Who Lives Near Here? Part II

by Jeffreys Copeland and Haemer

Welcome back to our series on common problems we face in our day-to-day office work. For the past several months, we've been discussing the problem of maintaining an address book. We're finally getting around to solving the problem we've been putting off since the April issue, namely how to determine who lives close to whom. Last month, we attacked the first step of this problem by building an Expect script, `find-geo`, to connect to the geographic database server at the University of Michigan, (`telnet martini.eecs.umich.edu 3000`) and saving our interaction with it.

As we've discussed before, we prefer to use a text-based database for everything we can, including our address file, in large part because we can use our familiar UNIX tools on it. Unlike a database in a specialized DBMS, or worse, a DOS word processor, on UNIX, a file is a file is a file: `cat` works on everything. This approach to databases will be the one we'll use until our computers have the same capabilities as those on "Star Trek." ("Computer, where's the nearest Klingon chocolate

restaurant?" "As you know, your family has been concerned lately with your intake of empty calories. As an alternative to chocolate, there's a nice Andorean juice bar one

Jeffrey Copeland (`copeland@alumni.caltech.edu`) *is a member of the technical staff at QMS's languages group, in Boulder, CO. His recent adventures include internationalizing a large sales and manufacturing system and providing software services to the administrators of the 1993 and 1994 Hugo awards. His research interests include internationalization, typesetting, cats and children.*

Jeffrey S. Haemer (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

block to the west...") Then again, maybe our tools will work well into the 23rd century.

We Rebuild the Database

We have several steps toward determining geographic proximity. We begin by building a version of our address database containing latitude and longitude for each entry.

Warning: We're going to cover ground pretty quickly here, often without discussing the details of the code in our little programs. Most of them should be simple enough that it won't be a problem.

Let's start by collecting all ZIP codes from our existing database. There are two possible one-liners to do this extraction. We can say either

```
sed -n 's/.* \([0-9]\{5\}\).*/\1/p' addr* | \
   sort -u >zipcode-list
```

or

```
perl -ne 'print "$1\n" if /\s(\d{5}) \
   [-\s]/;' addr* | sort -u >zipcode-list
```

This gives us a file that looks something like

```
00123
01966
02138
02154
02401
02904
03053
03456
78726
```

(Exercise for the reader: Which one-liner is more correct, and why? What happens in the case of a phone number like 01185-22-528-6605? Or an entry in the database that includes a credit card number?)

Next, we need to look up those ZIP codes in the UMich database, to get their coordinates. We can use the version we wrote last month, but as an exercise, try writing one that we can invoke as

```
find-geo `cat zipcode-list`
```

This will result in only one Telnet session, rather than one for each ZIP code. The output of this exercise will be a massive lump of data, most of which we don't want. How do we distill out the useful parts?

We need a program to read in our list of ZIP codes and extract the longitude and latitude from the text we got from Michigan, outputting a list of ZIP code plus coordinate pairs. We can write an ugly extractor as a shell script, or a somewhat more elegant version in Perl (see Listing 1 on the following page).

What does this script do? It begins by reading our list of ZIP codes into an array (`@targets`). Then it opens the file containing results of our `find-geo` command, and finds the latitude and longitude corresponding to each ZIP code in `@targets`. It finishes by giving us a list of the ZIP codes that it didn't find in the output file. (Variation: The lines commented out with `#-#` can replace the lines following them to interact directly with the geographic server, rather than using an intermediate file.)

So, uttering

```
zipdb.pl zipcode-list geo > zipcodeLL
```

creates a `zipcodeLL` file, which contains lines like these:

```
01966  L  42  39  20 N 70 37  15  W
02138  L  42  21  54 N 71 06  18  W
02154  L  42  22  35 N 71 14  10  W
02401  L  42  05  00 N 71 01  08  W
02904  L  41  49  26 N 71 24  48  W
03053  L  42  51  54 N 71 22  28  W
03456  L  43  06  57 N 72 11  51  W
78726  L  30  16  01 N 97 44  34  W
```

Notice that there are some ZIP codes that don't appear in the geographic information server, such as 91351 (Sun Valley, CA). We need to look those up by hand–or ask for a nearby alternative, like Valencia–and insert them into our `zipcodeLL` file.

(Exercise: Write an alternate version of `zipdb.pl` that creates a `zipcodeLL` with the coordinates of *all* ZIP codes we got from the geographic information server, not just those we were explicitly looking up.)

Also, as we alluded to in an earlier exercise, we get some chaff in our list of ZIP codes such as 00123 (the first five digits of Jeffrey Copeland's United Airlines frequent flyer account) if our ZIP code-extracting one-liner is not careful enough. Lastly, notice that some ZIP codes, for example, 02138 (Cambridge, MA), get several hits from the GIS. How to handle multiple coordinates for a ZIP code? Should we average them? We've just taken the first hit.

Next, we need to insert the latitude and longitude into the address database.We'll use `@` as our tag. Again, a Perl script will solve this problem (see Listing 2 on Page 32).

We have another small problem: We don't have postal codes for foreign locations in our `zipcode.LL` file. Neither does our program recognize non-U.S. post codes. Worse yet, this script operates by removing every location it sees and looking it up again in our list, so foreign locations inserted by hand will get stripped out when we run this.

Listing 1

```
#! /usr/local/bin/perl
# Collect long/lat pairs from results of a telnet session with UMich

# $1 is the list of zip codes in our addr database
# $2 is the collected output of find-ll

$0 =~ s(.*/)();
#-# die "usage: $0 zipcodes\n" unless @ARGV == 1;
die "usage: $0 zipcodes find-geo-output\n unless @ARGV == 2;

    # first, suck in all the target zipcodes
    # - could be done directly from the address file
open(TGTS, $zips = shift) || die "Can't open zipcode file $zips: $!";
chop(@targets = <TGTS>);
close(TGTS);
grep($tgt{$_}++, @targets);

#-# $gdb="./find-geo @targets |";
$gdb = shift;

$/ = "\r\n\r\n";
open(GEO, $gdb) || die "Can't open $gdb: $!";

    # now suck in the response from the geographical name server
    # - could invoke it right here, too.
while(<GEO>) {
    @loc = split(/\r\n/, $_);

    foreach $line (@loc) {
        if ($line =~ /^([\w])\s+(.*)/) { # key-value pair
            $key = $1;
            $val = $2;
            $data{$key} .= $val . " ";
        }
    }
    @zipcodes = split(/\s+/, $data{'Z'});

        # now find the intersection of the two arrays
    ($zip) = grep($tgt{$_}, @zipcodes);
    if ($zip && $data{'L'}) {
        print "$zip L $data{'L'}\n";
            # delete from the list of targets
        @targets = grep($_ ne $zip, @targets);
    }
    undef \data;

} close(GEO);

warn "The following zipcodes weren't found: @targets\n";
```

(Exercises: Write a version of `zipdb.pl` that is not destructive to existing data. Write a version that recognizes foreign post codes or uses some other method to look up foreign addresses.)

Again, we get a useful file, this time containing things like:

```
Hon David Skaggs
2nd Cong District Colorado
Rm 1124 Longworth House
Office Bldg
Washington DC 20515
@ 38 53 42 N 77 02 12 W
SKAGGS@HR.HOUSE.GOV

Computer Literacy
2590 N First St
San Jose CA 95131
@ 37 20 07 N 121 53 38 W
#w: 408-435-0744
#f: 408-435-1823
```

What good does this do us? We now have a latitude and longitude for each entry in our address book. If we know where we're going, we can figure out who or what is nearby. At least until the 23rd century, as we postulated in our second paragraph, when we'll need to worry about coordinates on other planets.

Who Can I Call for Dinner?

How close is one location to another on the globe? We won't bother to make this an exercise in spherical trigonometry but will, instead, pluck the formula from our navigation textbooks. The distance in nautical miles from latitude and longitude (L,l) to (L',l') is 60 arccos [sin L sin L' + cos L cos L' cos(l-l')]. (Why nautical miles? One nautical mile is a minute of arc at the equator.)

Given this formula, it's quite simple to write a program to do the calculation, which we can invoke as `dist 40:00:54N 105:16:12W 30:16:01N 97:44:34W` to get the distance from Boulder to Austin:

```
/*  small program to determine distance between
    two points on the globe, given longitude &
    latitude of both */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>

#define USAGE "%s: long1 lat1 long2 lat2\n\t\
long or lat is dd:mm[:ss][C], e.g., 40:27:17N\n"

main( ac, av )
int ac;
char *av[];
{
    double lat1, lat2, long1, long2;
    double parse();
    double dist;
    int n;

    /* usage message if we need it */
    if( ac < 5 ) {
     printf( USAGE, av[0] );
     exit( 1 );
    }
    /* get the arguments */
    lat1 = parse( av[1] );
    long1 = parse( av[2] );
    lat2 = parse( av[3] );
    long2 = parse( av[4] );

    /* use the standard formula,
       which gives us nautical miles */
    dist = cos(lat1*M_PI/180.)
       * cos(lat2*M_PI/180.);
    dist *= cos((long1-long2)*M_PI/180.);
    dist += sin(lat1*M_PI/180.)
       * sin(lat2*M_PI/180.);
    dist = 60. * acos(dist) * 180. / M_PI;

    /* convert to statute miles */
    dist = dist * 6080. / 5280.;
    printf( "%.3f\n", dist );
    exit( 0 );
}


double
parse( s )
char *s;
{
    double z;

    z = atof( s );
    while( isspace(*s)
     || isdigit(*s) ) s++;
    if( ispunct(*s) ) {
     z += atof(++s) / 60.;
     while( isspace(*s)
```

Listing 2

```
#! /usr/local/bin/perl -s

# given a file LL containing a list
# zip codes with the latitude/longitude
# line we got from UMich, insert the
# appropriate geographic coordinates
# into our address file every time we
# find a zip-code.

# begin with the list of known zip codes and
#long/lat

$0 =~ s(.*/)();
$USAGE = "$0 zipcodefile addressfile";
#die "$USAGE" unless (@ARGV == 2);

$zipcodes = shift;
open(LL, $zipcodes) ||
    die "Can't open $zipcodes: $!";
@longlat = <LL>;
close(LL);

$zip_pat = '\s(\d{5})[-\s]';
    # our pattern for a zip-code
    # could be enhanced,
    # perhaps even to handle foreign codes

# now read the data from our address files
while( <> ) {
    # skip existing geo data
    /^@/ && next;

    # always print any other line we read
    print;

    # if we have a possible zip code
    if (/$zip_pat/) {
     $zip = $1;
     @found = grep( /^$zip /, @longlat );
     next unless @found;
     @ll = split(/L */, $found[0]);
     print "@ $ll[1]";
    }
}
```

Listing 3

```
#! /usr/local/bin/perl

$0 =~ s(.*/)();
$USAGE = "$0 distance latitude longitude";

sub fmt_locn {   # format up a presumptive location
    local ($deg, $min, $sec, $dir) = @_;
    return 0 unless ("$deg$min$sec" =~ /^\d*$/);
    return 0 unless ($dir eq 'N' || $dir eq 'S'
        | $dir eq 'E' || $dir eq 'W');
    return "$deg:$min:$sec$dir"; }

$dist = shift(@ARGV);
$long1 = shift(@ARGV);
$lat1 = shift(@ARGV);
## print "$long1 $lat1\n";

$empty = 0;

while( <> ) {
    chop;
    if( /^$/ )
    {
        $empty++;
        next;
    }
    if( $empty )
    {
        $who = $_;
        $empty = 0;
        next;
    }

    @line = split;
    next unless (($lead = shift(@line)) eq '@');
    $long2 = &fmt_locn(@line[0..3]) || next;
    $lat2 = &fmt_locn(@line[4..7]) || next;

    $howfar = `./dist $long1 $lat1 $long2 $lat2`;
    print "$who: $long2 $lat2 ... $howfar"
        if ( $howfar <= $dist );
}
```

Figure 1

```
Hon David Skaggs: 38:53:42N 77:02:12W ... 0.000
Debbie & Ian Copeland: 38:48:17N 77:02:50W ... 6.263
Ed & Josie Ercegovic: 40:01:25N 79:53:03W ... 170.729
Victor & Rose Pallotta: 39:58:22N 79:52:40W ... 168.898
Vicki Scarmazzi: 40:15:45N 80:11:15W ... 192.542
```

```
        || isdigit(*s) ) s++;

    if( ispunct(*s) ) {
        z += atof(++s) / 3600.;
        while( isspace(*s)
          || isdigit(*s) ) s++;
    }
    }

    /* south and west of the
       zero points are negative */
    if( tolower(*s) == 's'
        || tolower(*s) == 'w' ) z = -z;

    return z;
}
```

Notice that we've made the output of `dist` a distance in statute miles.

(Exercise: How would you adjust this program to reply in arbitrary units?)

Our last step is to write another Perl script to find all the entries in our address book within some distance of a given latitude and longitude (see Listing 3).

For example, if we know we are going to Washington, D.C., (latitude and longitude 38:53:42N 77:02:12W) and are willing to make a side trip up to 200 miles, we can say: `near 200 38:53:42N 77:02:12W <addr` and get the results shown in Figure 1.

(Exercise: Print out the address and phone number of each person we find in our search, or an envelope and letter asking when you can join them for dinner.)

Future Development

Obviously, we've left a bit undone here. So, we leave you with a variety of additional exercises to fill those gaps:

Exercise: Write a makefile or some other script to keep a local supplement to your address database with latitudes and longitudes. Use that database to update your address book regularly. Do a telnet to `umich` only if needed.

Exercise: Rewrite `near` to take a location and a distance, rather than coordinates. That is, let us say `near 50 Boston` or `near 70km Brussels`, rather than having to know the latitude and longitude of Boston or Brussels.

Next month we'll begin talking about "to do" lists and reminders. Think about how you handle this now by hand and how you'd do it mechanically.

Until then, happy trails! ▲