Constructing a Geographical Database

Richard A. Becker

Allan R. Wilks

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The `map()` function in S permits a wide range of line and filled area maps to be drawn for use with map-oriented data analysis. The data underlying these maps is stored in a *geographical database*. This report describes the format of the geographical database and the steps involved in constructing one. The process is illustrated by our experience in creating a database for the coastlines and national political boundaries of the world.

The world database allows data to be displayed on a map of the entire world or on smaller subsets such as Europe or individual countries. The data comes from the World Data Bank II (US Dept. of Commerce) but we have performed edits to correct digitizing errors and have added accurate descriptions of the polygonal regions that make up each country. Because the database contains polygonal information, choropleth maps can be drawn that use data values to control the color of regions, and because it is so detailed, maps can be made in a wide range of scales.

While we do not expect most people will want to build an S geographical database, there are many possible applications requiring new base maps and this report documents the process of constructing such a database so that it is accessible to anyone who wants to try.

September 16, 1997

Constructing a Geographical Database

Richard A. Becker

Allan R. Wilks

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

Our previous document, "Maps in S" (Becker and Wilks, 1991), described a set of S functions for displaying maps. The software to implement these functions was introduced with the August, 1991 version of S, along with a county-level database of the United States. Almost immediately following this introduction, we received requests for other map databases: of LATAs, particular countries, or even the entire world. This report is an answer to those requests. It describes how we produced a world database for S, documenting the process step by step. It also describes the format of the data expected by the S mapping facility. The world database that we constructed was made available on the statlib server† in April, 1992 and is sufficiently detailed that it should immediately answer many of the map requests we have received. In addition, the description of how it was built provides a model for constructing other databases.

2. The Data

There are a number of steps in creating a geographical database for use with the `map()` function in S. Our goal in this report is describe each of these steps as they were actually carried out in a large example. The final product of this process is what we call a *geographical database*, which consists of a set of three files, containing information on polylines, polygons, and region names. The actual format of the data in these files will be described later. For now, we need to know the structure of the information in these files in order to understand the process used to create them.

The *polyline file* describes a set of polylines, each of which is a connected sequence of points on the earth's surface—the vast majority of the data in the polyline file consists of the coordinates of these points. The *polygon file* says how to fit the polylines together to form closed polygonal regions, each representing a country or state or province, for example. The relationship between the polylines and polygons—the topology of the map—is also stored in these two files. Each polyline is directed; in other words, it has a beginning and an end (its first and last points) and therefore a left side and a right side, as

† The `statlib` server is a computer at Carnegie-Mellon University that provides a repository for statistical programs and data, much like the `netlib` server (Dongarra and Grosse, 1987) provides facilities for numerical software. To retrieve the world database, send the message `send world.map from S` to `statlib@lib.stat.cmu.edu`. The server will send a description of the access procedure via return mail.

if you were standing at the first point of the polyline, looking toward the second. Each polyline is given a number, and the definition of a polygon is a list of these numbers, together with instructions on the direction in which to follow each polyline—forward or reverse—when making a complete circuit around the polygon. Each polygon is given a number, and the definition of a polyline also contains the information about which polygon is to the left of it and which is to the right. Finally, the *region name file* gives a name to each of the polygons.

These three files must be constructed from whatever raw information is available. We assume that someone has already digitized the desired information, giving coordinates of the endpoints of the individual line segments that comprise the map. In other words, we are assuming that we start with at least the individual line segments of the polylines. We may, however, have more information than this. For example, the Census Bureau data, which we used to construct the state and county maps, included for each line segment, the region (state and county) to each side of it. The world data, described in detail here, provided individual line segments already joined into polylines but no information was included about the polygons bounded by the polylines.

The World Data Bank II (WDBII) data consists of 33827 polylines, each of which is given as a sequence of latitude/longitude pairs. The polylines are divided into regions of the world (North America, South America, Europe-Africa, and Asia) and into four classifications within the regions. The classifications are: international boundaries or limits of sovereignty; coast, islands, and lakes; rivers; and internal boundaries. These classifications are further subdivided by importance of the features. We extracted data for 4115 of the polylines. These were comprised of the 2976 polylines of coasts, islands and lakes denoted in the database by

- coast, islands and lakes that appear on all maps

and the 1139 political boundary polylines labelled as

- demarcated or delimited (1038)
- indefinite or in dispute (57)
- lines of separation or sovereignty on land (38)
- no defined line; guide only; South Arabian Peninsula (6)

We omitted the remaining approximately 30,000 polylines. These corresponded to smaller islands and lakes, as well as reefs, salt pans, ice shelves and glaciers. We also left out about 150 polylines corresponding to ill-defined boundaries, as well as most of the river polylines, since they don't serve to separate the land belonging to nations. (This is a statement about the database, not about boundaries in general; when a political boundary lies along a river, WDBII will mark it as a political boundary and hence it will be included in the subset we chose.) We also included none of the internal boundary polylines (US states and Canadian provinces), as our goal was to make a map only of nations.

With the major changes that have occurred in the world political situation since 1990, it is important to note here that our data was collected at a prior point in time and corresponds to an outdated world. Of course, the coast, islands, and lake data is still correct, as are most of the political boundaries. However, we did update boundaries that were easy to do. In particular, we removed the boundaries between East and West

Germany and between Yemen and the Peoples Republic of Yemen. We did not, however, attempt to introduce the boundaries corresponding to the Baltic states or the Commonwealth of Independent States. To do so would have required digitization of those boundaries.

3. Editing the data

Having extracted a subset of the WDBII data consisting of 4115 polylines, comprised of 1685078 vertices, we needed to make sure that certain constraints were satisfied, to make it possible to construct the sort of database that the `map()` function expects. There are three such constraints; we give two of them here and defer the third until later. If we think of the polylines as defining an abstract graph embedded in the plane† where the segment endpoints are the vertices and the segments themselves are the edges, then to ensure that the plane is covered by simple closed regions, the data must satisfy:

Constraint #1: the valence of each vertex is at least two
Constraint #2: intersecting segments do so only in a common endpoint

The valence of a vertex is the number of edges having the vertex as an endpoint. A vertex that has valence one is *dangling* and corresponds in this database to a polyline with an endpoint that doesn't connect to anything else in the database. A vertex with valence zero can't happen in WDBII because of the structure of the data. Individual segments of polylines can intersect in many ways (in a point interior to both segments, in a subinterval of each segment, in the interior of one segment and the endpoint of another, etc.) but only one type of intersection is allowed, viz, when two segments share a single common endpoint.

Our next step was to construct a program that graphically displayed the data, and automatically found places where one of the two constraints was violated. There turned out to be many such violations. We began by looking for dangling vertices; there were about 2000 of these, or roughly one third of the polyline endpoints. To deal with these we included some editing facilities in our program. The idea behind the editing was to modify the selected polylines in such a way that the constraints became satisfied. We made an early decision not to introduce any new coordinates into the polylines; in this way, each edit could be described as a discrete manipulation of the existing vertices.

We also thought that a sequence of edits would be an ideal way to export the data. The idea was that anyone wanting a world database would be able to request the World Data Bank II from the Department of Commerce and the file of edits from us. They could then extract the appropriate subset of the data, and run a small program that would apply our edits, giving them a clean database. We would not have to ship a large database of world coordinates and the approach would be appropriate even if the world data had been proprietary. We ultimately decided against this technique for several reasons:

† The graph should actually be considered as embedded in a sphere, in order to preserve line segments that cross the 180 degree meridian. This could be achieved by first stereographically projecting the data from the globe to a plane, using, say, one of the poles for the projection point. We have chosen not to do this, so that as we worked with the data graphically, the shapes were the familiar ones of the world map— stereographic projection would have destroyed this property. Empirically, very few segments crossed the 180 degree meridian.

first, manual effort is necessary to get WDBII; second, WDBII is public domain data and can be freely distributed; third, our file of edits became large itself.

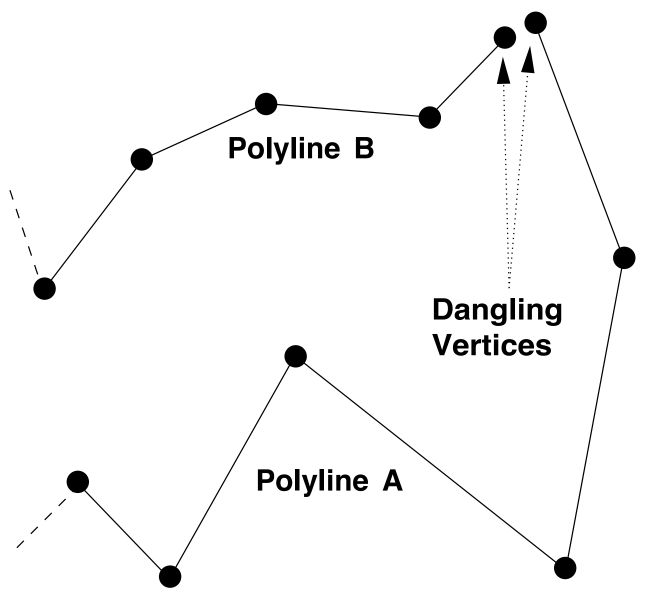


Figure 1. Two polylines, A and B, each with a dangling vertex. The editing solution here is to ''snap'' the dangling endpoint of A to be equal to the dangling vertex of B (or the other way around—it doesn't matter).

A common situation is represented schematically in Figure 1, where we show two polylines, called A and B, each with a dangling vertex. More than likely, the map that was being digitized showed just one line here, so that the two dangling vertices ought to be one and the same. This can happen, for example, when returning to the starting point of an island. In any case, we make this assumption and force the connection. Because of our decision not to introduce new vertices, we choose to correct this situation by ''snapping'' the dangling end of A to be equal to the dangling end of B (or the other way around—it doesn't matter). This is recorded by saying that vertex *n* of A takes on the coordinates of vertex *m* of B.

Each of the roughly 2000 dangles was manually corrected in this way after visual inspection of the situation. In cases where two dangling endpoints were very near one another, the snapping process could have been automated. However, we found from our visual inspection that there were many other situations where human intervention was necessary in order to choose the correct way to snap. In any case, the process of *finding* the dangles was automated and our editing software enabled us to move rapidly from one dangle to the next.

Having satisfied Constraint #1, we then turned to Constraint #2. To do this we computed every intersection among the two million segments. This computation was facilitated in two ways. First we computed the bounding box of each polyline, and then only compared two segments if the bounding boxes of their polylines overlapped. Second, having found two polylines with overlapping bounding boxes, we computed the intersections of their line segments with a sweep-line algorithm. Each time we found an intersection not satisfying Constraint #2 we recorded it for later editing.

There were a number of situations that were uncovered by this computation. A simple one was duplicate polylines—for an unknown reason, there were a number of polylines in the database that were simply repeated. The edit for these was simply to remove the extra copies. Another common situation was the crossing of two political boundaries, without a vertex at the intersection point. This situation is illustrated in Figure 2. The solution is to pick the closest pair of vertices and snap one to the other (as well as to split the polylines at this common vertex as described below). Automation of this task would have been very difficult.

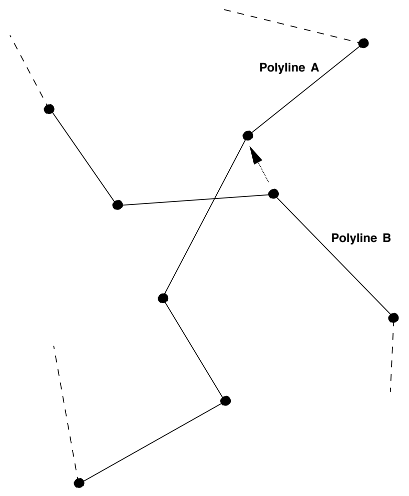


Figure 2. An illegal crossing of two polylines—the intersection point should be a vertex of each polyline. The solution is simple; just snap a vertex of Polyline B to a nearby vertex of Polyline A, as indicated by the arrow.

Though the solution just described for repairing an illegal crossing is sufficient to achieve satisfaction of Constraint #2, it leaves the two polylines in a state that makes them unsuitable for use as polylines in the final database. Of course, we could take the point of view that once the editing done, we will split each polyline into its component segments and send these to a database constructor as raw data. In fact, this is the approach we used for the construction of a UK database to be described in section 9. It seemed a waste, however, not to take advantage of the fact that WDBII already organized its data in polylines, and to use these polylines as much as possible. However, there are some constraints associated with compiling our original set of vertices and line segments into a geographical database, and if we are to keep the WDBII polylines, we must ensure that these constraints are still satisfied.

Here, then, is a further constraint:

Constraint #3: vertices interior to a polyline have valence two

The purpose of this constraint is to ensure that when we establish the regions of the map, each polyline will have a unique region to each side of it. Looking at Figure 2, we see that the region to the left of polyline B, looking along the polyline from the upper left to the lower right of Figure 2, changes at the newly created intersection point. The same is true of the region to the right of polyline B and the regions to both sides of polyline A.

The problem is that for both polylines, the intersection vertex is an interior vertex with valence 4 in the graph.

To repair this problem, we added another editing operator—‘‘split polyline’’—which breaks one polyline into two at the indicated vertex. The actual sequence of edits in Figure 2 would be first to split each polyline at the vertex that will become the intersection point, giving four polylines in place of the two. Then each of the two new endpoints of (former) polyline B is snapped to the intersection vertex.

Another way in which Constraint #2 failed to be satisfied happened for polylines representing a long spit of land extending into the ocean, such as at Cape Hatteras in North Carolina. When the width of the spit was small compared to the digitizing error (presumably) the polyline tended to cross itself, sometimes as much as 30 or 40 times. Though this is really a special case of the situation illustrated in Figure 2, we nevertheless took care of it with a new editing operator—‘‘untwist.’’ The idea of untwist is to travel along the polyline until just before the bad intersection, then ‘‘hop’’ to the later vertex that happens just before the intersection occurs again. At that point, follow the polyline in the reverse direction and cross over again where the intersection is. This is illustrated in Figure 3.

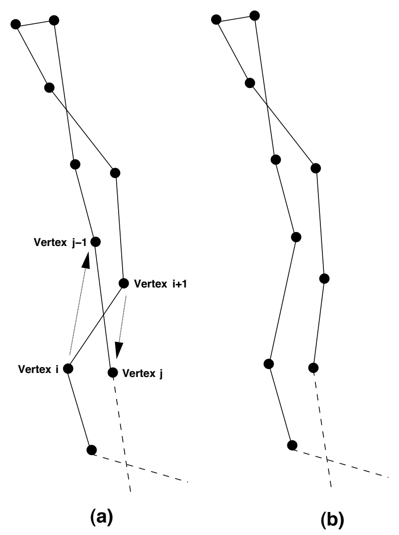


Figure 3. A polyline that crosses itself multiple times can be ‘‘untwisted.’’ In (a), vertex i and vertex j are chosen for the untwisting operation. Assuming i is less than j, the part of the polyline between these two vertices is rearranged by following first the arrow from i to $j-1$, then the polyline in reverse order from $j-1$ to $i+1$, then the arrow from $i+1$ to j. The result of the operation is shown in (b). In this example, at least two more untwists are needed to remove all intersections.

Each of these situations, and several others, were disposed of using our interactive program and the four editing operators described above. When we had completed all of the edits, we checked the Constraints and found them to be satisfied. In all, we applied 7860 edits to clean up the data:

6737 snap a vertex to another vertex
237 split a polyline at an interior vertex
448 delete a polyline
438 perform an untwist operation

One final editing operation was carried out automatically when the edited database was written. The points along each polyline were checked for adjacent duplicates and those duplicates were omitted. This operation made it unnecessary to have an explicit ‘‘delete point’’ edit, because points could be deleted by snapping them to an adjacent point. Thus, some of the 6737 ‘‘snap’’ edits really correspond to ‘‘delete’’ operations.

It is possible to check the validity of Constraints #1 and #2 without using a graphical interface to the data, though repairing failures non-graphically is tedious. In Appendix A we give code to do the validity checks.

4. Constructing the geometry

We now describe the process of constructing the geometrical information in a geographical database from a graph that satisfies Constraints #1 and #2. In what follows, we assume that one is given just the graph: the vertices and line segments (as pairs of vertices) of the map. This is rather less information than we began with in the WDBII data; starting with the segments joined into polylines means that some of what we are about to say is unnecessary. But since in our experience, geographical data comes in a variety of forms, we decided to present the following algorithms assuming the bare minumum of initial data.

The construction process involves these steps:

- find the connected components of the graph
- embed the graph in the plane
- construct the graph dual of each component
- construct polylines from individual segments
- find regions between the polylines

We describe each of these in turn.

4.1. Connected components

Two vertices are *connected* if one can be reached from the other by moving along edges of the graph. Maximal groups of such vertices, together with their incident edges, form the connected components of the graph. These presumably correspond to large land masses or islands in the map. Finding components can be done easily with a short recursive procedure that begins by visiting each vertex connected directly to the first vertex of the graph, then visiting each of the vertices connected to these and so on. Each vertex visited gets marked with a 1, say. When no more unmarked vertices can be reached, the next unmarked vertex in the graph is found and the procedure is repeated, using a mark of 2 for visited vertices. In this way, each vertex is visited and marked with a tag corresponding to the component of which it is a part. In our edited WDBII data, there were 2110 such components, of which most were islands.

4.2. Embedding the graph

The basic idea in finding the polygons of the map is to start at some vertex and walk around a polygon. This requires that whenever we reach a vertex of valence 3 or more, we must make a decision about which edge to use next. This decision can be made if we know how the edges incident at a vertex are ordered in a clockwise (or counterclockwise) direction. Figure 4 illustrates this.

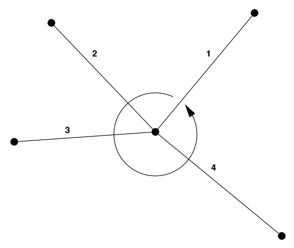


Figure 4. The embedding of a vertex of valence 4. Embedding simply means numbering the edges incident with the vertex in such a way that they trace around the vertex in order. It is irrelevant which edge is first; one consequence of this is that the only non-trivial computation is for vertices with valence at least 3.

4.3. Finding the duals

The original graph divides the plane up into a number of regions. The goal of the next step is to find the boundaries of those regions. The dual of the original graph is a graph with a vertex for each of the regions and an edge connecting two vertices (regions) whenever a segment of the original graph separates the two regions. Thus, its edges are in one-to-one correspondence with the edges of the original graph. The dual computation is done once for each connected component of the original graph and is accomplished by tracing out the component region by region, going around each face in counterclockwise order (say), by always making as sharp a left turn as possible at vertices with valence greater than 3 (hence the need for the previous step.) At the conclusion of this process, each component has split the plane into a number of regions, exactly one of which is unbounded—the ''ocean.''

Here's how the size of the duals came out with WDBII:

# faces	2	3	4	5	31	133
count	2100	4	2	2	1	1

This implies that 2100 of the 2110 components are islands with no internal political boundaries—just an inside face and an outside face. The component with 31 faces is the North/Central/South America land mass and the one with 133 faces is the Europe/Asia/Africa land mass.

4.4. Building the polylines

Next we find the polylines of the graph. By Constraint #3, the polylines can have no internal vertices of valence more than 2. We choose to use polylines of maximal length, viz, the endpoints always have valence at least three. Note that this is not satisfied in the case of an island consisting of a single polyline. Finding the polylines is easy—find a vertex of valence of at least 3 and trace outward from one of its incident edges until another vertex of valence at least 3 is reached. As we do so, we tag each segment with an indication of which polyline is being traced. Our world data had 3904 such polylines. This is 211 less than the original 4115 polylines from the raw data, a reflection of the fact that there were duplicates among the original polylines and that some polylines were joined and a few split in the editing process.

4.5. Finding the regions

The most interesting computational step is finding the regions. If there were only one connected component of the original graph, the regions would correspond to vertices of the dual graph. But with more than one connected component, an entire component can be contained within a face of another component. This situation is illustrated in Figure 5 where we have a graph with two components with component B entirely contained within face 2 of component A. The region between B and A is not a face of either component, and must get a single label, i.e., it must be discovered that face 0 of B intersects face 2 of A, and the intersection must be identified as a single region. If this is not done then the polyline between B2 and A2 will be recorded as having region B2 on one side and the ‘‘ocean’’ on the other (assuming that A0 is the ocean), which is wrong. The simplest case of this problem is that each component's unbounded face must be identified as part of the same region, otherwise there will be many names for the ‘‘ocean.’’

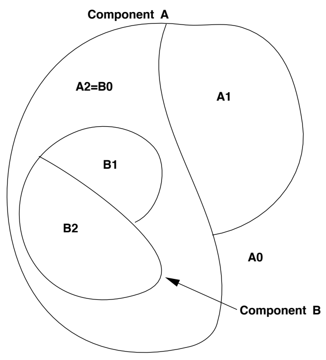


Figure 5. Finding regions. Two components, each with three faces, with the unbounded face numbered as face 0 in both cases. The region between the components is a face of neither, but rather the intersection of what would be face 2 of A and face 0 of B. The problem is to regions such situations and label them appropriately.

To find these regions we make a pass through the components. Each component is contained in some face of each other component (usually the unbounded face)—we find the smallest such containing face. The workhorse of this computation is the point-in-polygon algorithm: given a point and a polygon, determine if the point is inside or outside the polygon. The computation is greatly speeded up by using the polygon bounding boxes computed earlier for intersections. For the WDBII database there were 2285 regions, including the ‘‘ocean.’’

5. Database format

Now that all of the geometrical information is computed, we have only to save it on the files that make up a geographical database. Geographical databases for the `map()` facility in S can have two forms, an ASCII form and a binary one. The first is what one normally produces when creating a database, and is useful for shipping databases between computers. Software supplied with S allows conversion between the two formats. The ASCII format is very simple. We number the polylines, beginning with number 1, and we similarly number the regions, except that the unbounded region for the original graph is numbered 0. The polylines are written to a file whose name typically ends in `.line` with the format:

$left_1 \quad right_1$
$p_{11} \cdots p_{1n_1}$
`EOR`
$left_2 \quad right_2$
$p_{21} \cdots p_{2n_2}$
`EOR`
...

Here, $left_i$ and $right_i$ are the numbers of the regions to the left and right of the polyline as one ``travels'' on it from the starting vertex to the ending vertex; p_{ij} represents vertex j of polyline i and is given as a pair of coordinates, with longitude first (n_i is the number of vertices in polygon i). White space may be used freely, and the final literal `EOR` indicates the end of the polyline record.

The polygon file, typically named ending in `.gon`, has this format:

$l_{11} \cdots l_{1m_1}$
`EOR`
$l_{21} \cdots l_{2m_2}$
`EOR`
...

Here, each l_{ij} is either a polyline number or the negative of a polyline number. The absolute values of $l_{11} \cdots l_{1m_1}$ are the numbers of the polylines that must be traced, in order, to form the boundary of region 1. When the polyline number is positive, the vertices of the polyline should be traced in the order given in the polyline file; when negative, the polyline should be traced from its end to its beginning. As before, the end of a polygon record is indicated by the literal `EOR`.

Notice that it would be possible to detect a non-simply connected region (one with holes) using this scheme because normally, in going from one polyline to the next, the last vertex of one polyline is equal to the first of the next. If a break occurs, this indicates that another piece of the boundary is being traced out. Currently our software only outputs the outer boundary of each region.

6. Naming the regions

The final step in constructing a geographical database for S is to name each of the regions defined in the polygon file. More than likely this must be done by hand, though there are some possible ways to automate the process. For example, if we had a list of cities, together with their country, latitude and longitude, we could apply the point-in-polygon algorithm to determine which polygon each city was in and therefore identify the country that the polygon corresponded to. Unfortunately, this didn't work for WDBII because most of the polygons corresponded to small islands.

In the end we resorted to naming by hand. We had our interactive graphics program show the latitude and longitude as it zoomed in on the regions one by one, and we compared what we saw with an atlas. This was a tedious process, and in retrospect we should probably just have thrown out the smaller islands. In any case we were able to name all but 295 of the regions, the unnamed ones all being small islands. Most of these were in the following countries:

106	Maldives
35	Greenland
26	USSR
24	Chile
18	Bangladesh

At one point, being rather tired of poring over the atlas, we made a list of the remaining islands, together with their centroid and area and mailed this to the S-news mailing list, asking for help. Several people responded and contributed at least a hundred new names. Late in the project we also obtained a copy of the Relational World Data Bank II (RWDBII), an updated version of WDBII with many place names added. Some of these gave us new names, while others served as a check on the names we had already found.

There is a naming convention described in Becker and Wilks (1991) that turned out to be useful and important in the world map. The convention is that the hierarchy among regions is represented by names with several parts separated by colons. This is merely a convention—none of the mapping software knows about it. It is useful, though, because it is possible to choose a group of names by matching an initial substring of those names. In naming the regions of the world, we always used the form `country:name` so that each region was assigned to some country. Thus it is possible to plot all the parts of a country quite easily. This is not without problems, however, as illustrated in the examples of section 8.

7. Problems

All through the process we have been describing we always kept the data at the original resolution. The raw numbers were given to the nearest second, but empirically the distance between successive vertices was about a kilometer, or roughly 40 seconds of latitude. (The WDBII data came from a 1:2.5 million scale base map.) In any case, this meant that the final database was huge—the ASCII version of the polyline file occupied about 40 megabytes. This presented two problems. First, we didn't want to distribute S with such a large chunk of data, since the size of S itself was much less that this. Our solution was to make the data available on the statlib server at CMU, where anyone can retrieve it using anonymous ftp. Using some special purpose compression code we were able to squeeze the entire package into about 4 megabytes†.

The second problem with the large amount of data is that S can be unhappy dealing with such large datasets. In fact, when we tried simply: `map('world')`, S used up too much swap space and the operating system killed the process. To alleviate this problem, we created a new geographical database that had the same topological structure, and exactly the same polygon and name files. The polyline file was different, however, because each polyline in the polyline file was a *thinned* version of the original, consisting of fewer points but remaining close (in a certain sense) to the original polyline when drawn. Our thinning algorithm is described in Becker and Wilks (1991). The new database is called `world.thin` and is appropriate for maps that cover a large area of the world. It consists of about 35,000 points as compared with about 2,000,000 for the `world` version.

8. Examples

We now give several examples of maps made using the new database. At the top of each example is the set of S expressions that produced the map. First, a map of France in Figure 6. France itself comes out quite tiny due to the fact that the second argument to the call to `map()` matched all the names the begin with `France`, including 23 French islands around the world.

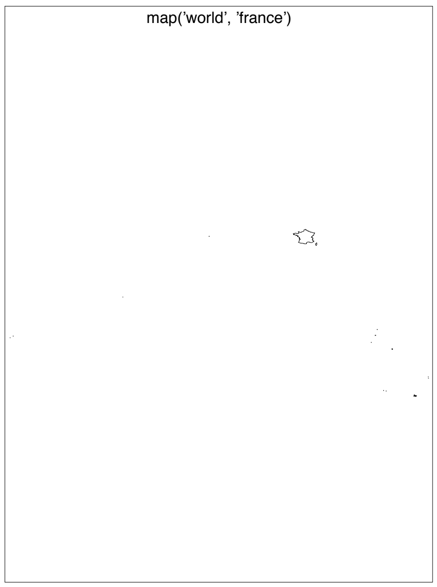


Figure 6. A map of France including all its possessions.

To fix this situation, we added a new argument, `exact=FALSE`, to the `map()` function. If `TRUE`, only exact matches are allowed (recall from Becker and Wilks, 1991, that all matches are anchored to the beginning of the names). Using this argument we may now produce a nice map of France itself, as shown in Figure 7.

†S is now exclusively distributed by StatSci and is the base for their S-Plus product. The map software and database organization in S-Plus is identical to what is described here.

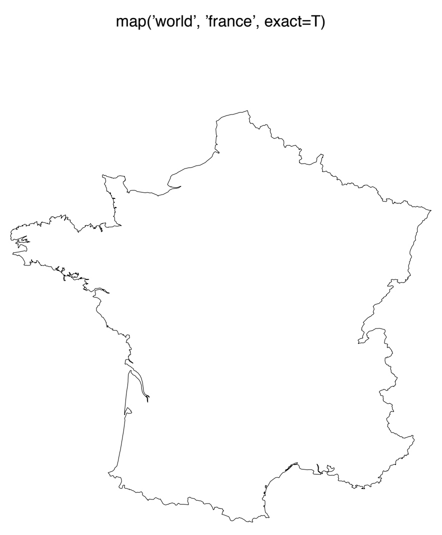


Figure 7. Another map of France, this time without all the French islands.

As a third example, we show a map of Europe. The names in the world database only have one extra level of implicit hierarchy, viz, the country of ownership. They do not include, for example, the continent the the country belongs to. Hence, to make a Europe map, we need explicitly to construct a vector of the names of the countries we wish to view:

```
europe <- c("france", "belgium", "netherlands", "germany",
     "denmark", "luxembourg", "spain", "portugal", "italy",
     "switzerland", "austria", "ireland", "northern ireland",
     "liechtenstein", "andorra", "great britain")
```

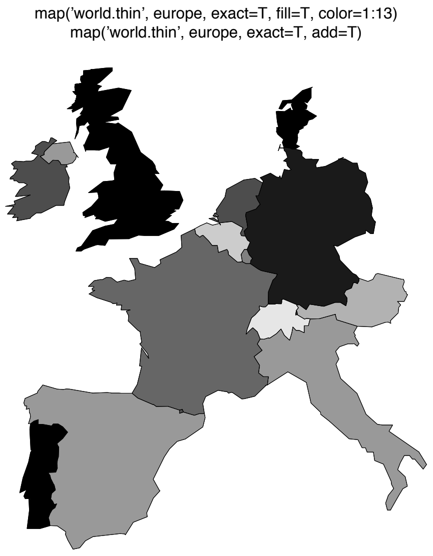
In Figure 8 we show a filled map of Europe, using random colors. Notice that we used the `world.thin` database, because otherwise some of the output polygons had too many sides for the PostScript printer to process.

As a final example, Figure 9 is a map of Antarctica, with the South Pole marked.

9. Further databases

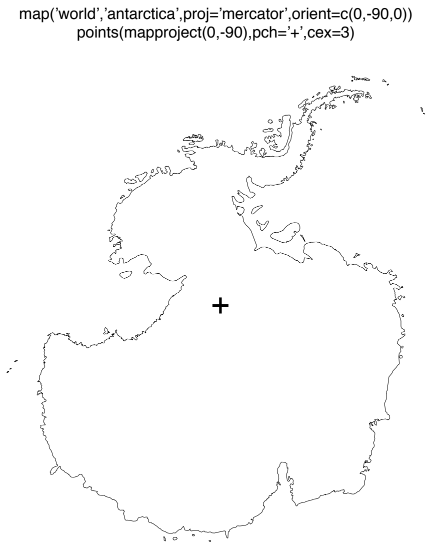
As time goes on, we hope to be able to add other geographical databases to S for use with the `map()` function. One relatively simple way to do this is through subsetting an existing database. In Appendix B we show a short shell program (mostly using `awk`) that will take a geographical database, together with a set of names from the names file, and construct a new database that has just those regions given by the names. This could be used to extract a Europe map, for instance.

Constructing databases from scratch, however, is hard, as illustrated by the process described in this report. Of the three main steps, data cleaning, geometry construction and region naming, only the second can be completely automated. In Appendix A we give a set of simple tools that can aid in all three steps.



```
map('world.thin', europe, exact=T, fill=T, color=1:13)
map('world.thin', europe, exact=T, add=T)
```

Figure 8. A map of Europe using random colors. A real use would match the colors to some data of interest. The second call to `map()` adds the outlines of the regions to give a crisper image.



```
map('world','antarctica',proj='mercator',orient=c(0,-90,0))
points(mapproject(0,-90),pch='+',cex=3)
```

Figure 9. A map of Antarctica, with a plus at the South Pole. Note the use of the `orient` argument, and the implicit use of the previous projection in the call to `mapproject`.

We have already constructed two more geographical databases for S using some of the tools described. One of these comes from data supplied by Mark Monmonier in the Department of Geography at the University of Syracuse. He developed what he calls a ‘‘visibility base map’’ of the USA, in which all states, even Rhode Island, are visible. The map is schematic, and consists of polygons that are each ‘‘islands.’’ Figure 10 shows this map. We also received data describing the counties of the UK from Barry Rowlingson in Lancaster, UK. This data was already quite clean, and required only two small fixes to make it correct. It is illustrated in Figure 11.

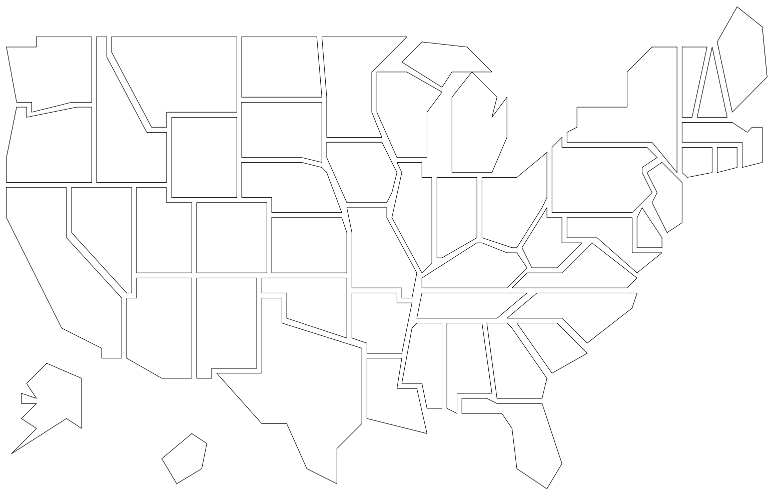


Figure 10. A ''visibility base map'' of the USA. This was produced with the S expression `map('state.vbm')`.

References

Becker, Richard A., and Allan R. Wilks, "Maps in S", AT&T Bell Laboratories Statistical Research Report No. 96, June 1991
(URL: `http://www.research.att.com/areas/stat/doc/93.2.ps`).

United States Department of Commerce, National Technical Information Service, WDB II General Users Guide, PB87-184818, World Data Bank II Set, PB87-184768.

Jack J. Dongarra and Eric Grosse, ''Distribution of Mathematical Software Via Electronic Mail'', *Comm. ACM* (1987) 30,403-407.

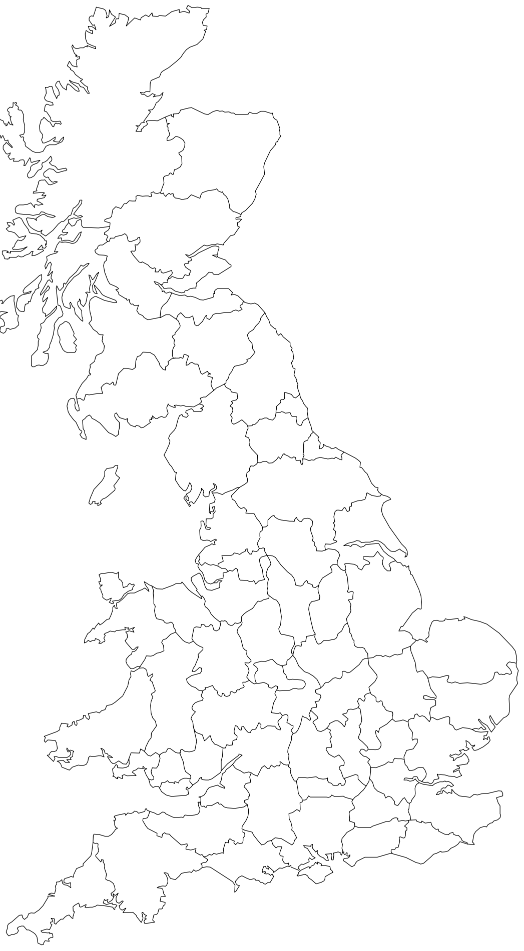


Figure 11. A county map of the United Kingdom. This was produced with the S command `map('uk')`.

Appendix A

In this Appendix we describe a simple set of tools that can be used to help construct a geographical database for S. All of these are freely available at `http://www.research.att.com/~rab/geo.html`.

We describe the process of building a geographical database as a sequence of steps, similar to the steps given in our earlier narrative.

Step 1: Produce a list of segments

Geographical data comes in many forms. The process we describe here assumes a minimal amount of information in the data: an ASCII file in which each line describes one line segment of the database. Each line should contain four numbers, the two coordinates of one endpoint of a line segment and then the two coordinates of the other endpoint, all separated by white space. This is basically the least amount of information needed to construct the database; in particular, it does not contain any information the original data might contain about connectivity (such as we saw in the polyline descriptions of the WDBII data) or regions (such as comes with the Census boundary data, in which each segment is given with the names of the regions to its left and right).

The mapping software in S can deal with two sorts of coordinate pairs; planar and spherical. In the planar case, the elements of a coordinate pair can have any values, but in the spherical case, each coordinate pair should represent a *(longitude,latitude)* position on the sphere; i.e., the first coordinate of the pair should be the longitude of a point in radians east of the prime meridian (so negative is west) and the second should be its latitude in radians north of the equator. The preferred format is spherical, for then map projections can be used. Planar data might result, for example, as the output of a digitization process. You need to decide at the outset what sort of database you are constructing, planar or spherical, and if the latter, you need to make sure that all the longitude coordinates are between $-\pi$ and π, and that the latitude coordinates are between $-\pi/2$ and $\pi/2$.

In addition, you need to decide how precise your data is. You should think of each x or y coordinate as a fixed-point real number with a certain fixed number of decimal digits after the decimal point. As a guide for spherical data, four digits after the decimal point gives an accuracy of about a kilometer in latitude. Also, one second of arc corresponds to about 5 millionths of a radian, so if your data is really that precise, six digits after the decimal point will suffice to represent it. We will refer to the number you pick as `prec`. Before proceeding to the next step, you should round all numbers in your data file to `prec` digits. We will assume that your ASCII file is named `segfile` for the purpose of this discussion. Here is a little shell file, called `round`, that will do the job:

```
#!/bin/sh
prec=$1; shift;;
(echo $prec; cat "$@") |
awk '
function round(z) {
     if(z > 0)
          return int(z*factor+0.5)/factor
     else
          return -int(-z*factor+0.5)/factor
}
NR == 1 {
     prec = $1
     factor = 10^prec
}
NR > 1 {
     printf "%.*f %.*f %.*f %.*f\n",
          prec, round($1),
          prec, round($2),
          prec, round($3),
          prec, round($4)
}'
```

If, for example, your value for `prec` were 4, you could say:

```
round 4 segfile > j
mv j segfile
```

Step 2: Remove duplicate and degenerate segments

Here is a little shell program called `gdclean` that reads its input either from a named file or from standard input and prints on standard output a cleaned-up version of the segments:

```
#!/bin/sh
sort "$@" |
uniq |
awk '$1 != $3 || $2 != $4'
```

This will do three things: it will print all coordinates out in a standard form so that strings representing equal numerical values will themselves be equal as strings; it will elide any duplicate copies of segments that appear more than once; and it will elide any degenerate segments, for which the two endpoints are the same. In what follows we will assume that `segfile` has been replaced by the output of `gdclean segfile`:

```
gdclean segfile > j
mv j segfile
```

Step 3: Check for dangles (Constraint #1)

No vertex should appear with valence 1, according to Constraint #1. The following short shell program, `dangle`, checks this condition, again reading its input either from a named file or from its standard input:

```
#!/bin/sh
cat "$@" |
awk '{
     v = sprintf("%g %g", $1, $2); count[v]++; line[v] = NR
     v = sprintf("%g %g", $3, $4); count[v]++; line[v] = NR
}
END {
     for(v in count)
          if(count[v] == 1)
               print line[v], v
}'
```

The output from `dangle` is a sequence of lines, each of which describes a violation of Constraint #1. The format of an output line is three numbers giving the line number of a segment in the input file, and then the coordinates of one of the vertices of that segment that appeared only once in the input, and was therefore dangling. If both endpoints of the segment are dangling (an isolated segment), there will be two output lines, one for each vertex.

If there are dangles, they must be repaired before proceeding with the next step. We offer no tools for this activity, though the main body of this paper gives some ideas about how to proceed.

```
dangle segfile > dangles
# now deal with each line of dangles and fix segfile
```

Step 4: Check for illegal intersections (Constraint #2)

The second Constraint says that segments may only intersect in a common endpoint. The C program `intersect.c` (not reproduced here so as to save space) will check this constraint. It reads the named file or its standard input for a list of segments and produces as output a sequence of lines describing illegal intersections. Each such line consists of a pair of segment numbers (line numbers in the input file) of segments that intersect illegally, together with a verbal description of the type of intersection.

The `intersect` program expects its arguments to be integers; this allows the segment intersection code to work precisely, with no need for a fuzz factor (it will actually work with arbitrary floating-point input, but it may flag spurious intersections or miss others). Since the `segfile` was rewritten in Step 1 to have a constant number `prec` decimal digits after the decimal point for each number, it can easily be turned into integers by merely eliding the decimal points, as in

```
sed 's/.//g' segfile | intersect
```

It is important to note that this trick (of eliding the decimal point) will only work if the numbers in segfile all have the same number of digits after the decimal point. This will be true if the `round` shell file was used, as recommended in Step 1.

The intersection code will always give correct answers if the range of x-values and the range of y-values are not too large. The definition of "too large" depends on the floating-point hardware, but with IEEE arithmetic, it is the 25th power of 2, or 33554432. In other words, as long as the difference between the smallest and largest x-values in the input to `intersect` is less that 33554432, and the same for the y-values, `intersect` will operate correctly. This number would allow a value of 7 for `prec` for spherical data (since it is larger by a tad than the number of ten-millionths in radians), which would allow segments to span just 4 meters on the surface of the earth; this should be sufficient accuracy for any spherical database.

Bad intersetions must be repaired before proceeding, and we offer no tools to aid in this process.

Step 5: Build the database files

At this point it is assumed that `segfile` contains a set of segments with no duplicates, no degenerate segments, no dangles and no illegal intersections. Note that it may be necessary to do Steps 2, 3 and 4 several times until this is true.

The next step is to create ASCII files for the geographical database in the format detailed earlier in Section 5. The C program `gdbuild.c` can be used for this purpose. It expects to read on its standard input the clean data, but in a new format. The new format looks like this:

```
nv
vertex 0
 ...
vertex nv-1
ne
edge 0
 ...
edge ne-1
```

The first line gives the number of unique vertices among the segments and then each of those vertices is listed once. Next the number of edges (segments) is given, followed by the segments themselves. The difference between an edge and a segment is that the edge is a pair of vertex numbers (according to the scheme established in the first part of the input), not four coordinates. Here is the shell program `gdgraph` to change the `segfile` into the new format:

```
#!/bin/sh
cat "$@" |
awk '
function vertex(x,y) {
     s = x " " y
     n = v[s]
     if(n == "") {
          n = v[s] = nv
          c[nv] = s
          nv++
     }
     return n
}
BEGIN {
     nv = ne = 0
}
{
     va[ne] = vertex($1,$2)
     vb[ne] = vertex($3,$4)
     ne++
}
END {
     print nv
     for(i = 0; i < nv; i++)
          print c[i]
     print ne
     for(i = 0; i < ne; i++)
          print va[i], vb[i]
}
```

The `gdbuild` program reads data in the new format on its standard input. It takes a single argument, as in:

```
gdgraph < segfile | gdbuild foo
```

where `foo` is the name of the database being created. Its output is two files, `foo.line` and `foo.gon`, containing the polyline and polygon information in ASCII.

Step 6: Make the binary database files

The binary versions of the polyline and polygon files are a simple transformation of the ASCII versions, and are produced with the `Lmake` and `Gmake` programs normally distributed with S or Splus. For completeness, their source code, `Lmake.c` and `Gmake.c` is included in the archive mentioned earlier.

The syntax of `Lmake` is this:

```
Lmake precision type output-type input-file output-file
```

`Lmake` can convert from ASCII to binary or from binary to ASCII; which one it does is controlled by the `output-type` argument. If `output-type` is `b` a binary file is produced (`input-file` names an ASCII file) and if `output-type` is `a` an ASCII file is produced (`input-file` names a binary file).

The `precision` argument may be ignored, as it refers to the output precision (actually the number of places after the decimal point) of the printed numbers when an ASCII file is being produced; you can set it to anything when producing the binary file. (You would use this argument when producing an ASCII version of the database to ship to someone else, for example. In that case, a good value for `precision` would be the value you chose previously for `prec`.)

The database format allows the polyline file to record coordinates on the plane or on the sphere; it is only the latter type of data that will work with map projections. If your input data is spherical, the first coordinate of each coordinate pair should be the longitude of a point in degrees east of the prime meridian (so negative is west) and the second should be its latitude in degrees north of the equator. You indicate this by calling `Lmake` with a value of `s` for the `type` argument. Otherwise, use `p`. The latter might be the case if your data were the raw output from a digitization process, for example.

`Gmake` is a little simpler than `Lmake`. Its syntax is:

```
  Gmake output-type input-file output-file output-line-file
```

where the `output-type` is as before. The reason that `Gmake` requires the name of the output file from the call to `Lmake` is that it will compute a bounding box for each polygon by looking at the bounding boxes of its constituent polylines, and this information is in the polyline file.

Since the `map()` function expects binary polyline files to have an extension `.L` and binary polygon files to have an extension `.G`, a typical pair of calls to these programs might look like this:

```
    Lmake 0 s b foo.line foo.L
    Gmake b foo.gon foo.G foo.L
```

Step 7: Create a dummy name file

This is an easy one; here is the shell file `protoname`:

```
    #!/bin/sh
    awk '$0 == "EOR" {print ++n}' |
    sed 's/.*/polygon &@&/' |
    tr '@' '\011'
```

Usage is:

```
    protoname < foo.gon > foo.N
```

This will give polygon 20, for example, the unimaginative name `polygon 20`. To do better than this we proceed to:

Step 8: Name the regions

There is a pair of S functions to aid in naming the regions, now that the geometric part of the database is built. First the function `pip()` (for Point In Polygon) takes a vector of `x` values, a parallel vector of `y` values and the name of a database (`'foo'` in our example) and returns the polygon numbers of the polygons containing the given points. The returned polygon number for a point is 0 if no region contains the point and it is –1 if the point was so close to the boundary of a region that it was hard to tell which region it was in. `Pip()` is just a call to a C routine that resides in the C file `mapget.c`, and this file also contains some of the other routines used in the mapping software. To use it, you will have to load the code in `mapget.c` with your S executable, either statically or dynamically.

Incidentally, if you set the shell environment variable `S_MAP_DATA_DIR` to point to the directory in which you are creating this database, the code in `mapget.c` will look there instead of its usual place, which is `$SHOME/library/maps/data`.

The other function, `gdname()`, uses `pip()` to name the regions interactively. It is a simple loop that draws the full map, prompts for graphical input, draws a blowup of the polygon selected and prompts for a name for that region. Its return value is a list giving the locations pointed at so far, together with the names that have been associated with their polygons. This list may be given as the second argument to `gdname()` to initialize that data in a subsequent naming session. So you can do something like this:

```
# interactively name some things until you need a break; when
# you are prompted to point and click at a region, just give
# the device's standard indication of end-of-graphical-input
> z <- gdname("foo")
# start from where you left off ...
> z <- gdname("foo", z)
```

The `gdname()` function is quite simple, and could be made fancier in a number of ways. But it should serve the purpose as it stands. When you have named all the regions (and saved the result in, say, `z`), you can now turn this into a more meaningful names file:

```
cat(paste(z$name,seq(z$name),sep="\t"),sep="\n",file="foo.N")
```

That's it! You now have the three files `foo.L`, `foo.G`, and `foo.N` that completely describe the `foo` database. You can either install these in the standard place, or set your `S_MAP_DATA_DIR` variable when you want to use `foo`.

Here is a brief recap of the commands described above. First, in the shell:

```
# produce segfile, a list of segments, one per line
round 4 segfile | gdclean > j
mv j segfile
dangle segfile > dangles
# deal with each line of dangles and fix segfile
sed 's/.//g' segfile | intersect > intersections
# deal with each line of intersections and fix segfile,
# then make sure that there are still no dangles
gdgraph < segfile | gdbuild foo
Lmake 0 s b foo.line foo.L
Gmake b foo.gon foo.G foo.L
protoname < foo.gon > foo.N
# set environment variable S_MAP_DATA_DIR to point to current directory
# load mapget.c code into S
```

and then in S:

```
z <- gdname("foo")
cat(paste(z$name,seq(z$name),sep="\t"),sep="\n",file="foo.N")
```

Appendix B

Also included with the code mentioned in Appendix A is a shell script, called `sub-gdb`, to extract part of a geographical database and use the data to create a new one. The two input arguments are the names of the old and new databases. The numbers of the regions to be extracted are read on the standard input. The result of running the script is three files: a `.line` file, a `.gon` file and a `.N` file. `Lmake` and `Gmake` must then be used to create the binary `.L`and `.G` files.

For example, suppose we wish to make a database for just Europe using the `europe` variable that we created earlier as a list of the coutries that should be in the database. The ASCII files can be created with

```
echo "cat(mapname('world',europe),sep='\n')" | S |
     subgdb $SHOME/library/maps/data/world europe
```

(The `mapname()` function is used by `map()` to turn region names into region numbers). This creates the files `europe.line` and `europe.gon` in the current directory, and the binary versions can be produced with:

```
$SHOME/library/maps/Lmake 4 s b europe.line europe.L
$SHOME/library/maps/Gmake b europe.gon europe.G europe.L
```

Afterward, `map("europe")` will access the new database. Don't forget to set the `S_MAP_DATA_DIR` environment variable if the new data does not reside in the standard place for map data (as described in Appendix A.)