The Stata Journal

The *Stata Journal* publishes reviewed papers together with shorter notes or comments, regular columns, book reviews, and other material of interest to Stata users. Examples of the types of papers include 1) expository papers that link the use of Stata commands or programs to associated principles, such as those that will serve as tutorials for users first encountering a new field of statistics or a major new technique; 2) papers that go "beyond the Stata manual" in explaining key features or uses of Stata that are of interest to intermediate or advanced users of Stata; 3) papers that discuss new commands or Stata programs of interest either to a wide spectrum of users (e.g., in data management or graphics) or to some large segment of Stata users (e.g., in survey statistics, survival analysis, panel analysis, or limited dependent variable modeling); 4) papers analyzing the statistical properties of new or existing estimators and tests in Stata; 5) papers that could be of interest or usefulness to researchers, especially in fields that are of practical importance but are not often included in texts or other journals, such as the use of Stata in managing datasets, especially large datasets, with advice from hard-won experience; and 6) papers of interest to those who teach, including Stata with topics such as extended examples of techniques and interpretation of results, simulations of statistical concepts, and overviews of subject areas.

The *Stata Journal* is indexed and abstracted by *CompuMath Citation Index*, *Current Contents/Social and Behavioral Sciences*, *RePEc: Research Papers in Economics*, *Science Citation Index Expanded* (also known as *SciSearch*), *Scopus*, and *Social Sciences Citation Index*.

For more information on the *Stata Journal*, including information for authors, see the webpage

http://www.stata-journal.com

Subscriptions are available from StataCorp, 4905 Lakeway Drive, College Station, Texas 77845, telephone 979-696-4600 or 800-STATA-PC, fax 979-696-4601, or online at

http://www.stata.com/bookstore/sj.html

Subscription rates listed below include both a printed and an electronic copy unless otherwise mentioned.

U.S. and Canada		Elsewhere	
Printed & electronic		**Printed & electronic**	
1-year subscription	$115	1-year subscription	$145
2-year subscription	$210	2-year subscription	$270
3-year subscription	$285	3-year subscription	$375
1-year student subscription	$ 85	1-year student subscription	$115
1-year institutional subscription	$345	1-year institutional subscription	$375
2-year institutional subscription	$625	2-year institutional subscription	$685
3-year institutional subscription	$875	3-year institutional subscription	$965
Electronic only		**Electronic only**	
1-year subscription	$ 85	1-year subscription	$ 85
2-year subscription	$155	2-year subscription	$155
3-year subscription	$215	3-year subscription	$215
1-year student subscription	$ 55	1-year student subscription	$ 55

Back issues of the *Stata Journal* may be ordered online at

http://www.stata.com/bookstore/sjj.html

Individual articles three or more years old may be accessed online without charge. More recent articles may be ordered online.

http://www.stata-journal.com/archives.html

The *Stata Journal* is published quarterly by the Stata Press, College Station, Texas, USA.

Address changes should be sent to the *Stata Journal*, StataCorp, 4905 Lakeway Drive, College Station, TX 77845, USA, or emailed to sj@stata.com.

The *Stata Journal* (ISSN 1536-867X) is a publication of Stata Press. Stata, Stata, Stata Press, Mata, Mata, and NetCourse are registered trademarks of StataCorp LP.

The Stata Journal (2015)
15, Number 3, pp. 845–853

mqtime: A Stata tool for calculating travel time and distance using MapQuest web services

John Voorheis
Department of Economics
University of Oregon
Eugene, OR
jlv@uoregon.edu

Abstract. In this article, I describe `mqtime`, a new Stata library that provides functionality to perform a variety of mapping tasks, including calculating travel time, distance (driving, biking, or walking), and estimated fuel use. `mqtime` uses an overlooked free and open-source mapping service provided by MapQuest. This service has significantly more attractive terms of use than widely used alternatives (for example, Google Maps), which limit use to a few thousand queries per day. Hence, `mqtime` makes analysis with even very large datasets practical. I also provide a convenient function for geocoding character addresses to geographic coordinates.

Keywords: dm0083, mqgeocode, mqtime, MapQuest, HERE Maps, geocoding

1 Introduction

In many applications, the distance between two locations can be a crucial factor in explaining behavior. There are numerous ways to calculate distances, and these methods have varying degrees of realism. One can, for instance, calculate a straight-line or "great circle" distance between two points using their latitude and longitude coordinates. If one is modeling transportation, however, the straight-line distance may be quite different from the distance actually traveled, for example, by car or bicycle. Calculating the true driving distance is a much more complex task than calculating straight-line distance, but when accurate measures of distances are important to a particular analysis, this extra complexity may be warranted.

One way to calculate distances is to use third-party mapping services. Indeed, the `traveltime` library (written by Adam Ozimek and Daniel Miles [2011]) was written for use with Google Maps. Unfortunately, `traveltime` was written for the now-obsolete v2 of the Google Maps application programming interface (API). Since the spring 2013 transition to the new Google Maps API v3, `traveltime` has been rendered nonfunctional. Additionally, before the API change, Google implemented restrictions that severely limited the number of requests that a researcher could make each day. An extension of the `traveltime` library has been written that supports the Google Maps API v3, but it is still subject to the same rate limits.

`mqtime` is an attempt to provide off-the-shelf travel-time calculation without the disadvantages associated with tools that use the Google Maps API. This utility takes

advantage of a valuable but overlooked service provided by MapQuest. MapQuest provides an API that accesses its commercial mapping service (the same service one would access through http://www.mapquest.com); however, this service is available only for a steep fee. MapQuest also provides a second API (the open API) that accesses the OpenStreetMap (OSM) service. The OSM project[1] is a partially crowd-sourced project to produce and maintain a publicly available, open-source street map covering as much of the world as possible.

`mqtime` (and the associated convenience function `mqgeocode`) is written to mimic the syntax of `traveltime`, thereby easing the learning curve for users who are already familiar with the `traveltime` tool. However, the under-the-hood functionality of making the API requests and then parsing them into a format that is readable by Stata differs significantly. The `insheetjson` (Lindsley 2012) library provides functionality to parse the type of data object (a JavaScript Object Notation [JSON] object) returned by the MapQuest service.[2] The approach in `mqtime` can be easily extended for use with other similar API services, which are increasingly provided by companies and are an underused resource for economists.

The chief advantage of `mqtime` over `traveltime` is freedom from the strict usage limits imposed by Google. Google places a firm limit of 2,500 API requests (for example, directions or geocoding) per day per IP address. These limits can be very restrictive, even for medium-sized datasets. A dataset used by one of the testers of `mqtime` involves approximately 500,000 unique address pairs. Processing these data with the Google Maps API would take over six months. The terms of use (http://info.mapquest.com/terms-of-use/) for the MapQuest Open API are much more permissive. By using the MapQuest Open API, `mqtime` allows for the processing of large datasets in reasonable amounts of time.[3]

2 The mqgeocode command

Although the OSM API will take either text address information or latitude and longitude coordinates as input, users may wish to generate latitudes and longitudes from a text address for other purposes (for example, for calculating simpler straight-line distances or for use within geographic information systems software.) Geocoding of addresses can be accomplished with geographic information systems rather easily, but doing so requires leaving the confines of Stata. `mqgeocode` is provided for convenience. Like the `mqtime` command (see next section), the `mqgeocode` command first attempts to geocode using the OSM API. If this geocoding fails, `mqgeocode` will then query a second API provided

1. See http://wiki.openstreetmap.org/wiki/Legal_FAQ#I_would_like_to_use_OpenStreetMap_maps._How_should_I_credit_you.3F and http://www.openstreetmap.org for more information.
2. The MapQuest APIs are written with web-oriented languages like JavaScript and Python in mind. It is easy for an experienced programmer to write a Python script that accomplishes the same thing as `mqtime`, but the goal here is to make the information as accessible as possible for general Stata users.
3. In testing, I timed the average request at about 0.5 seconds, most of which is HTTP overhead, so our tester's dataset would require several days to complete.

by the HERE Maps service.[4] By default, `mqgeocode` will display a running total of the number of addresses that have been geocoded as well as the number of requests made to the HERE Maps API.

2.1 Syntax

`mqgeocode` [*if*] [*in*], {`address(`*varname*`)` | `lat(`*varname*`) long(`*varname*`)`}
[`outaddress(`*varname*`) here_id(`*string*`) here_code(`*string*`)`]

2.2 Options

`address(`*varname*`)` specifies the variable holding the plain text addresses (for example, `123 First Ave., Eugene, OR 97402`) to be geocoded. This option cannot be combined with options `lat()` and `long()`. `address()` or `lat()` and `long()` are required.

`lat(`*varname*`)` specifies the variable holding the latitude to be reverse geocoded. This option must be used with option `long()` and cannot be used with option `address()`. `lat()` and `long()` or `address()` is required.

`long(`*varname*`)` specifies the variable holding the longitude to be reverse geocoded. This option must be used with option `lat()` and cannot be used with option `address()`. `lat()` and `long()` or `address()` is required.

`outaddress(`*varname*`)` specifies the variable name to be used for the output, which will be either a plain text string (`Ann Arbor, MI`) or a string holding a latitude, longitude pair (`32.377588,-86.301882`). The default is `outaddress(coords)`.

`here_id(`*string*`)` and `here_code(`*string*`)` specify user-specific HERE Maps API information (ID and code). By default, the `mqtime` command will use the built-in API credentials associated with the author. However, if users wish to use their own API credentials, they may do so. See the following comments for more discussion.

2.3 Remarks

Traditional geocoding is relatively straightforward—it requires only that the user has created a variable holding the full text addresses. If a user has separate variables (for example, city and state), concatenating them is trivial.

```
generate newvar = cityvar + "," + statevar
```

4. More information about the HERE Maps service is available at https://developer.here.com/.

In general, `mqgeocode` will do only basic string formatting (most importantly, replacing spaces with `%20`), so the user must ensure that there are no disallowed characters or spelling errors. The API is permissive in terms of which address formats it will accept, including `city`, `state`; `address`, `city`, `state`, `zipcode`; and `address`, `zipcode`. For certain addresses located in unincorporated areas, `address`, `zipcode` seems to perform better.

When reverse geocoding, `mqgeocode` will return a text address at the most granular level available (this will range, in practice, from the zip code level to the exact address). In either traditional or reverse geocoding, the geocoded variable is returned as a single string. If separate latitude and longitude variables are required in traditional geocoding, the `outaddress()` variable can be split using

```
split latlongvar, p(",")
```

2.4 Example

Suppose we think that the distance between state capitol buildings is important for some application and that we have a dataset with addresses of capitol buildings that looks like this:

```
. insheet using "statecap.csv", clear
(6 vars, 10 obs)
. rename v1 address
. rename v2 city
. rename v3 state
. rename v4 address1
. rename v5 city1
. rename v6 state1
. list address city state
```

	address	city	state
1.	600 Dexter Ave.	Montgomery	Alabama
2.	120 4th St.	Juneau	Alaska
3.	1700 W. Washington	Phoenix	Arizona
4.	300 W. Markham St.	Little Rock	Arkansas
5.	1315 10th St	Sacramento	California
6.	200 E. Colfax Ave.	Denver	Colorado
7.	2210 Capitol Ave.	Hartford	Connecticut
8.	411 Legislative Ave.	Dover	Delaware
9.	402 S. Monroe St.	Tallahassee	Florida
10.	206 Washington St. SW	Atlanta	Georgia

Our first step, then, is to create the new variable

```
. generate capitol = address + "," + city + "," + state
```

We can now geocode these addresses using the following:

```
. mqgeocode in 1/10, address(capitol) outaddress(coords)
Observation 1 of 10 geocoded using the OpenStreetMaps API.
Observation 2 of 10 geocoded using the OpenStreetMaps API.
Observation 3 of 10 geocoded using the HERE Maps API. (1 total requests this run)
  (output omitted)
```

When execution is complete, the data will contain the latitude and longitude for each observation.

```
. list capitol coords
```

	capitol	coords
1.	600 Dexter Ave.,Montgomery,Alabama	32.377588,-86.301882
2.	120 4th St.,Juneau,Alaska	58.301945,-134.410453
3.	1700 W. Washington,Phoenix,Arizona	33.4485703,-112.0944824
4.	300 W. Markham St.,Little Rock,Arkansas	34.748601,-92.273452
5.	1315 10th St,Sacramento,California	38.576718,-121.494911
6.	200 E. Colfax Ave.,Denver,Colorado	39.739237,-104.984795
7.	2210 Capitol Ave.,Hartford,Connecticut	41.763585,-72.691498
8.	411 Legislative Ave.,Dover,Delaware	39.154514,-75.519517
9.	402 S. Monroe St.,Tallahassee,Florida	30.437994,-84.280724
10.	206 Washington St. SW,Atlanta,Georgia	33.749747,-84.38857

3 The mqtime command

3.1 Syntax

`mqtime` [*if*] [*in*], {`start_x(`*varname*`) start_y(`*varname*`)` | `start_add(`*varname*`)`} {`end_x(`*varname*`) end_y(`*varname*`)` | `end_add(`*varname*`)`} [`here_id(`*string*`) here_code(`*string*`) km mode(`*string*`)`]

3.2 Options

`start_x(`*varname*`)` and `start_y(`*varname*`)` specify the longitude (x) and latitude (y), in degrees, of the origin location. These cannot be used with option `start_add()`. `start_x()` and `start_y()` or `start_add()` is required.

`start_add(`*varname*`)` specifies the variable holding the plain text address of the origin location. This option cannot be used with options `start_x()` and `start_y()`. `start_add()` or `start_x()` and `start_y()` are required.

`end_x(`*varname*`)` and `end_y(`*varname*`)` specify the longitude (x) and latitude (y), in degrees, of the destination location. These cannot be used with option `end_add()`. `end_x()` and `end_y()` or `end_add()` is required.

`end_add(`*varname*`)` specifies the variable holding the plain text address of the destination location. This option cannot be used with options `end_x()` and `end_y()`. `end_add()` or `end_x()` and `end_y()` are required.

`here_id(`*string*`)` and `here_code(`*string*`)` will use a user-supplied MapQuest API key in place of the built-in key. By default, the `mqtime` command will use the built-in API credentials associated with the author. However, if users wish to use their own API credentials, they may do so. See the following comments for more discussion.

`km` specifies whether the distances returned will be in kilometers or miles (default is miles).

`mode(`*string*`)` specifies which mode of travel is to be used. This option must be one of `walking`, `bicycle`, or `transit` (the default is `mode(driving)`, if unspecified).

3.3 Comments

By default, `mqtime` will first attempt to query the MapQuest OSM API for each origin and destination pair. As noted earlier, the OSM service has no preset rate limit and is therefore much more amenable to processing large datasets. However, it does have slightly less coverage than commercial mapping alternatives.[5] As a backup, `mqtime` will query the HERE Maps API if the OSM API fails to return a valid route. The HERE Maps API has considerably better coverage, although it does have stricter usage limits. By default, `mqtime` will query the HERE Maps API only if it receives a route-failure error from the OSM API.

By default, `mqtime` will also use the HERE Maps credentials associated with the author of this article. This means, however, that all requests to the HERE Maps API will be pooled for all users. If users find themselves needing to make a relatively large number of requests for which the OSM service fails, it may be advantageous to request separate API credentials. HERE Maps developer accounts are free and can be requested at the HERE Maps developer site.[6] To ensure `mqtime` uses a user's own API credentials, you must specify the `here_id()` and `here_code()` options. The actual API credentials are randomly generated strings, so it may be easiest to attach them to local macros, for instance,

```
local here_id = "your_id_here"
local here_code = "your_code_here"
```

Users can then instruct `mqtime` to use these credentials.

```
mqtime, ... here_id("`here_id´") here_code("`here_code´")
```

With this modification, `mqtime` will make requests using the user's own personal HERE Maps credentials.

5. In testing, I found that the OSM failed to generate a route for approximately 2% of origin and destination pairs.
6. See https://developer.here.com/get-started.

The `mqtime` program takes the user's inputs and builds a URL that is the desired API request. This URL will return a JSON data object.[7] This JSON object is a text file containing nested key:value pairs. Many general-purpose programming languages have built-in functionality to parse these objects, but Stata has no base functionality to deal with them. Fortunately, Lindsley's (2012) `insheetjson` command can parse JSON files into Stata-readable data. `mqtime` calls the `insheetjson` command to parse the JSON object and return the relevant information (travel time, driving distance, and fuel use) and discards the rest.

Users can specify the origin and destination locations either as latitude and longitude or as a text address. The two locations need not be in the same format. For example, if one had latitude and longitude for a list of origins and text addresses for a list of destinations, one could execute

```
mqtime, start_x(lngvar) start_y(latvar) end_add(addvar)
```

The other options are largely self-explanatory, with the exception of the `mode()` option. The nondriving travel modes available in `mode()` may be slightly less accurate than the driving information for the OSM service. For multimodal (transit) mode, the user must specify a time of day (if no time is provided, the API request will be for the time at run time).

`mqtime` will, by default, create four new variables: `travel_time`, `distance`, `service`, and `fuelUsed`. Travel time is returned in minutes by default, and distance is returned in miles (or kilometers if `km` is specified). The estimated fuel-use variable is generated based on MapQuest's estimate of fuel use in gallons for the trip.[8] Note that the HERE Maps API will not return an estimate of fuel use. The OSM and HERE Maps APIs return the fastest route based on posted speed limits, but they do not take into account real-time traffic information.

3.4 Examples

We return to the example using state capitol buildings to illustrate how `mqtime` can query the API to obtain driving directions from text addresses. Suppose we want to generate directions from the 10 state capitol buildings used in the previous example to 10 other arbitrarily matched capitol buildings. The first five rows of our data then look like this:

```
. rename capitol capitol_origin
. generate capitol_destination = address1 + "," + city1 + "," + state1
. keep capitol_origin capitol_destination
```

7. An example URL (requesting the driving route between Eugene, OR, and Springfield, OR) is http://open.mapquestapi.com/directions/v2/route?key=YOUR_KEY_HERE&from=Eugene,OR&to=Springfield,OR&outFormat='json'&narrative='none'.
8. It is possible to supply the MapQuest API with specific fuel-efficiency assumptions, but this is not yet implemented in `mqtime`.

```
. list
```

1.	capitol_origin: 600 Dexter Ave.,Montgomery,Alabama
	capitol_destination: 488 N 3rd St,Harrisburg,Pennsylvania
2.	capitol_origin: 120 4th St.,Juneau,Alaska
	capitol_destination: 82 Smith St.,Providence,Rhode Island
3.	capitol_origin: 1700 W. Washington,Phoenix,Arizona
	capitol_destination: 1100 Gervais St.,Columbia,South Carolina
4.	capitol_origin: 300 W. Markham St.,Little Rock,Arkansas
	capitol_destination: 500 E. Capitol Ave,Pierre,South Dakota

(*output omitted*)

To generate the driving time, distance, etc., we can then execute the following:

```
. mqtime, start_add(capitol_origin) end_add(capitol_destination)
Processed 1 of 10 using the OpenStreetMaps API.
Processed 2 of 10 using the OpenStreetMaps API.
Processed 3 of 10 using the HERE Maps API. (2 total requests this run)
  (output omitted)
Processed 10 of 10 using the OpenStreetMaps API.
. drop capitol_origin capitol_destination
```

When execution is complete, we can check that `mqtime` has generated the correct data; we can see that all the routes were mapped without error, and only one required the backup HERE Maps API.

```
. list
```

	travel~e	distance	fuelUsed	service
1.	805.0667	882.5601	42.06	OSM
2.	5216.317	4026.689	164.76	OSM
3.	1887.45	2064.501	.	HERE Maps
4.	904	1004.535	48.93	OSM

(*output omitted*)

4 Conclusion

Taking advantage of `insheetjson`'s ability to parse JSON files (the industry standard of data provision from an API), I have shown how to generate travel times, distances, and estimated fuel use. This is accomplished using a convenient but underused service provided by MapQuest. Unlike previous implementations, `mqtime` can process even very large datasets without running afoul of terms of use. `mqtime` can also be easily patched to stay up to date with API changes.

Several features are available in the OSM API and the HERE Maps API of which `mqtime` does not yet take full advantage. Future revisions of the `mqtime` code base, conditional on user input, will seek to incorporate these features. Chief among these potential additional features are the following: 1) providing vehicle miles per gallon to calculate more precise fuel use; 2) taking advantage of real-time traffic data to provide more precise travel times; and 3) the ability to request directions more precisely (for example, avoiding toll roads).

5 Acknowledgments

I thank Sonja Kolstoe and Jason Query for providing datasets and testing, and I thank Trudy Cameron for encouragement and advice. None of this would be possible without the work of Adam Ozimek and Daniel Miles (2011) (for writing the original `traveltime`) and Erik Lindsley (2012) (for writing `insheetjson`). Any remaining errors are my own.

6 References

Lindsley, E. 2012. insheetjson: Stata module for importing tabular data from JSON sources on the internet. Statistical Software Components S457407, Department of Economics, Boston College. https://ideas.repec.org/c/boc/bocode/s457407.html.

Ozimek, A., and D. Miles. 2011. Stata utilities for geocoding and generating travel time and travel distance information. *Stata Journal* 11: 106–119.

About the author

John Voorheis is a PhD student in economics at the University of Oregon. His research focuses on measuring income inequality and poverty at subnational geographic scales and examining the effects of local income distribution on economic outcomes of interest. He is also interested in learning and developing new tools for economists to take advantage of the "big data" revolution.