plyr: divide and conquer

Hadley Wickham

August 18, 2008

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

This section motivates the use of `plyr` by previewing one of the case studies. Ozone measurements are stored in a 3d array: the first two dimensions are location and the third is time. We want to remove seasonal effects, keeping the data in the same form. We also want to keep the models in 2d array so we can reference a local model in a similar way to referencing a local time series. Here I'll just use the code with little explanation, but hopefully you will get some hint of the succinctness of plyr for these tasks. This is example is explained in more detail in Section 5.2.

```
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)

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

Another way of storing the ozone data would be stored in a data frame with columns lat, long, time, month, and value. `plyr` has been written to work well with the three major R data structures: arrays, lists and data frames. 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 data in this format is often ragged, there's no particular advantage to using the for loops here, and we'll use `split()`, `lapply()` and `mapply()` to complete the task. Here the split-apply-combine strategy maps closely to R functions: we split with `split()`, apply with `lapply()` and then combine the pieces into a single result with `rbind()`.

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.

```
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 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)
```

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 array made out of a list, i.e. an list that has had dimensions imposed upon it with `dim`. (as opposed to an atomic vector, as is more usual).

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).

Plyr 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. Note that arguments to the `**ply` functions end in "`.`" - this is to prevent clashes with argument names in the processing function.

The `progress.` argument controls displaying of a progress bar, and is described at the end of Section 4.

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 split into pieces based on 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!)

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

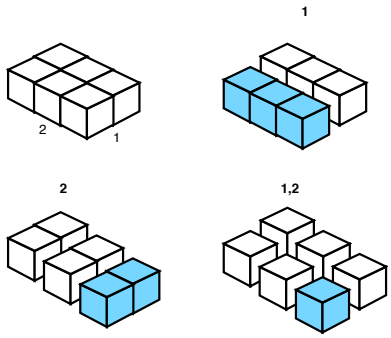

Figure 1: The four ways to split up a 2d matrix. Original matrix shown at top left, with dimensions labelled. Blue indicates a single piece of the output.

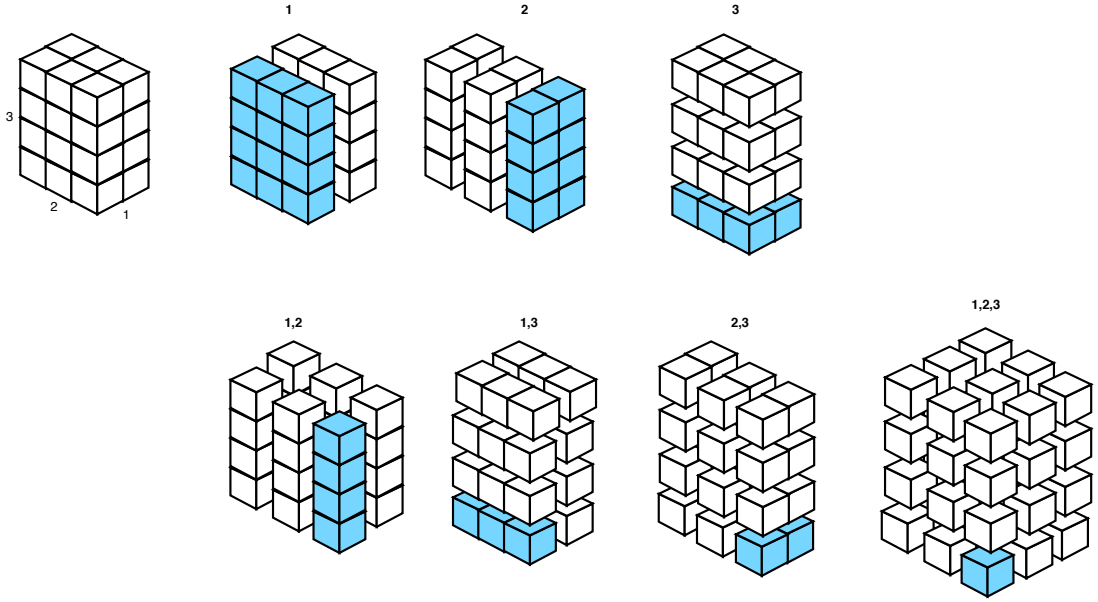

Figure 2: The eight ways to split up a 3d array. Original array shown at top left, with dimensions labelled. Blue indicates a single piece of the output.

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`. `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.

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).

- The interaction of multiple variables are taken: `.(a, b, c)` breaks the data frame into the same groups that `interaction(a, b, c)` would, but labels rows with all three variables.
- Functions of variables: `.(round(a))`, `.(a * b)`
- Variables in the global environment `.(anothervar)`

You can override the default names by using a named list:

- `.(first = a, second = b, third = c)`
- `.(product = a * b)`

3.1.3 Input: list (`l*ply`)

Processing lists is the simplest

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.

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	`NA`

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

3.2.1 Output: array (`*aply`)

With array output the dimensionality is determined by the input splits.

- A list will produce a single dimension
- a data frame will have a dimension for each variable split on
- a array will have a dimension for each dimension that it was split on

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, `adply` will return a logical vector of length 0.

`*aply()` also has 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.

Figures 3 and 4 illustrate

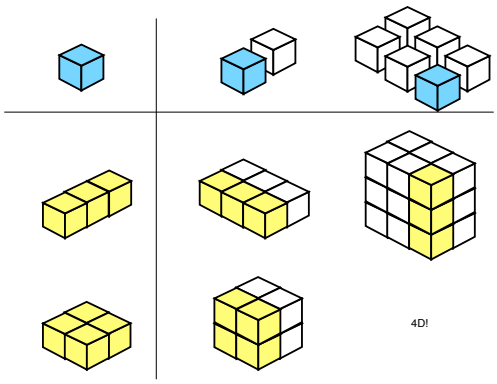

Figure 3: Results from outputs of various dimensionalty from a single value, shown top left. Columns indicate input: (left) a vector of length two, and (right) a 2*times*2 matrix. Rows indicated the shape of a single processed piece: (top) a vector of length 3, (bottom) a 2×2 matrix. Extra dimensions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

The dimnames of the array will be the same as the input, if an array, or the extracted from the subsets if a data frame.

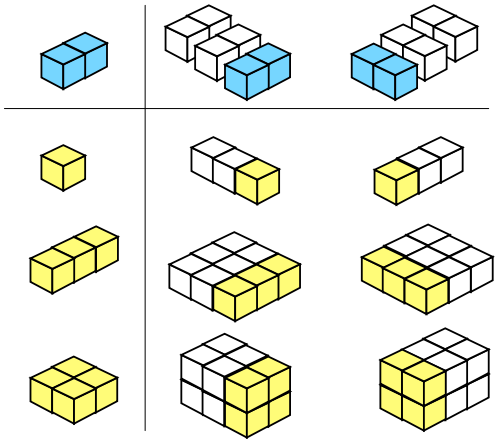

Figure 4: Results from outputs of various dimensionalty from a 1d vector, shown top left. Columns indicate input: (left) a 2×3 matrix split by rows and (right) and 3×2 matrix split by columns. Rows indicate the shape of a single processed piece: (top) a single value, (middle) a vector of length 3, and (bottom) a 2×2 matrix.

3.2.2 Output: data frames (`*dply`)

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.

The output data frame will be supplemented with columns that identify the subset of the original dataset that each piece was computed from. These columns make it easier to merge the old and new data. If the input was a data frame, this will be the values of the splitting variables. If the input was an array, this will be the dimension names.

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`.

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.

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 5, 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.

```
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) {
```

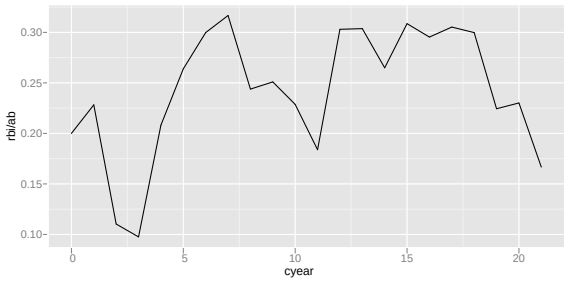

Figure 5: Runs per bat for Babe Ruth.

```
  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")
```

Figure 6 displays the distribution of r-squared across the models. The models generally do a very bad job of fitting the data! Figure 7 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.

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, and then combine results into a single summary.

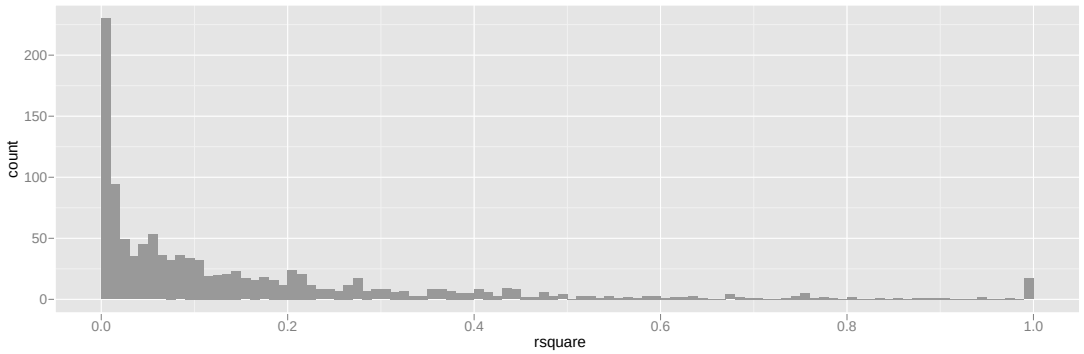

Figure 6: Histogram of model r-squared with bin width of 0.05. Most models fit very poorly! The spike of models with a r-squared of 1 are players with only two data points, found by inspecting `ldply(models[coef$rsquare == 1], "[[", "model")`

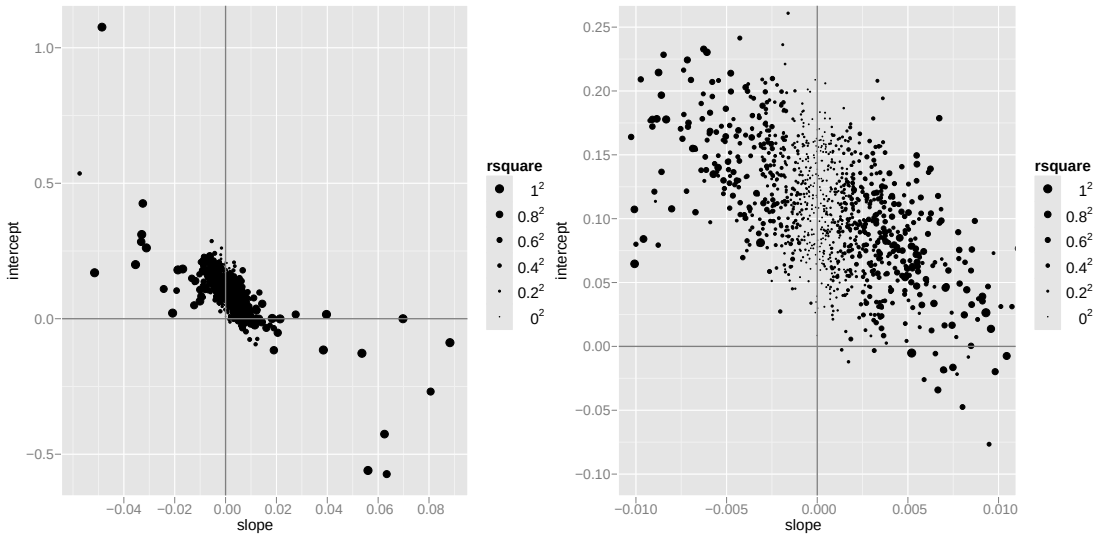

Figure 7: A scatterplot of model intercept and slope, with one point for each model (player). The size of the points is proportion to the R-square of the model. Vertical and horizontal lines emphasis the x and y origins.

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

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 8 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.

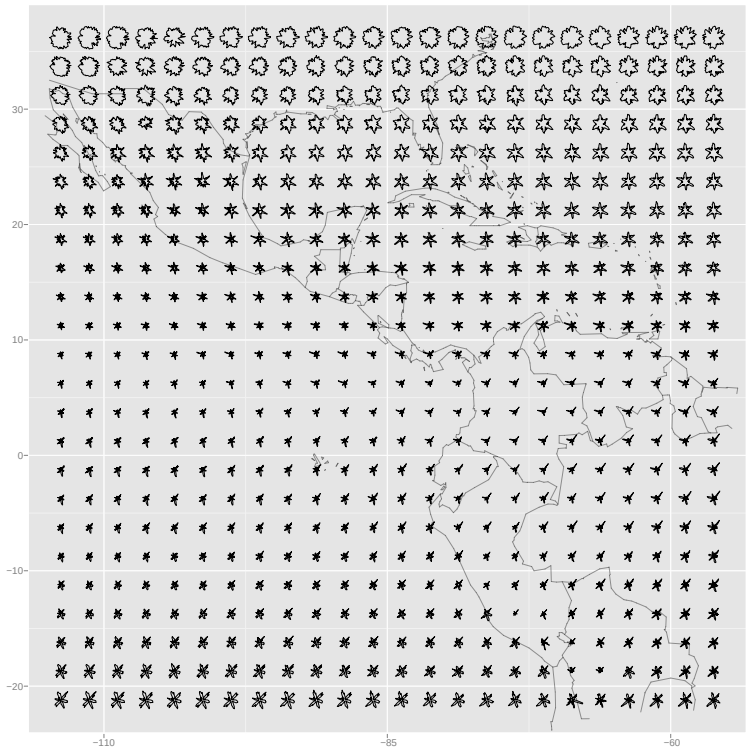

Figure 8: Star glyphs showing variation in ozone over time at each spatial location.

Again we will start with the simplest case: a single time point. We can display 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.

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

```
time <- 1:72
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)

qplot(time, value, geom="line")
qplot(month, value, geom="line", group = year)
```

We are going to use a very crude method to remove the season variation: we'll look at the residuals from a robust linear model.

```
library(MASS)

deseas <- rlm(value ~ month - 1)
qplot(month, resid(deseas), geom="line", group = year)
qplot(month, unname(coef(deseas)), geom="line", group = 1)
```

We next turn this into a function and apply to all of the spatial locations:

```
source("ozone-map.r")
deseasf <- function(value) rlm(value ~ month - 1)

models <- alply(ozone, 1:2, deseasf)
coefs <- laply(models, coef)
deseas <- laply(models, resid)

coef2 <- ldply(models, function(x)
  data.frame(month = factor(month.abbr, levels=month.abbr), coef = unname(coef(x)))
)

qplot(month, coef, data=coef2, geom="line", group=interaction(lat,long))

qplot(long, lat, data=subset(coef2, month =="Jan"), fill = coef, geom="tile") + map

coef_limits <- range(coef2$coef)
coef_mid <- mean(coef2$coef)
monthsurface <- function(mon) {
  df <- subset(coef2, month == mon)
  qplot(long, lat, data = df, fill = coef, geom="tile", main = mon) +
  scale_fill_gradient2(limits = coef_limits, midpoint = coef_mid) + map
}

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

seassum <- ddply(coef2, .(lat, long), function(df) {
```

```
  each(mean, sd)(df$coef)
})
qplot(long, lat, data=seassum, fill = sd, geom="tile")
qplot(long, lat, data=seassum, fill = mean, geom="tile")
qplot(mean, sd, data=seassum)

ozm <- melt(deseas)
names(ozm) <- c("lat", "long", "time", "value")
ozm[1:2] <- llply(ozm[1:2], function(x) as.numeric(as.character(x)))

small_mult(ozm) + geom_line(aes(group = interaction(lat,long)))

ozm <- melt(coef2)
ozm[1:2] <- llply(ozm[1:2], function(x) as.numeric(as.character(x)))
ozm$time <- as.numeric(ozm$month)

small_mult(ozm) + geom_line(aes(group = interaction(lat,long)))
```

For many other types of operations, it is useful to convert this array structure to a data frame. The `melt` function in the `reshape` package is one way to do that which preserves the dimension labels as much as possible. This is the power of plyr: you don't need to worry about whether your data is a list, data frame or array, you can use whatever feels most natural.

```
library(reshape)
ozonem <- melt(ozone)

models <- dlply(ozone, .(lat, long), deasf)
coefs <- ldply(models, coef)
deseas <- ldply(models, resid)
```

5.3 Other uses

Randomisation within groups. Simulation.

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. 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 `apply`, `aaply` returns the dimensions in a different order so as to be idempotent - i.e. `apply(x, a, function(x) x) == x` for all `a`.

Avoid any ambiguity about what you'll get back from one of these functions. This replaces the `simplify` argument that many of the `apply` functions in base R has, and means that you can depend on the output of each function being a given type (which makes programming with the results easier).

base	from	to	plyr
`apply`	a	a	`aaply`
`lapply`	l	l	`llply`
`sapply`	l	a	`laply`
`mapply`	a	a/l	`maply` / `mlply`
`by`	d	l	`dlply`
`aggregate`	d	d	`ddply` + `colwise`

Table 4: Mapping between apply functions and plyr functions.

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`.

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

- The `doBy` package
- The `gdata` package
- The `scope` package
- Data manipulation in R, by Phil Spector
- Chapters in MASS, R intro?

7 Future plans

If slow, might want to look at the profr package to speed up.

However, 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 also plan to investigate a connection to the `papply` function to allow for easy parallelisation across multiple instances of R (particularly for multi-core machines).

multir

References

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.

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