SECTION 1-B

1.2 Data Visualization with R

It is very important to gain a feel for the data that we are investigating.
One way to do this is by visualization.
We will do this by starting with a simple dataset that has some nice features.

Flea Beetles

This data is from a paper by A. A. Lubischew, "*On the Use of Discriminant Functions in Taxonomy*", Biometrics, Dec 1962, pp.455-477.
There are three species of flea-beetles: **C**. **concinna**, **Hp**. **heptapotamica**, and **Hk**. **heikertingeri**, and 6 measurements on each.

tars1 - width of the first joint of the first tarsus in microns (the sum of measurements for both tarsi).
tars2 - the same for the second joint.
head - the maximal width of the head between the external edges of the eyes in 0.01 mm.
aede1 - the maximal width of the aedeagus in the fore-part in microns.
aede2 - the front angle of the aedeagus (1 unit = 7.5 degrees).
aede3 - the aedeagus width from the side in microns.

1.2.1 Reading Data

The first thing we have to do is get the data (for this data set we will read in a text file). The following illustrates how to read the file(s). (Note the use of the UNIX type path separator with `/` rather than `\`.

```
drive <- "I:"
code.dir <- paste(drive, "DataMining", "Code", sep="/")
data.dir <- paste(drive, "DataMining", "Data", sep="/")
   # Set the files to be read
d.file <- paste(data.dir, "fleas", "flea.dat", sep="/")
[1] "I:/DataMining/Data/fleas/flea.dat"
d.col <- paste(data.dir, "fleas", "flea.col", sep="/")
[1] "I:/DataMining/Data/fleas/flea.col"
```

We now have paths for two files: `d.file` points to the data and `d.col` points to the column headers (variable names).

The function `scan` can be used to read in the data. If the data has characters in it, we need to indicate that with `""`

```
(headers <- scan(d.col))
Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings, :
        scan() expected 'a real', got 'tars1'
(headers <- scan(d.col, ""))
Read 6 items
[1] "tars1" "tars2" "head" "aede1" "aede2" "aede3"
(n.var <- length(headers))  # The vector length gives the number of variables
[1] 6
```

For reading in the data we can do a `scan` to read the data into a vector, and then convert the vector to a matrix with `n.var` columns. Because data files are typically stored by rows and R does the conversion to a matrix by column, we need to indicate that with `byrow=T`.

```
d.flea.s <- matrix(scan(d.file), ncol=n.var, byrow=T)
Read 444 items
d.flea.s[1:5,]    # This displays the first 5 rows and all the columns.
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  191  131   53  150   15  104
[2,]  185  134   50  147   13  105
[3,]  200  137   52  144   14  102
[4,]  173  127   50  144   16   97
[5,]  171  118   49  153   13  106
```

Note that if the file contains a mixture of numbers and text, we need to use the `""` in the scan

```
d.flea.str <- matrix(scan(d.file, ""), ncol=n.var, byrow=T)
Read 444 items
d.flea.str[1:5,]
     [,1]  [,2]  [,3] [,4]  [,5] [,6]
[1,] "191" "131" "53" "150" "15" "104"
[2,] "185" "134" "50" "147" "13" "105"
[3,] "200" "137" "52" "144" "14" "102"
[4,] "173" "127" "50" "144" "16" "97"
[5,] "171" "118" "49" "153" "13" "106"
```

We can see that the numbers are read in as strings. In order to do arithmetic on them, they have to be converted to numbers.

The command `as.numeric(...)` will convert a string to a number

```
as.numeric(d.flea.str[1,1])
[1] 191
```

but when it is applied to an array it makes the array into a vector.

To correct this we could try

```
d.flea.s <- matrix(as.numeric(d.flea.str), ncol=n.var)
d.flea.s[1:5,]
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  191  131   53  150   15  104
[2,]  185  134   50  147   13  105
[3,]  200  137   52  144   14  102
```

```
[4,]  173  127   50  144   16   97
[5,]  171  118   49  153   13  106
```

which appears to correct the problem.

A better way in many cases is to read the data as a *table*

```
d.flea <- read.table(d.file)
d.flea[1:5,]
   V1  V2 V3  V4 V5  V6
1 191 131 53 150 15 104
2 185 134 50 147 13 105
3 200 137 52 144 14 102
4 173 127 50 144 16  97
5 171 118 49 153 13 106
```

Note the different form for the row and column headers.
The latter form is better in that it gives 'names' to the rows and columns - not just positions.

Consider taking a sub-matrix

```
d.flea.s[c(5,10,15,20), c(2,4)]
     [,1] [,2]
[1,]  118  153
[2,]  115  142
[3,]  130  147
[4,]  121  147
```

We have no way of identifying from where the entries came. On the other hand

```
d.flea[c(5,10,15,20), c(2,4)]
    V2   V4
5  118  153
10 115  142
15 130  147
20 121  147
```

retains the information as to the rows and columns.

It is possible to improve on the first case by assigning row and column name information

```
dimnames(d.flea.s) <- list(1:dim(d.flea.s)[1], headers)
```

To see what is happening in this consider

```
dim(d.flea.s)
[1] 74  6
```

This gives the dimension of the matrix so `1:dim(d.flea.s)[1]` creates a vector of integers from 1 to the number of rows in the matrix. `dimnames` assigns the values in the `list` as row and column labels.

```
d.flea.s[c(5,10,15,20),]
   tars2 aede1
5    118   153
10   115   142
15   130   147
20   121   147
```

This is much more useful and is similar to the information displayed by the *data frame* version. In fact it is more informative because it uses the true header information. To improve the data frame we

can replace the generic column headers by the correct values (the row headers are good) by

```
colnames(d.flea) <- headers
d.flea[c(5,10,15,20), c(2,4)]
   tars2 aede1
5    118   153
10   115   142
15   130   147
20   121   147
```

Now that we have all the data, we can use further information to specify the species for the cases. For some purposes we may find it best to have characters to represent the species while for others, numerical values may be best. We will create both

```
flea.species <- c(rep("C",21),rep("Hp",22),rep("Hk",31))
species <- c(rep(1,21),rep(2,22),rep(3,31))
```

Here we have used information that was not contained in the data to set the species. This information is found in

```
(d.row <- paste(d.data.dir, d.basename, ".row", sep = ""))
(row.headers <- noquote(scan(d.row, "")))
Read 74 items
 [1] Concinna  Concinna  Concinna  Concinna  Concinna  Concinna  Concinna
 [8] Concinna  Concinna  Concinna  Concinna  Concinna  Concinna  Concinna
[15] Concinna  Concinna  Concinna  Concinna  Concinna  Concinna  Concinna
[22] Heptapot. Heptapot. Heptapot. Heptapot. Heptapot. Heptapot. Heptapot.
[29] Heptapot. Heptapot. Heptapot. Heptapot. Heptapot. Heptapot. Heptapot.
[36] Heptapot. Heptapot. Heptapot. Heptapot. Heptapot. Heptapot. Heptapot.
[43] Heptapot. Heikert.  Heikert.  Heikert.  Heikert.  Heikert.  Heikert.
[50] Heikert.  Heikert.  Heikert.  Heikert.  Heikert.  Heikert.  Heikert.
[57] Heikert.  Heikert.  Heikert.  Heikert.  Heikert.  Heikert.  Heikert.
[64] Heikert.  Heikert.  Heikert.  Heikert.  Heikert.  Heikert.  Heikert.
[71] Heikert.  Heikert.  Heikert.  Heikert.
```

A further refinement is to bind things together in what is called a `data frame`. As it happens, the table version is a data frame.

```
is.data.frame(d.flea)
[1] TRUE
```

Many functions require the use of a **data frame**. (It might be best to also bind in the species information, but which one depends on what we are doing.)

```
df.flea <- data.frame(d.flea.s)
```

We can read in some functions that we will use.

```
source(paste(d.code.dir, "DispStr.r", sep = ""))
source(paste(d.code.dir, "pairs_ext.r", sep = ""))
source(paste(d.code.dir, "MakeStereo.r", sep = ""))
```

`source` reads code from a file just as though it had been typed or pasted into R. Now we can look at the data as something other than just numbers - i.e. visualization of the data.

1.2.2 Scatterplot Matrices

The first type of visualization that we will consider is the **scatterplot** matrix.
This plots the variables pairwise and allows us to see if there are some types of relationships between variables by displaying pairwise plots.

We will look at a couple of versions of it.

```
pairs(d.flea)
```

As the figure below shows, there is duplication above and below the diagonal. We can define some functions that will allow us to display more information.

```
panel.smooth.asp <- function (x, y, col = par("col"), bg = NA, pch = par("pch"),
    cex = 1, col.smooth = "red", span = 2/3, iter = 3, asp,...)
{
    points(x, y, pch = pch, col = col, bg = bg, cex = cex, asp = 1)
    ok <- is.finite(x) & is.finite(y)
    if (any(ok))
        lines(lowess(x[ok], y[ok], f = span, iter = iter), col = col.smooth,...)
}
     ## put (absolute) correlations on the upper panels,
     ## with size proportional to the correlations.
panel.cor <- function(x, y, digits = 2, prefix = "", cex.cor) {
   usr <- par("usr"); on.exit(par(usr))
   par(usr = c(0, 1, 0, 1))
   r <- abs(cor(x, y))
   txt <- format(c(r, 0.123456789), digits=digits)[1]
   txt <- paste(prefix, txt, sep = "")
   if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)*r
   text(0.5, 0.5, txt, cex = cex.cor)
}
     ## put histograms on the diagonal
panel.hist <- function(x, ...) {
   usr <- par("usr"); on.exit(par(usr))
   par(usr = c(usr[1:2], 0, 1.5) )
   h <- hist(x, plot = FALSE)
   breaks <- h$breaks; nB <- length(breaks)
   y <- h$counts; y <- y/max(y)
   rect(breaks[-nB], 0, breaks[-1], y, col = "cyan", ...)
}
```

Notice the structure of the function.
The first line says that a function with the arguments (`(x, y, digits = 2, prefix = "", cex.cor)`) is assigned to the name (e.g. `panel.cor`). The body of the function is enclosed by braces `{...}`.

```
pairs(d.flea, upper.panel=panel.cor, diag.panel=panel.hist)
```

.

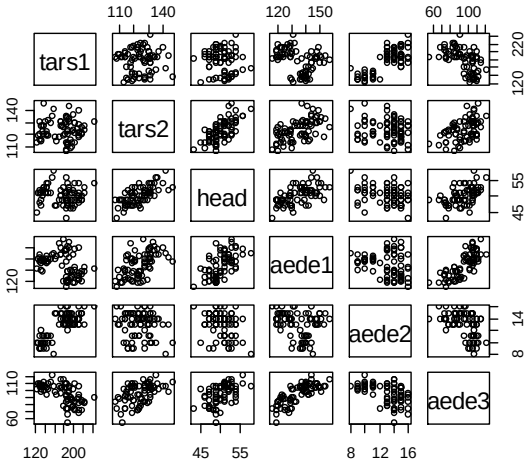


Figure 2. The first command gives the standard `pairs` plot. It shows duplicate plots.

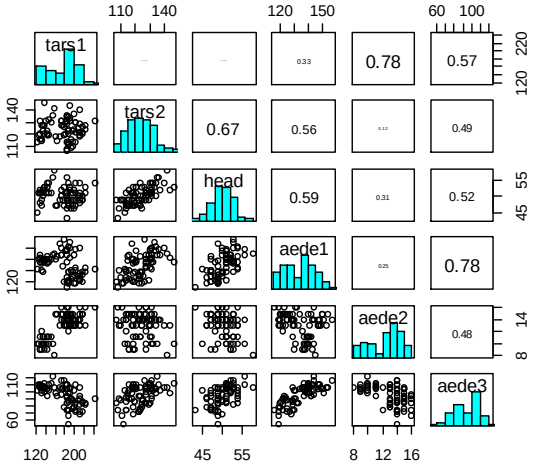


Figure 3. The second command also shows histograms and correlations (the size of the number relates to the degree of correlation)

It should be noted that high correlations (visual or numeric) indicate some linear relationship between pairs of variables but low correlations tell us nothing - the data may be related in a nonlinear fashion or related in combination with other variables.

At this point we might consider how we can see what is going on within these functions. We will consider two closely related methods - `debug(fun)` and `browser()`.

Consider

```
debug(panel.cor)
pairs(d.flea, upper.panel=panel.cor, diag.panel=panel.hist)
debugging in: lower.panel(as.vector(x[, j]), as.vector(x[, i]), ...)
debug: {
    usr <- par("usr")
    on.exit(par(usr))
    par(usr = c(0, 1, 0, 1))
    r <- abs(cor(x, y))
    txt <- format(c(r, 0.123456789), digits = digits)[1]
    txt <- paste(prefix, txt, sep = "")
    if (missing(cex.cor))
        cex.cor <- 0.8/strwidth(txt) * r
    text(0.5, 0.5, txt, cex = cex.cor)
}
```

(On start-up, the function that is being debugged is displayed. We can step through by typing `n` or `next` which executes the current command and displays the next one.)

```
Browse[1]> n
debug: usr <- par("usr")
```

```
Browse[1]> n
debug: on.exit(par(usr))
Browse[1]> n
debug: par(usr = c(0, 1, 0, 1))
Browse[1]> n
debug: r <- abs(cor(x, y))
Browse[1]> n
debug: txt <- format(c(r, 0.123456789), digits = digits)[1]
Browse[1]> x
 [1] 131 134 137 127 118 118 134 129 131 115 143 131 130 133 130 131 127 126 140
[20] 121 136 141 119 130 113 121 115 127 123 119 120 131 127 116 123 135 132 131
[39] 116 121 146 119 127 107 122 114 131 108 118 122 127 125 124 129 126 122 116
[58] 123 122 123 109 124 114 120 114 119 111 112 130 120 119 114 110 124
Browse[1]> y
 [1] 191 185 200 173 171 160 188 186 174 163 190 174 201 190 182 184 177 178 210
[20] 182 186 158 146 151 122 138 132 131 135 125 130 130 138 130 143 154 147 141
[39] 131 144 137 143 135 186 211 201 242 184 211 217 223 208 199 211 218 203 192
[58] 195 211 187 192 223 188 216 185 178 187 187 201 187 210 196 195 187
Browse[1]> r
[1] 0.02634653
Browse[1]> n
debug: txt <- paste(prefix, txt, sep = "")
Browse[1]> n
debug: if (missing(cex.cor)) cex.cor <- 0.8/strwidth(txt) * r
Browse[1]> txt
[1] "0.026"
Browse[1]> Q
```

To stop the debugging, we can type `Q`.

The next time we call a function that has been set for debugging, it will again be debugged. To turn the debugging off, we type

```
undebug(panel.cor)
```

If we wish to debug a function that we have written, we can put a `browser()` statement inside the function body.

The advantage of this is that when we have several spots at which we want to look at the behaviour, we can type `c` to continue the execution until we hit the next `browser()` statement - quite useful with loops.

In this case we know the species corresponding to each case so it is instructive to look at the relationship among the variables and species by the use of colour `col`. The colour can be given by a number or by a name such as `"red"`.

```
pairs(d.flea, col = species + 1)
```

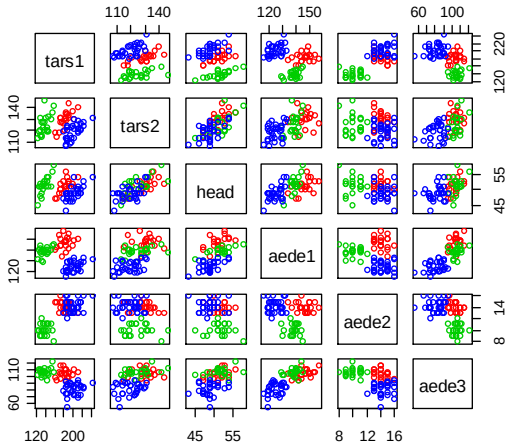


Figure 4.

We notice that in several plots the species are not mixed together e.g. `tars1` vs. `aede2`, `tars1` vs. `aede1` etc.

1.2.3 Conditional Plots

To investigate some of the more complicated relationships we can look at conditional plotting. There is more than one version of this. The first one is part of the standard package. Note the arguments. The `aede3 ~tars1 | aede1` says that we are plotting `aede3` against `tars1` conditioned against `aede1.` That is, we will get several plots corresponding to different ranges of `aede1.` Note that `~` is frequently used to indicate a formula. The `data = df.flea` allows us to use the variable names in the formula because those names are part of the data frame. (The `overlap=0.1` will be explained later.)

```
coplot(aede3 ~tars1 | aede1, data = df.flea)
coplot(aede3 ~tars1 | aede1, data = df.flea, overlap = 0.1)
```

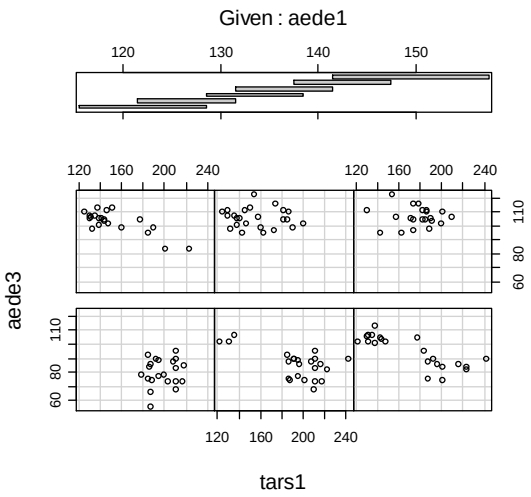


Figure 5.

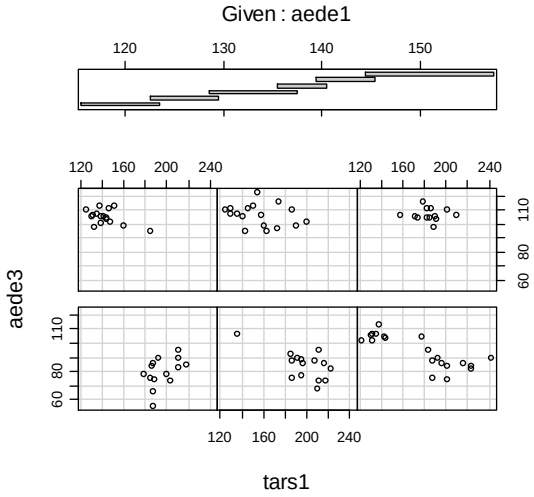


Figure 6.

In both the above figures, the lower six panels show the pairwise plots for `aede3` against `tars1` for different ranges of `aede1` as shown in the upper panel. The defaults for this function are to select 6 different subsets of the third variable with an *equal number of cases in each*. *In addition an overlap of* 0.5 *is allowed*. The second example has reduced the overlap to 0.1. We get a different view if we colour our points by species.

```
coplot(aede3 ~tars1 | aede1, data = df.flea, overlap = 0.1, col = species + 1, pch = 16)
```

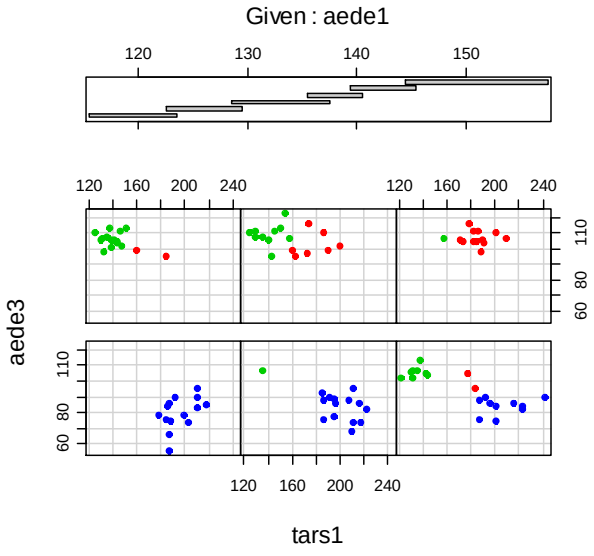


Figure 7.

This further illustrates the relationships noted earlier.

Another version of this is found in the `lattice` package, but before using this we might consider a function that will allow us to condition on fixed interval lengths rather than fixed count.

```
library(lattice)
equal.space <- function(data, count) {
      # range(data) gives the max and min of the variable data.
      # diff takes the difference between the two values so
      # diffs gives the width of each interval.
   diffs <- diff(range(data))/count
      # min(data)+diffs*(0:(count-1)) gives the starting values
      #    for the intervals.
      # min(data)+diffs*(1:count) gives the ending values
      #    for the intervals.
      # cbind treats two(or more) vectors as column vectors
      #    and binds them as columns of a matrix.
   intervals <- cbind(min(data)+diffs*(0:(count-1)),
                      min(data)+diffs*(1:count))
      # shingle takes the interval structure and the data
      #    and breaks the data into the appropriate groups.
   return (shingle(data, intervals))
}
```

The following uses the conditional plotting from the `lattice` package with
a) equal cases in each grouping and
b) equal spacing in each grouping.

```
C1 <- equal.count(df.flea$aede1, number = 6, overlap = 0.1)
xyplot(aede3 ~tars1 | C1, data = df.flea, pch = 19)
C2 <- equal.space(df.flea$aede1, 6)
xyplot(aede3 ~tars1 | C2, data = df.flea, pch = 19)
```

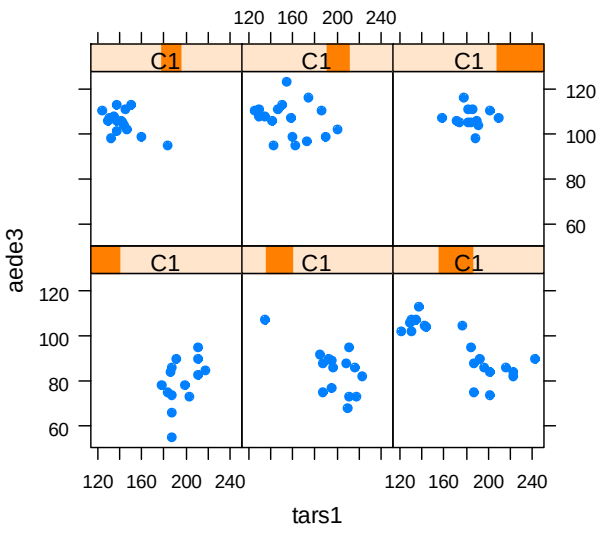


Figure 8. Equal cases in each grouping

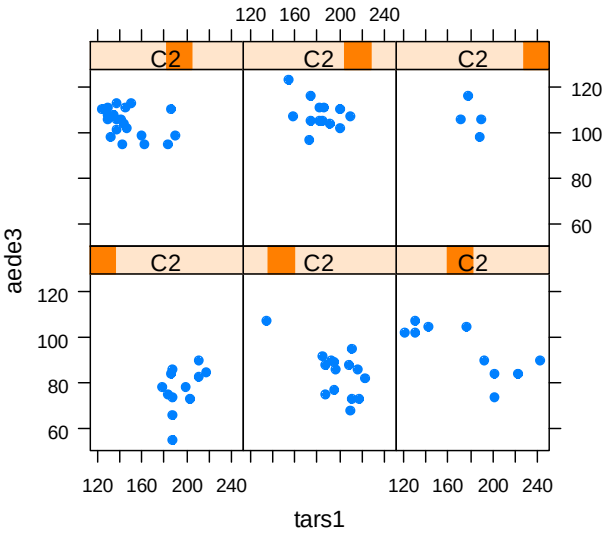


Figure 9. Equal spacing in eachgrouping

This version does not show the values of the conditioning variable.

It is also possible to **condition against two variables**, but before doing that we will create a synthetic data set. For now we will not go into detail about the nature of the data.

```
source(paste(d.R <- code.dir, "ellipseOutline.r", sep=""))
ec.t1 <-
for (t in -20:20)
   ec.t1 <- rbind(ec.t1, cbind(ellipse.outline(20,20,10,5,t,0,(200-t^2)/10),t))
}
ec.t1 <- data.frame(ec.t1[sample(dim(ec.t1)[1], dim(ec.t1)[1]),])
```

We can plot the scatterplot matrix

```
pairs(ec.t1, upper.panel = panel.cor, diag.panel = panel.hist)
```

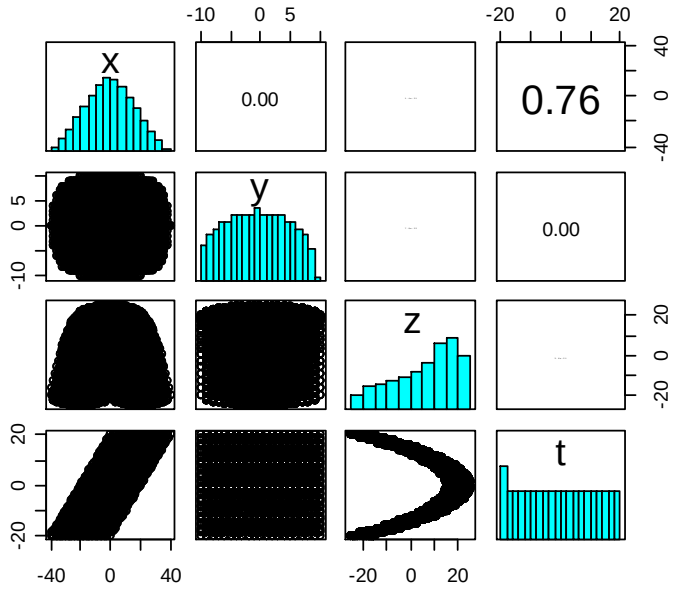


Figure 10.

In the lines that follow, note the use of the $ symbol. In this case it is used to reference the columns of a data frame; in other cases it references parts of other objects.

```
X <- equal.space(ec.t1$x, 25)
Y <- equal.space(ec.t1$y, 25)
Z <- equal.space(ec.t1$z, 25)
T <- equal.space(ec.t1$t, 25)
```

In the following, note the use of `aspect`. R, like many other languages, tries to use as much of the plotting region as possible. While this works well if the data has no intrinsic shape, it is a severe problem in other situations. For example, if you try to plot an ellipse you will get a circle. To avoid this you need to force the plot routine to use equal scales along the axes. This is often done by use of the aspect ratio (as below) but different plot routines use different methods (and for some it is up to you to find a way to force the appropriate scaling). Note the use of `x11()` as a method of creating another plot window rather than plotting over the current one.

```
xyplot(z ~x | Y, data = ec.t1, pch=".", main ="z ~x | Y",
                  aspect = diff(range(ec.t1$z))/diff(range(ec.t1$x)))
x11()
xyplot(y ~x | Z, data = ec.t1, pch=".", main ="y ~x | Z",
                  aspect = diff(range(ec.t1$y))/diff(range(ec.t1$x)))
x11()
xyplot(z ~y | X, data = ec.t1, pch=".", main ="z ~y | X",
                  aspect = diff(range(ec.t1$z))/diff(range(ec.t1$y)))
x11()
xyplot(z ~x | T, data = ec.t1, pch=".", main ="z ~x | T",
                  aspect = diff(range(ec.t1$z))/diff(range(ec.t1$x)))
```

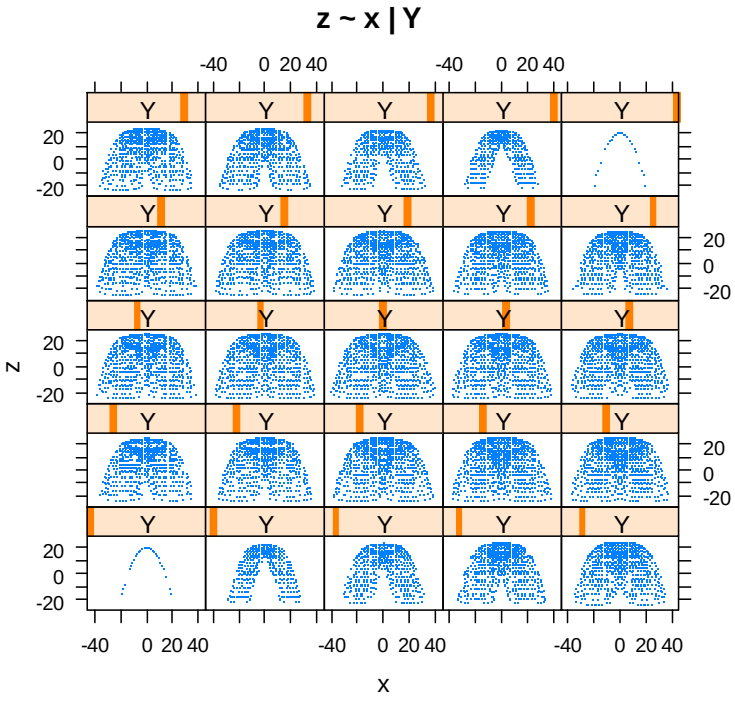


Figure 11.

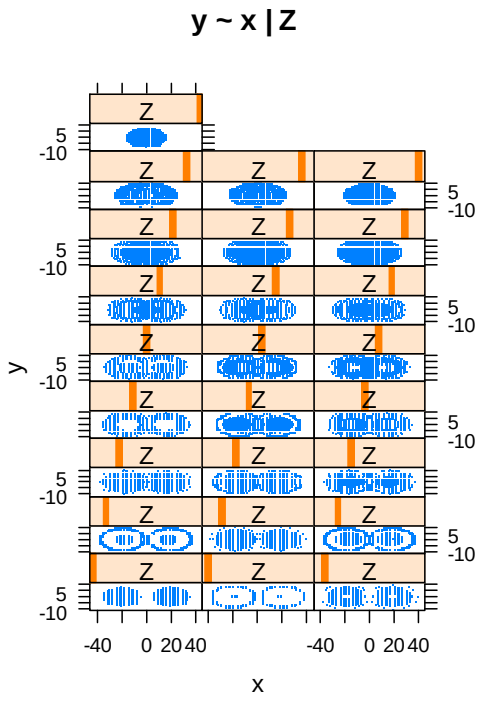


Figure 12.

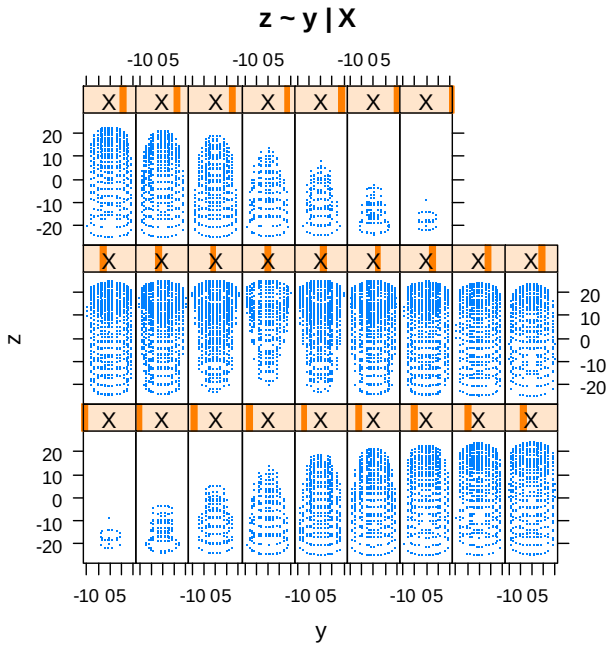


Figure 13.

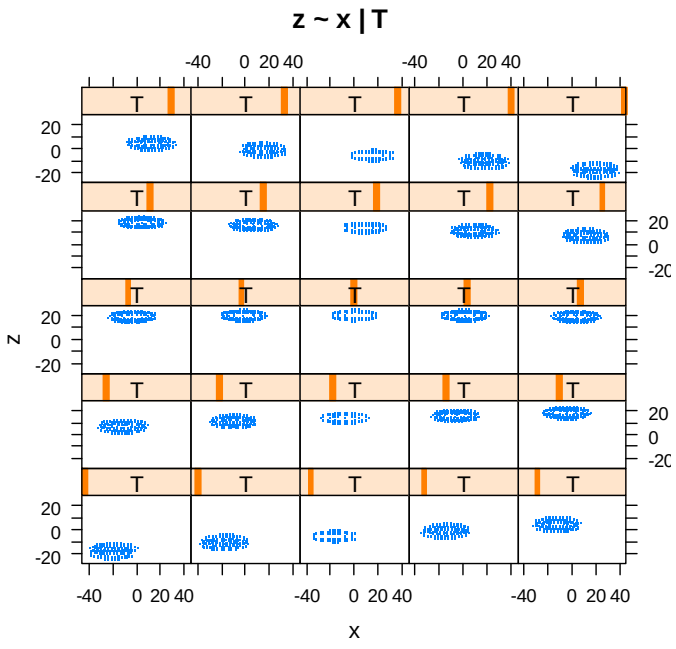


Figure 14.

```
Z5 <- equal.space(ec.t1$z, 5)
T5 <- equal.space(ec.t1$t, 5)
xyplot(z ~x | T5*Z5, data = ec.t1, main ="z ~x | T5*Z5", pch=".",
                    aspect = diff(range(ec.t1$z))/diff(range(ec.t1$x)))
```

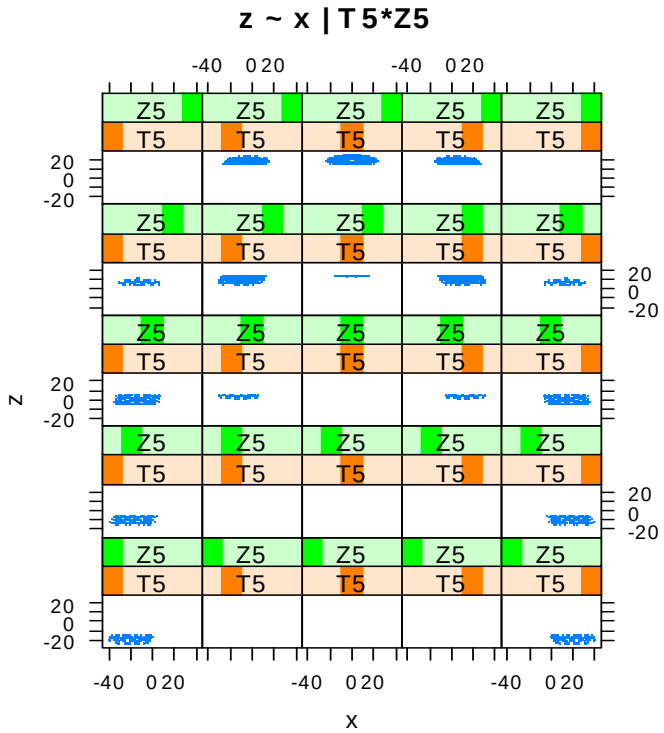


Figure15.

```
r <- 1
c <- 1
for (i in -20:15) {                # Loop through i from -20 to 15
   ind <- ec.t1$t==i               # Get the cases for which the t value == i
   X <- ec.t1$x[ind]               # And the corresponding x,y,z values
   Y <- ec.t1$y[ind]
   Z <- ec.t1$z[ind]
      # In the following - ( ?cloud)
      # print - displays the
      # cloud - a function that creates a cloud of points,
      #         with xlim, ylim, zlim (the range of values on the axes)
      #         set to the maximum range (x) to give proper scaling.
      # subpanel - the function use to plot the points.
      # groups - allows classes to be identified.
      # screen - sets the viewpoint.
      # split  - c(col, row, cols, rows)
      # more   -
   print(cloud(Z ~X*Y, xlim = range(ec.t1$x),
               ylim = range(ec.t1$x),zlim = range(ec.t1$x),
               subpanel = panel.superpose, groups=rep(1, dim(ec.t1)[1]),
               screen = list(z = 10, x = -80, y = 0), data = ec.t1),
               split = c(c, r, 6, 6), more = TRUE)
   c <- c+1
   if (c%%6 == 1) {   # Remainder mod 6
      c <- 1
      r <- r+1
   }
}
```

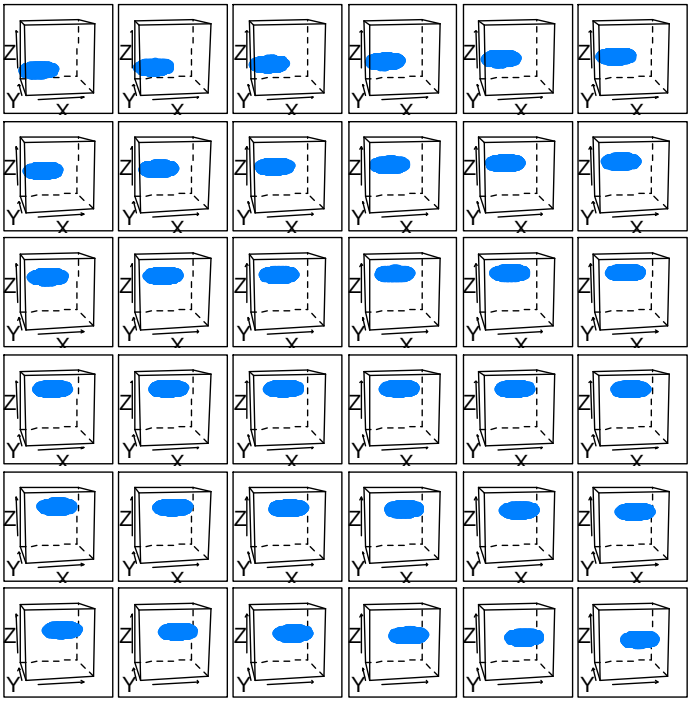


Figure 16.

1.3 Data Visualization in Ggobi

For high dimensional data, dynamic graphics will reveal more relationships.
For that purpose we will use Ggobi. This is a package that may be called from R or used alone.

```
library(rggobi)
g <- ggobi(d.flea)
```

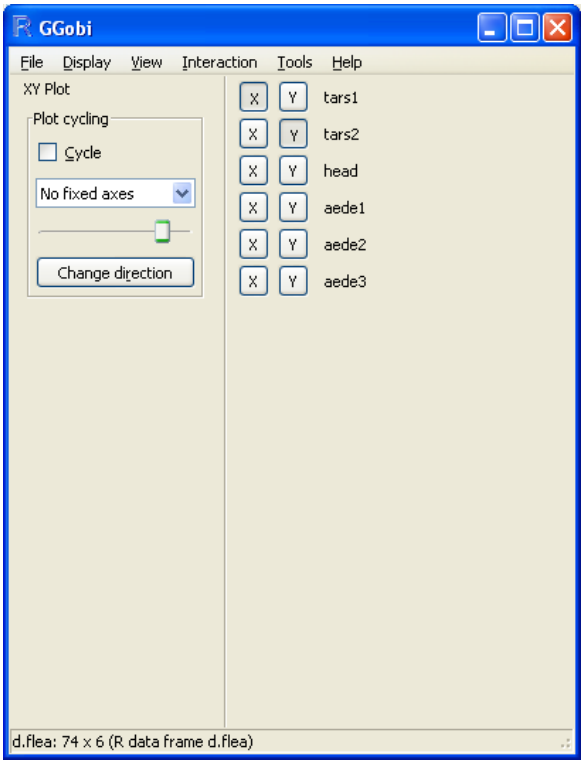


Figure 17. GGobi console

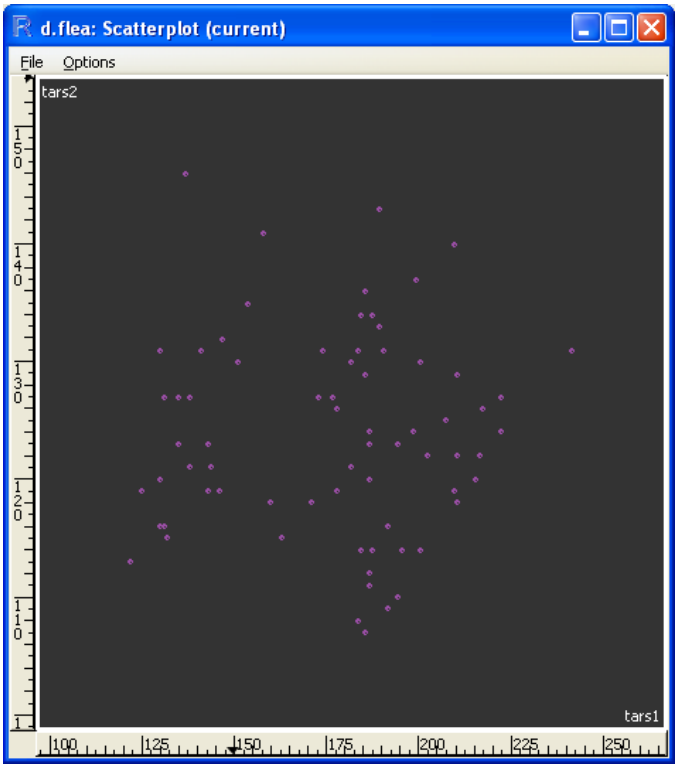


Figure 18. Scatterplot

1.3.1 Scatterplot Matrix

Ggobi starts with a console and a simple scatterplot as shown although we can also have a scatterplot matrix display.

```
display(g[1], "Scatterplot Matrix")
```

or [**Display**][**New scatterplot matrix**], from the Ggobi console.

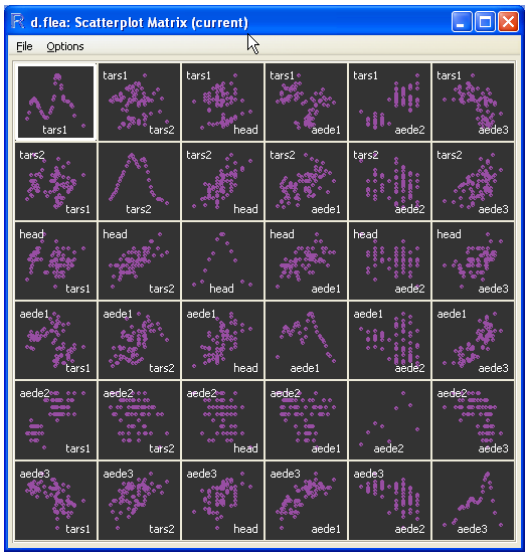


Figure 19. Scatterplot Matrix

1.3.2 Grand Tour

In order to investigate the data, we will start with a "grand tour"

```
display(g[1], "2D Tour")
```

or [**View**][**2D Tour**]

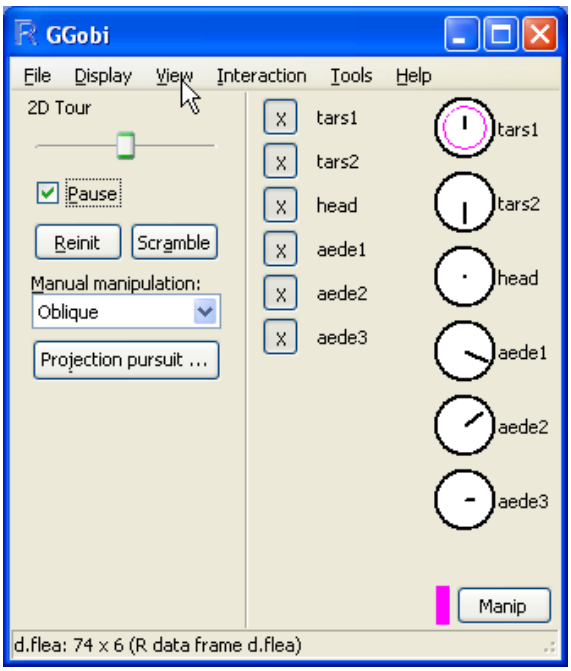


Figure 20.

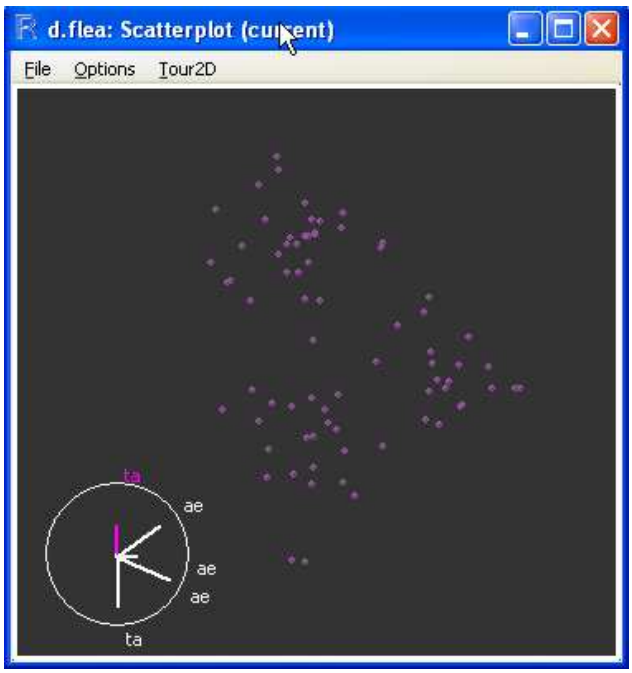


Figure 21. 2D Tour

This shows the console and the opening display. Note the circle with the lines. This represents the projection of the six axes on the two dimensional display. The process used in the grand tour is that a projection direction is selected and then a new direction is selected and the projection is changed smoothly in that direction. This allows the user to see the data from all directions (although it is

possible to move the projected direction by use of the mouse).

This gives a 2D tour of the 6 dimensional data. The portion of each variable in the view is shown by the representation of the axis in the bottom corner (and on the console).

1.3.3 Brushing

As the tour runs, 3 clusters will appear. When they do, you can click [**Pause**] and apply brushing, -[**Interaction**][**Brush**].- to group cases.

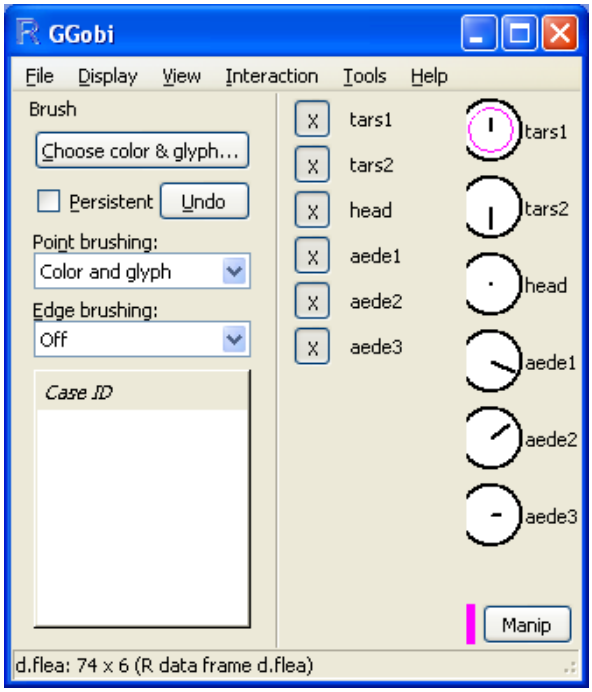


Figure 22. Brushing

You can change the colour and **glyph** (symbol) of data points.

The process involves selecting the colour and glyph and moving the brush over the points (we can select [**Persistent**] - if not selected, the brushing is transient).

We can or [**Choose color** & **glyph**] as shown below.

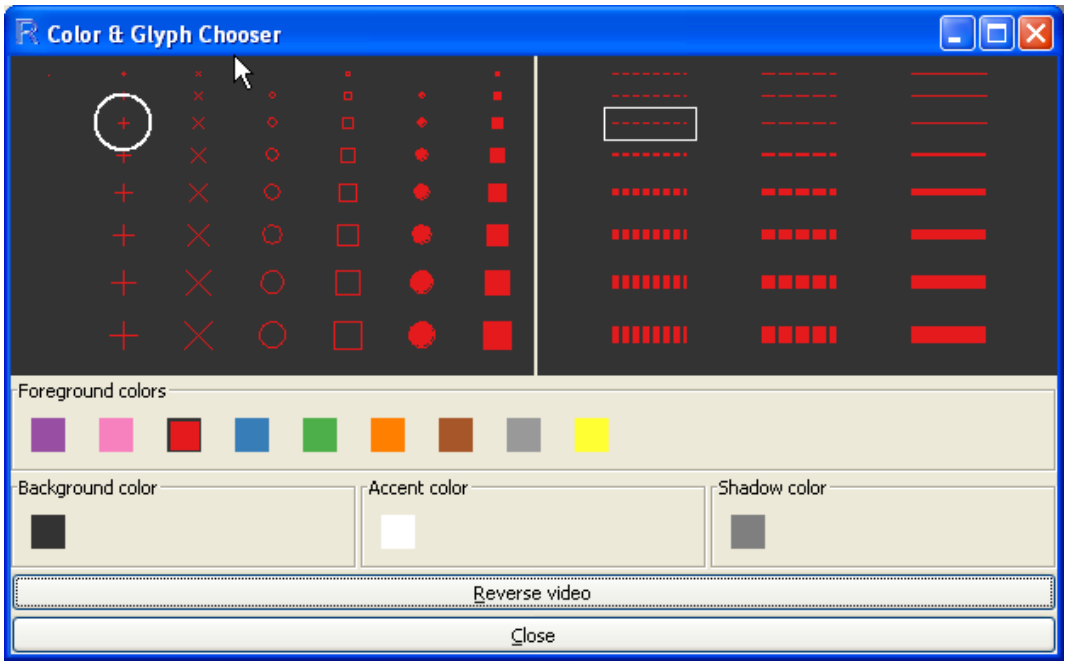


Figure 23. Select a red + (3^{rd} smallest size)

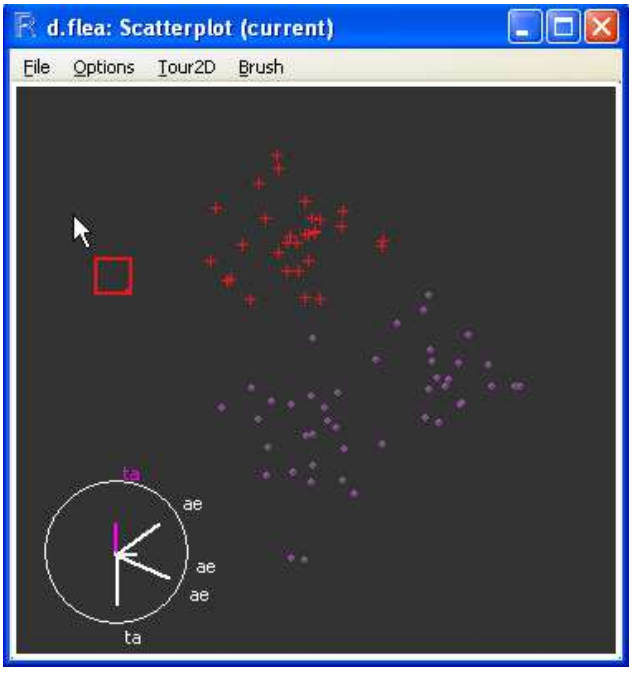


Figure 24. One cluster brushed
- red box for point brushing

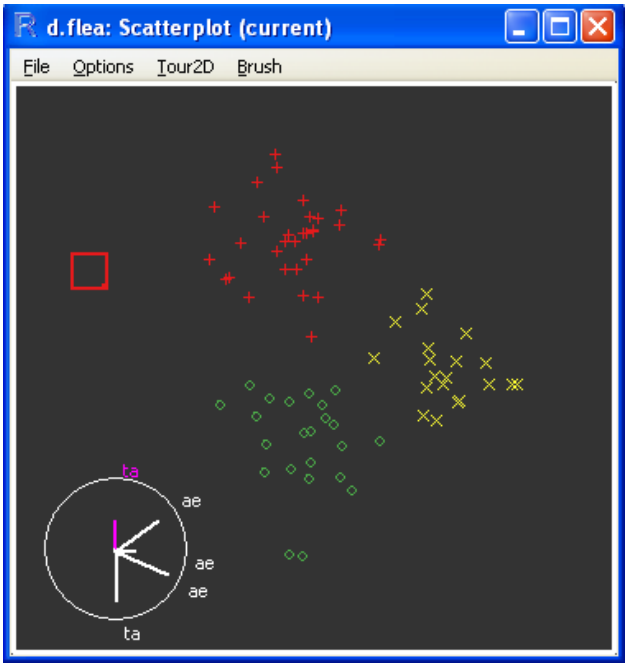


Figure 25. Other clusters brushed

In Figure 24, one apparent cluster is set to a red plus, while in Figure 25 another cluster is a yellow cross and the third is a green circle.

We can now return to a grand tour and see if the points of the same colour move together. Notice that for much of the time, the projection of the clusters are mixed together.

If we feel that we have the clusters properly coloured, we can again [**Pause**] and from R find out

what colours the points are

```
(old.col <- glyph_colour(g[1]))
F F F F F F F F F C F F F F F F F F F C C C C C C C C C C C C C C C C C C C A
9 9 9 9 9 9 9 9 9 5 9 9 9 9 9 9 9 9 9 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 3
A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

It turns out that we know the species of the flea beetles so we can compare the clustering that we observed with the true classification.

```
(noquote(rbind(flea.species,old.col)))
            F F F F F F F F F C F F F F F F F F F C C C C C C C C C C
flea.species C C C C C C C C C C C C C C C C C C C C C Hp Hp Hp Hp Hp Hp Hp Hp Hp Hp Hp
old.col      9 9 9 9 9 9 9 9 9 5 9 9 9 9 9 9 9 9 9 5 5 5 5 5 5 5 5 5 5
             C C C C C C C C C C A A A A A A A A A A A A A
flea.species Hp Hp Hp Hp Hp Hp Hp Hp Hp Hp Hp Hk Hk Hk Hk Hk Hk Hk Hk Hk Hk Hk Hk Hk
old.col      5 5 5 5 5 5 5 5 5 5 5 3 3 3 3 3 3 3 3 3 3 3 3 3
             A A A A A A A A A A A A A A A A A A A
flea.species Hk Hk Hk Hk Hk Hk Hk Hk Hk Hk Hk Hk Hk Hk Hk Hk Hk Hk
old.col      3 3 3 3 3 3  3  3  3 3 3 3 3 3 3 3 3
```

(notice that the 11th case is of class **C** but is the same colour as the **Hk** but all the others have the colour corresponding to a class of flea beetle) and then reset the colours and glyphs for the next part.

```
glyph_colour(g[1]) <- rep(2, 74)    # 74 purple
glyph_type(g[1]) <- rep(4, 74)      # 74 circles
```

1.3.4 Parallel Coordinates Plot

Another type of plot is the **Parallel Coordinates Plot**. This can be a good method for investigating high dimensional data. Consider the point (3, 1, 2) as shown in Figure 25 below. In parallel coordinates, it would be

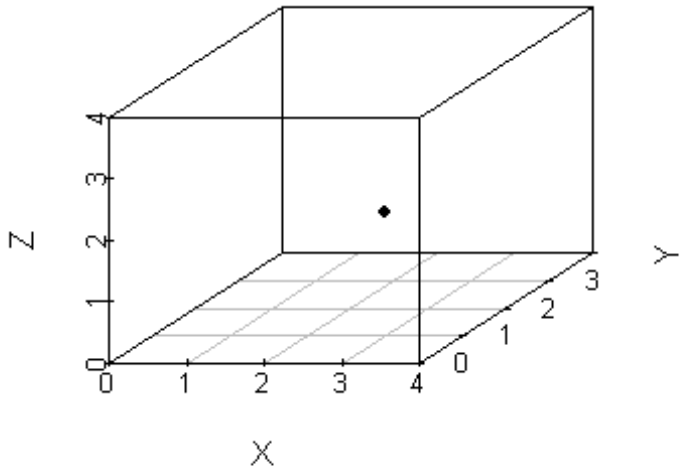


Figure 26. The point (3, 1, 2)

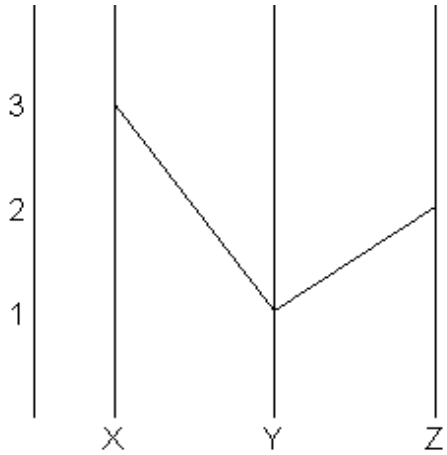


Figure 27. The point (3, 1, 2) in parallel co

If we want 4 (or more) dimensions, we need only add another (or more) parallel line(s).

```
display(g[1], "Parallel Coordinates Display")
```

or [**Display**][**New parallel coordinates plot**].

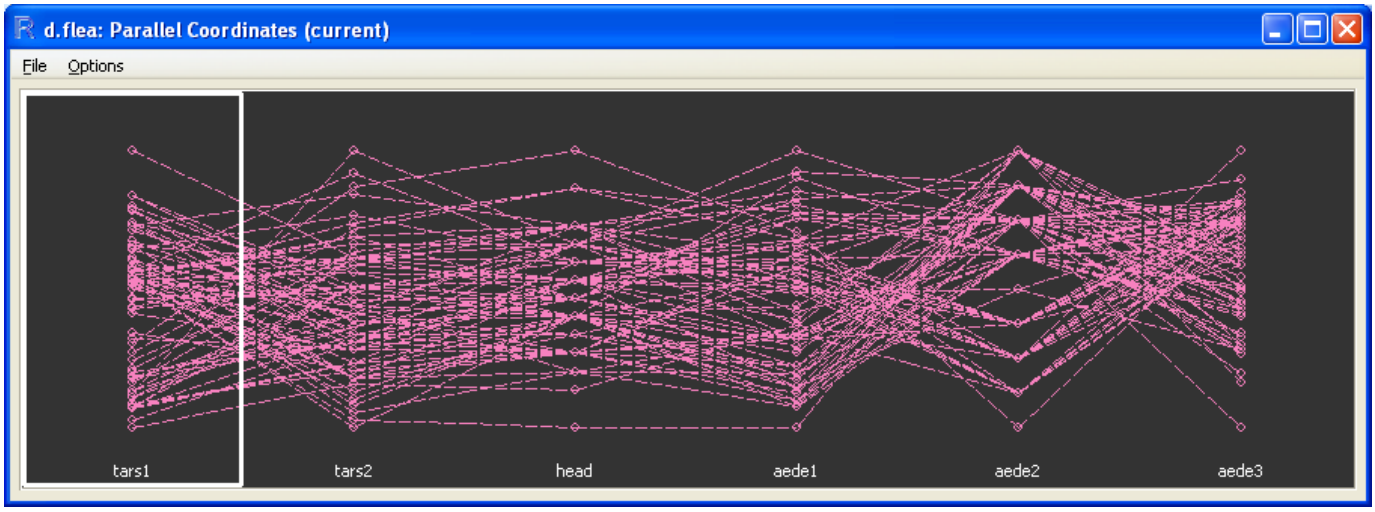


Figure 28. Parallel Coordinates Plot

Each data point has a value on each of the axes which are plotted vertically rather than at right angles to each other.

1.3.4.1 Parallel Coordinates Brushing

The value of brushing is greatly enhanced when we have more than one display. Below we see a scatterplot with 5 points being brushed and we see that the parallel coordinates display has 5 points (lines) coloured yellow. This shows which points correspond.

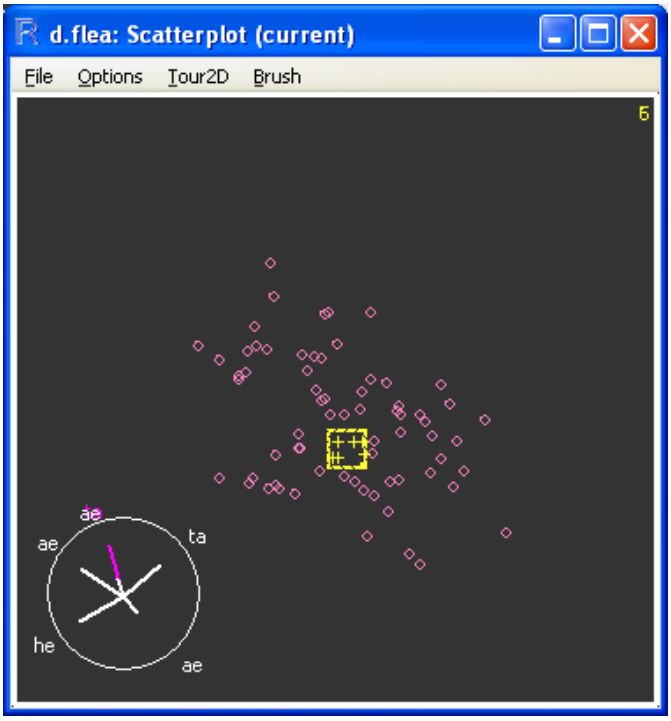


Figure 29. Brushing

1.3.4.2 Parallel Coordinates Linked Brushing

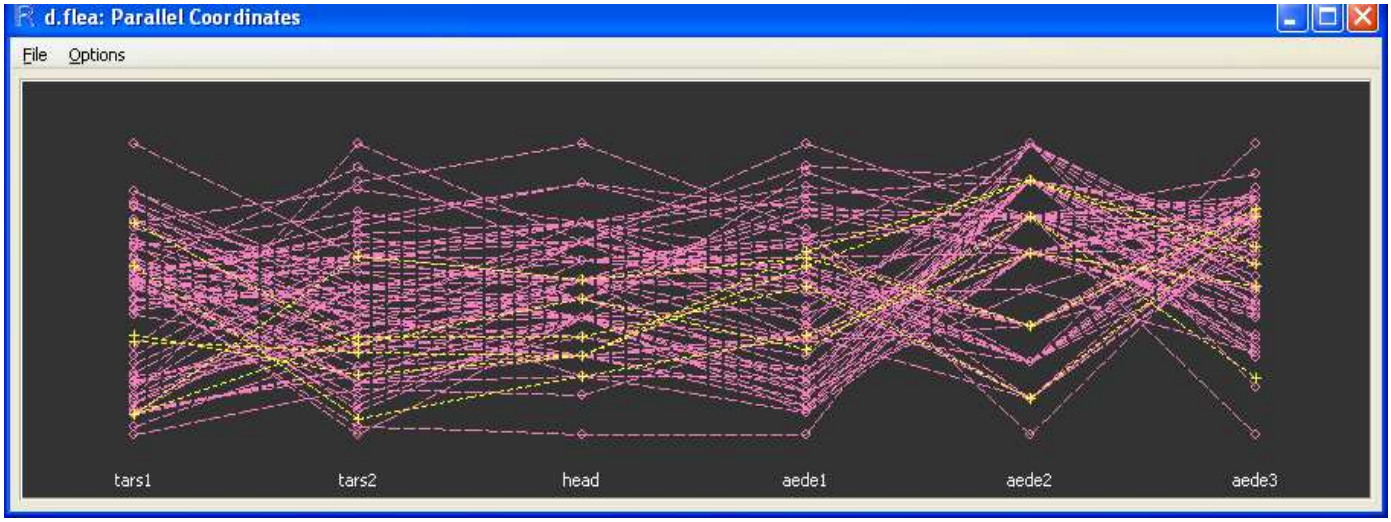


Figure 30. Linked brushing

For the next part, we set the colours to correspond to the species

```
glyph_colour(g[1]) <- c(rep(6,21),rep(4,22),rep(9,31))
```

It is also possible to get information about the clustering from parallel coordinates. We will start by moving the axes - put the mouse on the white frame and drag *aede*3 to the first position (a corner will appear as the cursor).

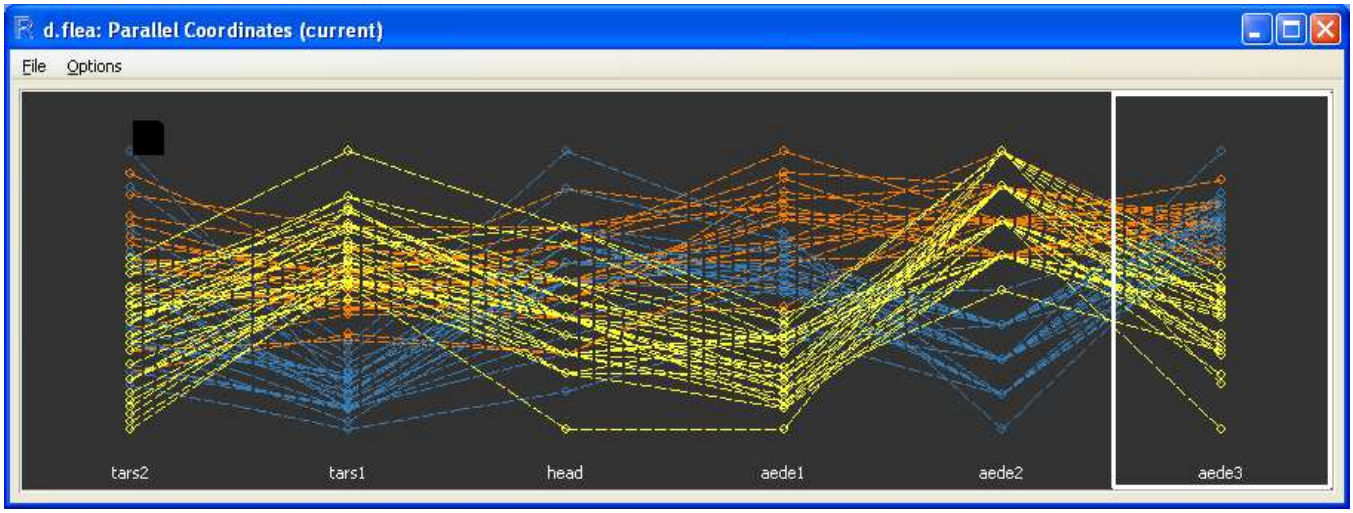


Figure31.

Repeat until we have the axes arranged as below.

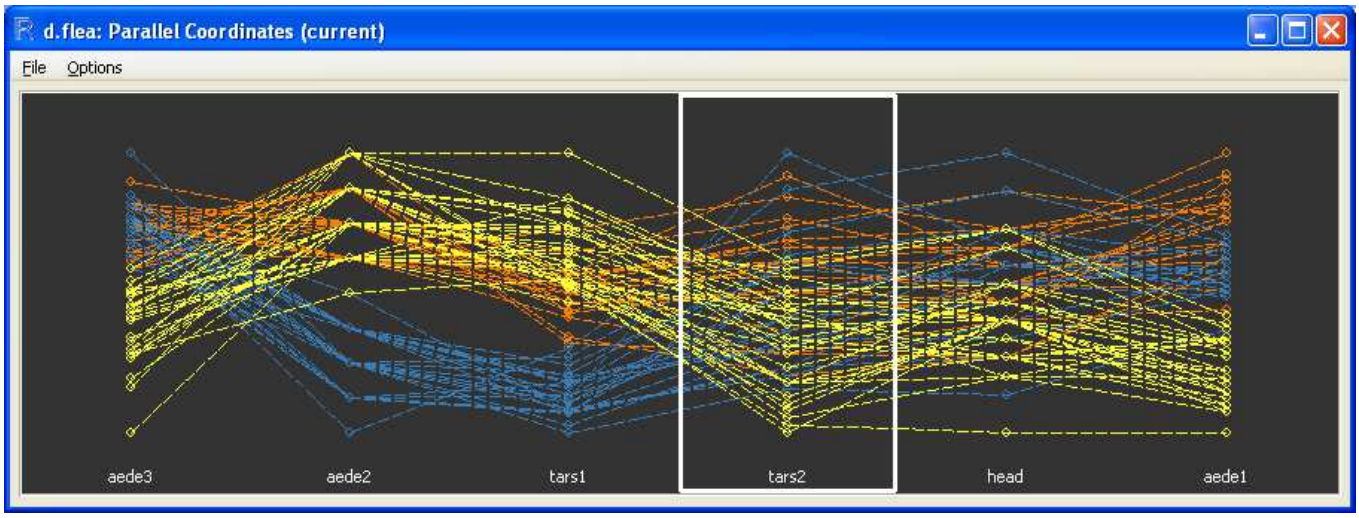


Figure 32.

When we look at this we see that there seem to be values of *aede*3 and *tars*1 that split the data.

1.3.4.1 Parallel Coordinates Identification

Before we find the values, we will introduce [**Tools**][**DataViewer**] which allows us to look at our data.

ggobi data viewer

Row Label	tars1	tars2	head	aede1	aede2	aede3
1	191.000000	131.000000	53.000000	150.000000	15.000000	104.000000
2	185.000000	134.000000	50.000000	147.000000	13.000000	105.000000
3	200.000000	137.000000	52.000000	144.000000	14.000000	102.000000
4	173.000000	127.000000	50.000000	144.000000	16.000000	97.000000
5	171.000000	118.000000	49.000000	153.000000	13.000000	106.000000
6	160.000000	118.000000	47.000000	140.000000	15.000000	99.000000
7	188.000000	134.000000	54.000000	151.000000	14.000000	98.000000
8	186.000000	129.000000	51.000000	143.000000	14.000000	110.000000
9	174.000000	131.000000	52.000000	144.000000	14.000000	116.000000
10	163.000000	115.000000	47.000000	142.000000	15.000000	95.000000
11	190.000000	143.000000	52.000000	141.000000	13.000000	99.000000
12	174.000000	131.000000	50.000000	150.000000	15.000000	105.000000
13	201.000000	130.000000	51.000000	148.000000	13.000000	110.000000
14	190.000000	133.000000	53.000000	154.000000	15.000000	106.000000
15	182.000000	130.000000	51.000000	147.000000	14.000000	105.000000
16	184.000000	131.000000	51.000000	137.000000	14.000000	95.000000

d.flea

Figure 33.

Next we will use [**Interaction**][**Identify**]

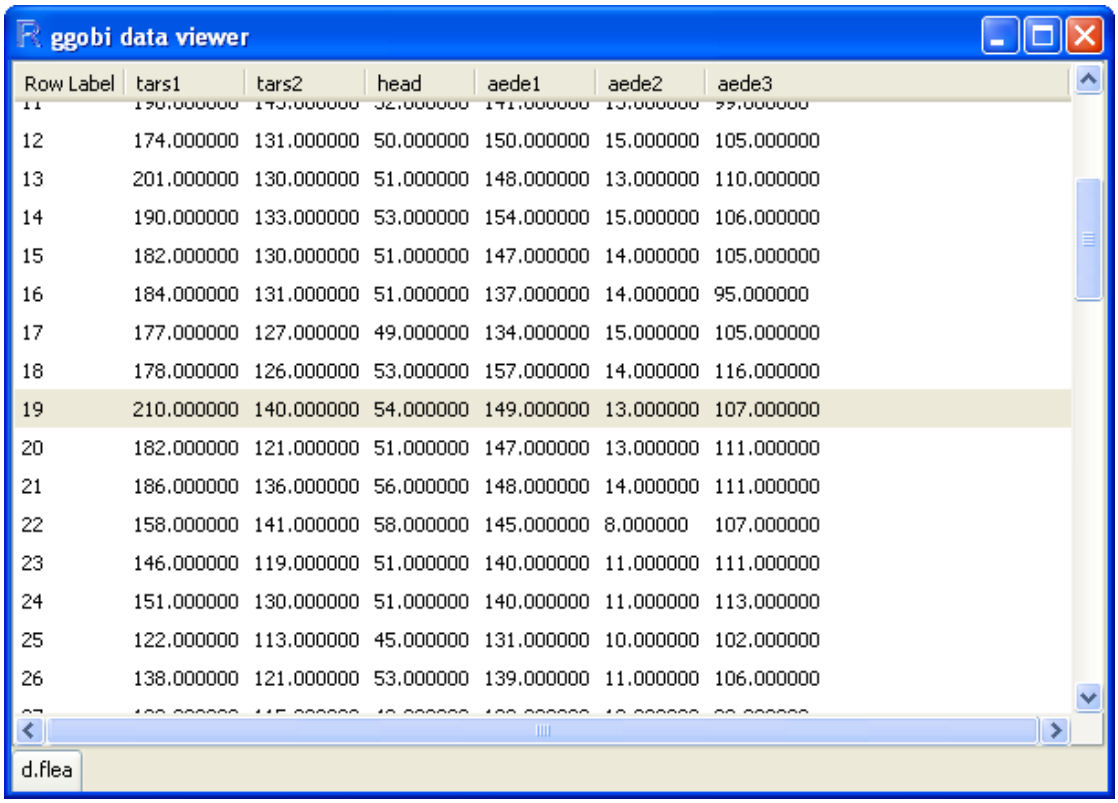


Figure 34. Linked identification

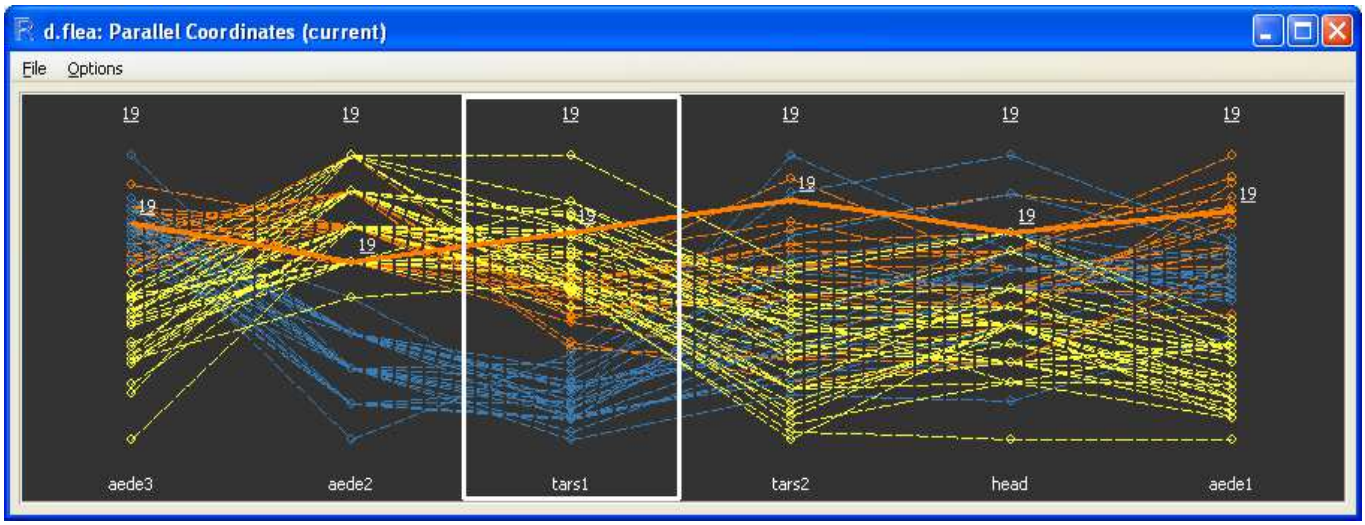


Figure 35. Linked identification

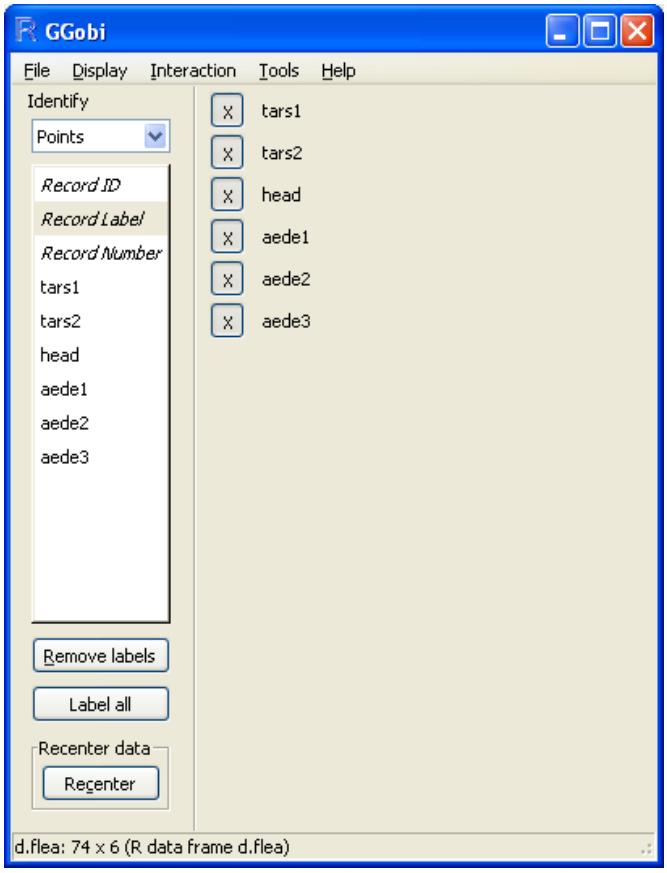


Figure 36. Identification using record label

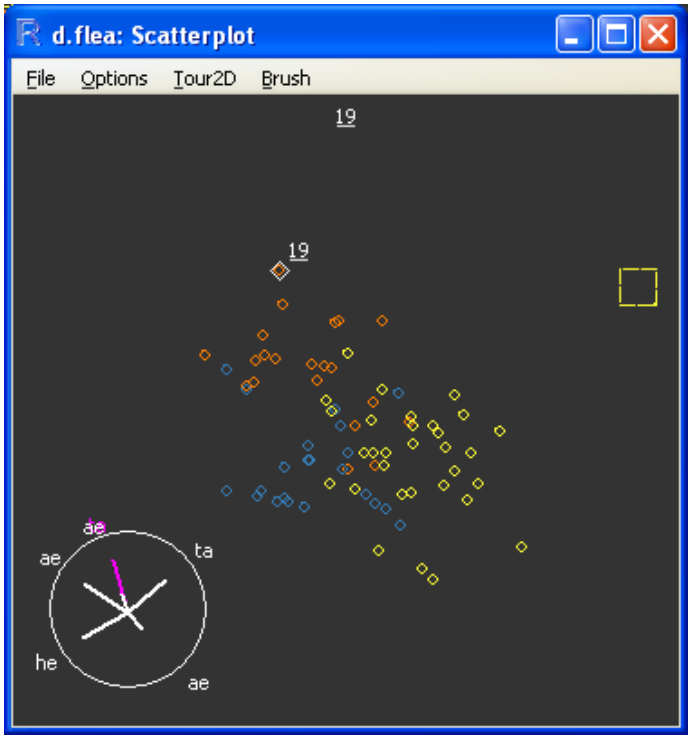


Figure 37. Linked identification

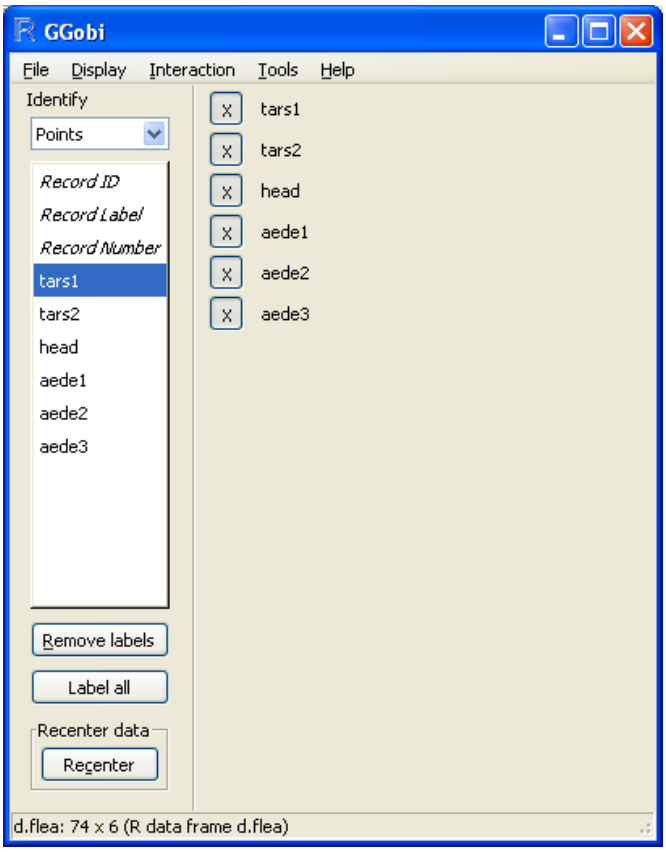


Figure 38. Identification using data value (tars1)

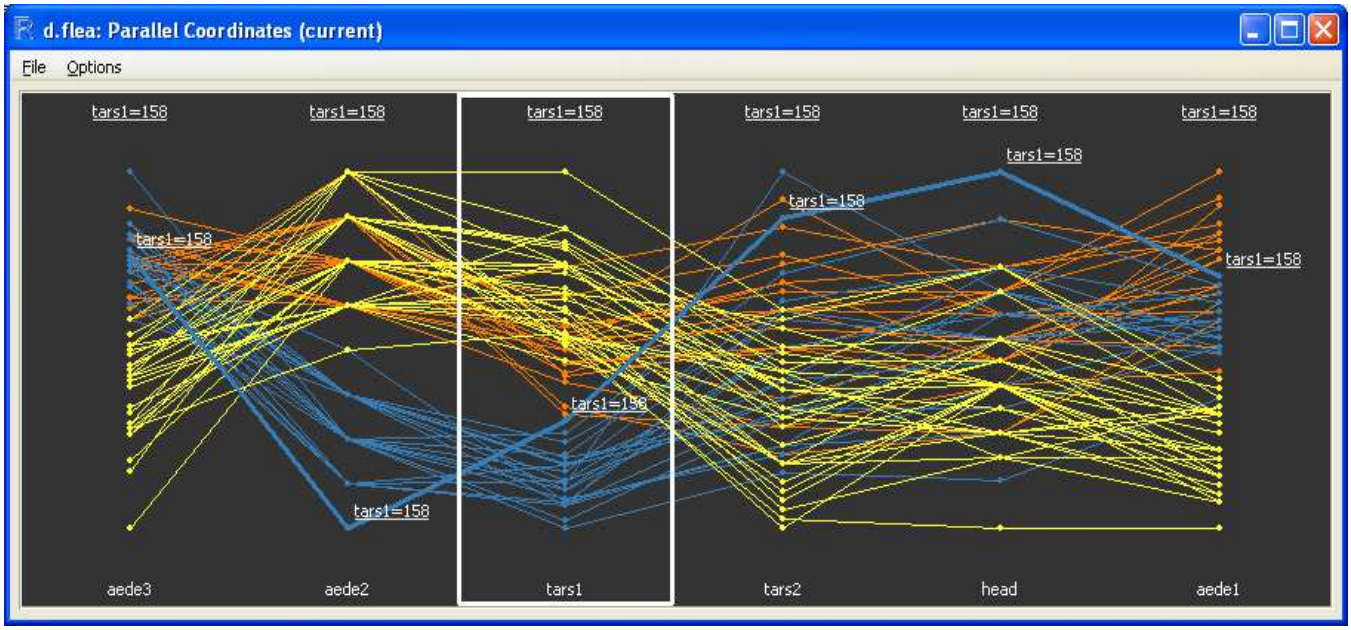


Figure 39. Linked identification using data value (tars1)

In the above we see that if *tars*1 < 160 we have one group (blue) split from the other two

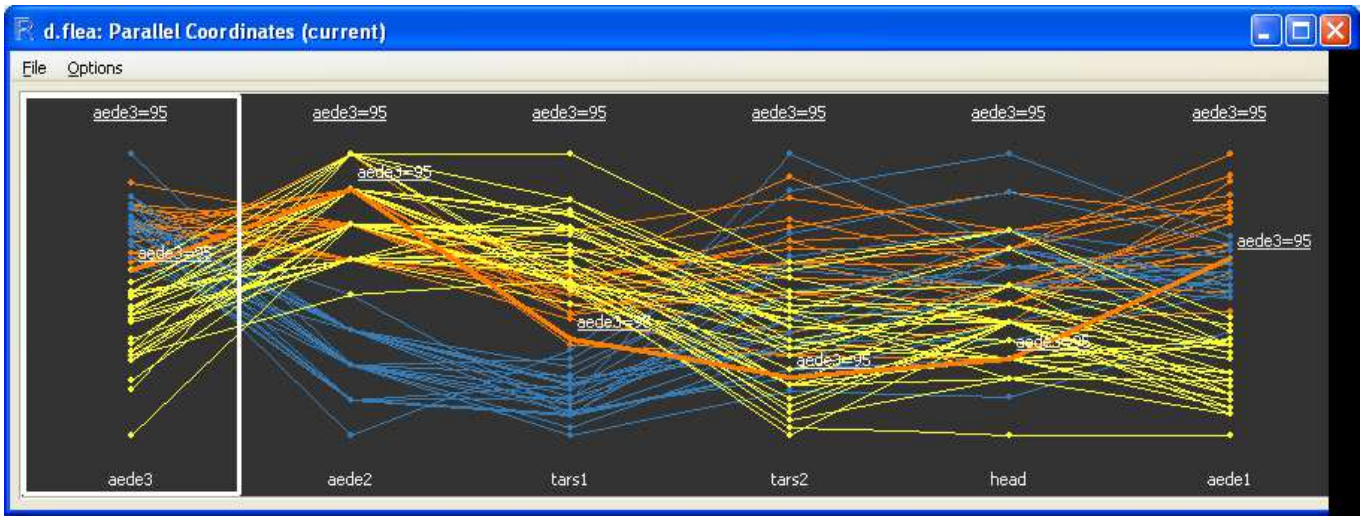


Figure 40. Linked identification using data value (aede3)

In the above we see that if *aede*3 < 95 we have one group (yellow) almost split from the other two. It appears that we can do an almost perfect split with this information. (We will see this forms the basis for a recursive splitting process that we will see later.)

We can do similar things from R.

```
cols <- rep(6, 74)
cols[which(d.flea[,6] < 95)] <- 9
cols[which(d.flea[,1] < 160)] <- 4
glyph_colour(g[1]) <- cols
close(g)
```

1.3.5 Stereo

An interesting view of the data can be obtained by looking at the data from slightly shifted viewpoints -

```
source(paste(code.dir,"MakeStereo.r",sep="/"))
make.Stereo(d.flea[,c(1,5,6)], species, Main=" Flea beetles", asp=" F",
            Xlab=" tars1", Ylab=" aede2", Zlab=" aede3")
```

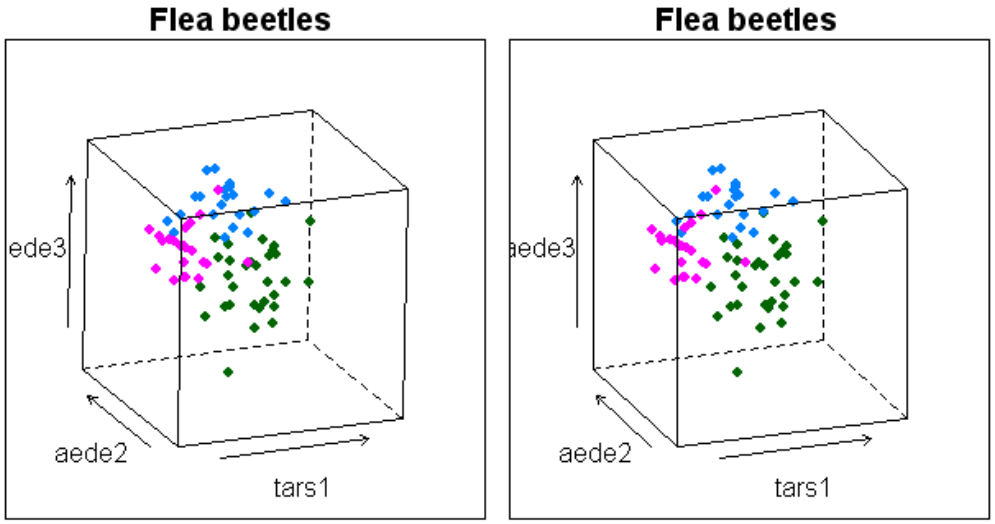


Figure 41. Stereo projection

```
make.Stereo(d.flea[,c(6, 5, 1)], species, Main="  Flea beetles", asp=" F",
            Zlab="  tars1", Ylab=" aede2", Xlab=" aede3")
```

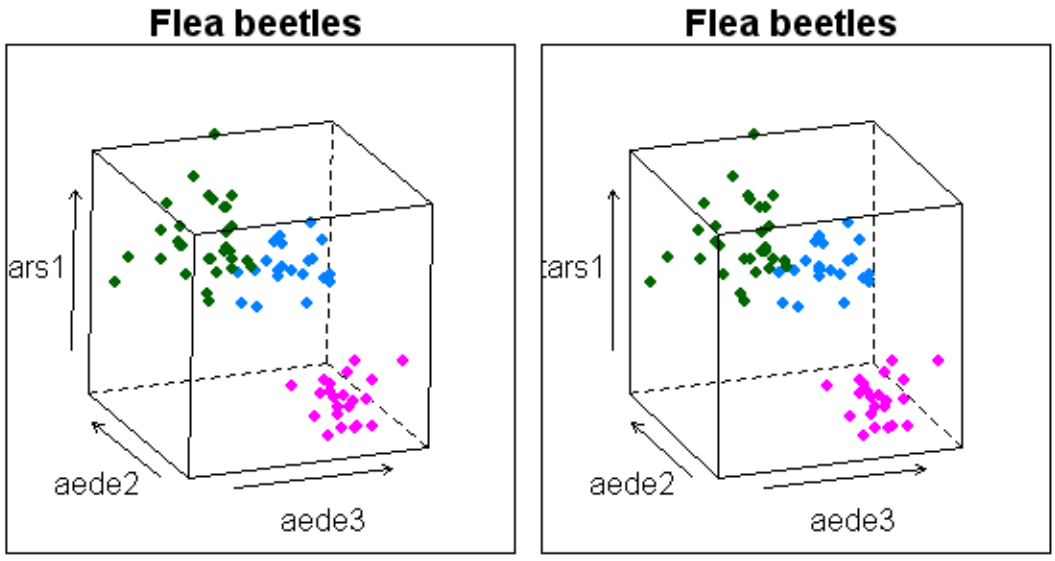


Figure 42. Stereo projection

1.3.6 RGL

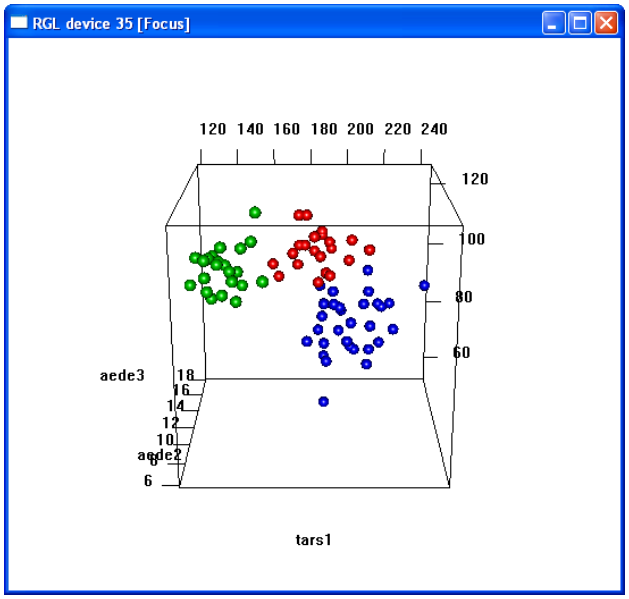
We could also use another package

```
library(rgl)
```

which allows interactive visualization.

Consider

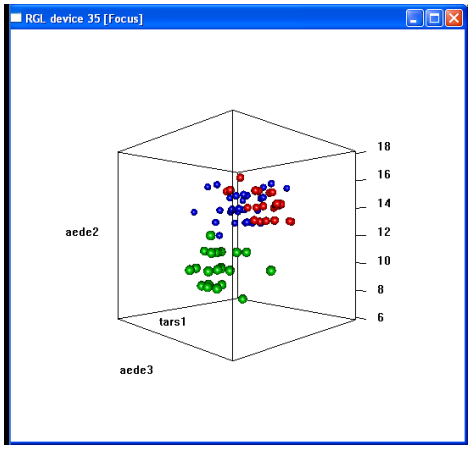
```
plot3d(d.flea[,1], d.flea[,5], d.flea[,6], xlab="tars1",ylab="aede2",zlab="aede3",
       col=species+1, size=0.5, type="s")
```

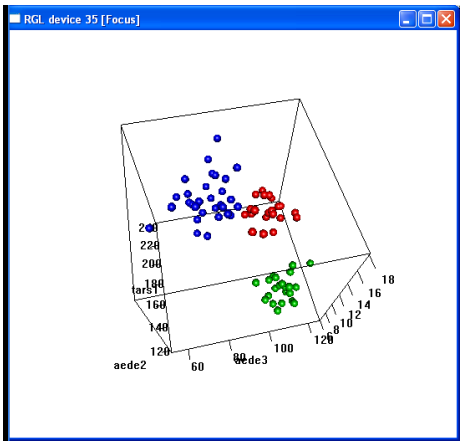


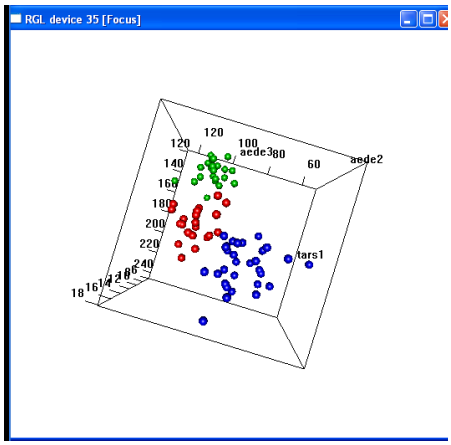
Now, apply the following code

```
for (j in seq(0, 90, 10)) {
   for(i in 0:360) {
      rgl.viewpoint(i, j);
   }
}
```

The `rgl.viewpoint(i, j)` changes the point from which you view the object, making it appear that the object is rotating. The first argument (`i` in this case) is the spherical coordinates angle θ, while the second is ϕ. Hence, this code "rotates" the object about the z-axis (`i in 0:360`) for $\phi = 0$ to $\pi/2$.







It is also possible to

- hold the left mouse button down to rotate the image the image;
- hold the right mouse button down (or use the mouse wheel) to zoom in/out.

1.4 Examples

1.4.1 Randu

Randu is a random number generator that had a ′slight′flaw.

```
d.file <- paste(data.dir, "randu.dat", sep = "/")
d.randu <- read.table(d.file)
pairs(d.randu, upper.panel=panel.cor, diag.panel=panel.hist)
```

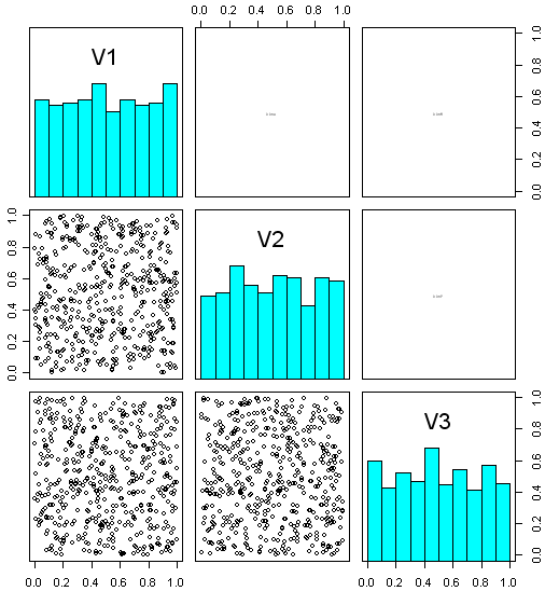


Figure 43. Scatterplot matrix for **randu**

Looking at the scatterplotmatrix everything seems random but in Ggobi...

```
g <- ggobi(d.randu)
```

with [**View**][**2D tour**]

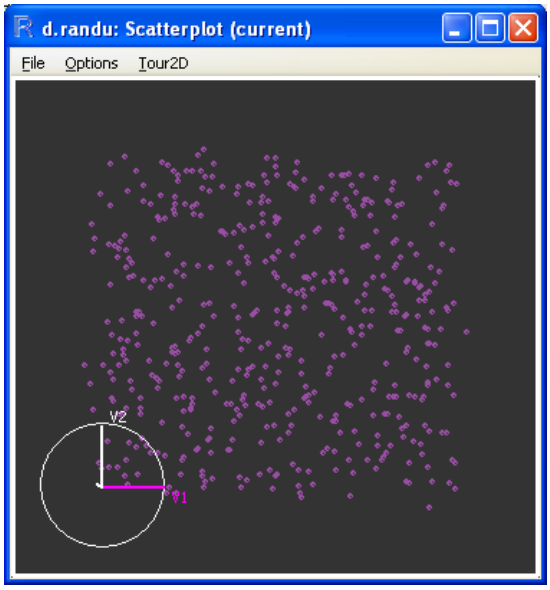


Figure 44. **randu** projection

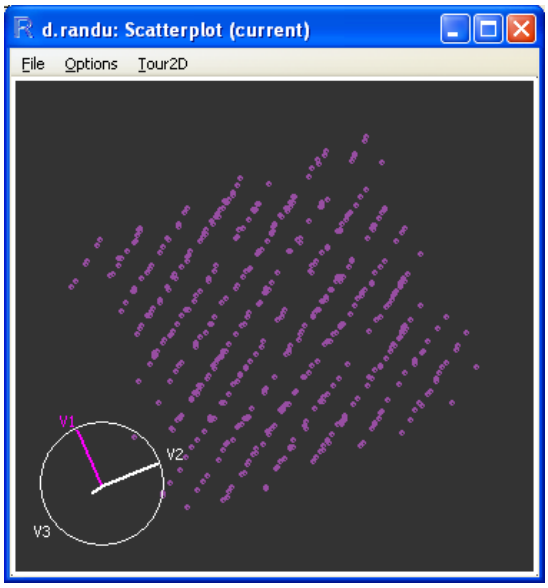


Figure 45. An *interesting* projection

1.4.2 Prim7

Prim7 contains 500 observations taken from a high energy particle physics scattering experiment which yields four particles. The reaction can be described completely by seven (7) independent measurements. The important features of the data are short-lived intermediate reaction stages which appear as protuberant "arms" in the point cloud.

```
prim <- read.table(paste(data.dir, "prim7.dat",sep="/"))
g <- ggobi(prim)
```

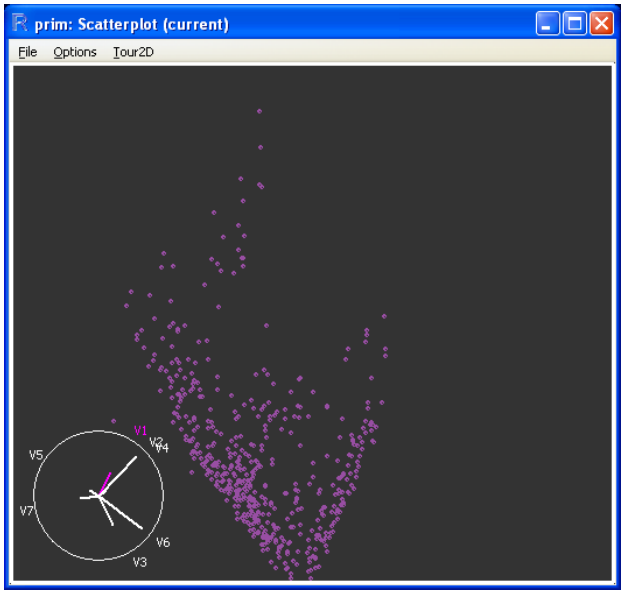


Figure 46. Prim7

We will look at this by looking at clusters that have been found in the past. It would be possible to do this by careful brushing and observation but the following sets the appropriate colours for the data.

```
new.col <- rep(1, 500)
col.2 <- c(2,3,4,14,15,16,17,18,21,23,30,34,37,41,43,46,49,50,53,54,55,
```

```
            57,58,63,65,66,69,70,72,73,74,75,77,78,79,85,86,88,90,91,92,
            93,94,95,99,100,102,104,105,106,107,109,110,113,114,116,120,
            121,124,125,126,127,129,130,133,139,140,141,143,145,147,150,
            152,153,157,158,159,160,161,164,166,169,172,175,176,177,178,
            180,185,194,195,198,200,203,204,209,210,211,212,218,219,220,
            222,223,226,228,229,233,234,236,238,240,242,244,245,246,248,
            249,252,253,257,259,263,264,265,266,267,269,270,273,277,278,
            280,281,282,283,284,286,292,294,296,297,300,305,310,311,314,
            315,317,323,331,332,333,334,335,341,342,343,346,351,356,359,
            360,361,362,365,370,372,374,375,377,378,379,380,383,386,388,
            389,390,391,393,397,398,400,402,403,405,407,408,413,414,415,
            417,418,419,420,425,427,428,429,430,432,433,434,436,437,438,
            440,444,445,447,448,452,453,454,455,456,463,465,467,470,471,
            473,476,477,478,480,481,482,484,485,487,488,489,490,491,494,
            497)
col.3 <- c(11,20,27,33,47,51,60,61,62,98,115,118,119,132,155,186,191,
            193,202,205,207,208,213,225,230,231,232,235,239,243,250,251,
            268,272,295,312,316,338,339,345,349,354,358,364,366,376,381,
            395,401,421,422,446,460,496)
col.5 <- c(5,8,13,19,26,32,39,48,56,71,81,96,111,136,137,144,149,156,
            162,165,188,199,201,216,255,262,274,279,289,291,301,320,322,
            326,327,329,344,348,353,363,367,369,384,399,404,406,411,423,
            441,442,443,469,474,479,483,495,499,500)
col.8 <- c(7,29,31,36,89,101,117,131,138,154,173,187,190,192,196,197,
            206,247,254,256,258,287,290,298,299,309,324,325,385,387,464)
col.9 <- c(1,12,22,24,25,44,45,52,64,83,103,108,122,123,134,135,146,151,
            167,168,170,174,179,181,184,221,224,237,261,271,285,293,304,
            306,307,308,319,328,337,352,355,357,368,396,410,424,426,435,
            439,449,451,458,461,462,466,472,475,493)
```

Now set all of the points belonging to one cluster to the colour with value 2.

```
new.col[col.2] <-  2
glyph_colour(g[1]) <-  new.col
```

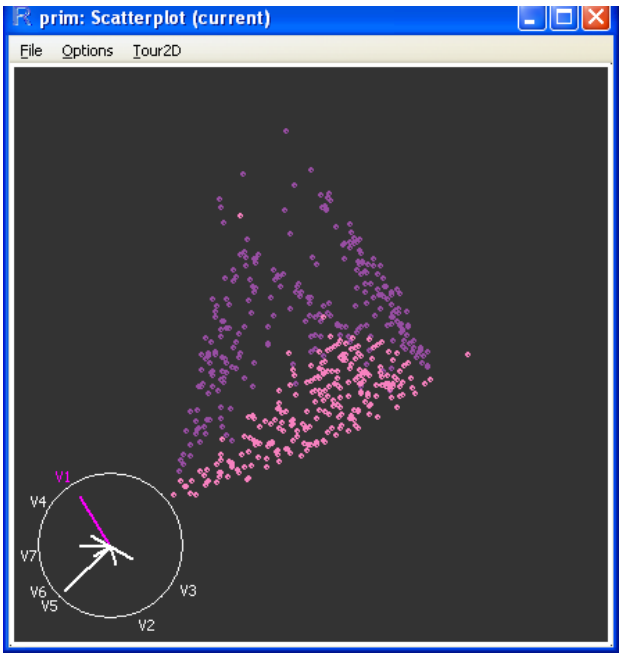


Figure 47. Prim7 with first cluster brushed

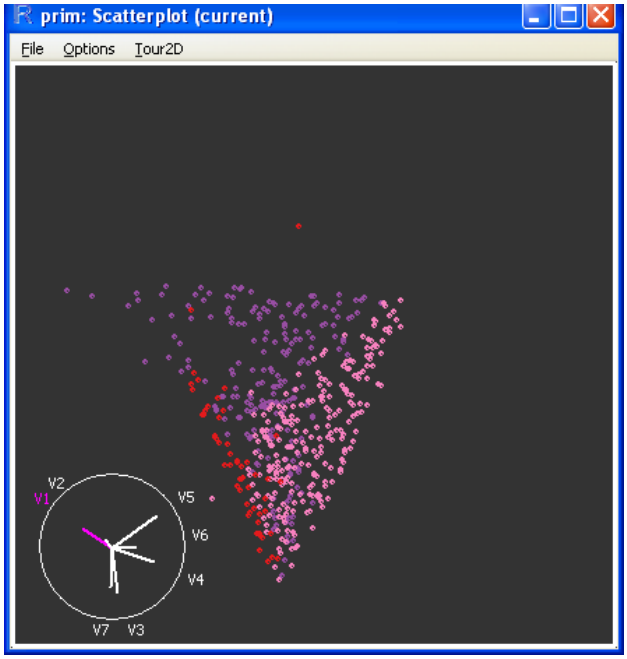


Figure 48. Prim7 with 2 clusters brushed

Now run though the other clusters.

```
new.col[col.3] <-  3
glyph_colour(g[1]) <-  new.col
new.col[col.5] <-  5
glyph_colour(g[1]) <-  new.col
```

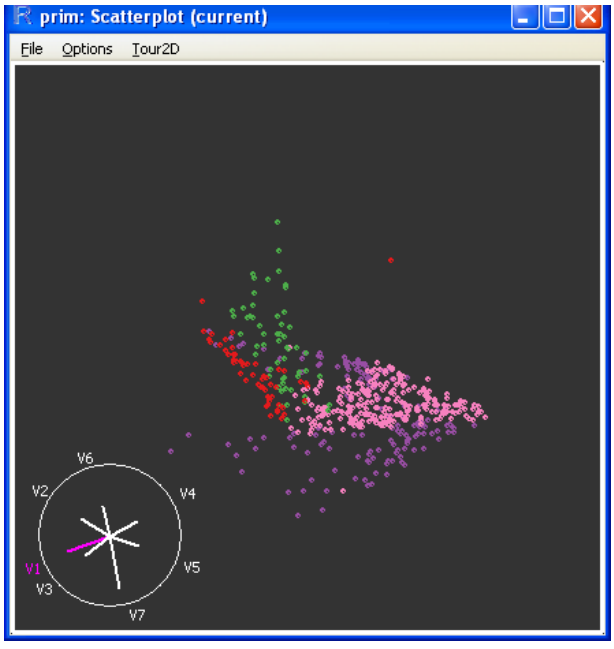


Figure 49. Prim7 with 3 clusters brushed

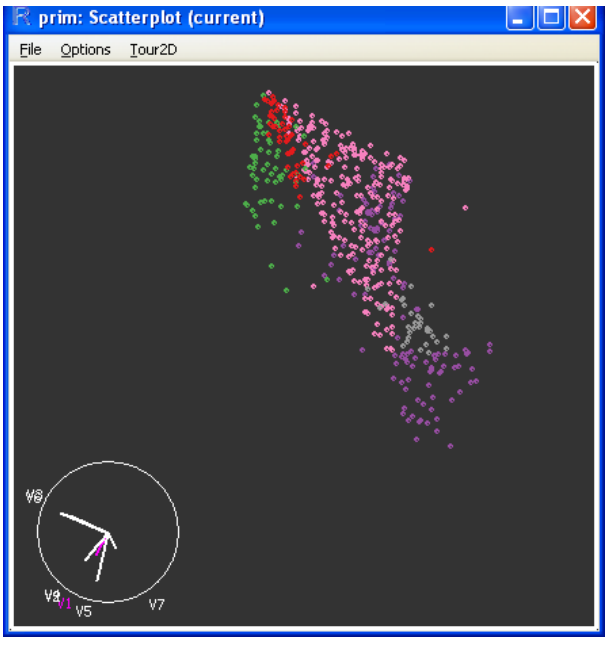
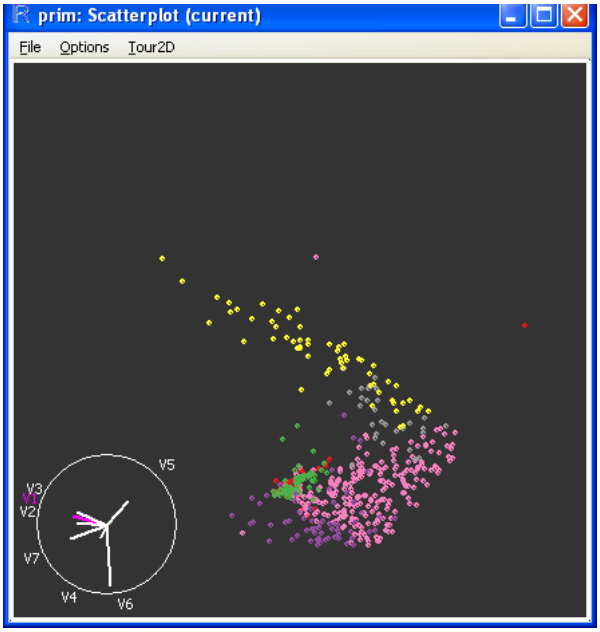


Figure 50. Prim7 with 4 clusters brushed

```
new.col[col.8] <-  8
glyph_colour(g[1]) <-  new.col
new.col[col.9] <-  9
glyph_colour(g[1]) <-  new.col
```



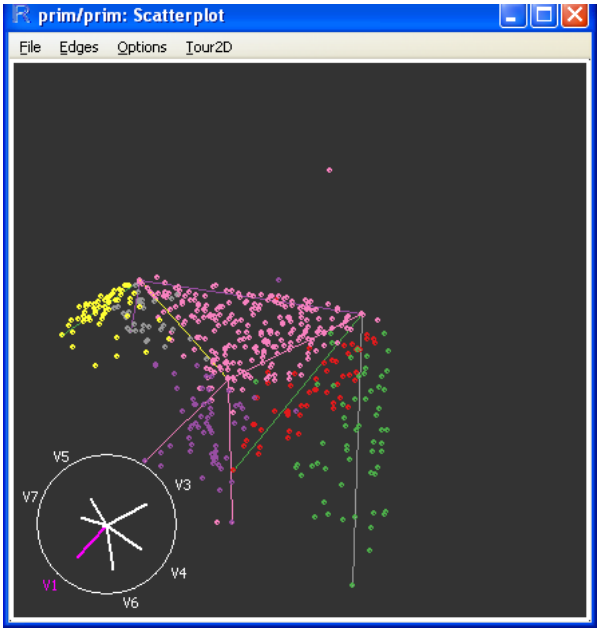


Figure 51. Prim7 with all clusters brushed **Figure 52**.

It is also possible to put lines on the data to outline possible structures. The lines are those determined by researchers

```
prim.lin <- read.table(paste(data.dir, "prim7.lines",sep="/"))
edges(g[1]) <- prim.lin
```

We can add lines by use of [**Interaction**][**Edit Edges**] and dragging the cursor from one point to another.

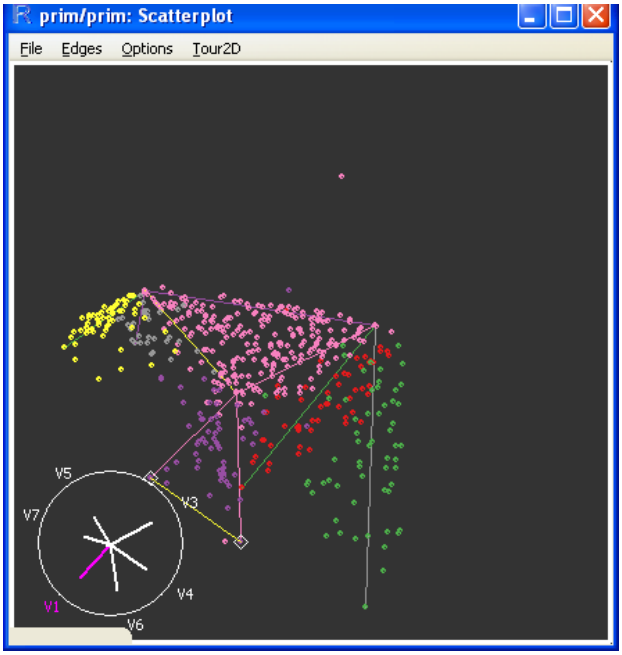


Figure 53. Line added

With careful exploration in a grand tour, and using *projection pursuit*, it is possible to discover structure in the data.

1.4.3 6-Dimensional Cube

The following shows how we might investigate the nature of the projection of a 6-dimensional cube.

```
g <- ggobi(paste(data.dir, "cube6.xml",sep="/"))
```

When Ggobi starts up, it will show 4 points in *xy* Plot mode. On the Scatterplot window select [**Edges**] and [**Attach edge set** ...] to show the edges.

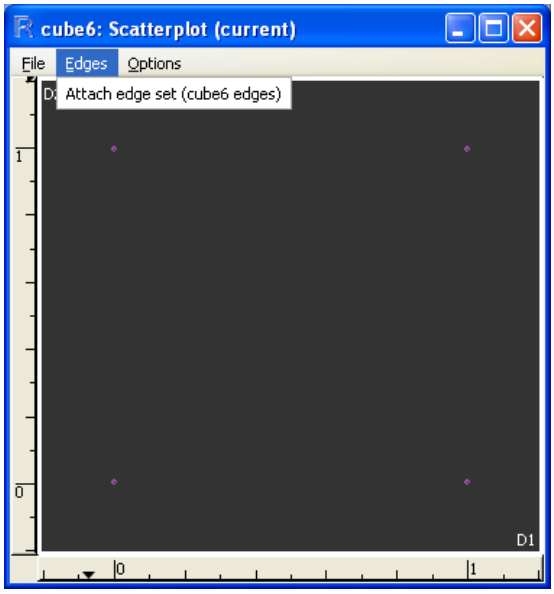


Figure 54.

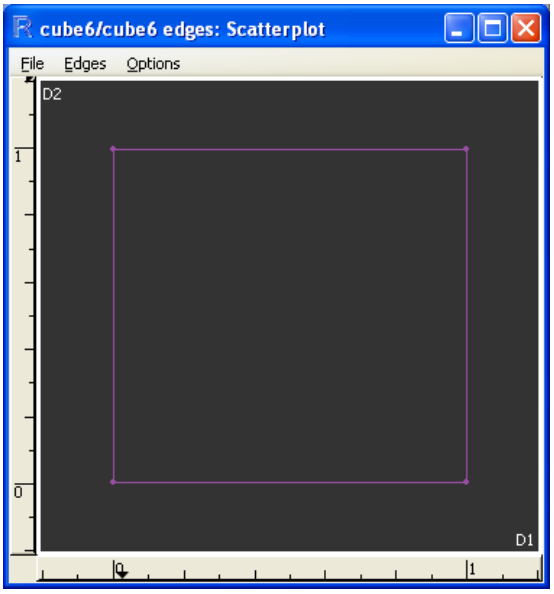


Figure 55.

Now on the console select [**View**][**2D tour**] which will show the projection. To inverstigate, we turn off the last 3 dimensions (D4, D5, D6) by clicking the selection box for each

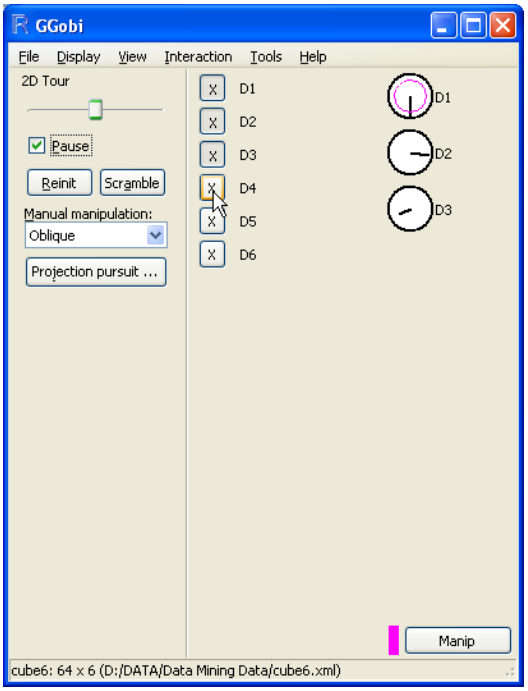


Figure 56.

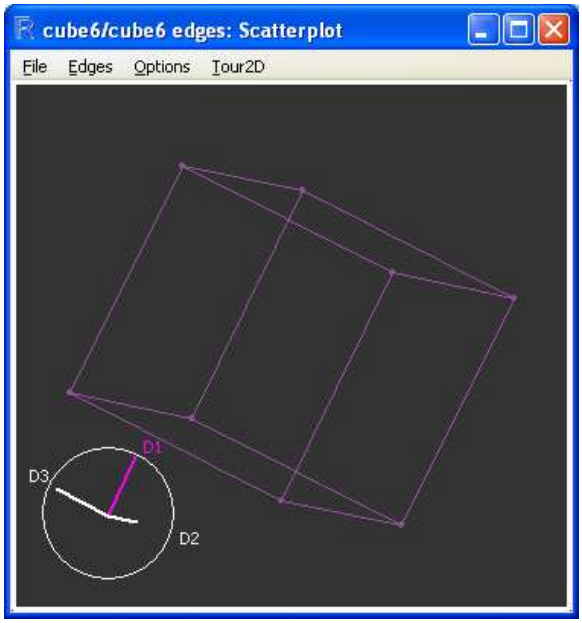


Figure 57.

Next, after pausing, select [**Interaction**][**Brush**], set [**Point brushing**] to *Off*, [**Edge brushing**] to *Color only* and select [**Persistent**]. Next move the brush to colour all the lines, return to the

[**Interaction**][**2D tour**], and deselect the [**Pause**].

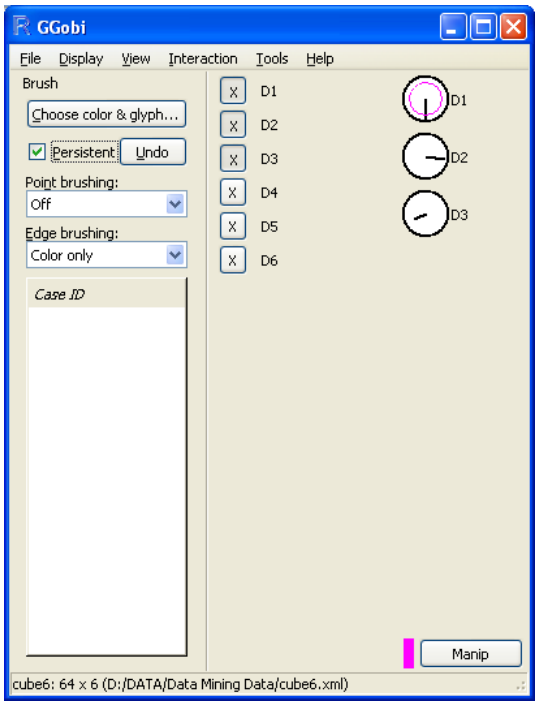


Figure 58.

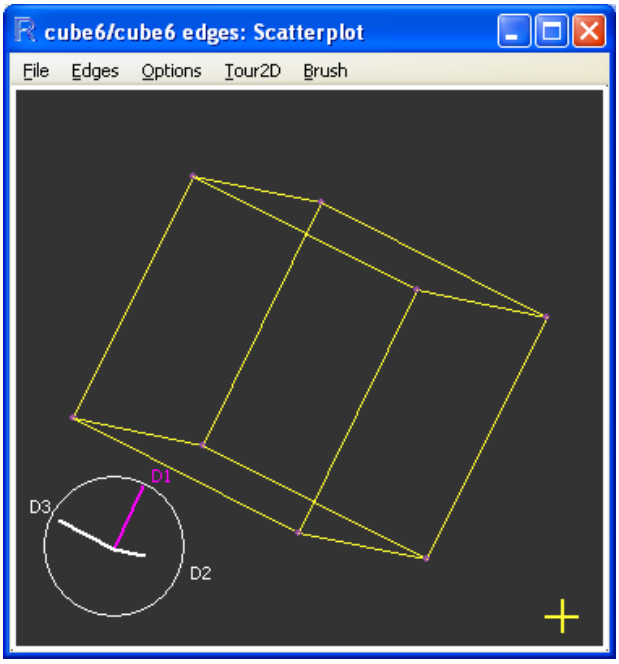


Figure 59. Yellow + for edge brushing

We now add in the D4, D5, D6 dimensions one at a time, each time brushing the new edges that arise (in a different colour obtained by selecting [**Choose color** & **glyph**]).

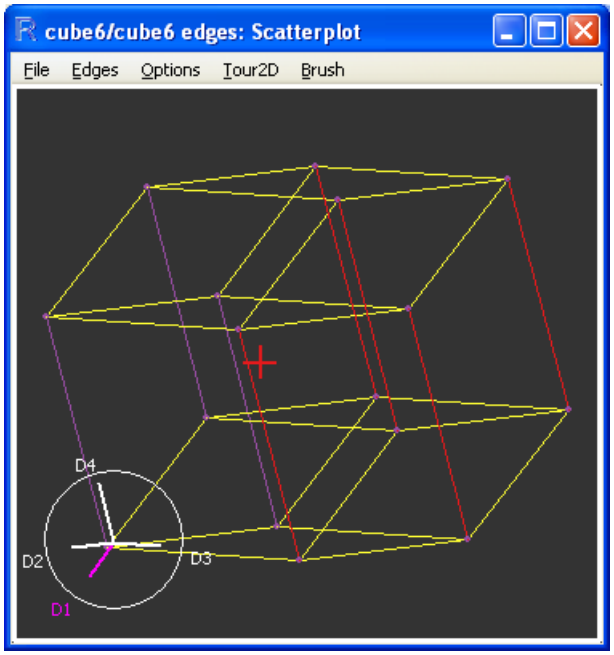


Figure 60. Showing 5 of the 8 4thdimension lines brushed

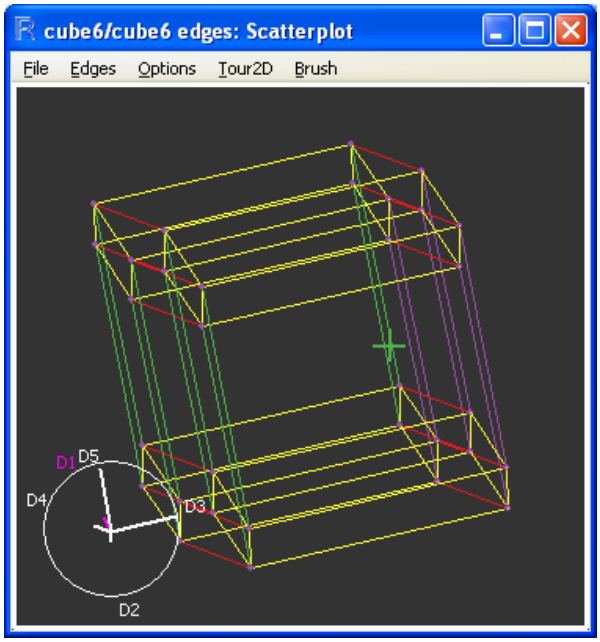


Figure 61. Showing 10 of the 16 5thdimension lines brushed

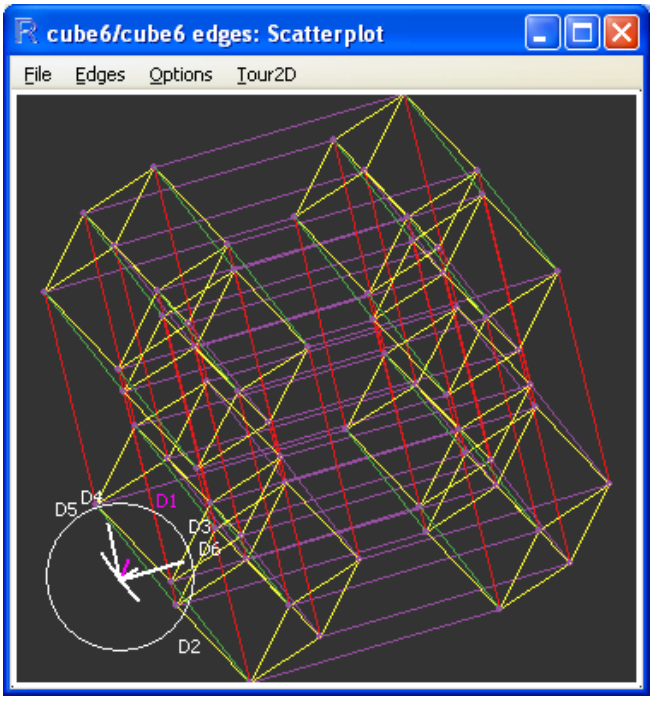


Figure 62. All 6 dimensions projected

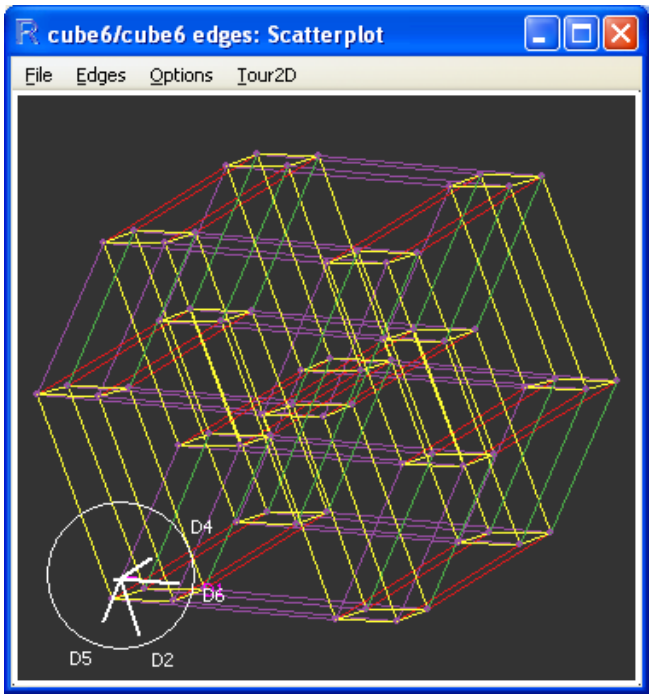


Figure 63. All 6 dimensions projected