Getting Started with LiveMap for XLISP-STAT

Dr. Chris Brunsdon

May 28, 1998

1 Introduction

This short document is intended as a primer for **LiveMap**, an add-on package for XLISP-STAT [6]. XLISP-STAT is an extended version of the XLISP package; the extensions incorporate a number of for statistical functions, and interactive graphics for exploratory data analysis. As the name suggests, the package is based on the LISP programming language, which has existed in some form since 1958. Readers wishing to know something about why LISP makes a good basis for a statistical computing package should consult the introductory chapter of [6]. I will now attempt to outline why a LISP-based statistical package is a good basis for LiveMap.

1.1 What is LiveMap?

LiveMap was initially conceived as a tool for exploratory spatial data analysis (ESDA), but it could be used as a general tool for spatial statisticians. It is a file of LISP code, which, when loaded adds several map drawing functions to the core of XLisp-Stat. The basic map drawing functions are designed with the 'linked plot' philosophy of XLisp-Stat in mind. If a map is linked to an XLisp-Stat graph window such as a scatterplot or histogram, then clicking on an item in the graph window highlights the corresponding geographical object in the map. In most cases, the converse is also true. This is the core of the ESDA approach to LiveMap. In addition to this, there are several spatial data handling functions provided. For example, given a set of geographical points (such as, say, locations of road accidents) and a set of geographical zone boundaries (such as census wards) it is possible to determine which ward each point lies in, or how many points lie in each ward. Functions of this sort can help to pre-process data before the ESDA stage, but they are also helpful in manipulating data for more traditional analytical approaches.

It is perhaps the interaction between exploratory and more formal approaches that sets XLisp-Stat apart from other statistics packages. Interactive graphics and quite powerful statistical 'number-crunching' methods are equally accessible from the command line. This encourages the user to adopt an exploratory and graphical approach even when carrying out quite intensive numerical computation. For example, if a model is being calibrated iteratively, one can explore the convergence of the iterative process graphically. Another thing to note is that XLisp-Stat is a statistical programming environment rather than a toolbox of commonly used techniques - rather than having ANOVAs and regression models hanging off the menu-bar [1] it provides an environment to develop new techniques and code one-off analyses and simulations.

These characteristics of XLisp-Stat form the justification of the existance of LiveMap. Whereas other packages, such as MANET [4] provide good exploratory spatial data analysis functions, the intention here is to combine such functions with a statistical programming environment where more formal spatial statistical procedures may be developed. In this way, it is possible to explore the output of spatial statistical models (such as residuals from regression models, for example) through interactive maps. Thus, LiveMap is essentially a set of LISP functions which are added to XLisp-Stat to provide the kind of facilities listed above. They are also designed to integrate fully with the rest of XLisp-Stat, so that they may be percieved as a 'natural' extension to the system. For example, it should be just as simple to produce a map of fitted y-values in a regression model as it is to produce a histogram of the same values.

1.2 Two Ways to Use LiveMap

The 'integration' principle for LiveMap works on two levels. Firstly, one can use the functions in LiveMap alongside all of the other XLisp-Stat functions. In this way, using XLisp-Stat is not much different from before, except that it now has facilities to work with geographical data. As suggested earlier, this should all be fairly seamless. However, more advanced XLisp-Stat users actually use the statistical and graphical features as 'building blocks' to build new features into the system. This approach is also possible with the features in LiveMap.

The former approach is relatively simple, and only requires the user to learn a number of new commands to access the geographical functionality. As one might expect, the latter approach is somewhat harder, and requires

[1] Although Forrest Young's ViSta package does provide — and indeed improve upon — this kind of environment extremely effectively

some familiarity with object-orented programming, and particularly with the object system provided in XLISP. This document is mostly concerned with the first option - the next section provides an introductory tutorial in LiveMap. Following this, some more advanced issues will also be discussed, giving a flavour of how LiveMap might be used as a basis for building more advanced application. In both cases, a strong emphasis will be placed on learning by example.

2 Livemap Basics

This section provides a tutorial in LiveMap. Some familiarity with XLisp-Stat is an advantage here — I suggest working through the tutorial chapter in [6] if you are new to the package.

Figure 1: XLisp-Stat with LiveMap Startup Window

2.1 Getting Started

Assuming you you have obtained the files that make up the LiveMap distribution, and placed them in a directory called 'LiveMap', the first thing you will need to do is to start XLisp-Stat running. XLisp-Stat is a multi-platform package, and the way it is started up will depend on the platform. Once XLisp-Stat has been started, use the 'File/Load' option on the menu (or press *command-L*) to load the file 'LiveMap.lsp'. When you have done

this, you should see something like figure 1 in the XLisp-Stat window [2].

You are now ready to use LiveMap — all you need is some data to map. One possibility is to use the training data set supplied. You can load this either from the 'File/Load' menu option, or by typing

```
(load "crime.lsp")
```

in the window. This will have loaded a number of variables, which will be used throughout the tutorial. In LiveMap, geographical information is stored in variables, just like any other data in LISP. For example, one variable you have just loaded is called `wards`. Typing the word `wards` in the window displays the variable - but this is not particularly meaningful. Since `wards` is a set of geographical zones a more useful thing to do is to map it. Type in

```
(map wards)
```

and a map of the area covered by the zones will appear in a new window - see figure 2. In itself this is not very exciting - it just shows the outline of

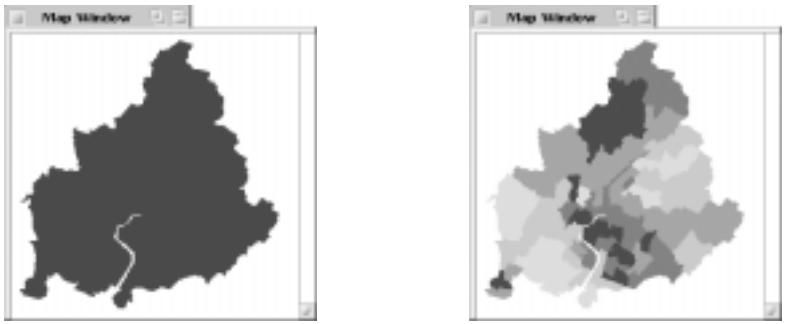

Figure 2: XLisp-Stat Maps; Produced by typing `(map wards)` (left) and `(map wards unemp)` (right).

the study area for this tutorial [3]. However, you also have a set of household burglary rates (per 10,000 households) in a variable called crimes. A map of this may be obtained by typing

```
(map wards crime)
```

[2] The style of the window you will see depends on you hardware and operating system — this is what you see on my Apple Mac

[3] This study area is fictitious, as is the rest of the data.

— this is also shown in figure 2. From this it may be seen that lower burglary rates occur near to the west of the river, except for one area in the south-eastern corner of the study area.

Now you have used the most basic - and perhaps most useful - LiveMap command, `map`. With one argument (a geographical variable) it simply shows the geographical form of that variable in a map window. If a second argument is provided, this is interpreted as a variable for a choropleth map. The geography for the choropleth map is provided by the first argument. At this stage it might be helpful to consider the form of the variables in more detail.

2.2 Types of LiveMap Geographical Variables

In LiveMap, there are three types of geographical variable, as listed below, and illustrated in figure 3:

- **Points:** These are the simplest kind of geographical entity. They correspond to the locations of pins stuck into a map - such as sometimes used by the Police to examine geographical patterns in crime incidence.
- **Lines:** These correspond to paths on a map. Examples of lines include include roads, and railway lines. A line consists of a list of points, joined together with line segments join-the-dots style.
- **Zones:** A zone is a filled area on a map. For example, Cornwall, or the City of Durham conservation area could both be represented by zones. Unlike lines or points, zones have a well-defined inside and outside.

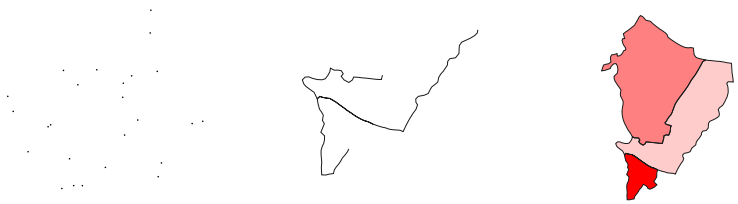

Figure 3: From Left to Right: Points, Lines and Zones.

The variable `wards` used in the last section is a *zones* variable. The information it contains is a list of zones, each one representing a single census ward. Similarly, variables could contain lists of points or lines. Each list of geographical entities is an ordered list. A standard LISP variable of the

same length as this list containing numerical values can be used to create choropleth maps. Essentially, each number in the variable is paired with a corresponding zone, point or line on the map, and used as a basis for deciding the way it is drawn. In the example earlier, the values in the list `crime` were used to determine the shading of the zones in `wards`. Type `crime` in the XLisp-Stat window and you will see that this is just an ordinary LISP variable.

Another example can be found in the geographical variable `house-sales`. This is a list of points, each representing a fictitious house that was sold in the study area in the same fictitious time period that the crime rates were measured. Another variable, `price` contains an ordered list of the sale prices for each of the houses represented in `house-sales`. Typing

```
(map house-sales price 750)
```

produces a proportional circles map of sale prices - see figure 4. Note that here you will require a third argument to `plot` - this gives the maximum circle size in map units. Note that the same function `map` is called, regardless

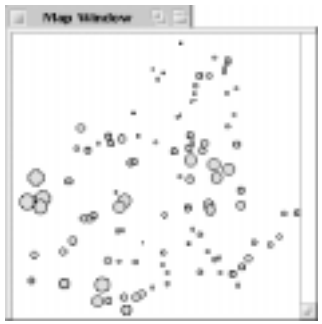

Figure 4: Proportional Circles Plot for House Prices

of whether you wish to produce maps of points, lines or zones.

You have now seen some point-based data, and some zone-based data. Unfortunately, the `crime.lsp` file contains no line-based data. However, it is possible to create a line-based geographical variable. To do this, the `outline` function is used. This takes a zone-based geographical list (such as `wards`) as input, and produces a line-based geographical list consisting of the outlines of the zones. In this case, the line-based components of the list are all closed polygons, but not all line-based data has this property. Because

the geographical information is stored in ordinary LISP variables, functions such as `outline` are just ordinary LISP functions. To create a new variable, `wardbounds`, containing the outlines of the zones in `wards`, type

```
(def wardbounds (outline wards))
```

this creates the new variable. To see it in map form, now enter

```
(map wardbounds)
```

and a window will appear, showing the outlines of the ward boundaries. Note the difference between this window and the one obtained by typing `(map wards)` previously.

2.3 Linked Plots

In this section, you will see how maps of the kind you have created earlier may be used in linked displays with other statistical graphs. If you have worked with XLisp-Stat already, you will be familiar with the principle of linked graphs. In XLisp-Stat, the graphs you create are not automatically linked — you must first select "Plot/Link View" from the menu. This is the same for maps.

To demonstrate the idea here, firstly create a map of wards using the command

```
(map wards)
```

Alongside the variable `crime` another variable is supplied in the `crime.lsp` file, called `unemp`. This gives the unemployment rates [4] (as percentages of economically active population) for each of the wards. Create a scatterplot of unemployment rate against crime rate by typing:

```
(plot-points unemp crime)
```

Now link both of these plots by carrying out the following steps: firstly, select the scatterplot window; then select 'Plot/Link View" from the menu; then select the map window; finally select "Plot/Link View" from the menu. Once this is done, it should be possible to click on points in the scatterplot in the usual XLisp-Stat way, and then see the zones corresponding to these points highlighted in yellow. For example, see figure 5. Here a series of high unemployment, high crime rate points are selected, and the geographical locations of these points are highlighted on the map.

[4] needless to say, these are fictitious!

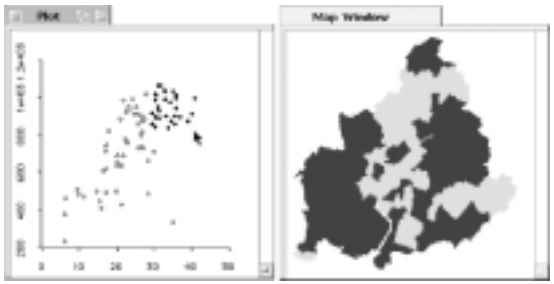

Figure 5: A Linked Zone Map and Scatterplot

As can be seen, there is a distinct geographical pattern here. Another exploratory task might be to find the geographical location of the outlying point seen to the lower right of the scatterplot.

If you attempt to select a zone in the map window at present, you will notice two things. Firstly, the cursor is a pointing finger, and secondly when you click on a point in the map window, the map co-oordinate is displayed, but the selected zones on the map are unaffected. Although this may be useful for locating map features, it does not help with linked plot exploration. To allow the selection of map features, you will need to alter the mouse mode. To do this, select "Plot/Mouse Mode" from the menu when the map window is selected. Now move the mouse back into the map window. You will now notice that the cursor is a crosshair, rather than a finger. Clicking on the map now alters the state of the object nearest to where you click, and also alters the corresponding selection on any linked plots. Thus, to see which point on the scatterplot any zone on the map corresponds to, click on that map zone, and the observe the change in the linked plot. Unfortunately, this 'one zone at a time' clicking is all that is available for map windows at present, but it is hoped to extend this functionality in the near future.

Note that it is also possible to create linked line and point maps, by following the same procedure as that given in the exercise above. For example data about traffic flows could be linked to a line-based geographical variable representing a road network, or data about items stolen, method of entry and so on could be linked to a point-based geographical variable giving locations of household burglaries.In each case, the objects in the statistical plots should have a one-to-one association with the geographical objects in the maps.

2.4 Variable Manipulation and Maps

Since the variables used to create choropleth maps, and proportional circles maps are ordinary LISP variables, they may be used in any normal XLisp-Stat calculation and the results may then be mapped. For example, you could type

```
(map wards (+ crimes (* 100 unemp)))
```

to obtain a map of the household burglary rate plus one hundred times the unemployment rate. This could be regarded as a map of a rather *ad hoc* quality-of-life indicator [5]. You could also combine mapping with the statistical analysis features of XLisp-Stat. Here, the residuals from regressing crime rates on unemployment rates are mapped:

```
(def regmod (regression-model unemp crime))
(map wards (send regmod :raw-residuals))
```

If you work through this, you may note that there appears to be some spatial pattern in the residuals, and that perhaps a spatial regression model (such as that proposed by Cliff and Ord [2]) or a geographically weighted regression [1] might be a more appropriate analysis technique for this data.

This demonstrates one of the main ways in which LiveMap is intended for use. Maps are used together with the statistical features of XLispStat to provide more understanding of the geographical patterns in the data, and to apply geographical diagnostics to the model fitting. You can combine mapping with any of the existing statistical techniques in XLisp-Stat. For example, a variant on the above residual mapping exercise might be to map *studentised* residuals:

```
(def regmod (regression-model unemp crime))
(map wards (send regmod :studentized-residuals))
```

However, as well as being able to map outputs from existing statistical features, LiveMap also provides some useful functions itself. These may be used in conjunction with maps, and existing XLisp-Stat functions. The first LiveMap function of this kind was introduced in the last section — `outline`. This took a geographical variable and returned another geographical variable. Another function of this kind is `centres`. This takes a zone-based variable and returns a point based variable with the same number of items. Each point-based item corresponds to a zonal centroid from the zone-based

[5] the factor of 100 is used as unemployment is a percentage, but crime is on a 'per-10,000' basis.

variable. For example, to re-draw the crime rate map as a proportional circles map one could obtain a ward centroid variable using `centres` and map this:

```
(def ward-centres (centres wards))
(map ward-centres crime)
```

Another form of function LiveMap provides extracts non-geographical variables from geographical ones. For example, suppose you have a point-based variable andd a zone based one. It might be helpful to find out which point is contained in which zone. To do this, the function `identity` is created. This takes two arguments, a point based variable and a zone-based one, and returns a list of numbers of the same length as the number of points. Each number in the list is the index number of the zone in the second variable that contains the point. For the tutorial example, enter

```
(def in-ward (identity house-sales wards))
(plot-points price in-ward))
```

the scatterplot here shows each of the sale prices for the houses, separated out on a ward-by-ward basis. This is a useful way of comparing the between-ward variance with the within-ward variance for house prices. However, it would be more useful to plot the house sale prices against one of the ward-based variables, `crime` or `unemp`. Since the variable `in-ward` gives the index of the containing ward for each point, the function `select` can be used to pick out the crime rates of the containing ward for each point:

```
(plot-points (select crime in-ward) price)
```

Doing this, a clear linkage between crime rates and house prices becomes apparent. Also, entering

```
(plot-points (select unemp in-ward) price)
```

shows a similar, if perhaps looser linkage between unemployment rates and house prices.

Although these are fairly crude analyses, they provide a useful elementary exploration of relationships between three variables - including one recorded in a different geographical format from the others. This can be done in only three lines of code. It is this kind of easy access to spatial exploration that is the design philosophy of LiveMap.

The complimentary functions to `identity` are perhaps `zonal-list`. This takes two geographical arguments, a point list and a zone list. It returns a list of lists of indices. Each list in the outer list contains the indices of the

points falling within the corresponding zone. Thus, the first inner list is a list of indices of points in the first zone, the second inner list is a list of indices of points in the second zone and so on ...

Try this function out by typing

```
(def zone-lists (zonal-list house-sales wards))
zone-lists
```

the second command prints out the `zone-lists` variable so you can see what it looks like. The `NIL` elements correspond to zones containing no points. Unfortunately, this is not the simplest of functions to use directly in calculations — more on this later in the tutorial. Easier functions to work with are `zonal-count`, `zonal-sum` and `zonal-mean`. `zonal-count` takes two arguments, a point-based variable and a zone-based one, and returns a list of the same length as the number of zones, whose elements are the number of points in each corresponding zone. Now type

```
(def counts (zonal-count house-sales wards))
counts
```

you can now see the number of sales occurring in each ward, as elements in the list `counts`. You can now produce a choropleth map of counts, on a ward-by ward basis.

```
(map wards counts)
```

Of course, you can also apply statistical techniques to the counts. For example, you may wish to test whether the counts in the wards follow a common Poisson distribution. This seems unlikely from the map, since larger wards tend to have more counts - rather than the count in each ward being independently and identically distributed. However, you can proceed with a formal test as follows: if μ, the mean count is larger than one, and n, the number of wards is reasonably large, then the *index of dispersion* I, given by

$$I = \sum_{i=1}^{n} (c_i - \bar{c})^2 / \{(n-1)\bar{c}\} \qquad (1)$$

where c_i is the number of counts in cell i, is approximately distributed as $\chi^2_{n-1}/(n-1)$ when the c_i are Poisson distributed. This can be used to test the Poisson count hypothesis. First compute I:

```
(def c-bar (mean counts))
(def i (/ (sum (** (- counts c-bar) 2))
       (- (length counts 1)) c-bar))
i
```


The final step prints out the estimate of I - this should be 2.74 (when rounded to two decimal places). This can be compared to the 5% and 1% points of $\chi^2_{n-1}/(n-1)$:

```
(/ (chisq-quant '(0.95 0.99) (length counts)) (length counts))
```

These are 1.28 and 1.41 respectively (again, rounded to two decimal places). Thus you have strong evidence to reject the null hypothesis of identical, independent Poisson counts for house sales - as the exploratory mapping suggested. If you wish to assess the proposal that larger zones tend to have more house-sales, you can use the `areas` function. This takes a zone-based variable as an argument, and returns the areas of these zones. For example, to investigate the relationship between the number of house sales and ward areas, enter

```
(plot-points (areas wards) counts)
```

The graph produced should confirm the idea that larger zones tend to have more house sales.

The function `zonal-mean` is similar to `zonal-sum` - it returns the mean value of some variable associated with the geographical points variable on a zone-by-zone basis. To obtain mean house prices on a ward-by-ward basis, enter

```
(def ward-price (zonal-mean house-sales price wards))
ward-price
```

The list obtained contains some mean values, and some elemennts taking the value -999. The latter represent *missing values* - these correspond to wards in which no house sales are observed - and so the notion of a mean house price is undefined. Missing values on a map are represented by a khaki colour - this is chosen to be clearly different from the red scale used for non-missing values.

```
(map wards ward-price)
```

The missing values can be removed from this list for analysis purposes, but the resulting variavble cannot be matched, as there is no longer a one-to-one matching between numerical values and zones. For example, consider the distribution of individual house-price levels in comparison to that for mean values aggregated to wards. Firstly, remove the missing values

```
(def ward-price
     (select ward-price
             (which (/= ward-price -999))))
ward-price
```

and then create a kernel density estimate plot of the price distribution, based on the averaged prices:

```
(def kernel (plot-lines (kernel-dens ward-price)))
```

Note that the distribution suggested here is bimodal, with a main mode at around 40,000 pounds and second mode at around 90,000 pounds. Next, add a kernel density estimate of the price distribution based on the *individual* prices.

```
(send kernel :add-lines (kernel-dens price)))
```

Here, the main mode is at roughly the same value, but the second mode is no longer noticeable. The aggregation appears to have accentuated the long upper tail of the distribution. It can be quite surprising what artefacts appear in data as an artefact of averaging or aggregation - see [5].

Finally in this section, you will look at contiguity and spatial smoothing. There are a number of useful functions for this in LiveMap. The basic one is called `contig`. Given a zone-based variable this function returns contiguity information in the form of a list of lists. Each inner list gives the indices of adjacent zones to a central zone - so there will be as many items in the outer lists as there are zones. This function is not particularly useful in itself, but the contiguity information is used in several other functions. For example, it is possible to carry out a "mean smooth" on some variable - that is, for each zone compute the average value of itself and its immediate neighbours. To do this, use the `mean-polish` function, which takes two arguments, the variable to be smoothed and the contiguity list. For example, a smoothed map is one way of investigating spatial trends in data:

```
(def cont (contig wards))
(def smoothed-unemp (mean-polish unemp cont))
(map wards smoothed-unemp)
```

A related function is `median-polish`. This is similar to `mean-polish`, but computes medians instead of means. This is another useful trend-finding device, but is more resistant to outliers. To test this out, type

```
(map wards (mean-polish unemp cont))
```

The *polish* functions provide smoothing on a zone-by-zone basis by applying some function to each zone and its neighbours. These are complemented by the *surround* functions, `mean-surround` and `median-surround`. These are identical to `mean-polish` and `median-polish` except that functions are only applied to the neighbours of the zones, not the zones themselves. Thus,

`mean-surround` returns a list of the mean values of the neighbours for each zone, and `median-surround` does the same with medians instead of means. These are both useful tools for finding spatial outliers - zones having very different values of some variable than their immediate neighbours. For example, enter

```
(plot-points crimes (median-surround crimes cont))
```

you should now see two wards that have notably different levels of crime to the median levels of their immediate neighbours. Linking this plot to a map of wards should identify where these two outlying wards are.

Two final smoothing functions are `surround` and `polish`. These are generalised versions of the functions described above. Each takes a third argument - a function lambda expression [6] which it applies to a list of values for each zone and its neighbours (`polish`) or just the neighbours `surround`. For example, the following maps local variability for the variable `crime`:

```
(map wards (polish crime cont #'standard-deviation))
```

If you have worked through this section, you will have used most of the data handling functions in LiveMap. As can be seen, they can be used alongside other XLisp-Stat functions to carry out statistical analysis and exploration of spatial data. The final skill you require is to compose more sophisticated and informative maps — the subject matter of the next section.

2.5 Map Composition and Combining Geographical Information

The default map produced by LiveMap is somewhat primitive - and deliberately so. The basic purpose of these maps is to explore spatial pattern, and the maps are designed to convey this information with as little extra information as is possible. However, there are occasions when you may wish to add to the information on the maps, either by adding keys and scales, or some contextual information. In this section you will see how this can be done.

Firstly, consider the map model in LiveMap. Each map consists of a number of *items* - these can be map information itself (ie a choropleth map, or a point map) or a scale or key. When an item consists of map information, it is refered to as a *layer*. Imagine each item or layer to be drawn on a transparent sheet - like an acetate sheet for an overhead projector. A map can then be thought of as a stack of these sheets. Although the sheets themselves are transparent, the drawings on them are not - so the top item

[6] if you don't know what one of these is, consult [6]

on the stack could obscure information in lower items. The `map` command produces produces a map with just one layer - which is determined by the arguments to map. Thus `(map wards)` produces a map with one layer, a map of wards in the study area. However, you can add or remove items from this map. To do this, firstly note that when a `map` command is entered, it returns a value. This value actually represents the map itself. If it is stored in a variable, you can then send it messages (using XLisp's object system) to add and subtract items.

As a practical example, firstly draw a map of census wards, and store it in a variable called `wardmap`:

```
(def wardmap (map wards))
```

This gives a map of wards, but since each one is filled in the same colour, one cannot distinguish ward boundaries. One way to solve this problem is to add a layer of ward boundaries. To do this, firstly create a line-based object containing the boundaries (using `outline`):

```
(def wardbounds (outline wards))
```

This can be turned into a map layer using the `map-layer` function. This function takes the same arguments as the `map` command, but only produces the map layer. For this to be displayed, you must add the map layer to the map by sending the `:add-item` message to the map variable:

```
(send wardmap :add-item (map-layer wardbounds))
```

You should now see the boundaries outlined on the ward map. Note that the boundary layer is the top item on the stack - if they were on the bottom you would not be able to see them, because the solid-filled wards would obscure them. Note also that this map may still be linked with other XLisp-Stat graphs - the map layer that links with the graphs will be the one at the top of the stack.

This last point is quite important, especially if you mix the geographies of the combined map layers. For example, you could add a set of house price proportional circles to the map you have just created:

```
(send wardmap :add-item (map-layer house-sales price 900))
```

This is shown in figure 6. Here, it is the circles from the house price map that will link to the other graphs, not the wards. This exercise also shows why it is sometimes useful to have several map layers. The map here provides much more information as to where house sales have occurred than does the more primitive map in figure 4.

Figure 6: Proportional Circles Plot for House Prices, Combined with Ward Outlines

The message `:drop-item` removes the top item from the stack of a map. For example `(send wardmap :drop-item)` would remove the proportional circle map. The message `:demote-item` does not remove the top item from the stack, but demotes it to the bottom of the stack. For example, if you have now dropped the circle layer from `wardmap` enter `(send wardmap :demote-item)`. Now the ward boundaries are below the filled wards - and the problem of boundaries being obscured becomes apparent.

You have now seen how to combine layers of map information. It is also possible to add keys and scales to the maps. These are treated as items, and can be added to maps using `:add-item`. The key item is created by the function `auto-key`. This takes four arguments - the variable being mapped, the x and y location of the map key (in pixels from the bottom left corner) and the title for the key (a string). Typically, one has to resize the map window to make space for the key (and stop it overwriting part of the main map). To see how this works, run through the following example:

```
(def crime-map (map wards crime))
```

At this point, re-size the new map window by about one and a half times. If you wish to be more exact than this, use the `:size` message for graphical windows:

```
(send crime-map :size 400 350)
```

Now add the key:

```
(send crime-map :add-layer
      (auto-key crime 10 220 "Crime Rates"))
```

At this point, a map with a key should appear. A similar function, `scale` produces a scale item. Again this takes four arguments - the x and y location of the scale in the window (in pixels from the bottom left hand corner), the length of the scale in map units, and trhe title for the scale (as a string). Add a scale to your map:

```
(send crime-map :add-layer (scale 10 20 1000 "1Km"))
```

Note that here, the map units are in metres. At the moment, the top stack item in this map is the scale - not much use for linking. A more useful top item would be a set of ward boundaries:

```
(send crime-map :add-item (map-layer wardbounds))
```

At this stage, you will have a map window as seen in figure 7.

Figure 7: Map Window with Key and Scale

Finally, you may wish to obtain hard copy of your maps. Until now, most of the maps included in this document are from raster screen-dumps. Although these are justifiable in a tutorial - where one is trying to obtain an accurate representation of what should appear on the screen - the quality is not really suitable for publication. To overcome this, maps have a message called `:eps-output`, which produces an encapsulated postscript (EPS) file of the map. When the message is sent, a dialogue to select the name of the output EPS file appears. This can be included in various types of documents for publication. Here, the EPS files are included in a LaTeXdocument. An example of this, based on the previous crime map, is given in figure 8.

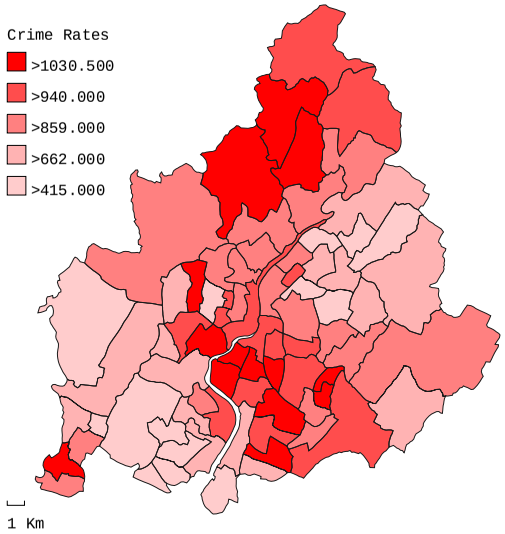

Figure 8: Map Produced using EPS File

2.6 Reading in Data

Up to this point, the data used has been supplied by the tutorial file. However, in real applications you will need to read in your own maps, and them to your own data. This section explains how this is done. Firstly, consider geographical data. This is read in using the functions `read-points`, `read-lines` and `read-zones`. In each case, the function takes only one argument, a string containing the name of an Arc/Info ungenerated file [3]. The value the function returns is a geographical variable of the corresponding type.

Unfortunately, the order in which the points, lines or zones appear in the new geographical variable will be arbitrary. This makes it difficult to associate numerical variables to the geographical objects in the correct order. To overcome this, Arc/Info assigns an ID code to each geographical code. When LiveMap reads in geographical data, this information is retained. Thus, provided you have a text-based data file which has one column containing the ID numbers, it is possible to match up the numberical variables to the geographical information. This is done using the `map-order` function. The

command

```
(map-order geog id x)
```

re-orders the variable `x` according to the IDs in the geographical variable `geog`, where the variable `id` contains the ordering of IDs in the variable `x`.

Often, the best approach is to convert the Arc/Info data into geographical variables and correctly ordered numerical variables at the start of a study, and then save these variables using the XLisp `savevar` command. From then on, one simply has to load this file to continue working. For example, the data in the tutorial could be saved using

```
(savevar '(wards unemp crime house-sales price) "crimes")
```

Of course, this approach is also useful for saving derived information, to save re-running computations in future analyses. For example one could enter

```
(savevar '(wards unemp crime house-sales price cont) "crimes")
```

to save the contiguity information calculated earlier in the tutorial alongside the other data.

2.7 Getting Help

The final part of this tutorial is perhaps one of the most useful. Most of the LiveMap commands have help information. They all make use of the XLisp help system. For example

```
(help 'map-order)
```

produces the text

```
MAP-ORDER                                        [function-doc]
Args: (geog id x)
Re-orders the elements of X so they match the element order
of GEOG. ID is a list of the element values of X in the
order matching X
```

3 Under the Bonnet

If you have been working through this exercise, will now have covered the basic functions in LiveMap. This section is somewhat more advanced, and addresses how parts of LiveMap can be used to produce your own functions,

extending the capabilities of the system. However, as this document is intended as a primer the treatment will be fairly brief - essentially the idea here is to give a flavour of this aspect of using LiveMap. To really take advantage of this, it is worth becoming very familiar with the LISP language and particularly XLisp-Stat.

3.1 Building New Mapping Functions

Firstly, consider function definition. XLisp-Stat contains the LISP programming language, and so allows the system to be extended by defining new functions. There is no reason why LiveMap functions cannot be used in this way. For example, to define a function to draw a choropleth map with ward boundaries added, the following code could be used:

```
(defun bound-map (zones x)
  (let
   ((bounds (outline zones))
    (map (map zones x)))
   (send map :add-item (map-layer bounds))
   map))
```

Again, familiarity with [6] will help in understanding what is going on here. Essentially the function uses two local variables, created in the `let` clause, containing the zonal boundaries and the map itself. In the main body of the function the boundaries are added to the map as an item. Finally, the last line causes the function to return the map itself as a value. This might be useful for adding other items to the map later, or saving it as an EPS file. Clearly, lots of other lines could be added to this function, for example adding a key and a scale to the map.

```
(defun bound-map (zones x key-title)
  (let
   ((bounds (outline zones))
    (key (auto-key x 10 210 key-title))
    (scale (scale 10 20 1000 "1 Km"))
    (map (map zones x)))
   (send map :size 400 350)
   (send map :add-item key)
   (send map :add-item scale)
   (send map :add-item (map-layer bounds))
   map))
```

Here, a few extra local variables have been added. Also, a third argument has been added to the function, a string variable containing the title for the key. Assume here that the map units are metres, and fix the scale title accordingly. The function could be made more sophisticated by adding the locations of the key and scale as keyword arguments:

```
(defun bound-map (zones x key-title
       &key (key-loc '(10 20))
            (scale-loc '(10 210)))
  (let
   ((bounds (outline zones))
    (key (auto-key x
         (first key-loc)
         (second key-loc) key-title))
    (scale (scale
           (first scale-loc)
           (second scale-loc)
           1000 "1 Km"))
    (map (map zones x)))
   (send map :size 400 350)
   (send map :add-item key)
   (send map :add-item scale)
   (send map :add-item (map-layer bounds)
   map)))
```

In this way, the locations of the key and scale default to the positions in the earlier definition, but if the function is called in the form

```
(bound-map zones x "Values"
           :key-loc '(20 20)
           :scale-loc '(20 220))
```

then the positions of the key and scale will change.

3.2 Data Manipulation

The previous section is an example of automating map production, but there are other, more computational uses for LiveMap in programming. These tend to make more use of the data manipulation functions. For example, one approach to spatial trend analysis for a set of zonal data might be to apply a median smooth repeatedly, until the spatial pattern becomes invariant to median smoothing. Once this is done, a single mean smooth is applied to remove the rough edges. This tends to produce a smooth map trend

which is not unduly influenced by spatial outliers. There are already the `median-polish` and `mean-polish` functions in LiveMap, so to carry out the iterated median smooth you simply need to incorporate these functions in a loop, using standard LISP programming:

```
(defun ims (x contig)
  (let
    ((old-x x)
     (new-x nil))
    (loop
      (def new-x (median-polish old-x contig))
      (if (< (max (abs (- new-x old-x))) 1.0e-4)
          (return))
      (def old-x new-x))
    (mean-polish new-x contig)))
```

For example, if you still have the contiguity variable for `wards`, then you could enter

```
(def smooth-crime (ims crime cont))
(map wards smooth-crime)
```

You could also keep a record of the convergence measure and return this as a list. This could then be inspected graphically to gain some insight into the way this process converges.

```
(defun ims-converge (x contig)
  (let
    ((old-x x)
     (converge nil)
     (epsilon nil)
     (new-x nil))
    (loop
      (def new-x (median-polish old-x contig))
      (def epsilon (max (abs (- new-x old-x))))
      (def converge (cons epsilon converge))
      (if (< epsilon 1.0e-4)
          (return))
      (def old-x new-x))
    (reverse converge)))
```

After defining this function, enter

```
(def converge-history (ims-converge crime cont))
(plot-points
   (iseq (length (converge-history)))
   converge-history)
```

to see that convergence to a fixed state is fairly rapid, and is virtually complete after 10 iterations.

3.3 Monte-Carlo Simulation

The random number generating functions in XLisp-Stat combined with the LiveMap functions make an ideal environment for Monte-Carlo simulations for spatial data. To demonstrate this, consider a non-parametric test of spatial association, based on counting the number of strong peaks and strong troughs in a zonal data set. A strong peak is defined as a zone whose value exceeds the largest of its neigbours by at least 10%, and a trough is defined as a zone whose value falls below the smallest of its neighbours by at least 10%. If there is a large number spatial outliers in the data, one could expect a relatively high number of strong peaks and strong troughs.

Firstly, a peak-count function can be defined, making use of the `surround` function. If you apply the `max` function to a zones neighbours, you will find the value of the largest neighbour for each zone. Comparing the zones own value with this then determines if the zone is a strong peak, and finally the peaks are counted using the `length` and `which` functions:

```
(defun peaks (x contig)
  (let
    ((surround-max (surround x contig #'max)))
    (length (which (> (/ x surround-max) 1.10)))))
```

A function called `troughs` can be defined in to compliment this:

```
(defun troughs (x contig)
  (let
    ((surround-min (surround x contig #'min)))
    (length (which (< (/ x surround-min) (/ 1.10))))))
```

For the `crimes` variable, you can compute the number of strong peaks and troughs by

```
(+ (troughs crime cont) (peaks crime cont))
```


Here, the statistic is 35. However, it is hard to tell whether this is unusually large or not, without considering a null distribution of this quantity under an assumption of no spatial pattern. This would be virtually impossible to consider analytically, but it could be investigated using Monte Carlo simulation. One approach to this is through a permutation test. Under the null hypothesis, any ordering of crime values to the zones is equally likely. Thus, the null distribution of the peak and trough count under random permutation of the crime values amongst the ward zones could be used as a basis for the test. Rather than derive this distribution analytically, one can simulate draws from this distribution by randomly permuting the `crime` variable with the `sample` function and computing the peak and trough count. Doing this n times, where n is at least 100, and comparing the distribution of the draws with the observed counts gives us the basis for a Monte-Carlo significance test. This can be coded as:

```
(defun monte-carlo (x contig n)
  (let
   ((l (length x))
    (permuted-x nil)
    (result nil))
   (dotimes (i n)
     (def permuted-x (sample x l))
     (def result
       (cons
         (+ (troughs permuted-x contig)
            (peaks permuted-x contig))
         result)))
   result))
```

For a 100 simulation Monte-Carlo test, enter

```
(def mc-results (monte-carlo crime cont 100))
(sort-data mc-results)
```

The results show values of the statistic ranging from 44 to 64 [7] suggesting that the observed count of 35 is unusually low, so that ward crime rates are more like their neighbours than one would expect under the null hypothesis. In other words, it would seem that there is some form of spatial autocorrelation in the data.

[7] Your results may differ - this is a simulation based on random numbers!

4 Epilogue

The previous sections of this primer should give you a grounding in the main functions of LiveMap, and also some idea of how it may be used in a more powerful way when combined with other XLisp-Stat functions. It is possible to delve even deeper "under the bonnet" and, for example define new types of map items or styles of map layer. However, this has been omitted from this primer, and will be the subject of a future document. To become an expert in this area, you will need to gain an understanding of the object system in XLisp-Stat, and the graphics commands.

Hopefully reading through this will give you some idea of what LiveMap could be used for. It has some of the functionality of a geographical information system, but not all. It is certainly not designed as a rival to Arc/Info. However, for certain numerically complex spatial analysis tasks it provides a powerful working environment. The general intention is to provide flexibility, by providing a set of building blocks which may be combined into functions designed to suit specific applications. By working through the exercises, and applying LiveMap to your own problems, you should begin to realise how versatile the environment is, even if it is not as simple to learn initially as, say, a desktop mapping system. The versatility is not solely due to LiveMap. To use Isaac Newton's phrase, it is sitting on giants shoulders - XLisp-Stat, and even more fundamentally LISP are both extremely flexible tools. This flexibility is intended to allow you to use your imagination to solve problems, rather than "shoe-horn" data into some pre-specified black box. The philosophy here is that the use of imagination should be encouraged — it is perhaps the most powerful analytical tool that you possess.

References

[1] C. Brunsdon, A.S. Fotheringham, and M.E. Charlton. Geographically weighted regression: A method for exploring spatial nonstationarity. *Geographical Analysis*, 28:281–289, 1996.

[2] A.D. Cliff and J.K. Ord. *Spatial Processes: Methods and Applications.* Pion, London, 1981.

[3] ESRI. *Understanding GIS: The Arc/Info Method.* Environmental Systems Research Institute, Redlands, CA, 1993.

[4] John Haslet, G. Wills, and Anthony Unwin. Spider - an interactive statistical tool for the analysis of spatially distributed data. *International Journal of Geographical Information Systems*, 4:285–296, 1990.

[5] Stan Openshaw. *CATMOG 38:The Modifiable Areal Unit Problem.* GeoAbstracts, Norwich, 1984.

[6] Luke Tierney. *LISP-STAT: An Object Oriented Environment for Statistical computing and Dynamic Graphics.* Wiley, Chichester, 1990.