Maps in S

Richard A. Becker
Allan R. Wilks

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The ability to draw geographical maps is an indispensable tool when analyzing or displaying geographically oriented data such as network data or census information. We describe a new map-producing mechanism integrated with the S system for data analysis. The map software features both line and area drawing, adaptation to plotting device resolution, map projections and a flexible user interface. The facility currently has a repertoire of three geographical databases, describing the national, state and county boundaries of the USA.

February 11, 1993

Maps in S

Richard A. Becker
Allan R. Wilks

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. The need for maps

The ability to draw geographical maps is an indispensable tool when analyzing or displaying geographically oriented data. This may simply mean being able to draw the national boundary of the USA, or it may mean being able to color each county of the USA a different color, according to the values of some variable. When specifying a map to be drawn, we may wish to name regions, such as New Jersey, or Northeastern USA, or we may wish to restrict the plotting to a range of longitudes and latitudes. We may wish to see national, state or county boundaries. We may wish to show the map in different projections. Rather than plotting, we may want to know the coordinates of the boundaries of various regions, perhaps for later plotting or for computing region areas or centers.

The `usa` function in S (Becker, et al 1988) provides a few of these facilities. The new `map` and `mapproject` functions in S, described in this memo, provide all of the facilities just described, and more. The fact that these functions are embedded in a data analytic environment is critical to their usefulness, in that the data analyst has a wide range of computational and graphical tools that are fully integrated with map drawing. In Section 2 we describe the capabilities of `map` and `mapproject`, by presenting a series of examples of their use. Then in Section 3 we give a discussion of the issues involved in organizing and accessing map data, which is typically massive and has complicated structure. In Section 4 we take up the problem of the parsimonious representation of connected sequences of line segments, and in Appendix A we give a linear time algorithm for computing such representations. Detailed documentation for `map` and `mapproject` are reproduced in Appendix B. Finally, information on map projections is given in Appendix C.

2. Making maps in S

The `map` function is an interface to geographical databases. In developing its capabilities we have used data supplied by the US Census Bureau (see References), which describes all county boundaries in the United States. From this data we constructed three geographical databases with information on national, state and county boundaries in the USA. The information in each geographical database is organized into three files. The first file has descriptions of *polylines*. These are sequences of points on the earth's surface, which, when joined in order by line segments, form a part of the map, typically a political or natural boundary. The second file describes *polygons* in terms of polylines, that is, each polygon is given as a list of polyline numbers, indexing polylines from the polyline file, which, when traversed in the given order, form a closed area of the map. Finally, a third file gives *names* to each of the polygons in the second file. It is primarily through these names that the map data is accessed. Polygons are named with a convention that allows several polygons to be accessed with one name, such as referring to the two parts of Michigan with the name ''Michigan.''

2.1. Simple Maps

A simple map of the national boundaries of the continental USA may be produced with the S expression `map('usa')` (see Figure 1(a)). To include state boundaries, use `map('state')`, and to get a full map of all county boundaries use `map('county')` (Figures 1(b) and 1(c)). In general, the first argument to `map` is the name of a geographical database, and with no other arguments, `map` will draw all the polylines for that database. Currently `usa`, `state`, and `county` are the only available databases. (But see Becker and Wilks (1991) for a description of how to build new databases.)

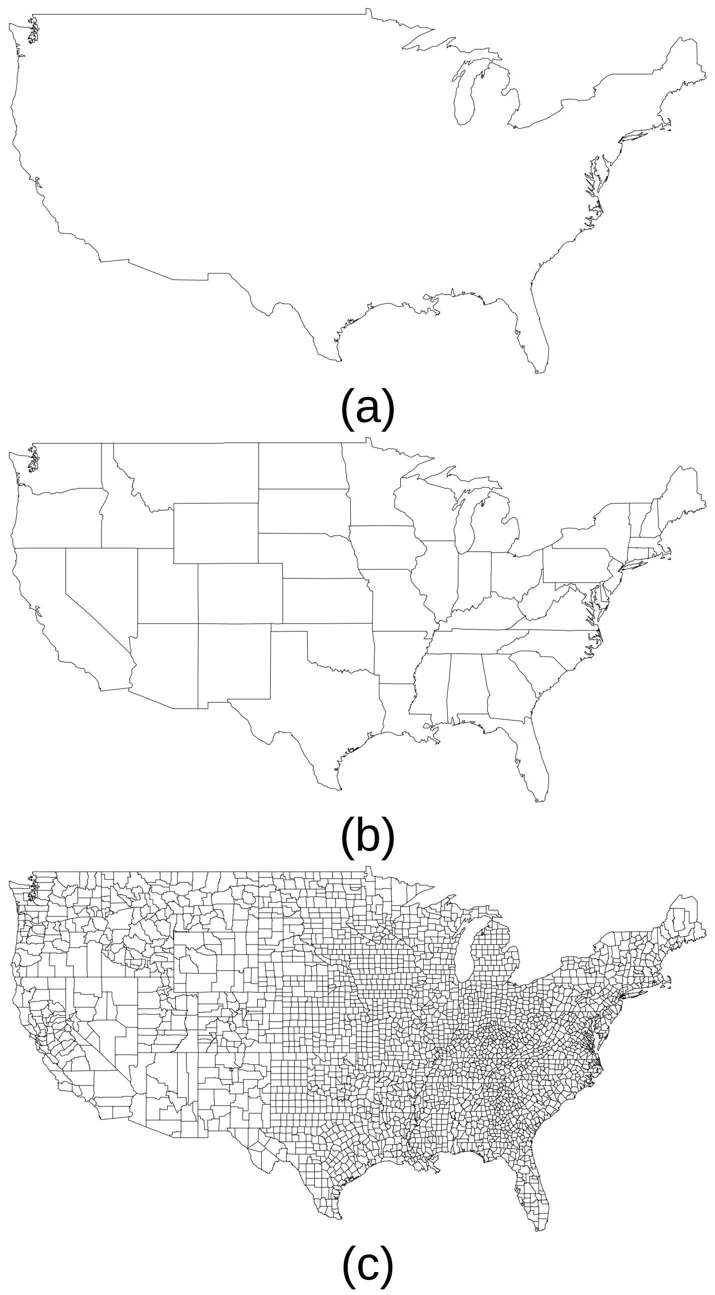

Figure 1. Three maps showing (a) national, (b) state, and (c) county boundaries.

The following table gives statistics for the sizes of these three databases:

Database	Segments	Polylines	Polygons
usa	7241	10	10
state	11350	169	63
county	46127	8941	3082

Nine of the ten polygons in the first database are islands (Manhattan, Staten Island, Nantucket, etc.). Only the continental USA is represented, so Alaska and Hawaii are missing. Thus the state database consists of the 48 continental states, plus the District of Columbia. There is quite a bit of variability in how individual states partition their territory into counties; sometimes they are not even called counties. For a discussion of the 3073 counties of the USA (in 1972), see Lerner's County and City Databook (1972), pages xxi-xxvii.

2.2. Selecting by Region

An optional second argument to `map` lists the names of particular regions to draw. For example,

```
> map('state', c('new york','New Jersey','penn'))
```

will produce a map of the boundaries of the three given states (see Figure 2). Notice that region names may be capitalized and abbreviated (by truncation). Illustrating the fact that a state may consist of more than one polygon, `map('state', 'mich')` produces a map of Michigan, which has two polygons, as in Figure 3.

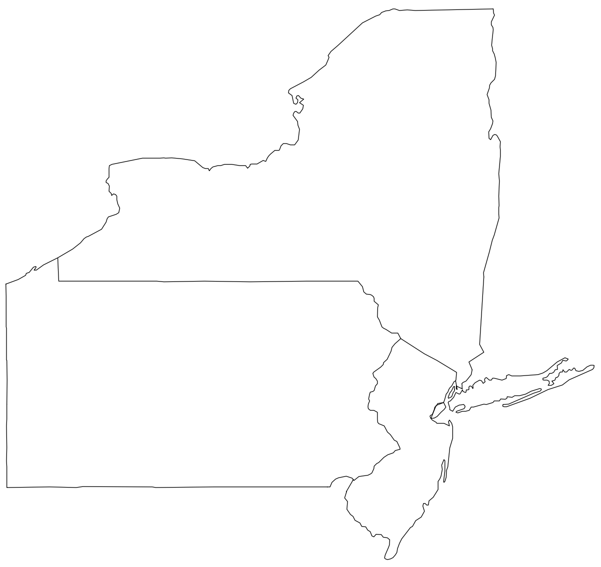

Figure 2. Map of New York, New Jersey and Pennsylvania.

Now it may be that we wish to refer to just one of these polygons; using `map`, this is possible, because each of the polygons has a unique name. To find the names of the Michigan polygons, but not do any plotting we may say

```
> map('state', 'mich', namesonly=T, plot=F)
[1] "michigan:north" "michigan:south"
```

This illustrates the scheme we have used in naming polygons: if a region is naturally composed of several polygons the individual polygons are named by a region name, followed by a ‘‘:’’ and a qualifying name. Region names in the county database include a state name and a county name, separated by a comma. For example:

```
> map('county', 'washington,san juan', namesonly=T, plot=F)
[1]   "washington,san juan:lopez island"  "washington,san juan:orcas island"
[3]   "washington,san juan:san juan island"
```

Incidentally, Washington, DC is denoted by `district of columbia` in the state database and by `district of columbia,washington` in the county database.

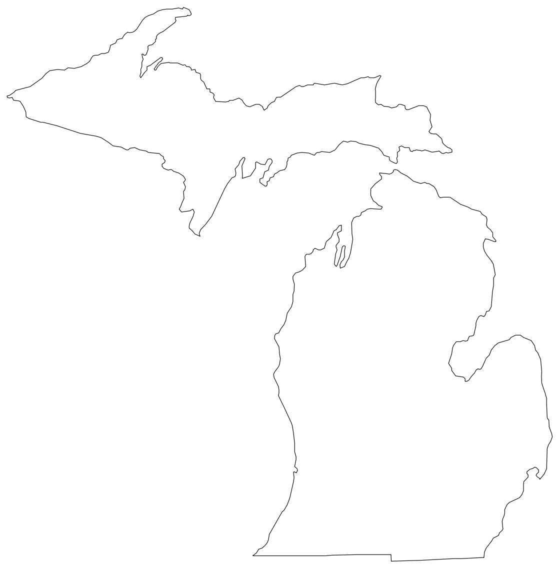

Figure 3. Outline of Michigan, with its two separate pieces.

The second argument to `map()` can be an arbitrary regular expression (see egrep(1) in the UNIX manual), to help pick out collections of regions. For example, `map('state','.*dakota')` could be used to make a map of the Dakota’s.

2.3. Selecting by Latitude and Longitude

Specifying a list of region names is one of two ways of saying what we want in the map to be plotted. The other way is to restrict drawing to a range of longitudes, a range of latitudes, or both. `Map` expects all longitudes and latitudes to be expressed in degrees east of Greenwich and degrees north of the equator. Longitude ranges from $-180°$ to $180°$ and latitude from $-90°$ to $90°$. This means, for example, that the USA is bounded by negative longitudes. Consider the standard S dataset `ozone.xy`, which gives the longitude and latitude of a number of ozone monitoring sites; for these sites `ozone.median` gives the median ozone concentration for a two-month period in 1974. We may plot this data (Figure 4) on a base map, with:

```
> map('state', xlim=range(ozone.xy$x), ylim=range(ozone.xy$y))
> text(ozone.xy, ozone.median)
> box()
```

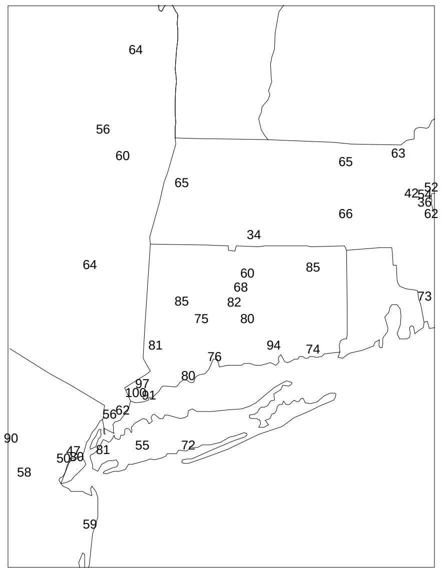

Figure 4. The ozone data is plotted on a map that covers its range.

2.4. Interior and Boundary Lines

In making a base map we sometimes would like to show the state boundaries in a color or line style different than the national boundaries. To do this, we distinguish between interior and boundary polylines. For a given call to `map`, an *interior* polyline is one that is part of the boundary of two of the polygons to be plotted; all other polylines (bounding only one such polygon) are called *boundary*. Thus to make our base map we might say

```
> map('state', interior=F)
> map('state', boundary=F, lty=2, add=T)
```

The result of these commands is shown in Figure 5.

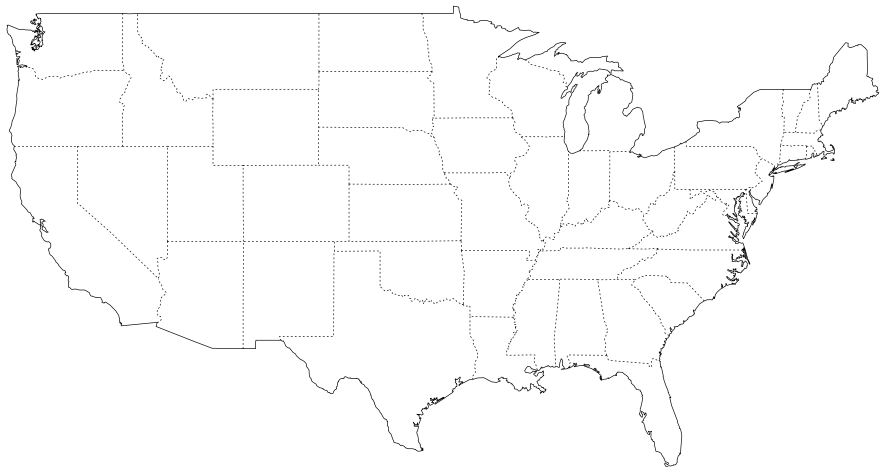

Figure 5. State map showing state boundaries in a different line style.

By default, both the `boundary` and `interior` arguments are `TRUE`. The `add` argument, `FALSE` by default, allows overplotting on the current map, and `lty=2` is a standard graphical parameter that requests line style 2.

2.5. Filled Regions

One of the main advantages of having polygon information in our maps is that we can draw filled regions according to statistical data. This is accomplished with the `fill=T` argument to `map`. As an example, we construct a map (Figure 6) showing the percent Republican vote in the 1900 election:

```
> state.names <- unix('tr "[A-Z]" "[a-z]"', state.name)
> map.states <- unix('sed "s/:.*//"', map(names=T, plot=F))
> state.to.map <- match(map.states, state.names)
> color <- votes.repub[state.to.map, votes.year == 1900] / 100
> map('state', fill=T, col=color)
> map('state', add=T)
```

The first expression changes uppercase to lowercase in the standard S dataset giving state names, so that these can be compared with the names returned by `map`. Next the complete set of state polygon names is requested (using `map(names=T,plot=F)`; the default database is `'state'`) and the trailing portions (from the ‘‘:’’ onwards) are removed so that we have a list of the state for which each polygon is a part or the whole. Then we create `state.to.map` that gives the translation from the ordering of the states known to S (alphabetical) to the ordering known to the mapping mechanism. By using this vector, as in the next expression, all the pieces of a state will be colored the same color. The `state.to.map` vector is a useful one to keep around, for it will work in any context where the ordering of the state data is as here. Notice that unless such a vector is being reused, it will usually be the case that there will be a step like this one, finding the translation between the ordering for the regions in your data and the ordering according to `map`. In general, the translation will have to be computed each time the set of selected polygons changes.

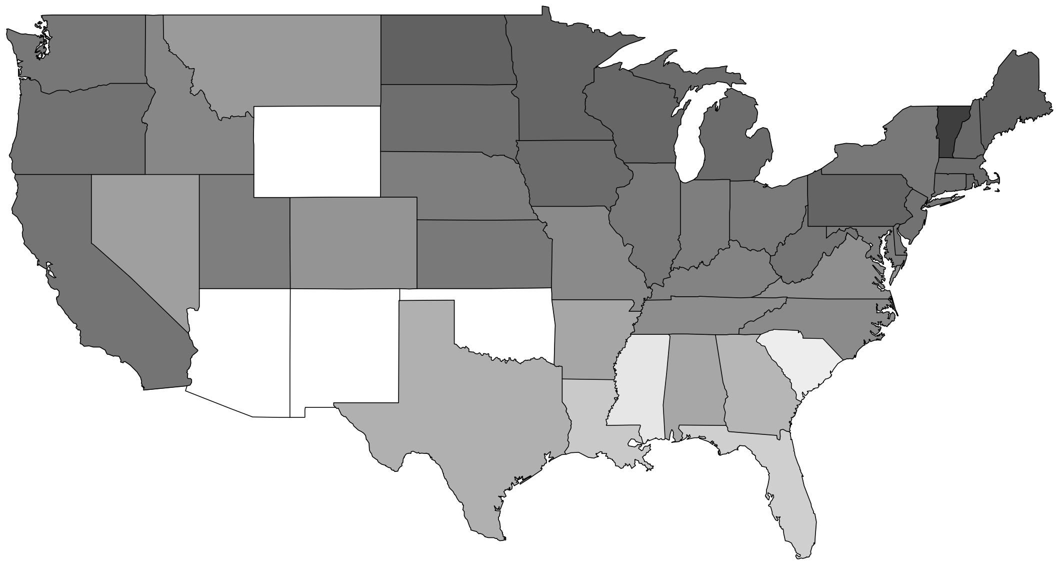

Figure 6. Percent Republican vote in 1900; darker regions correspond to higher values, but white means missing.

The expression that computes colors depends on the fact that the data are percentages, so they can simply be divided by 100 to get colors appropriate for a PostScript printer, via the `postscript()` device function in S. The `fill` argument in the second call to `map` says to fill polygons, rather than draw lines. The `col` argument is a vector of color numbers that are matched up with the selected polygons, which in this case is all of them. `Map` will use the i^{th} color to paint the i^{th} polygon (and will reuse colors cyclically, if necessary). The final call to `map` with `add=T` specified, overlays the picture with state boundaries. This is useful because there are some missing numbers in the 1900 results—Wyoming, Arizona, New Mexico and Oklahoma were not yet states. For missing colors `map` simply draws nothing, and putting on the state boundaries clearly marks out these missing regions.

Note that the i^{th} color is matched to the i^{th} polygon that `map` *finds*, and not to the i^{th} name you give it. This is important when region names are abbreviated; in the call

```
map('county', 'new jersey', fill=T, col=1:21)
```

all 21 counties of New Jersey are matched by `new jersey` and they are filled with the given 21 colors.

2.6. Map Coordinates

The return value of `map` is either the names of the polygons that were selected for plotting or the coordinates of the polylines or polygons that it retrieves from the databases, with the `fill` argument determining whether polylines or polygons are returned. If `map` actually does plotting, i.e., if `plot=TRUE`, the returned value is non-printing. In the coordinate case, the return value is a list and the coordinates are contained in components of the list named `x` and `y`. These components contain all of the coordinates in two long vectors; to be useful we must have a way of distinguishing where in these vectors one polyline (or polygon) ends and the next begins. This is done by following the end of the coordinates for each polyline (or polygon) with an `NA` in each vector. With no further processing this scheme is already useful, as most S graphics functions that accept *x* and *y* data will also accept `NA`s as well. For example, the `lines` function will ‘‘lift the pen’’ when it encounters `NA`; this means that handing the value `map(...,fill=F,...)` to `lines` we can get a map (up to projection) that `map` would have produced, assuming the coordinate system has been set up. Similarly, the value of `map(...,fill=T,...)` can be used in a call to `polygon`. Though this is convenient, we will usually want to do more with the returned vector, so it is useful to write a function `vapply` that takes a vector and a function and applies the function to each contiguous section of non-`NA` values in the vector:

```
vapply <- function(x, fun)
{
        n <- length(x)
        breaks <- (1:n)[is.na(x)]
        starts <- c(1, breaks + 1)
        ends <- c(breaks - 1, n)
        p <- length(starts)
        result <- logical(p)
        for(i in 1:p)
                result[i] <- fun(x[starts[i]:ends[i]])
        result
}
```

As an example of the use of `vapply`, suppose we wish to draw a map of New Jersey, with its counties outlined and labelled. We first get the county names. Then we call `map` with `fill=T` to get an `NA` separated list of the polygon coordinates. Using `vapply` on the `x` and `y` components of this returned data, we can compute the median *x* and *y* coordinates of each polygon. These medians are then used to position the text labels.

```
nj.name <- unix("sed 's/.*,//'",
     map('county', 'new jersey', plot=F, names=T))
nj.center <- lapply(
     map('county', 'new jersey', fill=T, plot=F)[c('x','y')],
     function(x) vapply(x, median))
map('county', 'new jersey')
text(nj.center, nj.name, cex=.75)
```

Figure 7(a) shows the result; part (b) shows what happens when means are used in place of medians, while the centers in (c) were are picked manually by eye. In general the automatic placement of labels is a hard problem. A reasonable procedure might be to move away from a middle value, looking for a long horizontal stretch, and stopping when enough room for the label is found. For some interesting recent progress on this problem, as well as further references, see Cook and Jones (1990).

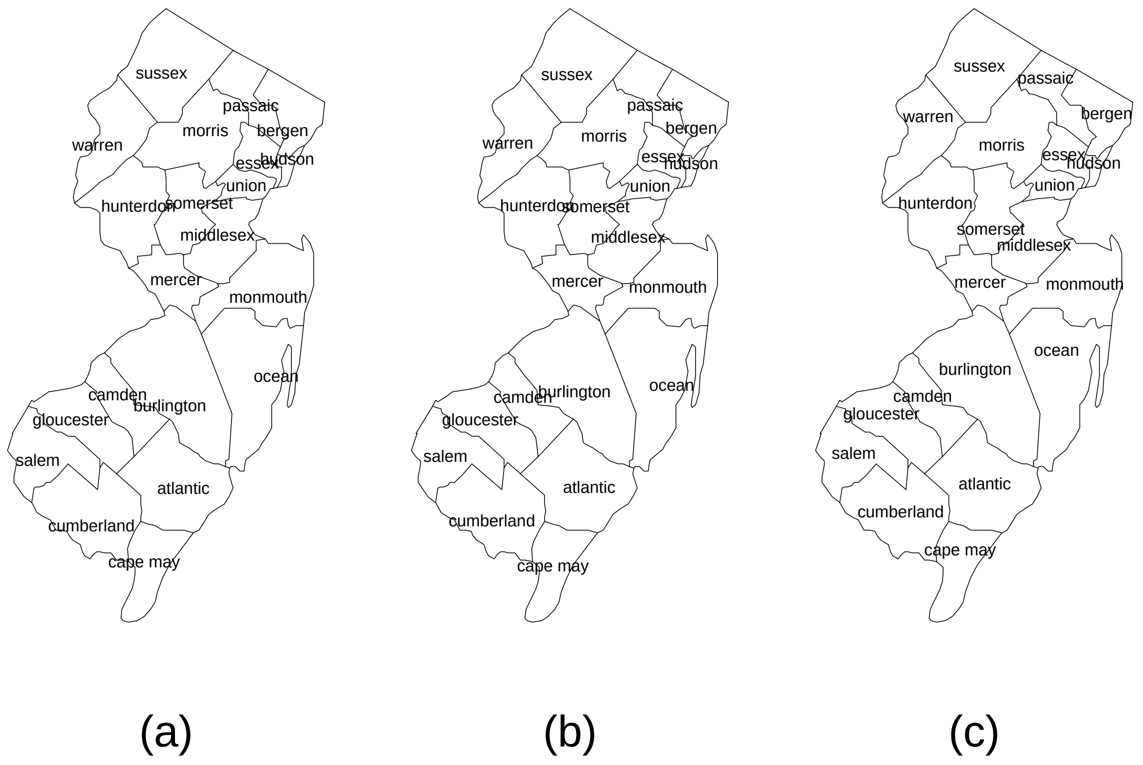

Figure 7. New Jersey with county labels; positions have been computed in several ways. In (a) the mean of the coordinates of the bounding polygon is used, while in (b) the median is used. The labels were placed manually by eye in (c).

2.7. Map Projections

Maps are flat representations of pieces of the nonflat (according to most people) earth. As such, a projection must always be used. The default used by `map` is to use longitute and latitude as *x* and *y* coordinates, adjusting the aspect ratio to be about 1 in the middle of the map. Many map projections have been proposed down through the centuries, and `map` can draw maps in a number of these by calling on the library of projection routines written by McIlroy (1990). By using the `projection` argument to `map`, a projection different than the default may be specified, and consequently a different coordinate system will be set up. Some projections require additional parameters, and these should be supplied in the `parameters` argument. For example,

```
> map('state', proj='albers', param=c(30,40))
```

produces a state map (Figure 8) using an equal-area Albers projection with true scale at latitudes 30° and 40°. Details on projections are supplied in Appendix C, which is adapted from McIlroy (1990).

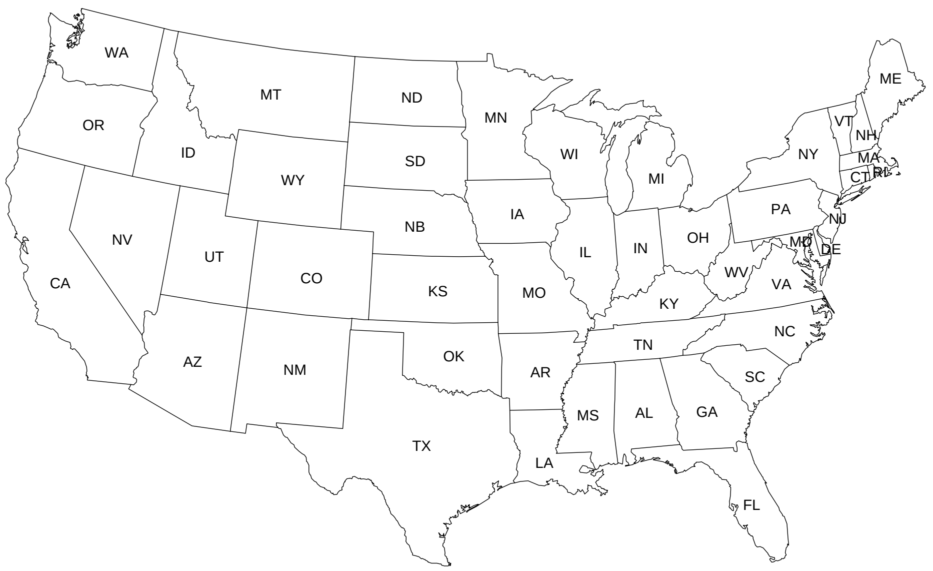

Figure 8. An equal area Albers projection of the USA with true scale at latitudes 30° and 40°.

Of course, once we have produced a map using a projection, we will probably want to plot some data on it, and this data likely needs to be projected into the same coordinate system. The `mapproject` function facilitates this task. It takes *x* and *y* data, together with `projection` and `parameters` arguments, just as in `map`, and returns a list with `x` and `y` components giving the projected data. As a simple example of its use, the state abbreviations in Figure 9 were added using the S datasets `state.center`, which contains longitude and latitude for central points in each state, and `state.abb`, which is a vector of state abbreviations. The command was

```
> text(mapproject(state.center), state.abb, cex=.75)
```

Normally, `mapproject` takes arguments like `map`, specifying the map projection and its parameters—in fact, when you give these arguments to `map`, it just passes them on to `mapproject`. Notice, however, that neither a projection nor parameters were specified in the call to `mapproject` above. This is possible because `mapproject` always saves a copy of the details of the last projection it did on the session (frame 0) dataset `.Last.projection`. Values from this dataset are used, as needed, when further calls to `map` or `mapproject` are made. In addition, if you don't specify the projection in a call to `map` or `mapproject`, you can give one or more parameters as `NA` and these will be filled in from the previous values. This gives a simple way of experimenting with the parameters of a projection. For example, you could try:

```
> map(proj='albers', par=c(30,40))
> map(par=c(20,50)) # another Albers projection
> map(par=c(NA,80)) # yet another; par=c(20,80) is implied
```

2.8. Resolution

The detail with which a map is drawn should ideally not exceed the resolution of the device on which it is being drawn. This can have an impact on the speed of drawing or the quality of the picture or both (see Section 4 for further discussion of this point). `Map` will automatically choose a level of detail that tries to match the current device resolution. However, this can be overridden with the `resolution` argument. If 0, this argument will force the map to be drawn to the full resolution available in the database. In the USA databases we have been using, this corresponds to roughly one kilometer. A nonzero value of `resolution` should be thought of as a number of device pixels. Roughly speaking, if successive points on a polyline are within this number of pixels of each other, they get merged into one point. Section 4 and Appendix A give more detail on how this is done. The default value of `resolution` is 1.

To illustrate the effect of changing resolution, see Figure 9, where the state map has been plotted at resolutions 0, 1, 5, and 20. The definition of a pixel in this case is a printer dot, about one three hundredth of an inch. The number of coordinate pairs defining these four maps is 11519, 3208, 1066 and 681, respectively.

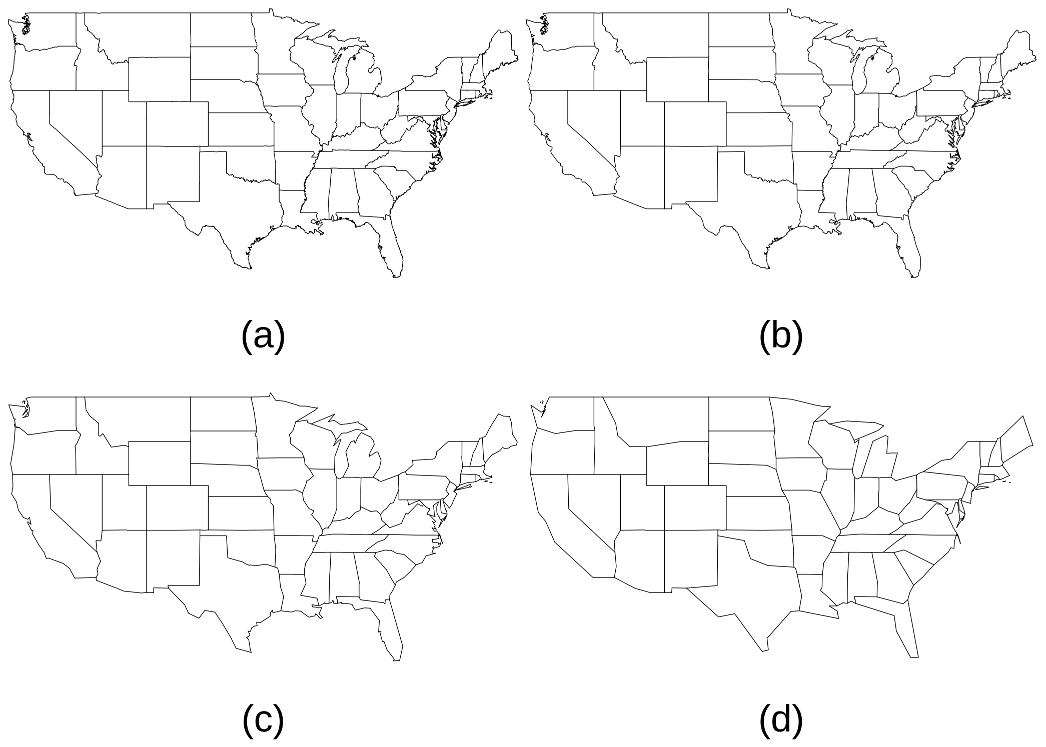

Figure 9. State map drawn at four resolutions: 0, 1, 5, and 20 respectively. The first map shows all the data in the state database, the second is adjusted to expunge consecutive points that are within a printer pixel of each other, while the remaining two pictures do the same thing for points within 5 and 20 pixels of each other. The number of coordinate pairs defining these four maps is 11519, 3208, 1066 and 681, respectively.

3. Geographical database organization

The typically large amounts of data associated with maps must be organized so that relevant information can be accessed in a timely fashion. Hand-drawn maps are made up of curved and straight lines. Such a line is approximated in our databases by a linear spline, that is, a sequence of short line segments, which, when joined together in order, never deviate too far from the original line. As mentioned earlier, we call such an object a *polyline*. The lowest level of information in one of our databases is therefore a collection of polylines, each of which is a list of longitude, latitude coordinate pairs. (It might be thought that we could, as a lower level of information, have a list of all coordinate pairs, with a polyline being a list of references to pairs. This leads, however, to more storage and longer access times.)

Besides being fundamental objects of interest, polylines are efficient, both for storage and for transmission to a plotting device. The older S `usa` function does just this. The problem with storing just polyline information is that there is no information on closed regions. Thus, for example, given all the polylines that describe the USA national and state boundaries, it is impossible to determine which ones bound, say, New Jersey. In fact, it may be that there is no set of polylines in the database that bound exactly New Jersey. This would be the case, for example, if the entire eastern seaboard coastline were stored as one polyline.

Thus, as a second level of information we keep a list of *polygons*, each of which is a list of references to polylines. These polylines, taken in order, exactly trace out the polygon. This implies some restrictions on which polylines are stored in the polyline file. In particular, if polylines are to be nonoverlapping, they must terminate at any point at which three or more line segments meet.

A map database consists of three files. In the first file, we store polyline information in the following form. Each polyline is given a reference number beginning at one; this is also done for the polygons. On one file, there is a list of all the polylines in the map, in order of their reference number. For each polyline we store its length (number of coordinate pairs) its minimum and maximum extent in both longitude and latitude, the reference numbers of the polygons to its left and right, and the list of coordinate pairs in order from its beginning to its end. (Left and right are well-defined because the polyline has a beginning and an end.) For polylines that bound only one polygon (coastal polylines, for example), zero is stored as the polygon reference number for the appropriate side. This is why polygon reference numbers are numbered from one.

In the polyline file, the information for each polyline, other than the actual coordinate list, is of fixed size. Therefore, we lay the file out with the fixed information for each of the polylines at the beginning of the file, together with a pointer for each polyline to the place where its coordinates are kept. This allows essentially random access to the polylines, at the expense of having to make an extra reference to the file for each one. If the file were not particularly large, this scheme would not be necessary, as the entire file could be read into main memory at the time it was first accessed. This would not work if the file were large, as the long startup time needed to read the file would be intolerable, especially for simple maps.

On the second file there is a list of all the polygons in the database, in order of their reference number. As with the polyline file, the polygon file begins with the fixed size information for each polygon, namely, its length (number of polylines), its minimum and maximum extent in both longitude and latitude, and a pointer to the sequence of reference numbers of its polylines. These reference numbers are stored as positive or negative, according as they are to be traversed in the direction given in the polyline file, or in the reverse direction. This is why polylines are numbered from one.

Finally, a third file contains names for each of the polygons. Each line of this ASCII file has a polygon number and a name. There is a naming convention that allows several polygons to be grouped conceptually into one region: the region is given a name and the individual polygons have the region name, a ‘‘:’’, and a qualifier.

Thus, each of our geographical databases is comprised of polygon, polyline and name files. The exact format of these files, together with information about constructing them, is given in Becker and Wilks (1991).

4. Parsimonious polylines

One frustration with plotting maps is that if the plotting device has fairly low resolution (say a CRT screen) and the map is highly detailed then there are many line segments ‘‘drawn’’ that are not even one pixel in length, so the plotting takes much longer than it needs to, and there may be an overly dense look in some regions. `Map` uses its `resolution` argument to attempt to produce polylines that are matched to the resolution of the plotting device. It does this by replacing each polyline with a new one that has the same endpoints as the first, but possibly deletes some of the interior vertices. We call this process *thinning* the polyline. McIlroy (1990) uses a simple algorithm to thin polylines: the user can supply an argument that says to delete every *n*th point of each polyline. This simple rule usually works quite well, but can obviously have problems if the deleted points of a polyline are important in determining its shape. In addition, it is difficult automatically to choose *n*.

A reasonable criterion for thinning is that the thinned polyline, when plotted on the given device, should look the same as if the unthinned polyline had been plotted. Pavlidis (1982) gives one solution to this problem in section 12.5. For a more recent paper, with a number of other bibliographic references, see Cromley and Campbell (1990). Pavlidis’ algorithm is based on the idea of near-colinearity of groups of points, and proceeds in a divide-and-conquer fashion to thin regions of a polyline that are almost colinear. (Incidentally, there is a notational conflict between our use of *thinning*, and the use Pavlidis makes of it, which is entirely different, and which involves reducing two-dimensional data rather than one-dimensional data.)

In Appendix A we present a ‘‘greedy’’ algorithm for thinning, whose running time is linear in the number of points in the original polyline. It is interesting that in spite of its linearity, the computation is sufficiently expensive that it may actually take longer to thin a polyline than simply to draw the whole thing. One possibility is to precompute several versions of the polyline database at various resolutions and to use the appropriate one at the time of plotting. Since polygons are lists of *references* to polylines, only the polyline portion of the database needs to be thinned. An earlier version of our software did this, but with current hardware, the speedup is now marginal.

It is worth noting that if thinning *is* done on the fly, it is important always to thin in the same direction for a given map, say in the positive orientation of polylines. This is because the algorithm may produce a different polyline when thinned in the reverse direction. If two adjacent polygons are to be drawn as filled regions, the polyline which is their common boundary gets thinned twice in opposite orientations and the two resulting polygons may overlap or have gaps on that boundary. Alternatively, polylines can be thinned symmetrically by thinning separately from each end to the middle of the polyline.

5. Directions

There are several directions in which we would like to carry our ideas. In the first place, we would like to develop more databases. An obvious first choice is a world map with country boundaries. The relevant data is available, but the problem is that it contains many regions (mostly islands), all of which would need to be manually named. However, this would be useful to have, given that AT&T is trying to do more international business, and can expect to collect more international data as time goes on. Other interesting databases might show the telephone

LATA's or the Metropolitan Statistical Areas of the United States, or the ZIP code regions.

Now that we have a general capability to produce base maps in S, it would be useful to augment the standard datasets in S to include other map data such as networks (telephone, road, rail, etc.) cities and towns.

Our current database organization is insufficient to represent more complicated relationships among geographical regions, such as the nesting that occurs with an island in a lake in a country. This shortcoming does not seem to matter for political maps, though, which tend to have a more regular, nonnested structure.

6. Summary

Through a series of examples, we have illustrated a new capability in S to create geographical maps of many varieties. This is based on a format for geographical databases that supports descriptions of both polylines and named polygons. Of the many features of this software, the most important are the ability to select map regions by name and to fill them with color (in constrast to merely outlining them).

The software described in this memo runs under the current research version of S. Please contact the second author for information about availability.

References

Becker, Richard A., John M. Chambers, and Allan R. Wilks (1988), *The New S Language*, Wadsworth & Brooks/Cole, Pacific Grove, California.

Becker, Richard A., and Allan R. Wilks (1991), ‘‘Geographical Databases for S’’, (to appear).

Cook, Anthony C. and Christopher B. Jones (1990), ‘‘A Prolog Interface To a Cartographic Database For Name Placement,’’ *Proceedings of the 4th International Symposium on Spatial Data Handling*, Volume 2, Zurich, Switzerland.

Cromley, Robert G. and Gerard M. Campbell (1990), ‘‘A Geometrically Efficient Bandwidth Line Simplification Algorithm,’’ *Proceedings of the 4th International Symposium on Spatial Data Handling*, Volume 1, Zurich, Switzerland.

Lerner, William (1972), *County and City Data Book 1972: A Statistical Abstract Supplement*, U.S. Bureau of the Census, Washington, D.C.

McIlroy (1990), documentation for proj(3) from *Tenth Edition UNIX manual*, Volume I.

Pavlidis, Theo (1982), *Algorithms for Graphics and Image Processing*, Computer Science Press, Rockville, Maryland.

US Department of Commerce, Census Bureau, *County Boundary File*, computer tape, available from Customer Services, Bureau of the Census, Washington, D.C. 20233.

Appendix A: Thinning

Part of the difficulty in thinning polylines is in formulating the correct statement of the problem. Our goal is to reduce the number of vertices in a polyline in such a way that only the unnecessarily fine detail is removed. This is partly because we wish to reduce the amount of information that is sent to the plotting device. In fact, the plot of a thinned polyline is often a *better* pictorial representation than the plot of the full polyline. This happens, for example, when a coastline has a lot of detail; the full polyline, when plotted on a low resolution device, may appear as just a very thick line, while the thinned version can be much crisper. The following formulation seems to capture the essence of these goals.

Problem: *Given n points $x_1, x_2, \cdots, x_n$ in the plane, and given $\delta > 0$, find integers $1 = i_1 < i_2 < \cdots < i_k = n$ such that the polyline determined by $x_{i_1}, \ldots, x_{i_k}$, thickened by δ, contains the entire polyline determined by the original points.*

The *thickening* of a polyline by δ is the set of all points that are within δ of the polyline. If we think of δ as the resolution of the plotting device, then it is approximately true that when a polyline is plotted, what actually gets drawn is its thickening by δ. Therefore, by plotting the polyline that solves the Problem, we are approximately plotting the full polyline. Further, if we think of the solution to the Problem as parameterized by δ, then by adjusting δ we can, for example, produce a version of a coastline segment that is crisp for a given plotting device.

Solving the Problem as given appears to be a relatively expensive optimization problem. Fortunately, there is a ‘‘greedy’’-type algorithm that is linear in *n*, and that appears to do fairly well at giving an optimum or near-optimum solution. The algorithm proceeds from one end of the polyline to the other, adding points to the reduced polyline only when needed. Suppose that we have added points $x_{i_1}, \cdots, x_{i_r}$ to the reduced polyline. We imagine circles of radius δ centered at each of the original points (see Figure 10). Draw a wedge, anchored at x_{i_r} whose rays are tangent to the circle around x_{i_r+1}. This wedge must contain x_{i_r+2}, if we intend to drop x_{i_r+1} from the list. In case the wedge does contain x_{i_r+2}, we drop x_{i_r+1} and consider x_{i_r+2} for deletion. To do this we create a new wedge, tangent to the circle around x_{i_r+2}, and intersect it with the old wedge (since x_{i_r+1} must continue to remain within δ of the new polyline). The new wedge is shaded at the end in Figure 10. We continue this process of refining our wedge until a point on the original polyline falls outside the wedge; the point just before that one then becomes $x_{i_{r+1}}$.

As mentioned at the end of Section 4, thinning must be consistent in a single map. An interior polyline on a map with `fill=T` will get used twice, once for each of the regions it bounds. If it is thinned in opposite directions each time, the region boundaries will not properly mesh. This can be solved by using *symmetric* thinning, in which each half of the polyline is thinned from the end to the middle. One simple consequence of this modification is that polylines with fewer than 5 points are never thinned.

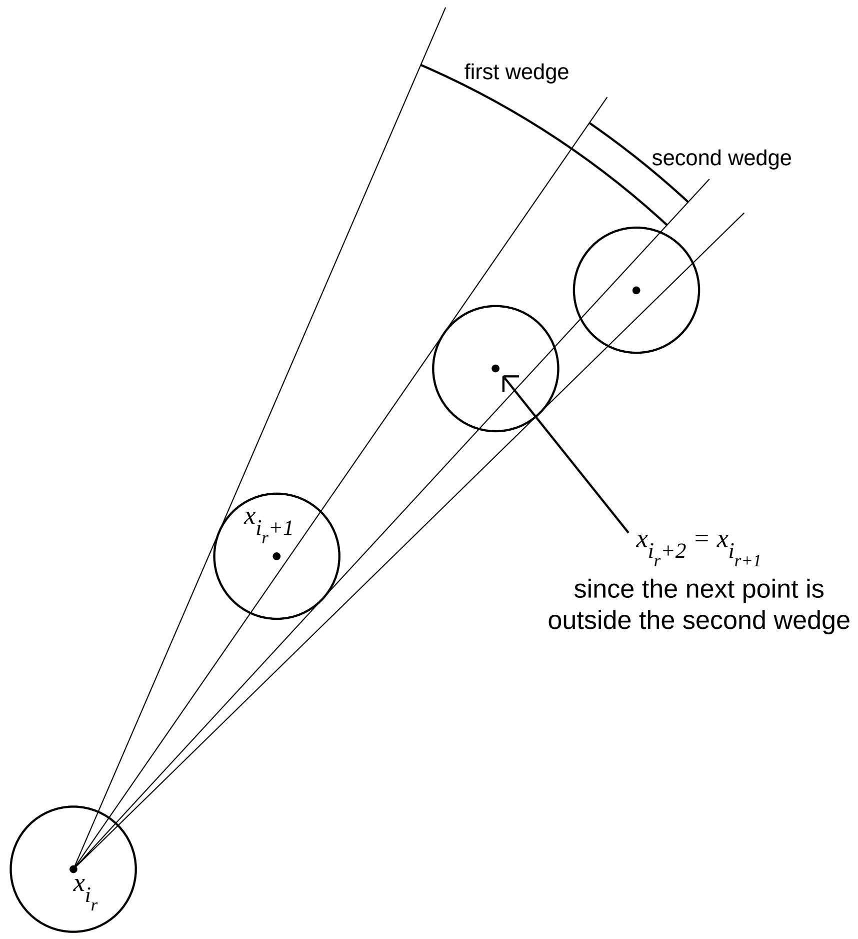

Figure 10. Illustration of the intersecting wedges that are kept as successive points are tested for deletion. All disks are of radius δ. The first wedge spans the tangents to the disk around $x_{i_r} + 1$ and the second wedge is the intersection of the first wedge with the span of the tangents to the disk around $x_{i_r} + 2$. Since the third point is within the (second) wedge, the second point may be deleted. However, the fourth point falls outside the wedge, so the third point is not deleted and thus becomes the next point ($x_{i_{r+1}}$) in the thinned sequence.

Appendix B: S documentation for `map` and `mapproject`.

map	Draw Geographical Maps	**map**

```
map(database, regions)  # simple form
map(database="state", regions=".", xlim=, ylim=, boundary=T, interior=T,
     fill=F, color=1, projection=, parameters=, orientation=,
     resolution=1, plot=T, add=F, namesonly=F)
```

ARGUMENTS

`database`	character string naming the geographical database from which `map` is to get its information. Currently the only choices are three USA databases: `"usa"` for national boundaries, `"state"` for state boundaries and `"county"` for county boundaries.
`regions`	character vector that names the polygons to draw. Each database is composed of a collection of polygons, and each polygon has a unique name. When a region is composed of more than one polygon, the individual polygons have the name of the region, followed by a colon and a qualifier, as in `michigan:north` and `michigan:south`. Each element of the `regions` argument is matched as a regular expression against all the polygon names in the database and matches are selected for drawing (but see `xlim`, `ylim` and `color=`, each of which can potentially modify this list). The default selects all polygons in the database.
`xlim`	two element numeric vector giving a range of longitudes, expressed in degreees, to which drawing should be restricted. Longitude is measured in degrees east of Greenwich, so that, in particular, locations in the USA have negative longitude. If `fill=TRUE`, polygons selected by `region` must be entirely inside the `xlim` range. The default value of this argument spans the entire longitude range of the `database`.
`ylim`	two element numeric vector giving a range of latitudes, expressed in degrees, to which drawing should be restricted. Latitude is measured in degrees north of the equator, so that, in particular, locations in the USA have positive latitude. If `fill=TRUE`, polygons selected by `region` must be entirely inside the `ylim` rang The default value of this argument spans the entire latitude range of the `database`.
`boundary`	logical flag that says whether to draw boundary segments. A boundary segment is a line segment of the map that bounds only one of the polygons to be drawn. This argument is ignored if `fill` is `TRUE`.
`interior`	logical flag that says whether to draw interior segments. An interior segment is a line segment of the map that bounds two of the polygons to be drawn. This argument is ignored if `fill` is `TRUE`.
`fill`	logical flag that says whether to draw lines or fill areas. If `FALSE`, the lines bounding each region will be drawn (but only once, for interior lines). If `TRUE`, each region will be filled using colors from the `color=` argument, and bounding lines will not be drawn.
`color`	vector of colors. If `fill` is `FALSE`, the first color is used for plotting all lines, and any other colors are ignored. Otherwise, the colors are matched one-one with the polygons that get selected by the `region` argument (and are reused cyclically, if necessary). A color of `NA` causes the corresponding region to be deleted from the list of polygons to be drawn. Polygon colors are assigned *after* polygons are deleted due to values of the `xlim` and `ylim` arguments.
`projection`	character string that names a map projection to use. See Appendix C of the Reference for a description of this and the next two arguments. The default is to use a rectangular projection with the aspect ratio chosen so that longitude and latitude scales are equivalent at the center of the picture.
`parameters`	numeric vector of parameters for use with the `projection` argument. This argument is optional only in the sense that certain projections do not require additional parameters. If a projection does require additional parameters, these must be given in the `parameters` argument. See Appendix C of the Ref-

erence for details.

`orientation` up to three numbers specifying the orientation of non-standard projections. Default is `c(90,0,m)`, where `m` is the middle of the longitude range. See Appendix C of Reference for details.

`resolution` number that specifies the resolution with which to draw the map. Resolution 0 is the full resolution of the database. Otherwise, just before polylines are plotted they are thinned: roughly speaking, successive points on the polyline that are within `resolution` device pixels of one another are collapsed to a single point (see the Reference for further details).

`plot` logical flag that specifies whether plotting should be done. If `plot` is `TRUE` the return value of `map` will not be printed automatically .

`add` logical flag that specifies whether to add to the current plot. If `FALSE`, a new plot is begun, and a new coordinate system is set up.

`namesonly` logical flag that says whether just the names of selected polygons will be returned as a character vector. If `FALSE`, map coordinates are returned.

Graphical parameters (see `par`) may also be supplied as arguments to this function.

VALUE

The polygons selected from `database`, through the `regions`, `xlim`, and `ylim` arguments, are outlined (`fill` is `FALSE`) or filled (`fill` is `TRUE`) with the colors in `color`. Names or coordinates of selected polygons are returned, depending on the value of the `namesonly` argument.

When `namesonly` is `TRUE`, the return value is a character vector of the names of the polygons that were selected for drawing. When `namesonly` is `FALSE`, the return value is a list with `x` and `y` components. If `fill` is `FALSE`, these vectors are the coordinates of successive polylines, separated by `NA`s. If `fill` is `TRUE`, the vectors have coordinates of successive polygons, again separated by `NA`s. Thus the return value can be handed directly to `lines` or `polygon`, as appropriate.

After a call to `map` for which the `projection` argument was specified there will be a dataset `.Last.projection` on frame 0, containing information about the projection used. This will be used for subsequent calls to `map`; see the documentation for `mapproject` for further details.

EXAMPLES

```
map()   # state map of the USA
map('usa')      # national boundaries
map('county', 'new jersey')         # county map of New Jersey
map(region=c('new york','new jersey','penn'))  # map of three states
map(proj='bonne', param=45)         # Bonne equal-area projection of states
map('county', 'washington,san', names=T, plot=F)
        # names of the San Juan islands in Washington state
map(xlim=range(ozone.xy$x), ylim=range(ozone.xy$y))
text(ozone.xy, ozone.median)
        # plot the ozone data on a base map
map(interior=F); map(boundary=F, lty=2, add=T)
        # national boundaries in one color, state in another
```

REFERENCE

R. A. Becker, and A. R. Wilks, "Maps in S", *AT&T Bell Laboratories Technical Memorandum,* December, 1990.

`mapproject` — Apply a Map Projection — `mapproject`

```
mapproject(x, y, projection="mercator", parameters=, orientation=)
```

ARGUMENTS

`x,y` two vectors giving longitude and latitude coordinates of points on the earth's surface to be projected. A list containing components named `x` and `y`, giving the coordinates of the points to be projected may also be given. Missing values (`NA`s) are allowed. The coordinate system is degrees of longitude east of Greenwich (so the USA is bounded by negative longitudes) and degrees north of the equator.

`projection=` optional character string that names a map projection to use. See Appendix C of the Reference for a description of this and the next two arguments.

`parameters=` optional numeric vector of parameters for use with the `projection` argument. This argument is optional only in the sense that certain projections do not require additional parameters. If a projection does require additional parameters, these must be given in the `parameters` argument.

`orientation=` optional; up to three numbers specifying the orientation of non-standard projections. Default is `c(90,0,m)`, where `m` is the middle of the longitude range. See Appendix C of the Reference for details.

VALUE

list with components named `x` and `y`, containing the projected coordinates. `NA`s project to `NA`s. Points deemed unprojectable (such as north of 80 degrees latitude in the Mercator projection) are returned as `NA`. Because of the ambiguity of the first two arguments, the other arguments must be given by name. Each time `mapproject` is called, it leaves on frame 0 the dataset `.Last.projection`, which is a list with components `projection`, `parameters`, and `orientation` giving the arguments from the call to `mapproject` or as constructed (for `orientation`). Subsequent calls to `mapproject` will get missing information from `.Last.projection`. Since `map` uses `mapproject` to do its projections, calls to `mapproject` after a call to `map` need not supply any arguments other than the data.

EXAMPLES

```
map(proj='bonne', param=45)
text(mapproject(state.center), state.abb)
        # Bonne equal-area projection with state abbreviations
```

REFERENCE

R. A. Becker, and A. R. Wilks, "Maps in S", *AT&T Bell Laboratories Technical Memorandum,* December, 1990.

Appendix C: Projections (adapted from McIlroy (1990))

Each standard projection is displayed with the Prime Meridian (longitude 0) being a straight vertical line, along which North is up. The orientation of nonstandard projections is specified by the three parameters *lat*, *lon*, and *rot*. Imagine a transparent gridded sphere around the globe. First turn the overlay about the North Pole so that the Prime Meridian (longitude 0) of the overlay coincides with meridian *lon* on the globe. Then tilt the North Pole of the overlay along its Prime Meridian to latitude *lat* on the globe. Finally again turn the overlay about its 'North Pole' so that its Prime Meridian coincides with the previous position of (the overlay's) meridian *rot*. Project the desired map in the standard form appropriate to the overlay, but presenting information from the underlying globe.

In the descriptions that follow, each projection is shown as a function call; if it requires parameters, these are shown as arguments to the function. The descriptions are grouped into families.

Equatorial projections centered on the Prime Meridian (longitude 0). Parallels are straight horizontal lines.

`mercator()` equally spaced straight meridians, conformal, straight compass courses
`sinusoidal()` equally spaced parallels, equal-area, same as *bonne(0)*
`cylequalarea(lat0)` equally spaced straight meridians, equal-area, true scale on *lat0*
`cylindrical()` central projection on tangent cylinder
`rectangular(lat0)` equally spaced parallels, equally spaced straight meridians, true scale on *lat0*
`gall(lat0)` parallels spaced stereographically on prime meridian, equally spaced straight meridians, true scale on *lat0*
`mollweide()` (homalographic) equal-area, hemisphere is a circle

Azimuthal projections centered on the North Pole. Parallels are concentric circles. Meridians are equally spaced radial lines.

`azequidistant()` equally spaced parallels, true distances from pole
`azequalarea()` equal-area
`gnomonic()` central projection on tangent plane, straight great circles
`perspective(dist)` viewed along earth's axis *dist* earth radii from center of earth
`orthographic()` viewed from infinity
`stereographic()` conformal, projected from opposite pole
`laue()` used in xray crystallography
`fisheye(n)` stereographic seen through medium with refractive index n

Polar conic projections symmetric about the Prime Meridian. Parallels are segments of concentric circles. Except in the Bonne projection, meridians are equally spaced radial lines orthogonal to the parallels.

`conic(lat0)` central projection on cone tangent at *lat0*
`simpleconic(lat0,lat1)` equally spaced parallels, true scale on *lat0* and *lat1*
`lambert(lat0,lat1)` conformal, true scale on *lat0* and *lat1*
`albers(lat0,lat1)` equal-area, true scale on *lat0* and *lat1*
`bonne(lat0)` equally spaced parallels, equal-area, parallel *lat0* developed from tangent cone

Projections with bilateral symmetry about the Prime Meridian and the equator.

`polyconic()` parallels developed from tangent cones, equally spaced along Prime Meridian
`aitoff()` equal-area projection of globe onto 2-to-1 ellipse, based on *azequalarea*
`lagrange()` conformal, maps whole sphere into a circle
`bicentric(lon0)` points plotted at true azimuth from two centers on the equator at longitudes $\pm$*lon0*, great circles are straight lines (a stretched gnomonic projection)

`elliptic(lon0)` points are plotted at true distance from two centers on the equator at longitudes ±*lon0*

`globular()` hemisphere is circle, circular arc meridians equally spaced on equator, circular arc parallels equally spaced on 0- and 90-degree meridians

`vandergrinten()` sphere is circle, meridians as in *globular*, circular arc parallels resemble *mercator*

Doubly periodic conformal projections.

`guyou()` W and E hemispheres are square

`square()` world is square with Poles at diagonally opposite corners

`tetra()` map on tetrahedron with edge tangent to Prime Meridian at S Pole, unfolded into equilateral triangle

`hex()` world is hexagon centered on N Pole, N and S hemispheres are equilateral triangles

Miscellaneous projections.

`harrison(dist,angle)` oblique perspective from above the North Pole, *dist* earth radii from center of earth, looking along the Date Line *angle* degrees off vertical

`trapezoidal(lat0,lat1)` equally spaced parallels, straight meridians equally spaced along parallels, true scale at *lat0* and *lat1* on Prime Meridian

Retroazimuthal projections. At every point the angle between vertical and a straight line to ‘Mecca’, latitude *lat0* on the prime meridian, is the true bearing of Mecca.

`mecca(lat0)` equally spaced vertical meridians

`homing(lat0)` distances to ‘Mecca’ are true

Maps based on the spheroid. Of geodetic quality, these projections do not make sense for tilted orientations. For descriptions, see corresponding maps above.

`sp_mercator()`

`sp_albers(lat0,lat1)`