Advanced Multivariate Continuous Displays and Diagnostics

ACTIVITY 37

This activity explores advanced multivariate plotting. With the advanced diagnostic functions, output graphs are more readable and useful for analyzing complex data structures.

Purpose

The purpose of this activity is to help you become familiar with customizing advanced arguments.

Learning Objective

Be familiar with some advanced graphical methods in R for continuous data, as well as some techniques for simple data mining

Required Resources

- R, R Studio
- Sample script file from class web site
- R Libraries (`RODBC`, `moments`, `lattice`, `MASS`, `classInt`, and `RColorBrewer`)

Time Allocated

110 minutes in class

Tasks

This activity begins assuming R Studio is open. If not, please start R Studio and load libraries `RODBC`, `moments`, `lattice`, `classInt`, `RColorBrewer`, *and* `MASS`.

A. Scatterplots

The scatterplot below (Figure 62) shows the relation between speed vs. volume at lower left and relation between speed vs. occupancy at loop detector ID 1594.

Next, we will break occupancy into certain amount of color intervals. R contains a function called `cut()` that divides the range of x into defined intervals. The intervals can then be attached to the dataframe by adding a column that defines each records interval. The idea of establishing intervals is to be able to assign different colors to the intervals providing a way to depict the variation in occupancy. To complete the addition of the 3rd dimension a vector of colors is required and can be completed using the `terrain.colors()` function which only requires the input of the number of colors that should be used to assign to the intervals.

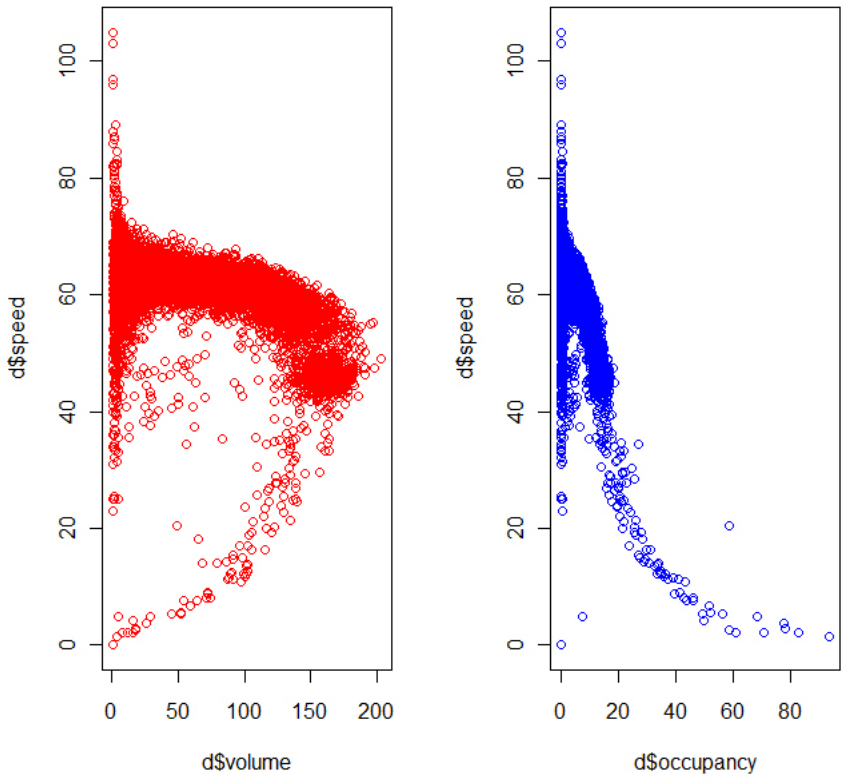

Figure 62 Scatterplots of speed versus volume (left), and speed versus occupancy (right)

Plotting the scatterplot only requires the call to plot as would normally be done, however the color argument is called using the numeric representation of the intervals, or `collist[as.numeric(d$col_occ)]` in the case of the accompanying script. Doing this establishes that the first interval, `[0,5]` receives the first color in the vector `collist` which was previously defined as the `terrain.colors()` function with 8 different colors. The resultant call returns Figure 63.

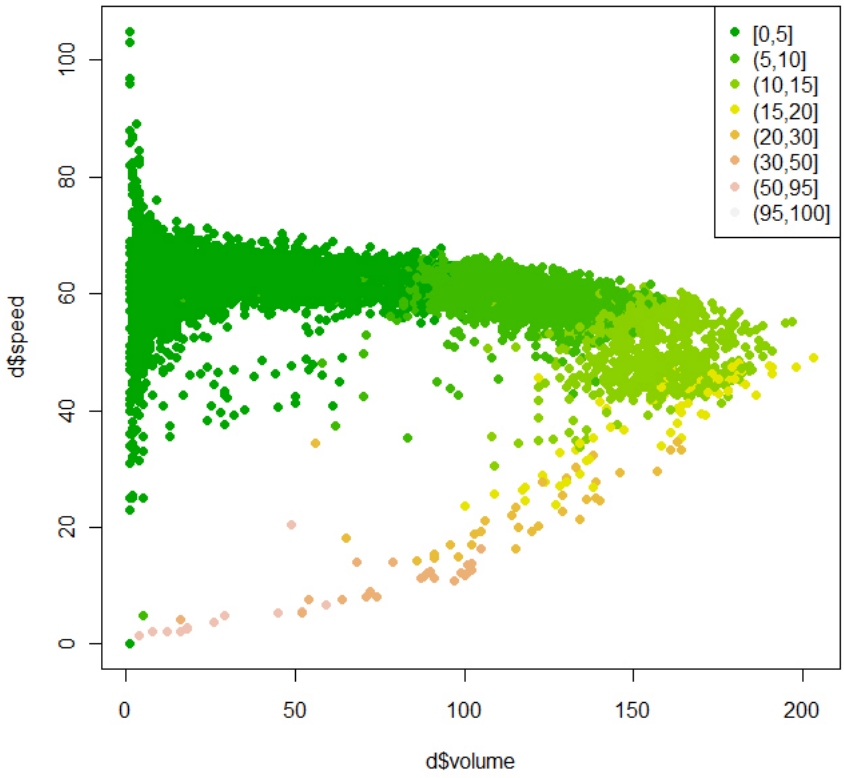

Figure 63 Scatterplot with third dimension as color

Similar to the change of color described in the previous discussion, other factors can be changed using functions found within the *graphics* package. In the case of Figure 64, the size of the data points are defined in the plot call using the function `symbols()`. Again, volume is plotted on the x-axis and speed on the y-axis with occupancy being added as the third dimension and reflected as the size of the data point with center (x, y).

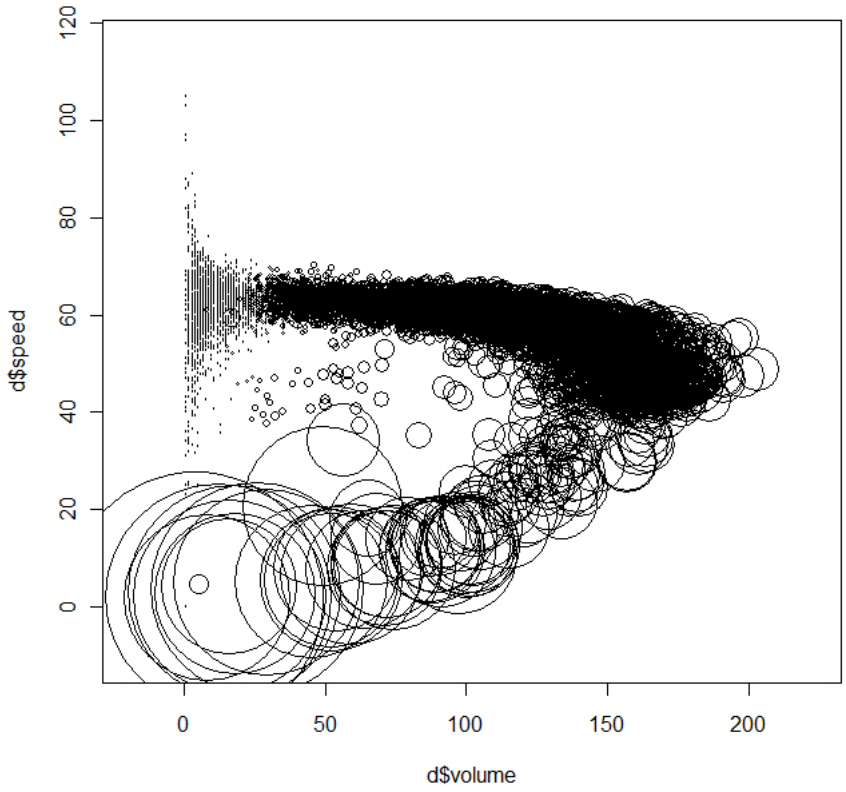

Figure 64 Scatterplot with third dimension as size

Another variation for the addition of a 3rd dimension is the addition of a second axis as demonstrated in Figure 65. The addition of the second axis is completed by using `par(new=T)` and the call for an additional axis is done using the `axis()` function. The `new=T` argument calls the next plot to be placed on the same graphic as the previous plot. Then the second axis values are placed by defining the location (the first argument of the `axis()` function) to be at the edge of the graphic. Defining the size of the graphic output with `par(mar=c())` makes the effort to establish the right side of the limit due to the user defined limits.

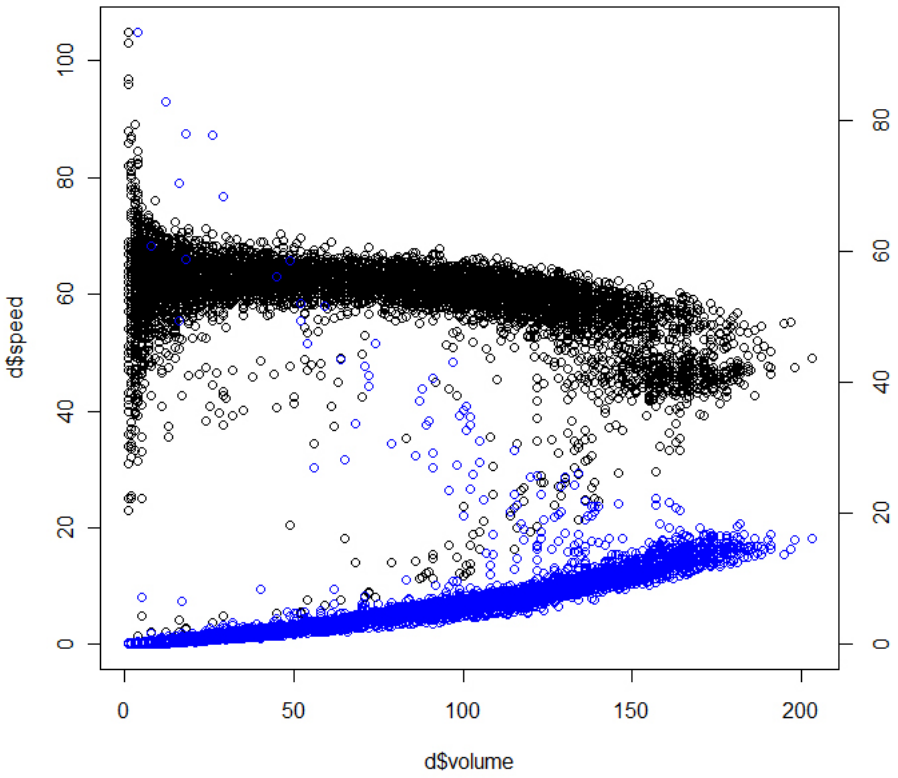

Figure 65 Scatterplot with third dimension as second axis

Earlier we established color variations using intervals. Another variation in color is to add transparency to the color as was done in Figure 66. This is accomplished by defining the color using the `rgb()` function. The key points of this function are the arguments `a=`, `r=`, `g=`, and `b=` which defines the level of the red, blue, and green primary colors. The alpha argument (`a=`) determines the level of transparency, the higher the value the more transparent. Take a few minutes to explore the variations available regarding this useful function.

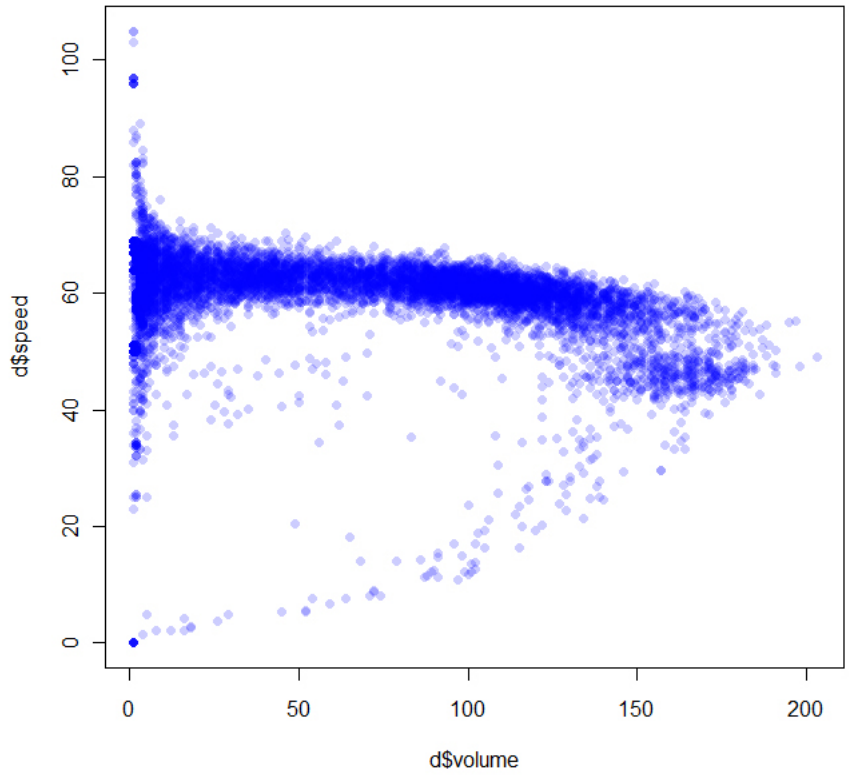

Figure 66 Scatterplot - overlay points with transparency

Another symbolic representation of a third dimension is the sunflower plot which plots a scatter plot of the non-duplicated values then over-plots the graphic demonstrating the shared value between multiple data records. For each repetition of value the function adds an additional 'petal' to depict the density of that data value. Figure 67 is a plot of the `sunflower()` function The call to plot is the function `sunflowerplot()` with similar arguments of the `plot()` function. The most notable argument is the ability to generate a rotation of the points based on a randomly assigned direction. This is an attempt to further limit what the sunflower plot attempts to negate which is confused overplotting.

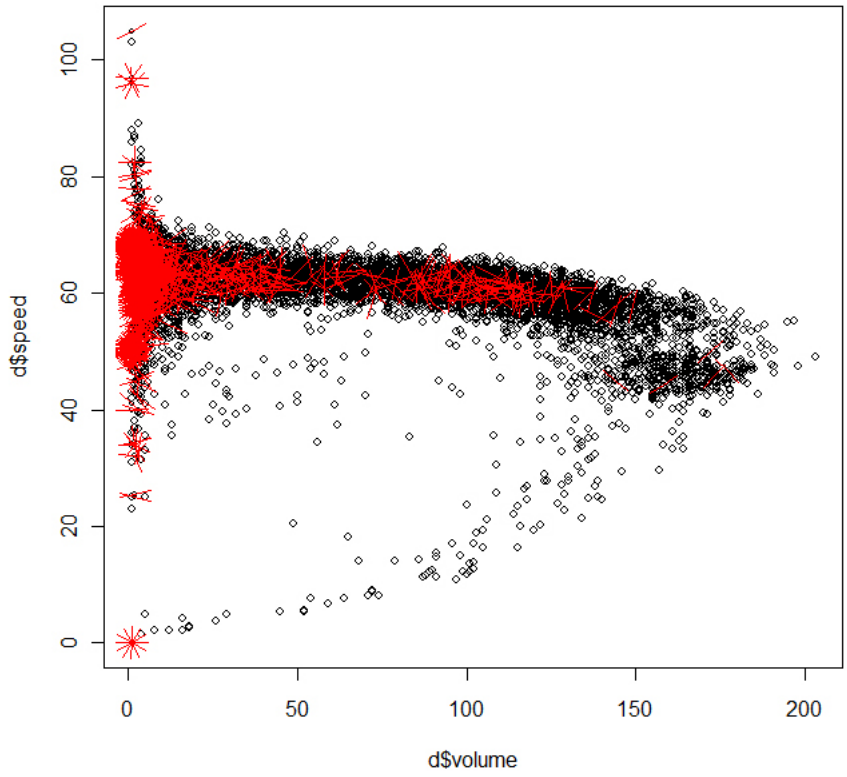

Figure 67 Scatterplot with sunflower overlay describing multiplicity of data points

As was noticed with the Figure 63 method of establishing breaks and assigning colors, the ability of the `terrain.color` function was limited regarding adjusting the range of colors. Using `brewer.pal()` allows substantial variability in the selection of colors. This function is found in the *RColorBrewer* library. Take a few moments to explore the color palette using the code from R Help that demonstrates the variability of the `brewer.pal()`.

B. Bivariate Scatterplots

This task primarily represents an introduction to the plotting method for bivariate graphs. Let's first use the basic plot function to plot simple scatter plots to compare these output plots. Use the accompanying script to generate the two 5 × 5 plots of the loop data set seen in Figure 68. Plotting the data in a minimal amount of graphic windows allows for quick comparison of the corridor, but does not provide detail regarding the specific sensors.

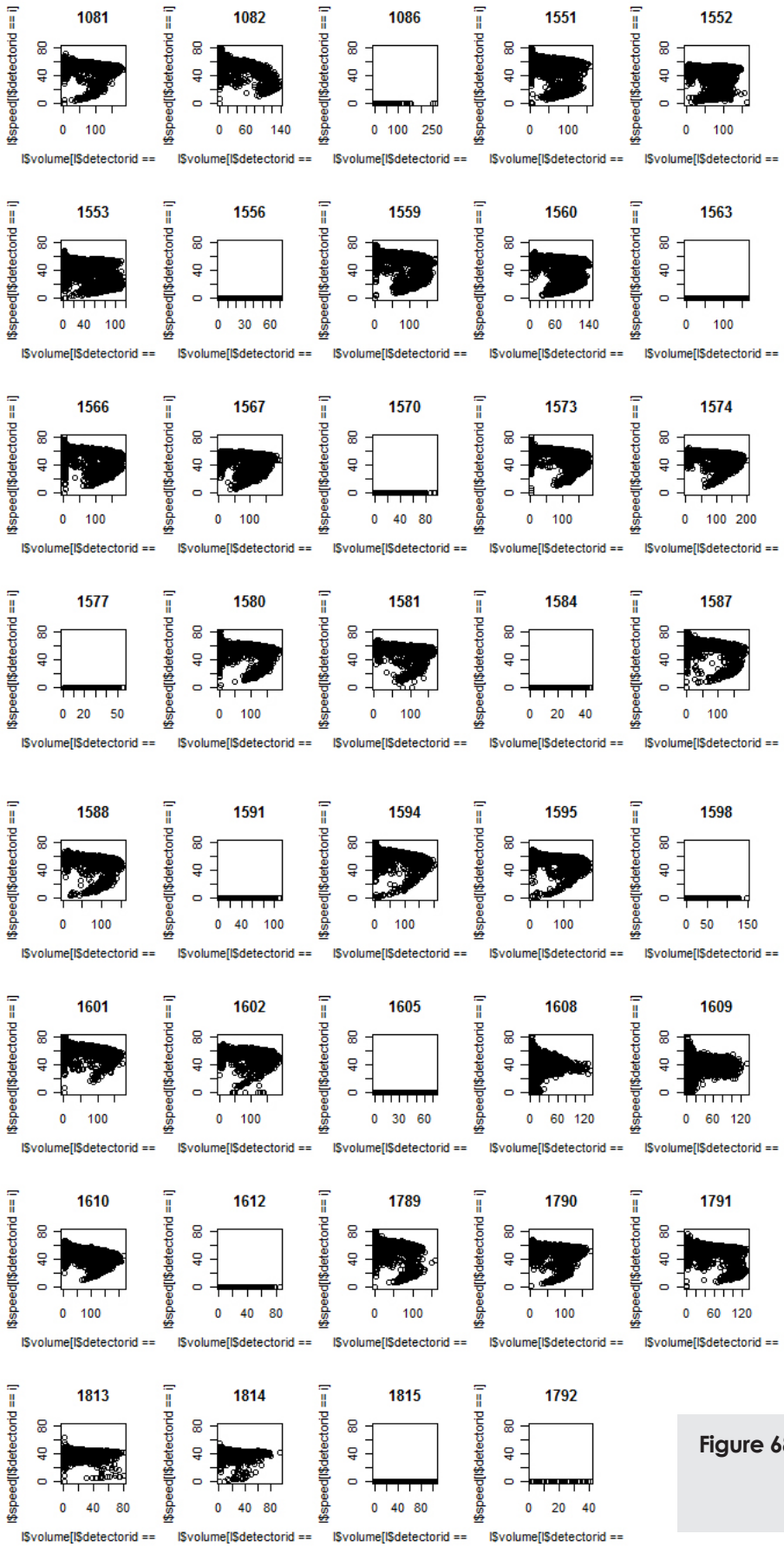

Figure 68 Bivariate plots of volume and speed from the loop dataset

A more refined graphic can be generated by xyplot within the *lattice* package. This function combines a set of arguments that include layout, aspects ratio, legends, axis annotation and many other details, and represents bivariate relation in a consistent way. An example of this function can be seen in Figure 69 (below) which generates scatterplots for the same data set as Figure 68.

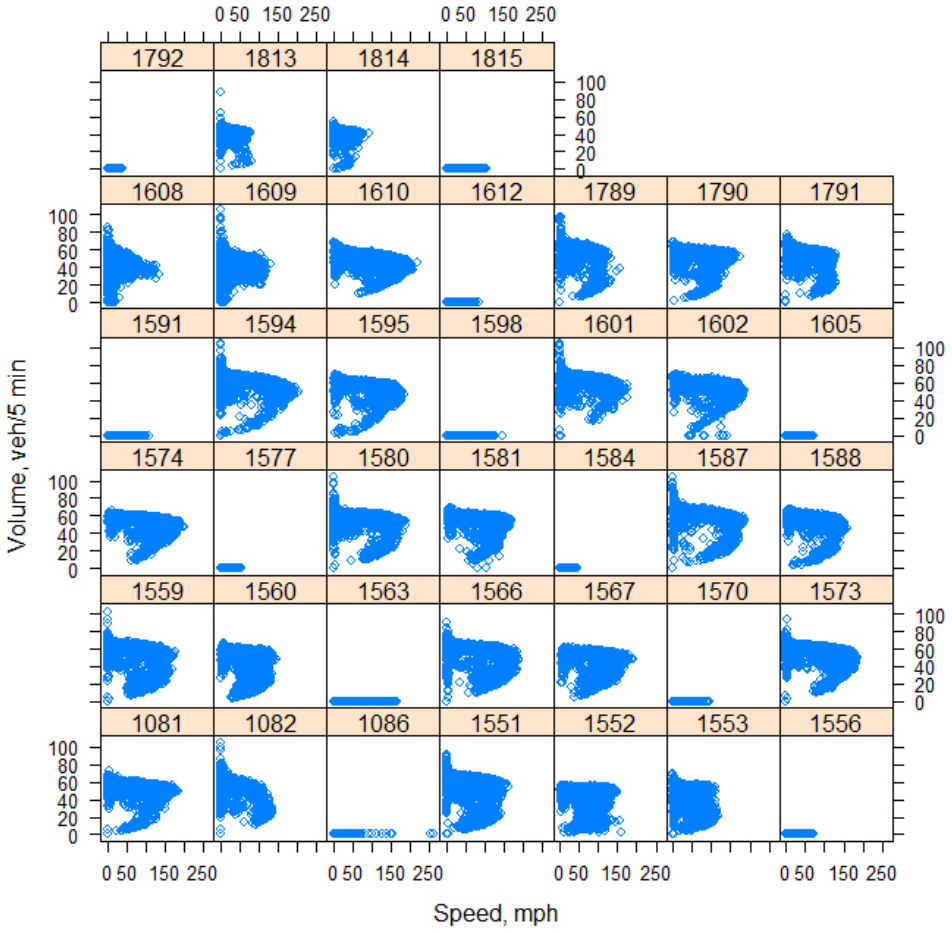

Figure 69 Graphic of speed and volume data from loop dataset using xyplot() in the lattice package

C. Breaks

The accompanying script further demonstrates the methods for establishing intervals by using other functions that have the same intention as the manual process presented and demonstrated by Figure 63. A key benefit to `classInterval` is the addition of variability regarding the calculation of breaks for the interval. The function calculates the break points based on user selected methodologies built into the function. Some of the available methodologies include: fixed, pretty based style, equal, and quantile.

The accompanying script also introduces the `findInterval()` function which takes the user defined breaks or the `classIntervals()` defined breaks and assigns each data point a bin number reflecting the interval that the data point has been defined to fall within.

D. Advanced Graphical Multivariate Displays

The final task within this activity is the introduction of more complex contour plots found within the *graphics* package. The first plot function is `image()` and generates a plot of the data in similar concept to that of a transparent graphic. Instead of adding the third dimension of the graphic using a third variable the graphic used a two-dimensional kernel density estimate function to establish a three dimensional matrix with the z variable being the estimated density of recurring values within the dataset.

The second and third plots are simplistic contour plots that demonstrate the 2-D kernel density estimation using contour lines. The `contour()` function generates a plot similar to a topographic map with the value of the line printed in the line. The number of levels can be controlled or where the levels occur using either the `nlevels=` or `levels=` arugments within the function. Similarly the levels can be controlled within the `filled.contour()` function using the same arguments. The concept of the filled contour and the simple contour are same, but the presentation of the filled contour lacks values within the graphic, only in the legend and the difference between the levels is defined by color (see Figure 70).

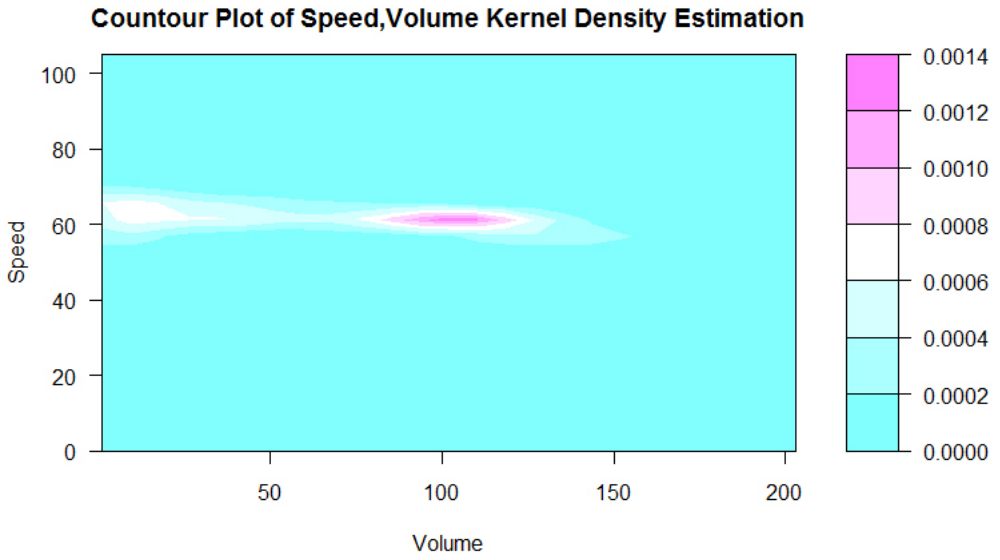

Figure 70 Filled contour plot of volume versus speed with third dimension of kernel density estimation

Finally, make sure to annotate your code and demonstrate your exploration of the functions within this activity. As always make sure your script executes.

Deliverable

This is a discovery activity (it helps you put together some things that you have learned to date). You don't need to expound too much on why it worked, but upload your R code cleaned and commented to the class dropbox.

Assessment

Your score is based on the following criteria:

Activity 37 Grading Rubric

	Excellent (10)	Good (8)	Poor (6)	NONE
Script	Organized, complete, accurate and executes.	Missing minor parts, but executes and is otherwise organized and accurate.	Missing significant portions of the activity, unorganized, inaccurate, but executes.	Code does not execute
Annotation	Annotations are complete and describe what the code is accomplishing.	Some annotations are incomplete or do not describe what the code is accomplishing.	No annotations were provided.	Code does not execute
Discussion / Commentary	Insightful discussion or commentary relating to the question at hand demonstrating student understanding of the task.	Discussion or commentary was incomplete.	Minimal to no discussion or commentary.	Code does not execute