Generating Google™ maps

A preprint version of a “Mathematical graphics” column from

Mathematica in Education and Research.

Mark McClure

mcmcclur@unca.edu

Department of Mathematics
University of North Carolina at Asheville
Asheville, NC 28804

Abstract

The Google Maps™ mapping service provides a relatively easy way to create interactive maps. To do so efficiently however, some interesting mathematical problems must be solved. *Mathematica*'s tools for reading, manipulating, and writing data make it a good choice to generate a Google map presenting complicated data.

▪ Mathematica Initializations

1. Introduction

Google Maps™ mapping service provides an exciting and relatively simple new way to create interactive maps called *Google* maps. While *relatively* simple, creating a *Google* map to display interesting geographic data can be complicated, depending on the nature of the data. The data is likely stored in a file which must be read, analyzed, and translated to javascript, the language of the Google Maps™ API. *Mathematica* is an outstanding tool for each step along the way. First, *Mathematica*’s XML capabilities make it easy to read several key types of files used to store geographic information. Second, the data might need to be simplified to be displayed efficiently; some interesting mathematics arises at this step. Finally, *Mathematica*’s string manipulation and file processing tools make it easy to write out the HTML and javascript representing the map. Description of this process seems like a natural topic for an issue focusing on the interface between *Mathematica* and other software.

Many readers are surely familiar with the main Google Maps™ website [1]. Using this site is quite easy and intuitive for most users and this article necessarily assumes a basic familiarity with that interface. It’s easy to set up a URL pointing to a *Google* map of a specific place. For example, the following URL points to a *Google* map of Asheville, North Carolina: http://maps.google.com/maps?q=asheville+nc.

It is also possible to set up a *Google* map residing in your own webspace using the Google Maps™ API. You can add clickable markers, complicated paths, and just about any tool you can program - in javascript. Since the Google Maps™ API is implemented in javascript and the map itself is displayed in a webpage, some familiarity with javascript and HTML would be helpful to the reader of this paper. It is not, however, mandatory. Indeed, a major point of this issue's column is to provide a simple tool for *Mathematica* users to create their own *Google* maps, without delving too deeply into javascript.

One logical use of the Google Maps™ mapping service is to display geographic data already stored in a file. Geographic data can be very complicated and the Google Maps™ API, being based in javascript, has difficulty handling large data. It is necessary to cluster large collections of markers and simplify complicated path data. *Mathematica* has built in tools to deal with the first problem and is an excellent programming tool to deal with the second.

A basic map

Before generating maps with complicated data, we should introduce the basics of the Google Maps™ API. Much more complete information is presented at the Google Maps™ API reference [2]. Two main pieces of code are required to define and display a *Google* map: an HTML file for the webpage and a javascript file that defines the map. The HTML code below defines a minimal webpage to hold a *Google* map. This is exactly the code contained in the BasicGoogleMap.html file in the SupplementaryFiles folder distributed with this column.

```
<html>
<head>
<title>A very basic Google map</title>
<script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key="
 type="text/javascript"></script>
<script src="BasicGoogleMap.js"
 type="text/javascript"></script>
</head>
<body onload="load()" onunload="GUnload()">
<h1>A very basic Google map</h1>
<div id="map" style="width: 500px; height: 300px"></div>
</body>
</html>
```

Code segment 1: HTML for a basic Google map.

This code has several important elements that are immediately relevant to the *Google* map. Inside the head are pointers to two javascripts. The first points to the Google Maps™ API file which defines the classes and functions you can use to make a *Google* map. The second points to a file, typically written by the programer, that uses the API to define the *Google* map in the web page. Inside the body element is a div element which holds the map. The style property of the div sets the size of the map and the id property names the map for reference by the javascript. Finally, the onload property of the body element calls the load method, which is defined in BasicGoogleMap.js; it is the load method which starts the action.

Here are the contents of the BasicGoogleMap.js file.

```
function load() {
 var map = new GMap2(document.getElementById("map"));
 map.setCenter(new GLatLng(35.6173, -82.5657), 16);
 map.addControl(new GSmallMapControl());
 map.addControl(new GMapTypeControl());
}
```

Code segment 2: Javascript for a basic Google map.

This defines the function load referred to in the HTML file. The first step of load initializes the map, the second sets the location and zoom level, and then a couple of controls are added. If you view the HTML file in a web browser, you should see the following map which is defined by the javascript.

```
Show[Import["BasicGoogleMap.tiff"],
 ImageSize → $ImageSize]
```

Note the empty key parameter at the end of the longest line in the HTML file. In order to display the map on a public webpage, you must sign up for Google Maps™ key at http://www.google.com/apis/maps/signup.html. This page contains a script which will generate a long and complicated string which is your Google Maps™ key. This string is tied to your webspace and is the proper value of the key parameter. For development purposes, it is not necessary to use a key to display a map off of your harddrive. After the addition of the key, the HTML file is as complicated as is necessary to create a *Google* map. We'll generate some much more interesting javascript to add long paths and large collections of markers, however.

■ A V6 goodie

It's easy to modify the code to display a map of your favorite city; simply replace the latitude and longitude (contained in the `GLatLng` function) by the latitude and longitude of the city you want and adjust the zoom level. The new V6 CityData command offers simple access to this type of information. For example the following command (executed in V6) yields `{35.5799,-82.5558}`, the latitude and longitude of Asheville, NC.

```
CityData[{"Asheville", "NorthCarolina", "UnitedStates"},
 "Coordinates"]
```

GPXToGoogleMap

Provided with this column is a package called **`GPXToGoogleMap`**, which automates the process of producing a *Google* map to display information stored in a file. This program has been under development nearly since the inception of the Google Maps™ API in early 2005. The current version is compatible with *Mathematica* versions 5.1, 5.2, and 6.0. The latest version of the program can be downloaded from the **`GPXToGoogleMap`** webpage [3], which contains much more extensive documentation.

There are many different filetypes used to store geographic information. **`GPXToGoogleMap`** reads GPX, a filetype for the storage of data obtained from a Geographical Positioning System (GPS) unit. The advantages of using GPX for this purpose include: its widespread use, the fact that each GPX element corresponds to a *Google* map object in an obvious way, and its implementation in XML allowing use of *Mathematica*'s XML capabilities. (Another natural choice of filetype to translate might be KML, *Google*'s proprietary, XML based filetype used with the Google Earth™ mapping service. KML is currently undergoing rapid development, however. Thus, the primary advantage of GPX over KML is its stability.)

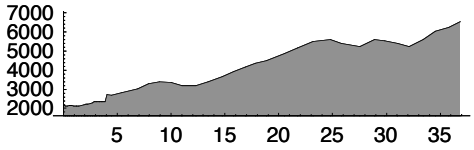
There are three major types of elements which may be described in a GPX document: tracks, waypoints, and routes. A track is simply a sequence of points defined by latitude, longitude pairs typically used to represent a path graphically on a map.

Other data, such as elevation, may or may not be present as well. If elevation data is present for all of the tracks, then **GPXToGoogleMap** will also generate elevation profiles for these tracks. A waypoint is defined by a latitude, longitude pair and typically has more data associated with it, such as a name and description. A route is a sequence of route points from one location to another. The route points usually have directional information associated with them, so a route is a good way to represent driving directions. Waypoints and routepoints are represented using markers on the *Google* map.

There are some interesting mathematical challenges that arise in the attempt to translate a GPX file to a *Google* map. It is not unusual for a GPX file to contain an extremely large number of waypoints and very complicated tracks. Javascript, while a natural choice to add dynamic enhancement to webpages, is not a particularly fast language. Thus, it is important that we represent our *Google* map as efficiently as possible. The Google Maps™ API offers several tools for this purpose, but we must still algorithmically translate the GPX file into a form that these tools understand. It is in this regard that **GPXToGoogleMap** stands out from the many other comparable tools available on the web. A GPX file with a few thousands waypoints and several tracks consisting of tens of thousands of points each can be handled fairly easily.

Use of **GPXToGoogleMap** is easy. Simply load the package and execute the command. The following command generates a *Google* map from the file MtMitchell.gpx in the SupplementaryFiles folder distributed with this paper.

```
Needs["GPXToGoogleMap`"];
GPXToGoogleMap["MtMitchell.gpx",
 DisplayFunction → $DisplayFunction]
```



GPXToGoogleMap accepts a number of options. In this example, we have used the option **DisplayFunction→ $DisplayFunction** to show the elevation profile, which is usually embedded in the webpage but not displayed. After executing this command, three files should appear on your harddrive: MtMitchell.html, MtMitchell.js, and MtMitchell-EC1.gif. The last of these is the elevation chart of the GPX track. If you open MtMitchell.html, you should see the following *Google* map.

```
Show[Import["MtMitchell.tiff"],
 ImageSize → $ImageSize]
```

The obvious difference between this map and the simpler map from the last section is the presence of a path, indicating the GPX track, and two markers indicating the beginning and ending of the path. The code to add these is, perhaps, a bit more mysterious than the simple code from before, particularly the code defining the path:

```
var ePolyline = new GPolyline.fromEncoded({
 color: "#0000ff",
 weight: 4,
 opacity: 0.8,
 points: "LongStrangeString1",
 levels: "LongStrangeString2",
 zoomFactor: 2,
 numLevels: 18
});
map.addOverlay(ePolyline);
```

Code segment 3: Javascript for an encoded polyline

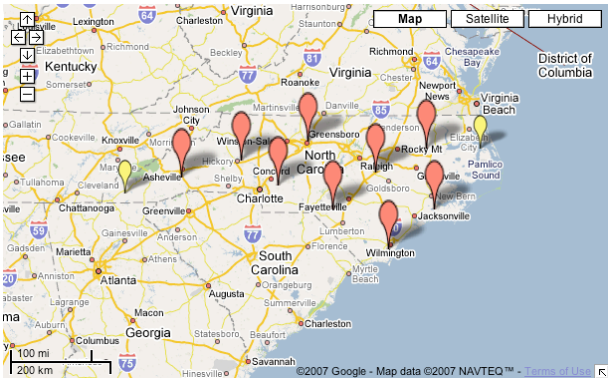
Code of this type defines a GPolyline object. The color, weight and opacity lines are simple style directives and the addOverlay function simply adds the path to the map. The strings which encode the points and levels, together with the zoomFactor and numLevels, are the trickiests aspect of this code. The levels string will be discussed in the next section. Of particular interest, though, is the fact that this path with over 700 points displays very quickly. Paths with several thousand points can render quite easily using the encoding technique.

It's also possible to generate a *Google* map which displays a few thousand markers. However, it is important that no more than a hundred or so be displayed at any given time. **GPXToGoogleMap** accomplishes this by clustering the markers. For example, the file NCCities.gpx contains waypoints for 650 different cities in North Carolina. We can still apply **GPXToGoogleMap**.

```
GPXToGoogleMap["NCCities.gpx"]
```

If you now view the *Google* map generated by this command in NCCities.html, you should see the following.

```
Show[Import["NCCities.tiff"],
 ImageSize → $ImageSize]
```



Each large marker represents a cluster of more than 20 cities, while the smaller markers on either side represent a cluster of between 2 and 20 cities. As we zoom in, the clusters split into smaller clusters as appropriate. If we zoom in to the Asheville area, for example, we see the following.

```
Show[Import["AshevilleAreaCities.tiff"],
 ImageSize → $ImageSize]
```

The new, smallest markers represent single cities. You can click on those markers to get the population and elevation of the city.

GPXToGoogleMap has a number of options, although their use requires a bit more knowledge of the Google Maps™ API. We refer the interested reader to the **GPXToGoogleMap** webpage [3].

Some mathematical details

The **GPXToGoogleMap** package is nearly 1000 lines of *Mathematica* code. Much of this is concerned with string manipulation and file input/output operations. It's actually quite a bit shorter than it might be, thanks to *Mathematica*'s XML capabilities and the fact that GPX is an XML extension. Much of it is useful, but not particularly exciting. There are, however, some interesting mathematical steps related to the clustering of the markers and encoding of the paths. We briefly discuss the ideas here and refer the reader to the **GPXToGoogleMap** documentation [3] for more details.

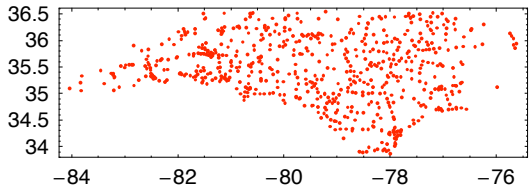
■ Clustering markers

As mentioned earlier, the performance of a *Google* map will start to degrade when more than about 100 or so markers are displayed at once. A class defined in the Google Maps™ API called the `GMarkerManager` class provides a workaround. The programmer can define marker icons that correspond to groups of markers and specify when they turn on and off using the `GMarkerManager`. It is still up to the programmer to cluster the markers appropriately at various levels. The **GPXToGoogleMap** package accomplishes this using the **Agglomerate** function in the **Statistics`ClusterAnalysis`** package.

```
If[$VersionNumber < 6.0,
  Needs["Statistics`ClusterAnalysis`"],
  Needs["HierarchicalClustering`"]];
```

To illustrate the process, we can **Import** the locations of the 650 cities from NCCities.gpx using the XML structure of the file.

```
dataXML = Import["NCCities.gpx", "XML"];
wpts = ToExpression[({"lon", "lat"} /. #1[[2]] & ) /@
    Cases[dataXML, XMLElement["wpt", __], Infinity]];
ListPlot[wpts, AspectRatio → Automatic,
 PlotJoined → False]
```



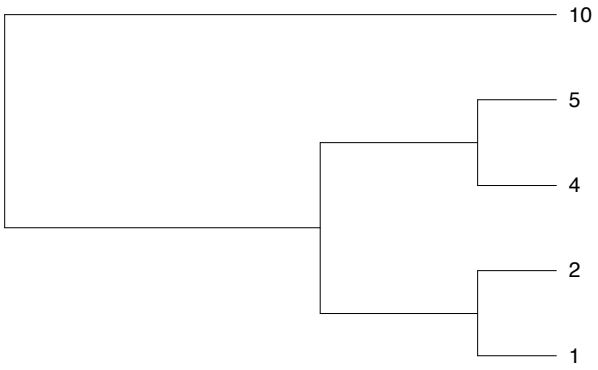
Given a particular distance tolerance, we wish to cluster the points so that each cluster contains a collection of points whose distance from one another is within that tolerance. Furthermore, we want to do so hierarchically; i.e. as we zoom in and decrease the error tolerance, each cluster further decomposes into sub-clusters. Creation of such a hierarchy is exactly the purpose of the **Agglomerate** command. **Agglomerate** produces a nested collection of **Cluster** objects. For example,

```
hierarchy = Agglomerate[{1, 2, 4, 5, 10},
  DistanceFunction → EuclideanDistance,
  Linkage → Average]
```

```
Cluster[Cluster[Cluster[1, 2, 1, 1, 1], Cluster[4, 5, 1, 1, 1], 3, 2, 2], 10, 7, 4, 1]
```

The format of a **Cluster** is **Cluster[cl1,cl2,dist,n1,n2]**., where **cl1** and **cl2** are sub-clusters, **dist** is the distance between the sub-clusters and **n1** and **n2** are the number of elements in the sub-clusters. The **DendrogramPlot** command makes it easy to visualize this hierarchy.

```
DendrogramPlot[hierarchy,
 Orientation → Left, LeafLabels → (# &)]
```



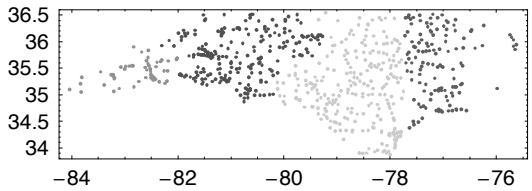
Now we apply **Agglomerate** to the city locations and **segregate** the clusters according to distance.

```
heirarchy = Agglomerate[wpts,
   DistanceFunction → EuclideanDistance,
   Linkage → "Average"];
segregate[Cluster[cl1_, cl2_, d_, _, _], tol_] :=
  MyClusters[cl1, cl2] /; d > tol;
segregate[mine_MyClusters, tol_] := segregate[#, tol] & /@ mine;
segregate[x_, _] := x;
cf[cl_Cluster] := ClusterFlatten[cl];
cf[x_] := {x};
clusters = cf /@ List @@ Flatten[FixedPoint[segregate[#, 1.5] &,
      MyClusters[heirarchy]]];
clusters // Length
```

```
4
```

This has yielded 4 clusters, which may be visualized used the following **ClusterPlot** command.

```
ClusterPlot[lists_] := Show[MapIndexed[
    ListPlot[#, PlotJoined → False,
      AspectRatio → Automatic,
      PlotStyle → GrayLevel[Mod[#2[[1]] / Length[lists], 0.7, 0.2]],
      DisplayFunction → Identity] &, lists],
   DisplayFunction → $DisplayFunction];
ClusterPlot[clusters]
```



If we were to generate a *Google* map showing these cities, then each of these clusters would yield a single marker at this level of resolution. As we zoom in, the process would be repeated using a smaller distance tolerance.

■ Polyline simplification and encoding

Polyline simplification is the process of removing points from a path while minimizing the distortion of the shape of the path. There is a well known algorithm for this purpose, called the Douglas-Peucker algorithm, that naturally extends to the encoding process offered by the Google Maps™ API. Most of the details of the encoding process need not concern us here; we refer the interested reader to Google Maps™ API documentation [2] and the author's webpage on encoding polylines [4]. The Douglas-Peucker algorithm is an interesting and important algorithm, however, and it is a short step from there to indicate the basic idea of the polyline encoding process. The original reference for the Douglas Peucker algorithm is [5].

The idea behind the Douglas-Peucker algorithm is as follows. We start with a zeroth level approximation to the path consisting of just the endpoints. Call these endpoints A and B. We then prescribe a small error tolerance - distances smaller than this error tolerance will essentially be ignored. Now, we scan the list of points in the path and find the one farthest away from the line segment [A,B]; call it C. If the distance from C to [A,B] is less than the error tolerance, then the process terminates and the polyline approximation will be a single line segment. Otherwise, label the point C with it's distance to [A,B] and call the procedure recursively on the portion of the line between A and C and the portion of the line between C and B. This procedure will eventually terminate, at which time unlabeled points will be discarded. In the classic Douglas-Peucker algorithm, the approximation consists of precisely the labeled points. In the polyline encoding algorithm, the labels are translated to characters indicating the "significance" of the point, as described shortly.

This algorithm is implemented by the **DPSimplify** command defined below. Each segment of the path is stored in a **seg**; the function **testSeg**, tests a segment to see if it needs to be split and performs the split if necessary. **DPSimplify** then repeatedly calls **testSeg** until all segments satisfy the requirements of the algorithm.

```
dist[q : {x_, y_}, {p1 : {x1_, y1_}, p2 : {x2_, y2_}}] :=
  With[{u = (q - p1).(p2 - p1) / (p2 - p1).(p2 - p1)},
   Which[u ≤ 0, Norm[q - p1],
    u ≥ 1, Norm[q - p2],
    True, Norm[q - (p1 + u (p2 - p1))]]];
testSeg[seg[points_List], tol_] :=
  Module[{dists, max, pos}, dists = dist[#, {points[[1]], points[[-1]]}] & /@
      points[[Range[2, Length[points] - 1]]];
    max = Max[dists];
    If[max > tol,
     pos = Position[dists, max][[1, 1]] + 1;
     {seg[points[[Range[1, pos]]]],
      seg[points[[Range[pos, Length[points]]]]]},
     seg[points, done]]] /; Length[points] > 2;
testSeg[seg[points_List], tol_] := seg[points, done];
testSeg[seg[points_List, done], tol_] := seg[points, done];
DPSimplify[points_List, tol_Real] := Append[First /@ First /@ Flatten[{seg[points]} //.
       s_seg :→ testSeg[s, tol]], Last[points]];
```

We can illustrate the algorithm using the first 360 points of the Mt. Mitchell path.

```
dataXML = Import["MtMitchell.gpx", "XML"];
track = Take[ToExpression[Cases[dataXML,
      XMLElement["trkpt", __], Infinity] /.
     XMLElement["trkpt", latLonRules_, ___] :→
      ({"lon", "lat"} /. latLonRules)], 360];
Show[Graphics[wholePath =
   {Line[track], PointSize[0.007], Point /@ track},
  AspectRatio → Automatic]]
```

This path exhibits characteristics common to GPS data. Reception is very good in some spots but not so in others. In this example, the better reception occurs on the straighter portions of the path. This is exactly the situation the Douglas-Peucker algorithm is built for. The following picture illustrates the result for several choices of the tolerance.

```
Show[GraphicsArray[{ListPlot[DPSimplify[track, #],
      Epilog → wholePath, Frame → False,
      DisplayFunction → Identity,
      AspectRatio → Automatic]} & /@
   {0.01, 0.005, 0.0007}]]
```

Note that the last approximation consists of only 35 points but is quite good at this level of resolution.

Finally, we briefly indicate how to translate this simplification process into an encoding. The polyline code shown in code segment 3 contains a "levels" parameter. This parameter is a string and its number of characters is equal to the number of points in the path. The character in a particular position in the string indicates the zoom level at which the corresponding point should be turned on. The efficiency in the process arises from the fact that not every point need be displayed at every zoom level. It is an easy process to turn the distances computed during the Douglas-Peucker algorithm into the appropriate characters for the encoding. See the author's webpage on polyline encoding [4] for more details.

■ Another V6 Goodie

There is an interactive Douglas-Peucker simplifier at the Wolfram Demonstrations site [6].

References

[1] Google Maps™ mapping service: http://maps.google.com/.

[2] Google Maps™ API Documentation: http://www.google.com/apis/maps/documentation/.

[3] Mark McClure, GPXToGoogleMap: http://facstaff.unca.edu/mcmcclur/GoogleMaps/GPXToGoogleMap/.

[4] Mark McClure, Encoding polylines: http://facstaff.unca.edu/mcmcclur/GoogleMaps/EncodePolyline/.

[5] D. Douglas & T. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature", *The Canadian Cartographer* **10** (1973) 112-122.

[6] Mark McClure, Polyline simplification (a Wolfram demonstration): http://demonstrations.wolfram.com/PolylineSimplification/.