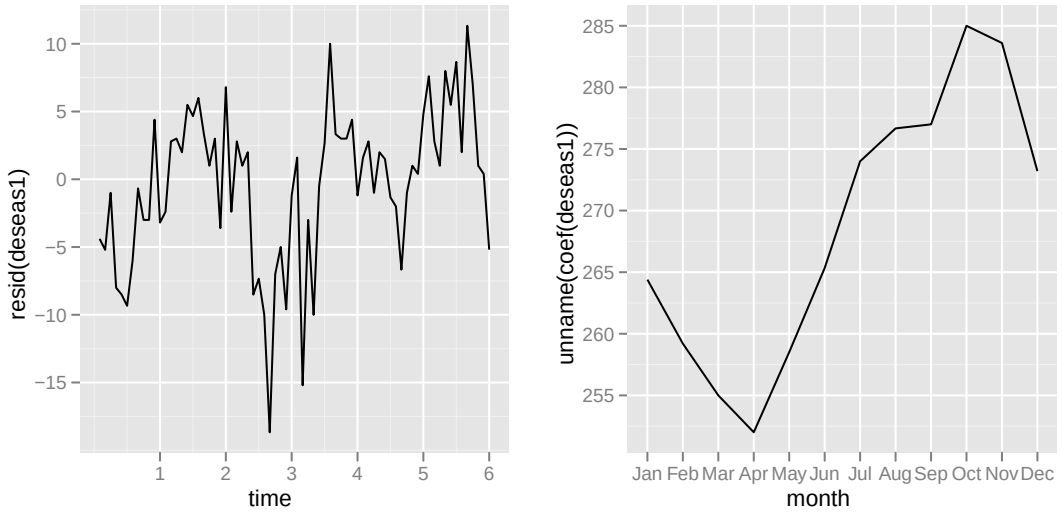


Figure 14: Deasonalised ozone trends. (Left) deasonalised trend over six years. (Right) Estimates of seasonal effects. Compare to Figure 13

```
WARNING: rlm failed to converge in 50 steps
WARNING: rlm failed to converge in 50 steps
WARNING: rlm failed to converge in 50 steps
> failed <- laply(models, function(x) !x$converged)
```

From those models we extract the deseasonalised 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.

```
> 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 15 and 16 visualise these results with star glyph plots. For plotting, it's more convenient to have the data in data frames. There are a 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 effort 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)
  month   lat long value avg   std
1   May -21.2 -114   264 269 0.928
2   Apr -21.2 -114   259 269 0.909
3   Aug -21.2 -114   255 269 0.895
4   Jan -21.2 -114   252 269 0.884
5   Sep -21.2 -114   259 269 0.907
6   Jul -21.2 -114   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 $24 \times 24$ grid is coloured according to its value. The following code sets up a function with constant scales to do that. Figure 17 shows the spatial variation of seasonal coefficients for January and July.

```
> 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 = brightblue, high = "yellow") +
+     map + opts(aspect.ratio = 1)
+ }
```

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. The `*_ply` functions are useful for making animations because we are only calling the plotting function for its side effects, not because we're interested in its value.

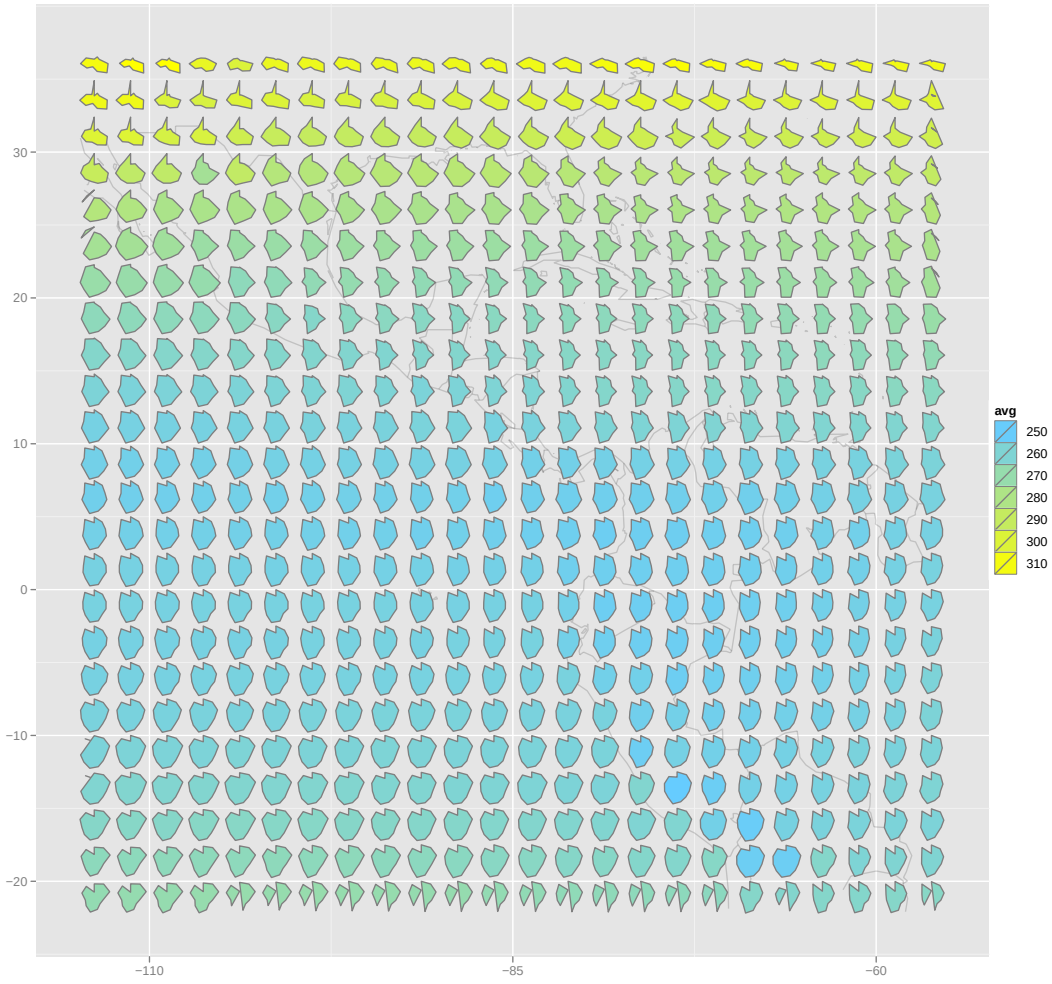


Figure 15: Star glyphs showing seasonal variation. Estimates of seasonal effects are standardised to have the same maximum at each location to make it easier to compare the general pattern. The glyph colours give the overall average ozone measurement. Note the strong spatial correlation: nearby glyphs have similar shapes.

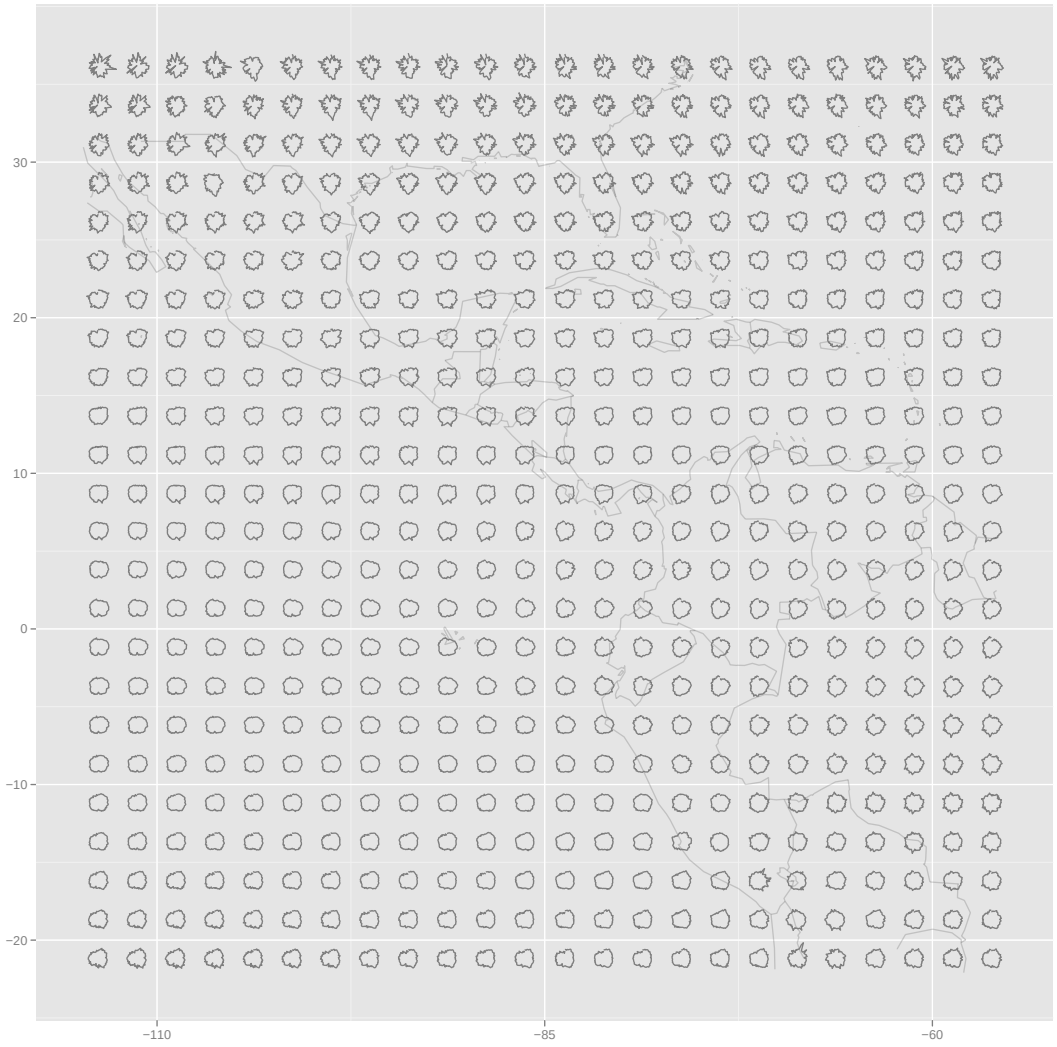


Figure 16: Star glyphs showing deasonalised trends. This plot contains a lot of data—over 40,000 observations—and rewards detailed study. Looking at a printed version also helps as the resolution of a printer (600 dpi) is much higher than that of the screen (1̃00 dpi).

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

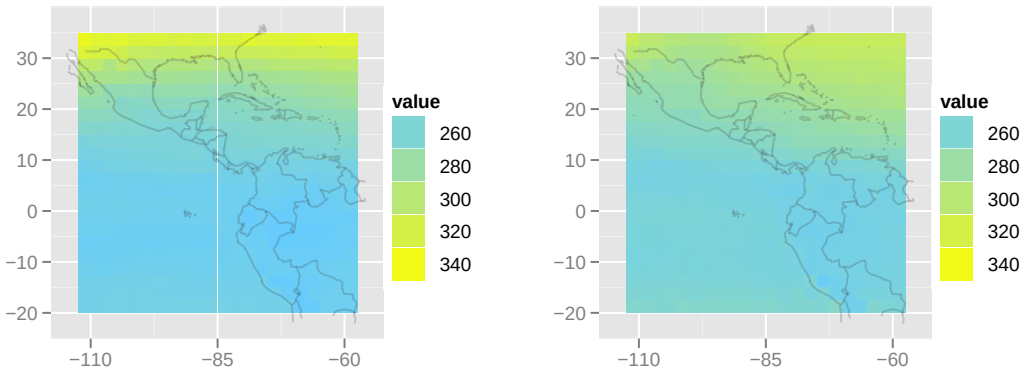


Figure 17: Tile plots of coefficients for January (left) and July (right).

### 5.3 Other uses

The `transform()` and `subset()` functions work will in combination with `plyr`. Transform makes it very easy to perform randomisation within groups. For example, if we wanted to break the dependence between ozone values and time within each spatial location, we could do:

```
ddply(coefs_df, .(lat, long), transform, time = sample(time))
```

This technique is useful for performing block bootstrapping and other related permutation tests, and is related to the `ave` function in base R. Scaling variables within a group is also trivial:

```
ddply(coefs_df, .(lat, long), transform, value = scale(value))
```

If we wanted to extract the observation in each group with the lowest value ozone of ozone, it's just as easy:

```
ddply(coefs_df, .(lat, long), subset, value == min(value))
```

For simulations, `mdply()` can be very useful, because it's easy to generate a grid of parameter values and then evaluate them. This can also be useful when testing many possible combinations to input to a function.

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

## 6 Related work

There are a number of other approaches to solving the problems that plyr solves. You can always use loops, but loops create a lot of book-keeping code that obscures the intent of your algorithm. This section describes other high-level approaches similar to plyr.

Table 4 describes the functions in base R that work similarly to functions in plyr. The built-in R functions focus mainly on arrays and lists, not data frames, and most attempt to return an atomic data structure if possible, and if not, a list. This ambiguity of the output type is fine for interactive use, but does make programming with these functions tricky. Compared to aaply, apply returns the new dimensions first, rather than last, which means it is not idempotent when used with the identity function. In contrast, aaply(x, a, identity) == aperm(x, a) for all a.

| Base function | Input | Output | plyr function |
|---|---|---|---|
| aggregate | d | d | ddply + colwise |
| apply | a | a/l | aaply / alply |
| 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.

Related functions tapply and sweep have no corresponding function in plyr, and remain useful. merge is useful for combining summaries with the original data.

Contributed packages also tackle this problem:

- 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 with a formula based interface, which, particularly for summary, makes it easy to only operate on selected columns.
- The gdata (Warnes and Gorjanc., 2008) package contains a bundle of helpful data manipulation functions, including frameApply which works like ddply or dlply depending on its arguments.
- The scope (Bergsma, 2007) package provides scope, scoop, skim, score and probe which provide a composable set of functions for operating symbolically on subsets of data frames.
- The reshape (Wickham, 2005) package is similar to Excel pivot tables and provides tools for rearranging matrices and data frames. The cast function in the reshape package is closely related to aaply.
- The sqldf (Grothendieck, 2008) package allows you to use SQL commands with R data frames. This gives the user access to a powerful set-based data access language.

## 7 Conclusion

Speed-wise `plyr` is competitive with R for small to moderate sized datasets, and generally a little faster for large datasets split by many different values. It is more memory-efficient than the naïve split-apply-combine approach because `plyr` is careful not to make an extra copy of the data in the split step. Further efficiency gains are possible, particularly by implementing key parts C for maximum speed and memory efficiency. The basic algorithm of `plyr` is trivially parallelisable, and future versions will integrate with `multicore` (Urbanek, 2009), `papply` (Currie, 2005) and related packages to make best use of multiple cores and multiple machines.

More generally, what are other common strategies used in data analysis? How can we identify these strategies and then develop software to support them? It is difficult to step back and identify these patterns as trivial details may obscure the common components; it took four years of thinking about related problems before I recognised this split-apply-combine strategy. However, the task is important because the patterns are so useful. Personally, identifying the split-apply-combine strategy has made it much easier for me to solve common data analysis problems, and I have also found useful when teaching others how to do data analysis.

## 8 Acknowledgements

Thanks go to Norman Josephy, Austin F Frank, Antony Unwin, Joseph Voelkel, Erik Iverson, and Jean-Olivier Irisson for their comments on early versions of this paper.

## 9 Supplemental files

**plyr.r** R code used in this paper.

**ozone-map.r** Functions to produce maps of ozone values.

## 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.

M. Friendly and J. Fox. Using APL2 to create an object-oriented environment for statistical computation. *Journal of Computational and Graphical Statistics*, 3:387–407, 1994.

G. Grothendieck. *sqldf: Perform SQL Selects on R Data Frames*, 2008. URL `http://code.google.com/p/sqldf`. R package version 0-1.4.

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.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2009. URL http://www.R-project.org. ISBN 3-900051-07-0.

S. Urbanek. *multicore: Parallel processing of R code on machines with multiple cores or CPUs*, 2009. URL http://www.rforge.net/multicore/. R package version 0.1-3.

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.