Linked Micromaps

Quinn Payton, Marc Weber,
Michael McManus, Tom Kincaid

October 16, 2012

Contents

		Page
1	An initial linked micromap	
	o Example 1. National Linked Micromap of Education and Poverty: Statistical summary in flat file format and geographic data from a shapefile.	1
2	Quick plotting tips	
	o Playing with panel margins and window size	5
	o Transforming a dot plot into "bullets"	
	o Rearranging panels	
3	Preparing data for use with the library	8
	o Example 2. Texas ecoregions: Steps for simplifying spatial polygons	
	o Steps for simplifying very large spatial data	
4	Full list of adjustable attributes	11
5	Creating a new panel type	15
	o Example 3. National Linked Micromap of Lung Cancer Data with Confidence Intervals and Creating a New Graph for Comparing Time Periods: Statistical summary and geographic data are both in flat file formats.	
6	Group-categorized micromaps (**lmgroupedplot** function)	21
	o Example 4. Vegetation Coverage from the National Wadeable Streams Assessment: Statistical summary in flat file format and geographic data from a shapefile.	

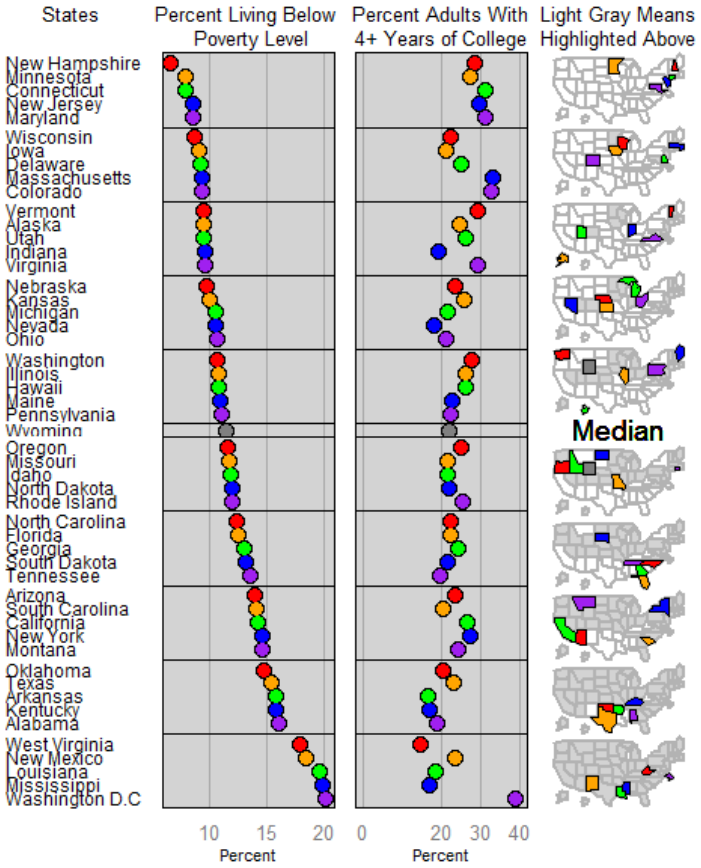
1 An initial linked micromap

First, we load the library and data:

```
library(micromap)
```

lmplot(stat.data, map.data, panel.types, panel.data, map.link, ord.by, grouping, …)

The first key to using either of these functions is to have your data set up correctly. For this first example, we would like to display the state names, a graph illustrating its poverty level, a graph illustrating its percentage of college graduates, and a micromap indicating which states are being referenced. In order to do this, all we need is a table with state names and estimates of each of the two metrics we're interested in. The dataset "edPov" included in the **micromap** library is in this form:




```
data("edPov")
head(edPov)
```

state	ed	pov	region	StateAb
Illinois	26.1	10.7	MW	IL
Indiana	19.4	9.5	MW	IN
Iowa	21.2	9.1	MW	IA
Kansas	25.8	9.9	MW	KS
Michigan	21.8	10.5	MW	MI
Minnesota	27.4	7.9	MW	MN

Next, we need a table of polygons to map. We can use the **create_map_table** function to take a spatial object file and create a small efficient table in the form that the lmplot function can use or we can construct our table directly. In order to do this successfully our table must end up with 4 essential columns that must be named as follows: ID; coordsx; coordsy; and poly. The ID column is what links to the table of statistics. The poly column is used to identify state polygons for the same ID (otherwise R will connect all the vertices with some odd looking results). For this first example we will use the "states" shapefile included with the library and use **create_map_table** in order to get the data in the right format.

Some preliminary steps are usually required to use the create_map_table function. First, many spatial objects are quite detailed, far more detailed than what is needed for a micromap. The size and complexity of these files will drastically reduce the speed at which plots can be produced and, in some cases, overwhelm R with the amount of data being handled causing it to crash. One option for reducing shapefile complexity is to use a "thinning" function from the **maptools** library which can be used to reduce the size and complexity of a spatial object. See Section 3, "Preparing data for use with the library", for more details and an example. The USstates data is very simple and therefore we will hold off discussion of the thinning function until later.

The second (and much simpler) step in successfully using the create_map_table function is assigning an explicit ID to each polygon. The data table associated with the spatial object must have an ID column (literally called "ID") to name each polygon. This is the column that will be used to link the information from the stat table to this built map

table. With this in mind we can check the data table from our USstates file by using the following @data syntax. The "@" syntax refers to grabbing the data object stored in this "slot" of a shapefile object. To examine the other slots of this shapefile one would use the slotNames() function.

```
data("USstates")
head(USstates@data)
```

	ST	ST_NAME	AREA_KM	PERIM_KM
0	AK	Alaska	1506038	60260.64
1	AL	Alabama	133761	2354.6
2	AR	Arkansas	137733.7	2172.208
3	AZ	Arizona	295267.5	2395.409
4	CA	California	409603.3	5682.304
5	CO	Colorado	269599.9	2100.092

Since there is no ID column in this table we can insert a second argument into **create_map_table** identifying which column we would like to use as our ID. The "ST" column will be very useful in linking to our stats table so that will be used:

```
statePolys <- create_map_table(USstates, IDcolumn='ST')
head(statePolys)
```

	ID	region	poly	coordsx	coordsy	hole	plug
1	AK	1	1	2	5	0	0
2	AK	1	1	7	10	0	0
3	AK	1	1	4	12	0	0
4	AK	1	1	7	15	0	0
5	AK	1	1	4	15	0	0
6	AK	1	1	4	17	0	0

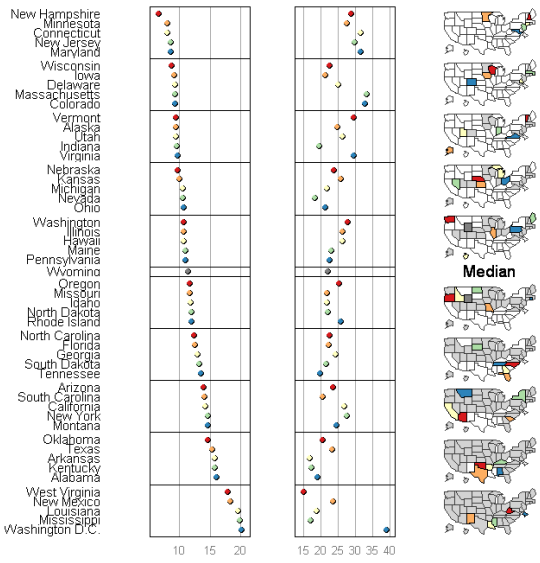
From here we can create the graph desired. To graph our poverty and college degree metrics we must specify the type of graph to be used. As of now, there are only 6 types of graphs built in:

- Dot plots (with or without confidence limits)
- Bar plots (with or without confidence limits)
- Box summary (5 and 7 point)

Additional graph types will be built and included as needed. Users can create and include new graph types as is explained in later in section 5, "Creating a new panel/graph type".

The most basic version of an LM plot can be made with this code:

```
lmplot (stat.data=edPov,
        map.data=statePolys,
        panel.types=c('labels', 'dot', 'dot','map'),
        panel.data=list('state','pov','ed', NA),
        ord.by='pov',
        grouping=5, median.row=T,
        map.link=c('StateAb','ID'))
```



A full explanation of all the function arguments is provided below but 3 things should be made clear here.

- panel.data is list of lists to specify which columns of the stat.data table to use in filling out the panels. For a panel needing multiple columns you would enter a sublist. There needs to be an entry for every panel even when specific data from the stat table isn't supplied by the user. As you can see here, the map panel has an NA entry. **These entries cannot be left out.**
  - Note: The order of the entries in panel.data and panel.types must coincide. If we want to rearrange the order of the panels, the entries of both panel.data and panel.types need to be rearranged.
- map.link is a vector specifying which column from the stat table matches which column for the map table respectively. In this example the StatAb column from the stat table matches each data line to its associated polygons in the map table labeled by matching entries in that table's ID column.
- Setting median.row=TRUE will insert a median row. As is noted below, this will override the default to force the x and y axis coordinates to stay respective to each other which will probably cause distortion in the maps being presented. Adjusting panel.width should be used to manually correct this.

This initial call will rarely result in high quality, "final" looking results. From here we can make notes on what adjustments would make this look better. We are attempting to replicate what Dan Carr (http://mason.gmu.edu/~dcarr/) created previously and so we must make some adjustments.

As with most R functions, a few plot wide adjustments can be made by simply adding in extra arguments in the function call (such as plot.height, colors, and map.color2 in this example). To adjust the individual panels, however, we must make a "list of lists" specifying which panel we are adjusting and then which attributes we would like to modify.

To make this more intuitive here is a quick and simple example. Suppose we just want to change the text alignment in panel 1 and the graph background colors in panels 2 and 3. First we make a list for each of these panels specifying the changes we would like to make with the first entry of each list specifying which panel is to be altered:

list(1, align='left**)**
list(2, graph.bgcolor='lightgray'**)**
list(3, graph.bgcolor='lightgray'**)**

Now we compile these lists into a "list of lists":

list(list(1, align='left), list(2, graph.bgcolor='lightgray'), list(3, graph.bgcolor='lightgray') **)**

Now we can just add *"panel.att= list(list(1, align='left), list(2, graph.bgcolor='lightgray'), list(3, graph.bgcolor='lightgray'))"* to our lmplot function call and see the changes. We have a lot more changes to make, though, so we might as well implement all of them at once. The following code was used to make the graph seen at the beginning of this document:

Note "\n" inserts a carriage return in the header. **Actual carriage returns have the same effect but should not be used as this will result in lmPlot being unable to properly align panels.**
e.g. **use:**
"… header='Percent Living Below \n Poverty Level' …"
not:
"… header='Percent Living Below
Poverty Level' …"

```
lmplot(stat.data=edPov,  map.data=statePolys,
        panel.types=c('labels', 'dot', 'dot','map'),
        panel.data=list('state','pov','ed', NA),
        ord.by='pov',  grouping=5,
        median.row=T,
        map.link=c('StateAb','ID'),

        plot.height=9,
        colors=c('red','orange','green','blue','purple'),
        map.color2='lightgray',

        panel.att=list(list(1, header='States', panel.width=.8, align='left', text.size=.9),
                       list(2, header='Percent Living Below \n Poverty Level',
                               graph.bgcolor='lightgray', point.size=1.5,
                               xaxis.ticks=list(10,15,20), xaxis.labels=list(10,15,20),
                               xaxis.title='Percent'),
                       list(3, header='Percent Adults With\n4+ Years of College',
                               graph.bgcolor='lightgray', point.size=1.5,
                               xaxis.ticks=list(20,30,40), xaxis.labels=list(20,30,40),
                               xaxis.title='Percent'),
                       list(4, header='Light Gray Means\nHighlighted Above',
                               inactive.border.color=gray(.7), inactive.border.size=2,
                               panel.width=.8)))
```

This seems pretty good. We have two options for storing this final figure. In the lmplot function call we can add a line the final list of panel attributes specifying a filename (and resolution if desired) as follows:

```
lmplot(stat.data=edPov, … , print.file='myFigure.tiff', print.res=300)
```

The ".tiff" tells the lmplot function that a tiff file is requested. Pdf, jpeg, and png files may also be produced in a similar manner. The other option is to store our function output in an R object as we build it. When we have results we are satisfied with we can use the **printLMplot** function to print it out:

```
myPlot  <- lmplot(stat.data=edPov, … )
printLMPlot(myPlot, name='myFigure.tiff', res=300)
```

2 Quick Plotting Tips

Quick tips for making higher quality figures with the lmplot function:

- Panel widths will almost certainly need to be adjusted in order to have the text correctly fit across the panel. Text in the labels and ranks panels are defaulted to fit vertically correctly. If text is overlapping vertically, it may be because enough vertical space isn't being provided on the plot. Adjusting plot.height (defaults to 7) and plot.pGrp.spacing (defaults to 1) can, and should, be used to correct this.

- Adjusting right and left panel margins are perhaps the most useful tool in making a plot look nice. Panels are printed out from left to right and many times a plot will overlap its preceding neighbor; therefore bringing in the left margin by setting left.margin=-.5 or even left.margin=-1 can be very helpful in clearing out white space. For neighboring graphs (such as those in the states graphic example above) adjusting the left panel's right margin and the right panel's left margin can cause them to share a border thus appearing attached.

- As is noted elsewhere, the micromaps are set to have the x and y axis coordinates set respective to each other. This causes quite a few unintended consequences one of which is micromap "shrinkage" if the panel.width is not wide enough. If your maps look too small at first, expanding the panel width will probably enlarge your graph quite a bit.

- Also, due to an artifact (some might call it a bug) in ggplot2, this coordinate "respectivity" in the micromaps goes away when adding a median row. Therefore, one should be careful in such situations and take care in setting the panel width of the map panel to correct any distortion that may present itself.

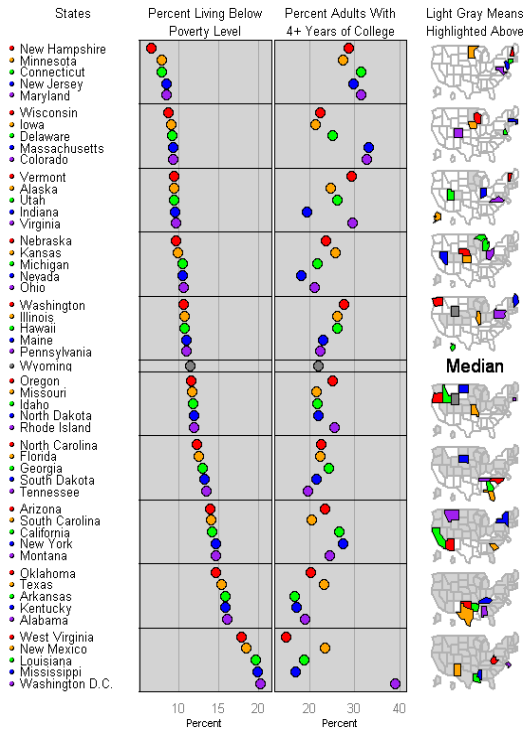
We can use a quick example to further illustrate the power of these options. Suppose we wish to add a series of color coded bullets in front of our state names in the original graphic we began with. Essentially this is just a dot with all the data points at the same value (note: we must insert a dummy column in our stat table).

edPov$points <- 0

	state	ed	pov	region	StateAb	points
IL	Illinois	26.1	10.7	MW	IL	0
IN	Indiana	19.4	9.5	MW	IN	0
IA	Iowa	21.2	9.1	MW	IA	0
KS	Kansas	25.8	9.9	MW	KS	0
MI	Michigan	21.8	10.5	MW	MI	0

Now this dot plot just needs a white background, without axis lines, axis text, and the border set to "white" to blend in with the background color. If we then drastically reduce the panel width and bring in the margins it will seamlessly blend with its neighboring labels panel.

lmplot(stat.data=edPov, map.data=statePolys,



Note the correspondence between the number of arguments for panel.types and panel.data

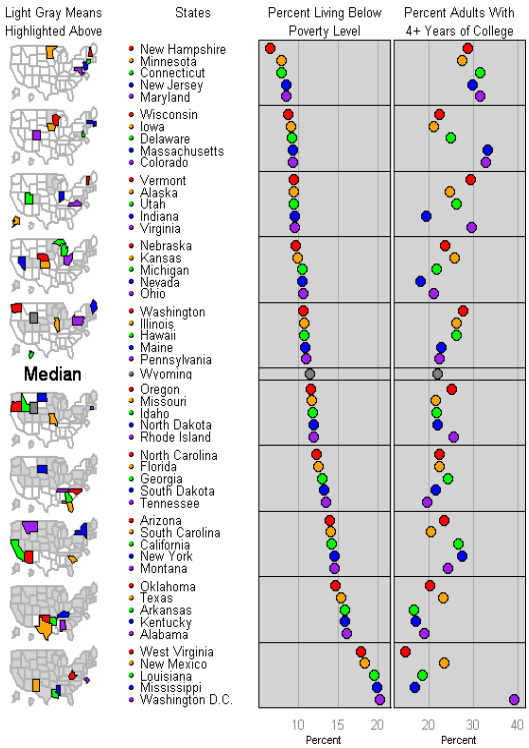
```
panel.types=c('dot',  'labels', 'dot', 'dot', 'map'),
panel.data=list('points', 'state', 'pov', 'ed', NA),
map.link=c('StateAb','ID'),
ord.by='pov',
grouping=5,
median.row=T,

plot.height=9,

colors=c('red','orange','green','blue','purple'),
map.color2='lightgray',

panel.att=list(list(1, panel.width=.15, point.type=20,
                    graph.border.color='white',
                    xaxis.text.display=F, xaxis.line.display=F,
                    graph.grid.major=F),

               list(2, header='States', panel.width=.8,
                    align='left', text.size=.9),

               list(3, header='Percent Living Below\nPoverty Level',
                    graph.bgcolor='lightgray', point.size=1.5,
                    xaxis.ticks=list(10,15,20), xaxis.labels=list(10,15,20),
                    xaxis.title='Percent'),

               list(4, header='Percent Adults With\n4+ Years of College',
                    graph.bgcolor='lightgray', point.size=1.5,
                    xaxis.ticks=list(20,30,40), xaxis.labels=list(20,30,40),
                    xaxis.title='Percent'),

               list(5, header='Light Gray Means\nHighlighted Above',
                    inactive.border.color=gray(.7), inactive.border.size=2,
                    panel.width=.8)))
```

Note the numbers have all been increased to adjust for the addition of a new panel

A final note, we can easily rearrange panels by changing the order of the panel.types and panel.data, then simply re-number the panel attributes section. Let's move the maps to the first panel:

```
lmplot(stat.data=edPov, map.data=statePolys,
       panel.types=c('map', 'dot',  'labels', 'dot', 'dot'),
       panel.data=list(NA, 'points', 'state', 'pov', 'ed'),
       map.link=c('StateAb','ID'),
       ord.by='pov',
       grouping=5,
       median.row=T,

       plot.height=9,

       colors=c('red','orange','green','blue','purple'),
       map.color2='lightgray',

       panel.att=list(list(2, panel.width=.15, point.type=20,
                           graph.border.color='white',
                           xaxis.text.display=F, xaxis.line.display=F,
                           graph.grid.major=F),

                      list(3, header='States', panel.width=.8,
                           align='left', text.size=.9),

                      list(4, header='Percent Living Below\nPoverty Level',
                           graph.bgcolor='lightgray', point.size=1.5,
```

```
        xaxis.ticks=list(10,15,20), xaxis.labels=list(10,15,20),
        xaxis.title='Percent'),

    list(5, header='Percent Adults With\n4+ Years of College',
        graph.bgcolor='lightgray', point.size=1.5,
        xaxis.ticks=list(20,30,40), xaxis.labels=list(20,30,40),
        xaxis.title='Percent'),

    list(1, header='Light Gray Means\nHighlighted Above',
        inactive.border.color=gray(.7), inactive.border.size=2,
        panel.width=.8)))
```



3 Preparing Data for use with the library

Example Steps for simplifying spatial polygons in spatial data set for the lmplot function:
Download an example shapefile – we'll use level 3 ecoregions of Texas as an example:
ftp://ftp.epa.gov/wed/ecoregions/tx/tx_eco_l3.zip

Method for simplifying polygons using simplification in GIS software such as ArcMap

- Add the shapefile to an ArcMap session
  File > Add Data > navigate to where you downloaded file

- Open the Simplify Polygon tool in ArcToolbox
  Generalization > Simplify Polygon

- Choose simplification algorithm, maximum allowable offset, and minimum area. Point remove is quick, bend simplify can take longer but gives more ascetically pleasing results
  Simplification Algorithm: POINT_REMOVE
  Maximum Allowable Offset: 1000 Meters
  Minimum Area: .001
  Handling Topological Errors: RESOLVE_ERRORS

- Read resulting shapefile into R using readOGR (uses readOGR from rgdal, loaded with the **micromaps** library):
  txeco <- readOGR(".","tx_eco_l3")

- Create an ID column in your spatialdataframe for the create_map_table function
  txeco@data$ID <- txeco@data$US_L3CODE

Method two is to simplify polygons within R, and this can be done in two ways. One way is to use the thinnedSpatialPoly function in maptools library. The other way is to use the gSimplify method in rgeos, which includes the step of converting a SpatialPolygon object in R into a SpatialPolygonDataFrame. The create_map_table function in the **micromap** library only works on a SpatialPolygonDataFrame.

- Read the shapefile into R from your working directory
  txeco <- readOGR(".","tx_eco_l3")

- Create an ID column in your spatial dataframe for the create_map_table function
  txeco@data$ID <- txeco@data$US_L3CODE

- Load the maptools library
  library(maptools))

- Plot the spatial object in R to see how it looks by default
  plot(txeco)

- Run the thinnedSpatialPoly function in maptools. You'll have to play with values that work best for the two parameters 'tolerance' and 'minarea'. If you have rgeos loaded it's advisable to set avoidGEOS=T but users can explore options and functionality for thinning using rgeos. Read function notes for more details.
  myPolys1<-thinnedSpatialPoly(txeco,tolerance=1000, minarea=0.001, topologyPreserve = F, avoidGEOS = T)
  plot(myPolys1)

- Alternatively, load the rgeos library
  library(rgeos)
  txeco2<-gsimplify(txeco, 10000, topologyPreserve=T)
  txeco2<-SpatialPolygonsDataFrame(txeco2, txeco@data)

Notice the difference in resolution, particularly along the southeastern coastline, before and after thinning using rgeos.

In the introductory code, we show code to make a basic and publication lmpot of the Texas ecoregions.

With either method, the function 'create_map_table' will result in a less memory-intensive map table being built for linked micromaps. The function deletes polygons that are smaller than a certain fraction of the total area of the full map when building the table (default is .001, users can play with but will need to be careful in adjusting levels if using function so that they don't delete any areas that should be represented in the linked micromap plot).

- We'll run the create_map_table.r function with the default threshold

```
txeco <- create_map_table(txeco, poly.thresh=.001)
```

Steps for simplifying very large spatial data:

For very large data you need to take extra steps to get manageable spatial data for use in linked micromaps. We'll use level 3 ecoregions for the conterminous US as an example. Note that these are one example of steps that work, other combinations of steps could possibly work better for other data – the point is to get rid of very small features and simplify line work as much as possible.

ftp://ftp.epa.gov/wed/ecoregions/us/Eco_Level_III_US.zip

In ArcMap:

- Get rid of the state boundaries:
  - Open Dissolve tool in toolbox
    Generalization > Dissolve
  - Choose US_L3CODE as the field to dissolve on
- Simplify newly created feature
  - Open Simplify Polygon tool
  - Choose simplification algorithm 'Bend Simplify', Reference Baseline 100 kilometers, minimum area 100 square kilometers, and handling toplogical errors 'resolve errors'
- Now simplify features you just created again, but using a different simplification algorithm
  - Open Simplify Polygon tool
  - Choose simplification algorithm 'Point Remove', Maximum allowable offset 10,000 meters, minimum area 10,000 square meters, and handling toplogical errors 'resolve errors'

This will create a sufficiently simplified shapefile to use with the lmplot function.

In R:
The best option for getting a sufficiently simplified spatial object that still looks reasonable is to use ArcMap. We have found it difficult to use existing simplification algorithms available through R packages to create visually acceptable, as well as simple enough, spatial objects for the lmplot function. However, methods to try in R are available in both maptools and rgeos library, such as:

```
eco3 <- thinnedSpatialPoly(eco3, tolerance=50000, topologyPreserve=TRUE, avoidGEOS=FALSE)
eco3 <- thinnedSpatialPoly(eco, tolerance=50000, minarea=100,avoidGEOS = TRUE)
eco3 <- gSimplify(eco3, tol=50000, topologyPreserve=TRUE)
```

Note that if using rgeos package you will want to check the validity of your shapefile topology , i.e.

```
gIsValid(eco3)
```

If you don't have valid topology, you will need to fix topology errors in your shapefile
If you try gSimplify method, you'll need to promote the object to a SpatialPolygonsDataFrame in R like:

```
df <- eco@data # your original SpatialPolygonsDataFrame prior to thinning
eco3 <- SpatialPolygonsDataFrame(eco2, df)
```

Using freely available online tool MapShaper:
Available here: http://mapshaper.com/test/MapShaper.swf
Note 'alpha version' on tool, contact tool authors with questions, we simply point out the resource as an available method for shapefile simplification outside of proprietary GIS software

For further reading on polygon simplification, we refer users to the following papers:
Douglas, D. and Peucker, T. (1973). Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. The Canadian Cartographer 10(2). 112-122.

Harrower, M. and Bloch, M. (2006). MapShaper.org: A Map Generalization Web Service. IEE Computer Graphics and Applications 26(4). 22-27.

Mansouryar, M. and Hedayati, A. (2012). Smoothing Via Iterative Averaging (SIA) A Basic Techniqu for Line Smoothing. International Journal of Computer and Electrical Engineering 4(3), 307-311.

Technical paper, ESRI, "Automation of Map Generalization: The Cutting-Edge Technology," 1996. It can be found in the White Papers section of ArcOnline at this Internet address:http://downloads.esri.com/support/whitepapers/ao_/mapgen.pdf

4 Full list of adjustable attributes

Attribute arguments recognized by the lmplot function:

cat – category column within stats table for a categorization type linked micromap.
colors – The color palette used within each perceptual group. (e.g. brewer.pal(5, "Spectral"))
grouping (required)– The number of lines per perceptual group. E.g. simply entering "5" will put 5 lines in each perceptual group or you can enter c(5,6,5,4) to have disproportionate numbers of lines in each group.
map.all – By default, the lmplot function will only plot the polygons associated with data in the stats table. Setting **map.all=T** will tell it to show all the polygons in your polygon table regardless of whether they are actively referred to.
map.color—The color to be used to fill in current region being described in a grouped plot
map.color2—The color to be used to fill in previously referenced polygons in the map panel
map.data (required) – Data table (probably created by "create_map_table" of polygon data
map.link (required) – A vector specifying which column from the stat table matches which column for the map table repectively (e.g. "c('StateAb', 'ID')")
median.row – Specifies whether a median row should be included. If an odd number of data lines are supplied a data line itself will be used as the median otherwise median entries will be calculated from the supplied data. Note that without a median row maps are forced into proper size. However, an artifact in ggplot2 removes this feature when a median row is added and so a user must use the **panel.width** (and **left.margin/right.margin** panel attribute) specifications in order to make map coordinates not distorted. (defaults to FALSE)
ord.by, grp.by (required) – ord.by specifies the stats.data column to be ranked for the ordering of the figure. grp.by is used for grouped plots in order to specify which data table column to sort the figure by.
panel.att – a list of panel specific attributes to be altered (described in more detail below)
panel.data (required) – A list of lists to specify which columns of the stat.data table to use in filling out the panels. For a panel needing multiple columns you enter a sublist. There needs to be an entry for every panel even when specific data from the stat table isn't supplied by the user. That is to say map and rank panels (as well as user created panel types) should have NAs.

e.g. panel.data=list('State', list('Estimate', 'Lower.Bound', 'Upper.Bound'), NA)

panel.types (required) – A vector specifying the panels of the plot. Note: each "panel.type" (e.g. 'map', 'labels', 'dot_cl', etc.) is the name of a function that will be called to create that panel. Therefore a user can create a new panel type (e.g. "new.graph.type") and the lmplot function will automatically go look for and call that function just by changing the entry here. Page down for more on this…
plot.footer – Not implemented yet
plot.footer.size – Not implemented yet
plot.footer.color – Not implemented yet
plot.grp.spacing – The verticle spacing between groups measured in lines. Defaults to 1.
plot.pGrp.spacing – The spacing between perceptual groups. "1" (the default) implies standard spacing.
plot.header – Not implemented yet
plot.header.size – Not implemented yet
plot.header.color – Not implemented yet
plot.height – The height of the plot window. (Defaults to 7) NOTE: I'm not sure what units width and height are measured in but the defaults of 7 & 7 make a rectangle so they are certainly not relative to each other!
plot.width – The width of the plot window. (Defaults to 7)
print.file – The full file name (i.e. including extension) to save the resulting figure to if one wishes. The extension tells the lmplot function which type of printing function to run. Pdf, tiff, png or jpeg are all recognized
print.res – The resolution desired for the resulting file
rev.ord – reverse the order for ranking the plot.
stat.data (required) – Data table of statistic
two.ended.maps – The resulting micromaps will highlight previously referenced polygons (see **map.color2**) up to the median perceptual group and then switch to highlighting all polygons that are still to be referenced later.

panel.att is a list object (simply referred to as "a" throughout the function) which contains a sublist of specifications for each panel. Some attributes are standard for all panel types (e.g. header, graph color, etc.), while other options are only available to alter for certain panels (bar size, point type, etc.). If a user tries to alter a panel specific attribute that isn't recognized (e.g. bar size on a dot plot), it is ignored and a warning is printed.

Standard attributes

graph.bgcolor –The background color within any graphs being drawn

graph.border.color – *(Defaults to "Black" on graphs, no borders are shown on maps, labels and ranks)*Alters the border color on graphs. Note this can be used to hide borders on graphs by setting it equal to white or whatever the specified panel background color is.

graph.grid.major – A boolean variable stating whether major grid lines should appear in the graph. (T/F or 0/1 should both work) *(Defaults to "TRUE" for graphs, "FALSE" for all others)*

graph.grid.minor – (see above)

panel.att – A list of panel specific attributes. These are to be entered as a list of lists, with the first entry of each sublist specifying with panel's attributes are being altered:

```
E.g. panel.att=list(    list(1,  …),
                        list(2, …),
                        …
                        list(n, …)
                    )
```

The following attributes can be specified:

left.margin, right.margin – set panel specific panel margins individually.

panel.bgcolor – The back ground color in each panel

panel.footer - ! panel footer not implemented yet !

panel.footer.size

panel.footer.color

panel.header – A title for the whole panel

panel.header.size – Size **relative to default.** All panels should have the same size header to keep proper alignment between panels. **If a user has specified unequal header sizes between panels, the function will return a warning.**

panel.header.color

panel.width - This is the relative panel width compared to the other panels

xaxis.color – The color of the x axis line

xaxis.labels – This is a list or vector of text to be written at each tick mark**. Note: if these are being explicitly specified then xaxis.ticks must be explicitly specified as well.**

e.g. xaxis.ticks=c("500","1000","1500","2000")

xaxis.line.display – A boolean variable stating whether the line of the x axis should appear on the graph. (T/F or 0/1 should both work) *(Defaults to "FALSE" on maps, labels and ranks panel types)*

xaxis.text.display – A boolean variable… Note: this is the text associated with each tick, not the axis title! (Defaults *to "FALSE" on maps, labels and ranks panel types)*

xaxis.ticks – This is a list or vector of points at which ticks should be drawn on the x axis.

e.g. xaxis.ticks=c(500,1000,1500,2000)

xaxis.ticks.display – A boolean variable stating whether the axis ticks should appear on the x axis. (T/F or 0/1 should both work) *(Defaults to "FALSE" on all graphs)*

xaxis.title – Specifies what the x axis should be labeled. Note: this defaults to **no** axis label

yaxis.labels – This is a list or vector of text to be written at each tick mark. Note: if these are being explicitly specified then **yaxis.ticks** must be explicitly specified as well.

e.g. xaxis.ticks=list("A","B","C","D")

yaxis.line.display – A boolean variable stating whether the axis ticks should appear on the y axis. (T/F or 0/1 should both work)

yaxis.text.display – A boolean variable… Note: this is the text associated with each tick, not the axis title!

yaxis.ticks – This is a list or vector of points at which ticks should be drawn on the y axis.

e.g. yaxis.ticks=list(-1,-2,-3,-4,-5)

yaxis.ticks.display – A boolean variable stating whether the axis ticks should appear on the y axis. (T/F or 0/1 should both work) *(Defaults to "FALSE" on all graph types)*

yaxis.title – Specifies what the y axis should be labeled. (Defaults to **no** axis label on all graph types)

Panel specific attributes

labels:

- **align –** Horizontal alignment. I.E. ‘center’, ‘right’, ‘left’
- **text.size –** relative to default size

ranks:

- **align –** Horizontal alignment. I.E. ‘center’, ‘right’, ‘left’
- **text.size –** relative to default size

dot:

- **add.line –** add a line at some specified x coordinate
  - add.line.col – specify color
  - add.line.typ – specify type**
- **median.line –** add a line at the calculated median
  - median.line.col – specify line color
  - median.line.typ – specify type**
- **point.border –** By default a black border will be placed around dots. To correct this, set this option to FALSE
- **point.size –** Size **relative to default**.
- **point.type –** The pch specification for points contained in a graph

dot_cl:

- **add.line –** add a line at some specified x coordinate
  - add.line.col – specify color
  - add.line.typ – specify type**
- **line.width –** Thickness of confidence bands relative to default
- **median.line –** add a line at the calculated median
  - median.line.col – specify line color
  - median.line.typ – specify type**
- **point.border –** By default a black border will be placed around dots. To correct this, set this option to FALSE
- **point.size –** Size **relative to default**.
- **point.type** – The pch specification for points contained in a graph

bar:

- **add.line –** add a line at some specified x coordinate
  - add.line.col – specify color
  - add.line.typ – specify type**
- **graph.bar.size –** relative to default size
- **median.line –** add a line at the calculated median
  - median.line.col – specify line color
  - median.line.typ – specify type**

bar_cl:

- **add.line –** add a line at some specified x coordinate
  - add.line.col – specify color
  - add.line.typ – specify type**
- **graph.bar.size –** relative to default size **median.line –** add a line at the calculated median
  - median.line.col – specify line color
  - median.line.typ – specify type**

box_summary:

- **add.line –** add a line at some specified x coordinate
  - add.line.col – specify color
  - add.line.typ – specify type**
- **graph.bar.size –** relative to default size
- **median.line –** add a line at the calculated median
  - median.line.col – specify line color
  - median.line.typ – specify type

** Here is a helpful site for a list of line types--
http://wiki.stdout.org/rcookbook/Graphs/Shapes%20and%20line%20types/

A user defined panel should be accompanied by an attribute list which a user can specify other attributes to be altered with by the function call. See the user created panel functions section for details on how this is done.

5 Creating a new panel type

Note: A general understanding of ggplot2 is needed and assumed throughout this section

Now let's say we would like to illustrate the change in lung cancer rates using arrows on a graph. We can build our own graph type by creating our own graphing function; we'll call it 'arrow.plot.build'. The lmplot function sends all graphing functions the same arguments (in this order): the panel ggplot2 object being worked on; the number of the panel; the stats data table; and the attributes list (this is a little involved so we won't get into it until a little later). (Note: the panel number tells you which sublist in the attribute list to work with). To start, let's get our data and store it in a new object:

```
myStats <- lungMort
head(myStats)
```

	StateAb	Rate_95	Count_95	Lower_95	Upper_95	Pop_95	StdErr_95	Rate_00	Count_00	Lower_00	Upper_00	Pop_00	StdErr_00	State
AK	AK	50	298	44.2	56.3	1089123	3.1	46.8	350	41.8	52.2	1122525	2.6	Alaska
AL	AL	40.2	4095	39	41.4	8124753	0.6	43.4	4630	42.2	44.7	8245919	0.6	Alabama
AR	AR	43.8	3079	42.3	45.4	5479988	0.8	48.3	3568	46.7	49.9	5661547	0.8	Arkansas
AZ	AZ	38.4	4794	37.3	39.5	10557561	0.6	38.5	5482	37.4	39.5	12066024	0.5	Arizona
CA	CA	41.5	26931	41	42	64354973	0.3	39.2	27406	38.7	39.6	68478617	0.2	California
CO	CO	31.5	2723	30.3	32.7	9245273	0.6	33.9	3265	32.7	35.1	10159130	0.6	Colorado

For the time being, we'll also remove Washington D.C. so that we have nice even grouping numbers and can momentarily avoid the median row topic.

```
myStats <- subset(myStats, !StateAb=='DC')
```

The data table that will actually be passed into our graphing function once we implement it into the function is not exactly like our stats table. Before constructing the panels, the lmplot function adds the extra columns "rank", "median", "color", "pGrp" and "pGrpOrd" that specify, respectively, the overall order to plot the information, whether the row should be seperated as a median, the color from the color list to use, the perceptual group each table entry belongs to and the order in each perceptual group of each entry. These columns are added using a built-in function called "create_DF_rank" --the syntax for this function is: create_DF_rank (data, ord.by, grouping)--. We need these columns to know the nature of what we are working with in order to build our new graph type.

For now, we can assume groups of 5 will look good and we will want our table ordered by the rate from 2000. To create a new table with these columns file we run:

```
myNewStats <- create_DF_rank(myStats, ord.by="Rate_00", grouping=5)
```

	StateAb	Rate_95	Count_95	Lower_95	Upper_95	Pop_95	StdErr_95	Rate_00	Count_00	Lower_00	Upper_00	Pop_00	StdErr_00	State	rank	median	pGrp	pGrpOrd	color
UT	UT	17.6	685	16.3	18.9	5036638	0.7	16.9	738	15.7	18.2	5488475	0.6	Utah	1	FALSE	1	1	1
ND	ND	30.6	574	28.1	33.3	1527853	1.3	31.4	608	28.9	34.1	1480915	1.3	North Dak	2	FALSE	1	2	2
NM	NM	31.8	1293	30.1	33.6	3899455	0.9	31.5	1420	29.9	33.2	4038163	0.8	New Mexi	3	FALSE	1	3	3
SD	SD	30.5	659	28.2	33.1	1710003	1.2	32.9	736	30.5	35.5	1716683	1.2	South Dak	4	FALSE	1	4	4
CO	CO	31.5	2723	30.3	32.7	9245273	0.6	33.9	3265	32.7	35.1	10159130	0.6	Colorado	5	FALSE	1	5	5
ID	ID	33	981	31	35.2	2976963	1.1	35	1158	33	37.1	3230513	1	Idaho	6	FALSE	2	1	1

Now, to build our new graphing function, we have 4 basic steps to go through:

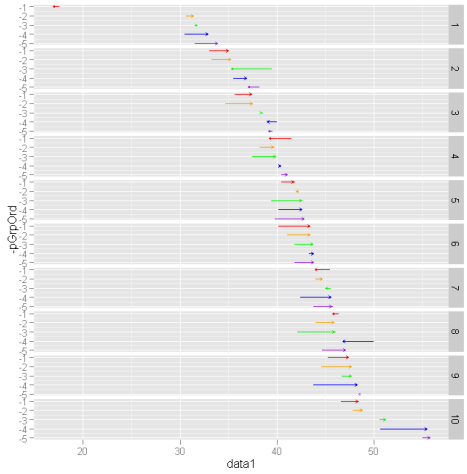
1. create the general graph's structure
2. generalize the inputs
3. integrate it with the lmplot function
4. enable user customization if desired

Step 1: First we use ggplot2 to create the general structure of the graphs as we would like to see them. We can use geom_segment function in ggplot2 to make arrows. On our graph we would like an arrow starting at the 1995 rate extending to the 2000 rate so these columns will obviously be used for our "x" and "xend" parameters. The y coordinates can be inferred from the 'pGrpOrd' column which has been created for just this purpose. Setting both

the “y” and “yend” parameters equal to “-pGrpOrd” should result in a flat arrow for each state, descending down our graph in an order which will match our label column as well as any other graphs being presented.

First, we can use the “color” column (which is calculated in create.DF.rank based on the pGrpOrd column) to vary the color of arrows within each perceptual group. Second, for various portions of the lmplot function code, we must use facet_grid instead of facet_wrap.

```
### ggplot2 code:
ggplot(myNewStats) +
geom_segment(aes(x=Rate_95, y=-pGrpOrd,
                 xend=Rate_00, yend=-pGrpOrd, colour=factor(color)),
                 arrow=arrow(length=unit(0.1,"cm"))) +
facet_grid(pGrp~., scales="free_y") +
scale_colour_manual(values=c('red','orange','green','blue','purple'),
                 guide=”none”)
```



Step 2: This graph looks like it is in the basic form we need. Good initial start but we need to change our x coordinate columns and color palette from being hard coded to being user specified. As was noted earlier, the lmplot function provides the panel object, the panel number, the stats data table and the attribute list. It is this attributes list through which the color and data specifications are going to be provided to our function. Without delving too far into the details of this list just yet, we can take for granted that the user specified color palette will be stored in the “colors” slot in the plot section of the object and the names of our data columns will be stored in the “panel.data” slot of one of the panel sections; the panel number tells us which panel section to look in.

In writing our function we can refer to the panel object, the panel number, stats table and the attribute list however we like. We’ve already been referring to the data table as myNewStats so, along those same lines, let’s call the other items myPanel, myNumber, and myAtts respectively. In the next section we will start referring to the myAtts and myNumber so it is helpful to set up a fake list and fake number to work with while we build our function that we can work with to test our code as we go along. The **sample.att** function will provide this list for us and we will simply set myNumber equal to 1.

```
myAtts <- sample_att()
myNumber <- 1
```

This is just a dummy attribute list for now so we need to overwrite its entries with our specifications from above so that we can continue to test and have everything work as expected:

```
myAtts$colors  <- c('red','orange','green','blue','purple')
myAtts[[myNumber]]$panel.data <- c('Rate_95','Rate_00')
```

We will pull out our color list and panel column list into variables called myColors and myColumns. This means myColumns will be a vector with the myColumns[1] referring to the start points and myColumns[2] referring to the end points of our arrows. The code to pull these items out of the attributes list will look like this:

```
myColors <- myAtts$colors                    # pulls color out of the plot level section of the “myAtts” attributes list
myColumns <- myAtts[[myNumber]]$panel.data   # looks in the panel level section numbered “myNumber” of the “myAtts” attributes list
```

We need to work around ggplot a bit in order for it to understand where to find our data. Telling ggplot “x=myColumns[1], xend=myColumns[2]” will just confuse it. Instead, we have to hard code the column names to look for (i.e. “x=data1, xend=data2”) and add those columns to myNewStats. This is illustrated with the following code:

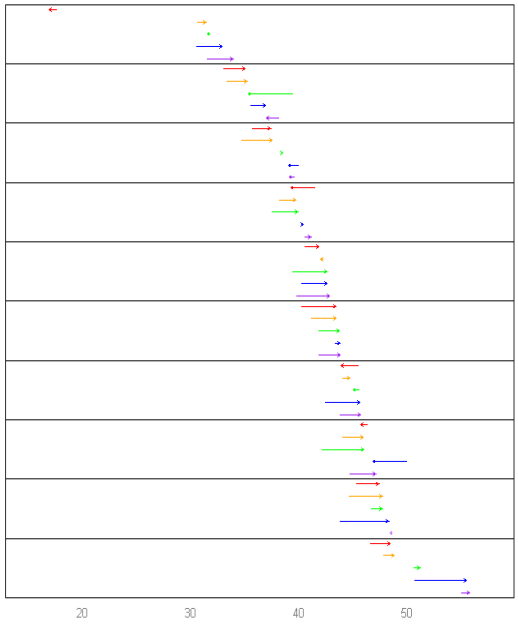
```
myColors <- myAtts$colors                    # pulls the “colors” attribute out of the plot level section of the “myAtts” attributes list
myColumns <- myAtts[[myNumber]]$panel.data   # looks in the panel level section numbered “myNumber” of the “myAtts” attributes list

myNewStats$data1 <- myNewStats[, myColumns[1] ]
```

```
myNewStats$data2 <- myNewStats[, myColumns[2] ]

myPanel  <- ggplot(myNewStats) +
            geom_segment(aes(x=data1, y=-pGrpOrd,
                             xend= data2, yend=-pGrpOrd, colour=factor(color)),
                             arrow=arrow(length=unit(0.1,"cm"))) +
            facet_grid(pGrp~.) +
            scale_colour_manual(values=myColors,
                             guide="none")

myPanel
```

Note that we have also gone ahead and stored this graph in the myPanel object as we will eventually be returning this back to the lmplot function anyways. This means the last line of code (simply "myPanel") has the dual purpose of telling R to show us our graph but will also return the panel object back to the lmplot function when we're finally ready to compile this into function form.

Step 3: We are getting close to being able to implement our graph but we still have to clean it up a bit in order for it to seamlessly match the rest of our linked micromap. There are several built in functions that work to this end. We have stored our plot in a variable called myPanel that we can send out to the **assimilatePlot** function to do all the needed work for us.

```
assimilatePlot(myPanel, myNumber, myAtts)
```



Our graph looks like it will probably fit right in with the rest of the linked micromap plot. Now, we just need to put our code in proper function form:

```
arrow_plot_build <- function(myPanel, myNumber, myNewStats, myAtts){
            myColors <- myAtts$colors
            myColumns <- myAtts[[myNumber]]$panel.data

            myNewStats$data1 <- myNewStats[, myColumns[1] ]
            myNewStats$data2 <- myNewStats[, myColumns[2] ]

            myPanel  <- ggplot(myNewStats) +
                        geom_segment(aes(x=data1, y=-pGrpOrd,
                                         xend= data2, yend=-pGrpOrd,
                                         colour=factor(color)),
                                    arrow=arrow(length=unit(0.1,"cm")))  +
             facet_grid(pGrp~.) +
             scale_colour_manual(values=myColors,  guide="none")

            myPanel <- assimilatePlot(myPanel, myNumber, myAtts)

            myPanel
}
```

Dealing with a median row:
We have one final piece of difficulty to overcome and that is dealing with inserting a median row. There is a built in function that should handle this fairly we called **alterForMedian**. If, after we've added our new columns, we simply hand that function our stats table and the attributes list, it should give us back one that has been altered as needed. We also need to slightly alter the "facet_grid" line to allow for the median to be a different size.

```
arrow_plot_build <- function(myPanel, myNumber, myNewStats, myAtts){
            myColors <- myAtts$colors
            myColumns <- myAtts[[myNumber]]$panel.data

            myNewStats$data1 <- myNewStats[, myColumns[1] ]
```

```
        myNewStats$data2 <- myNewStats[, myColumns[2] ]

        myNewStats <- alterForMedian(myNewStats, myAtts)

        myPanel  <- ggplot(myNewStats) +
                        geom_segment(aes(x=data1, y=-pGrpOrd,
                                         xend= data2, yend=-pGrpOrd,
                                         colour=factor(color)),
                                         arrow=arrow(length=unit(0.1,"cm")))  +
                facet_grid(pGrp~., space="free", scales="free_y") +
                scale_colour_manual(values=myColors,  guide="none")

        myPanel <- assimilatePlot(myPanel, myNumber, myAtts)

        myPanel
}
```

After we run this function, or saving it to a file and then sourcing that file, we'll be able to tell the lmplot function to build this graph simply entering a panel type of "arrow.plot".

Optional Step 4 – specializing user controlled attributes:
If we run the line of code:

```
print(myAtts)
```

We can see a full list of attributes available for alteration/specification by a user. All of these attributes (e.g. axis labels, background color, grid lines, etc.) are applied to the graph through the **assimilatePlot** function so if we like how our graph looks and don't feel the need to give the user any more control on its features we can stop here. However, there might be some changes that users would like to make such as the width of the arrows and lengths of the arrow heads. In order to allow these changes by users we need to: a) create extra slots in our panel level of the attributes list and b) alter our code to recognize these options.

Creating the extra slots in the attribute list is actually not a terribly difficult process. This is done for every graph that has already been built in to the **micromaps** library. What these built in graphs have that ours is still lacking is a personalized "attribute function". When the lmplot function sees a panel type of "arrow.plot", it's already looking for an attribute function called arrow.plot.att to supply the panel level list for our all encompassing attribute list that is being passed around but we haven't created this yet; so it settles on a built in function called standard.att. We'll use standard.att to build our new arrow.plot.att function.

In the code below we first start with standard.att to get our very useful base list and then well just append on the new attributes we'd like to control. We'll call these new attributes "line.width" and "tip.length".

```
myPanelAtts <- standard_att()
myPanelAtts <- append(myPanelAtts,
                      list(line.width=1, tip.length=1))
myPanelAtts
```

Note that the "=1" is setting our defaults for these 2 entries at "1". We can control what "1" actually implies later. Now let's put this into function form. Note that the lmplot function "sends" nothing to this function. It only wants a list of attributes back. Which makes our function simply look like:

```
arrow_plot_att <- function(){
        myPanelAtts <- standard_att()
        myPanelAtts <- append(myPanelAtts,
                              list(line.width=1, tip.length=1))
        myPanelAtts
}
```

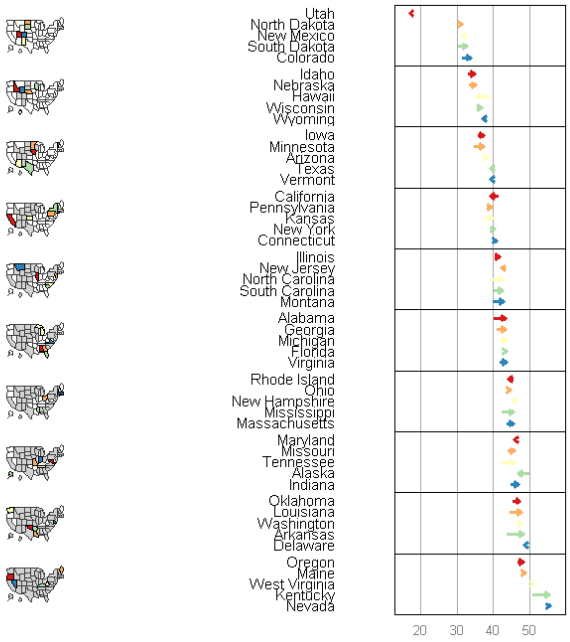
Simple enough. Now let's revisit our arrow.plot function and insert lines to pull these attribute specifications out of the attribute list and implement them in our graphing code:

```
arrow_plot_build <- function(myPanel, myNumber, myNewStats, myAtts){
        myColors <- myAtts$colors
        myColumns <- myAtts[[myNumber]]$panel.data
        myLineWidth <- myAtts[[myNumber]]$line.width      # Again, note that these are stored in the panel level section of the
        myTipLength <- myAtts[[myNumber]]$tip.length      #    attributes object


        myNewStats$data1 <- myNewStats[, myColumns[1] ]
        myNewStats$data2 <- myNewStats[, myColumns[2] ]

        myNewStats <- alterForMedian(myNewStats, myAtts)

        myPanel  <- ggplot(myNewStats) +
                geom_segment(aes(x=data1, y=-pGrpOrd,
                                 xend= data2, yend=-pGrpOrd,
                                 colour=factor(color)),
                                 arrow=arrow(length=unit(0.1*myTipLength,"cm")),    # Here, you'll notice the "1" default above
                                                                                    # is specifying length in tenths of a cm
                                 size=myLineWidth)  +
                facet_grid(pGrp~., space="free", scales="free_y") +
                scale_colour_manual(values=myColors, guide="none")

        myPanel <- assimilatePlot(myPanel, myNumber, myAtts)

        myPanel
}
```

Pretty painless, very useful.

Step Last: Now let's try to implement this new panel in a simple linked micromap (using the statePolys map data from the initial example) and adjust the line width and tip length while we're at it.

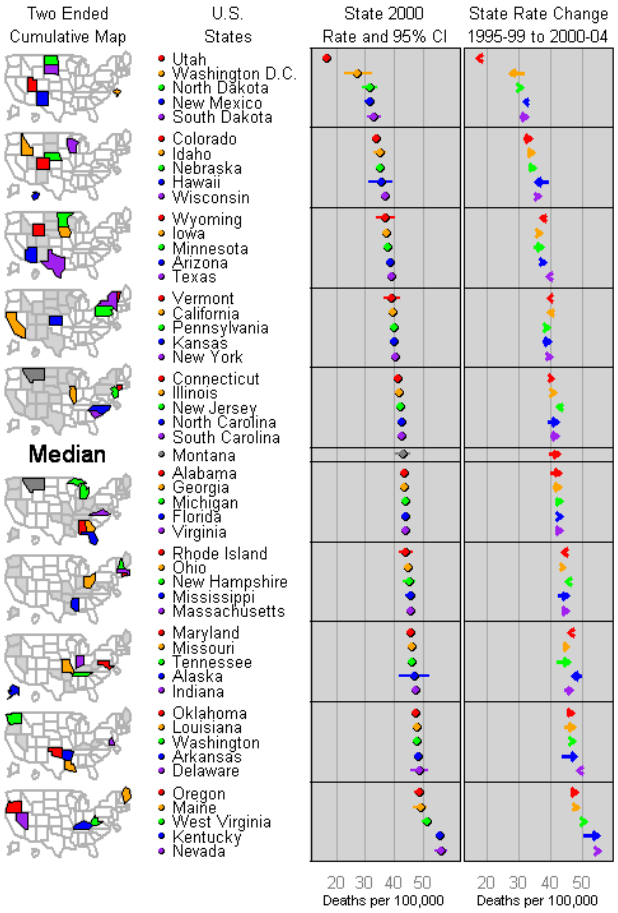
```
lmplot(stat.data=myStats,
        map.data=statePolys,
        panel.types=c('map','labels', 'arrow_plot'),
        panel.data=list(NA,'State', list('Rate_95','Rate_00')),
        ord.by='Rate_00',
        grouping=5,
        map.link=c('StateAb','ID'),

        panel.att=list(list(3, line.width=1.25, tip.length=1.5)))
```



It looks like our new graph has been implemented nicely. We can obviously still clean this up a bit and might as well add in some extra plots as well. Also, we should bring Washington DC back into the picture (ie use our original myStats table) and make sure our median row is displaying correctly with the new graph. Using our bullet points we created above and tweeking the panel attributes section quite a bit, we are ready to present the following:

```
data(lungMort)
myStats <- lungMort
myStats$points <- 0
lmplot(stat.data=myStats,
        map.data=statePolys,
        panel.types=c('map', 'dot',  'labels', 'dot_cl', 'arrow_plot'),
        panel.data=list(NA,
                        'points',
                        'State',
                        list('Rate_00','Lower_00','Upper_00'),
                        list('Rate_95','Rate_00')),
        ord.by='Rate_00', grouping=5,
        median.row=T,  two.ended=T,
        map.link=c('StateAb','ID'),

        plot.height=10,
        colors=c('red','orange','green','blue','purple'),
        map.color2='lightgray',

        panel.att=list(list(1, header='Light Gray Means\n Highlighted Above',
                               panel.width=1,
                               inactive.border.color=gray(.7),
                               inactive.border.size=2),

                       list(2, panel.width=.15,
                               graph.border.color='white',
                               xaxis.text.display=F,
                               xaxis.line.display=F,
                               graph.grid.major=F,
                               point.type=20),

                       list(3, header='U.S.   \nStates   ',
                               panel.width=.8,
                               align='left', text.size=.9),

                       list(4, header='State 2000\n Rate and 95% CI',
                               graph.bgcolor='lightgray',
                               xaxis.ticks=c(20,30,40,50),
                               xaxis.labels=c(20,30,40,50),
                               xaxis.title='Deaths per 100,000'),

                       list(5, header='State Rate Change\n 1995-99 to 2000-04',
                               line.width=1.25, tip.length=1.5,
                               graph.bgcolor='lightgray',
                               xaxis.ticks=c(20,30,40,50),
                               xaxis.labels=c(20,30,40,50),
                               xaxis.title='Deaths per 100,000')  ) )
```

Note the "two ended cumulative map" type



6 Group-categorized micromaps (lmgroupedplot function)

lmgroupedplot(stat.data, map.data, panel.types, panel.data, cat, map.link, …)

The **lmgroupedplot** function is very much similar to the **lmplot** function above. Previously we had multiple polygons per perceptual group with one line of data per polygon. Here, we have a data table with multiple categories of data per polygon and the perceptual groups are simply the region, or polygon, presently being referred to. Taking a look at an example data table should make this a bit clearer:

```
data(vegCov)
head(vegCov)
```

Type	Subpopul	Indicator	Category	NResp	Estimate.I	StdError.P	LCB95Pct.I	UCB95Pct.	Estimate.(	StdError.U	LCB95Pct.I	UCB95Pct.
National	National	CondClass	1:LEAST DI	698	47.61908	1.511643	44.65632	50.58185	516807	18907.16	479749.6	553864.3
National	National	CondClass	2:INTERMI	394	28.3188	1.533217	25.31375	31.32385	307342.2	17184.38	273661.4	341023
National	National	CondClass	3:MOST DI	291	19.33501	1.229759	16.92473	21.74529	209841.7	13516.94	183349	236334.4
ecowsa3	EHIGH	CondClass	1:LEAST DI	129	42.16749	2.597053	37.07736	47.25762	187505.4	13383.58	161274	213736.7
ecowsa3	EHIGH	CondClass	2:INTERMI	92	30.70827	2.685939	25.44393	35.97261	136549.9	12402.81	112240.8	160858.9
ecowsa3	EHIGH	CondClass	3:MOST DI	48	17.58212	1.990464	13.68088	21.48336	78182.07	8887.312	60763.26	95600.88
ecowsa3	PLNLOW	CondClass	1:LEAST DI	155	47.53334	2.753267	42.13704	52.92965	186008.4	11785.39	162909.4	209107.3
ecowsa3	PLNLOW	CondClass	2:INTERMI	111	24.45491	2.544108	19.46855	29.44127	95697.4	10313.98	75482.37	115912.4
ecowsa3	PLNLOW	CondClass	3:MOST DI	145	25.95537	2.341336	21.36644	30.5443	101569	9522.937	82904.42	120233.6
ecowsa3	WMTNS	CondClass	1:LEAST DI	413	57.06422	2.060667	53.02539	61.10305	140481.6	5750.971	129209.9	151753.3
ecowsa3	WMTNS	CondClass	2:INTERMI	191	30.50387	2.025976	26.53303	34.47471	75094.92	5483.507	64347.45	85842.4
ecowsa3	WMTNS	CondClass	3:MOST DI	98	12.22291	1.352445	9.57217	14.87366	30090.57	3381.678	23462.6	36718.54

Here we would like to list the 3 categories by "subpopulation" and look at the metrics within each. The included WSA3 shapefile has maps to link with this table. Note that this shapefile has already been thinned down and should be quite manageable in size. As we look at this shapefile's data table to determine a good ID column we see that a column has already been named as such. However, we also notice that there is no "National" map (since it is just a combination of the other 3).

```
data(WSA3)
print(WSA3@data)
```

	WSA_3	WSA_3_NM	area_mdn	ID	area_mdn	area_mdn
1	EHIGH	Eastern Highlands	1.2E+12	EHIGH	1.2E+12	1.2E+12
2	PLNLOW	Plains and Lowlands	3.95E+12	PLNLOW	3.95E+12	3.95E+12
3	WMTNS	West	2.64E+12	WMTNS	2.64E+12	2.64E+12

We can remedy this problem after we create an initial map table using the **create_map_table** function:

```
wsa.polys <- create_map_table(WSA3)
head(wsa.polys)
```

	ID	region	poly	coordsx	coordsy	hole	plug
1	EHIGH	1	1	659712.9	-3636.47	0	0
2	EHIGH	1	1	659500.8	-11891.7	0	0
3	EHIGH	1	1	644443.4	5378.044	0	0
4	EHIGH	1	1	659712.9	-3636.47	0	0
5	EHIGH	1	2	672579.2	49281.8	0	1
6	EHIGH	1	2	692236.2	27743.74	0	1

Basically, we want almost all of these polys as our National map. "Almost all" since, all we want is an outline, we don't need the plugs or holes. We do need to be careful not to have our polygons numbered the same but this is easily remedied as well by just combining them with region:

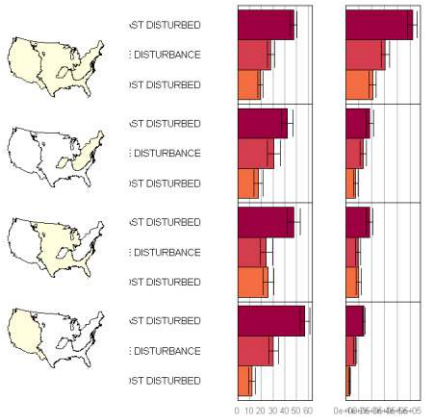
```
national.polys <-subset(wsa.polys, hole==0 & plug==0)
national.polys <- transform(national.polys, ID='National', region=4, poly=region*1000 + poly)
head(national.polys)
```

	ID	region	poly	coordsx	coordsy	hole	plug
1	National	4	1001	659712.9	-3636.47	0	0
2	National	4	1001	659500.8	-11891.7	0	0
3	National	4	1001	644443.4	5378.044	0	0
4	National	4	1001	659712.9	-3636.47	0	0
111	National	4	1004	1822280	563868.8	0	0
112	National	4	1004	1820338	549799	0	0

```
wsa.polys <- rbind(wsa.polys, national.polys)
```

Now that the data is set up we can move on to running our function. The structure of the **lmgroupedplot** function is similar to that of **lmplot**. The bare bones code needed to generate an figure such as this example would look something like this:

```
lmgroupedplot(stat.data=vegCov,
              map.data=wsa.polys,
              panel.types=c('map', 'labels', 'bar_cl', 'bar_cl'),
              panel.data=list(NA,'Category',

                              list('Estimate.P','LCB95Pct.P','UCB95Pct.P'),

                              list('Estimate.U','LCB95Pct.U','UCB95Pct.U')),
              grp.by='Subpopulation',
              cat='Category',
              map.link=c('Subpopulation', 'ID'))
```



We then can clean it up to a final state with code such as this:

```
lmgroupedplot(stat.data= vegCov,
              map.data= wsa.polys,
              panel.types=c('map', 'labels', 'bar_cl', 'bar_cl'),
              panel.data=list(NA,'Category',
                              list('Estimate.P','LCB95Pct.P','UCB95Pct.P'),

list('Estimate.U','LCB95Pct.U','UCB95Pct.U')),
              grp.by='Subpopulation',
              cat='Category',
              colors=c('red3','green3','lightblue'),
              map.link=c('Subpopulation', 'ID'),
              map.color='orange3',
              plot.grp.spacing=2,
              plot.width=7,
              plot.height=4,

              panel.att=list(list(1, header='Region', header.size=1.5,
                                     panel.width=.75),
                             list(2, header='Category',
                                     header.size=1.5,
                                     panel.width=1.7),
                             list(3, header='Percent', header.size=1.5,
                                     xaxis.ticks.display=T,
                                     xaxis.text.display=T,
                                     graph.bgcolor='lightgray',
                                     xaxis.title='percent',
                                     xaxis.ticks=c(0,20,40,60),
                                     xaxis.labels=c(0,20,40,60)),
                             list(4, header='Unit', header.size=1.5,
                                     xaxis.ticks.display=T,
                                     xaxis.text.display=T,
                                     graph.bgcolor='lightgray',
                                     xaxis.title='thousands',
                                     xaxis.ticks=c(0,200000,350000,550000),
                                     xaxis.labels=c(0,200,350,550))))
```