A numerical method for solving Poisson’s equation was developed on the assumption of a grounded boundary enclosing a charge distribution. This finite element method was then implemented in MATLAB and used to solve Poisson’s equation for the region enclosed by the boundary of the state of Maryland. Four charge distributions were considered. First, the charge density was made inversely proportional to the distance from a point to the author’s house located in Ellicott City, Maryland. Second, the charge density was defined as a constant over each county in Maryland proportional to that county’s population. Third, the density in each county was made proportional to that county’s income. Finally, the charge density was made proportional to the percentage of voters in each county who voted for the Democratic party in the 2006 senate election. All four cases showed a strong peak in the electric potential at the center of the state.

I. INTRODUCTION

When charges are stationary and no magnetic fields are present, Maxwell’s laws reduce to a single equation relating the electric potential to the charge distribution. Know as Poisson’s equation, the electric potential can be solved for as a boundary value problem. While Poisson’s equation can be solved analytically for simple, symmetric boundaries, numerical solutions are needed for the more arbitrary boundary conditions encountered in real life.

One common approach to solving partial differential equations with complex boundary conditions is the finite element method. This method is relatively straight-forward to implement in a high-level language like MATLAB and can be used to find solutions in regions with quite complicated boundaries. An example of such a region is the state of Maryland, as its southern border is formed by waterways. Maryland also offers a wealth of interesting charge distributions based on census data for its constituent counties.

Using Cooper’s textbook [1] as a reference for the theory behind the finite element method, MATLAB functions were developed to solve Poisson’s equation for an arbitrary grounded boundary given a triangulation of the enclosed region. These functions were then used to solve for the potential inside Maryland when the charge density is made proportional to county population, income, and voting patterns.

II. THEORY

According to Maxwell’s laws, the divergence of the electric field $\mathbf{E}$ is proportional to the charge density $\rho$, indicating that charges are sources and sinks of electric field lines.

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\varepsilon_0}$$

In general, the curl of the electric field is determined by time variance of the magnetic field $\mathbf{B}$ according to

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

In the electrostatic case, however, there are no time-varying magnetic fields, so $\nabla \times \mathbf{E} = 0$. Since the vector function $\mathbf{E}$ has zero curl, it can be expressed as the gradient of some scalar function $\phi$, known as the electric potential. The relationship between the electric field and the electric potential is expressed as

$$\mathbf{E} = -\nabla \phi$$

Maxwell’s equation for the divergence of $\mathbf{E}$ can then be re-written in terms of the Laplacian of $\phi$:

$$\nabla^2 \phi = -\frac{\rho}{\varepsilon_0}$$

This equation is known as Poisson’s equation.
The finite element method used in this study is based on a discussion of finite elements in Cooper [1]. The context of the problem is assumed to be an open region \( G \subset \mathbb{R}^2 \) (here, the state of Maryland) with a boundary \( \partial G \) composed of finitely many \( C^1 \) curves. The problem is stated as:

\[
-\nabla^2 u = q \text{ (in } G) \\
u = 0 \text{ (on } \partial G)
\]

Setting \( u = \phi \) and \( q = \rho/\varepsilon_0 \), this represents Poisson’s equation when the boundary is grounded.

The first step is to triangulate \( G \). The fineness of this triangulation is chosen to achieve the desired quality of solution within the time constraints on computation. The triangulation will contain some number \( N \) of interior nodes \((x_j, y_j)\) – triangle vertices that do not lie on \( \partial G \). For the \( j \)th interior node, define the region \( S_j \) as the union of all triangles with a vertex at that node. Define a “tent function” \( \varphi_j \) piecewise on \( S_j \) such that it is continuous, linear on each triangle in \( S_j \), and such that \( \varphi_j(x_j, y_j) = 1 \) and \( \varphi_j = 0 \) on \( \partial S_j \). This can be accomplished by expressing \( \varphi_j \) on each triangle (with vertices \((x_j, y_j), (x_k, y_k), (x_l, y_l)\)) as

\[
\varphi_j(x, y) = ax + by + c
\]

where the coefficients \( a, b, \) and \( c \) are solutions to the linear system

\[
\begin{align*}
ax_j + by_j + c & = 1 \\
ax_k + by_k + c & = 0 \\
ax_l + by_l + c & = 0
\end{align*}
\]

Define \( S = \bigcup S_j \). With a fine enough triangulation, \( \partial S \) is a good approximation of \( \partial G \). An approximate piecewise linear solution to the problem would be given by

\[
u(x, y) = \sum_{j=1}^{N} u_j \varphi_j(x, y)
\]

Thus, the problem has been reduced to finding a vector \( \mathbf{u} \) of length \( N \). To solve for \( \mathbf{u} \), compute the elements of the stiffness matrix \( A \) according to

\[
A_{i,j} = \int \int_S \nabla \varphi_j(x, y) \cdot \nabla \varphi_i(x, y) \, dx \, dy
\]

and define a vector \( \mathbf{q} \) as

\[
q_i = \int \int_S q(x, y) \varphi_i(x, y) \, dx \, dy
\]

Then \( \mathbf{u} \) becomes the solution to the linear system

\[
A \mathbf{u} = \mathbf{q}
\]

III. IMPLEMENTATION

The code to perform the finite element method was written for MATLAB due to existing functions for working with mesh geometry (most notably \texttt{tsearch} and \texttt{polyarea}). However, the MATLAB environment leaves much to be desired for a high-performance implementation. When functions cannot be vectorized, MATLAB’s \texttt{for}-loop is extremely slow. Furthermore, MATLAB is inherently single-threaded and resists any attempt at parallelizing functions. This is a significant drawback when dealing with the finite element method, as nearly all operations are independent of previous results and fall into the class of “embarrassingly parallel” problems. For example, the stiffness matrix could be partitioned into eight sections each solved for by independent processor cores.

For the scale of this study such parallelism was not necessary, as the longest computation was under 45 minutes, but given the growth in availability of multi-core systems, any further study would benefit greatly from parallelized routines. This would most likely imply re-writing all functions in a lower-level language like C or FORTRAN and using threads or MPI to distribute parallel tasks.

The finite element mesh is initially represented as a list of triangles, where a triangle is represented by three indices into a list of \( x \) and a list of \( y \) coordinates. A list of which pairs of \( x \) and \( y \) coordinates represent interior nodes is also
provided. When performing the finite element method, it is necessary to access a list of all triangles associated with a given interior node. Rather than generating such a list each time it is needed, it is more efficient to pre-compute these lists for all interior nodes. Since these lists depend only on the structure of the mesh, they can be reused for each charge distribution function considered, saving computation time. A cell array indexed by interior node number is an appropriate data structure for storing these lists. This array is generated by the following simple function:

```matlab
function c = getTrisByPoints(tri, inidx)
%GETTRISBYPOINTS List of triangles attached to interior nodes
% GETTRISBYPOINTS(TRI,INIDX) returns a cell array containing the indicies
% into TRI of triangles attached to each interior node indexed by INIDX.
%
% AUTHOR: Curran D. Muhlberger <cdmuhlb@umd.edu>
% DATE: 2006-12-10

if (nargin ~= 2)
    error('cdmuhlb:getTrisByPoints:NotEnoughInputs', 'Needs 2 inputs.');
end

for i = 1:length(inidx)
    k = 1;
    tmpvec = zeros(8, 1);
    for j = 1:length(tri)
        if ((inidx(i) == tri(j, 1)) || (inidx(i) == tri(j, 2)) || (inidx(i) == tri(j, 3))
            tmpvec(k) = j;
            k = k + 1;
        end
    end
    c{i} = tmpvec(1:(k - 1));
end
end

Next, it is necessary to compute the coefficients for the tent functions \( \varphi_j \) defined on the mesh. For the \( j \)th interior node, \( \varphi_j \) is defined piecewise on each of the triangles associated with that node. Therefore, the previous data structure can be used to determine over which of the triangles the function will be non-zero. The coefficients can be represented by a 3-dimensional sparse matrix, where each triangle will be associated with at most three sets of coefficients. Each set of coefficients is the solution to equation \( \Box \) which is efficiently solved using MATLAB’s backslash operator.

```matlab
function c = getCoefs(nodesx, nodesy, tri, inidx, tbp)
%GETCOEFS Coefficients for tent functions
% GETCOEFS(NODESX,NODESY,TRI,INIDX,TBP) returns a cell array containing
% the coefficients defining the tent functions on each triangle in the
% mesh TRI at each interior node indexed by INIDX into NODESX and NODESY.
%
% AUTHOR: Curran D. Muhlberger <cdmuhlb@umd.edu>
% DATE: 2006-12-10

if (nargin ~= 5)
    error('cdmuhlb:getCoefs:NotEnoughInputs', 'Needs 5 inputs.');
end

if (~isequal(size(nodesx), size(nodesy)))
    error('cdmuhlb:getCoefs:InputSizeMismatch', ...
        'NODESX and NODESY must be the same size.');
end
if (~isequal(size(inidx), size(tbp)))
    error('cdmuhlb:getCoefs:InputSizeMismatch', ...
        'INIDX and TBP must be the same size.');
end

for i = 1:length(inidx)
    for j = 1:length(tbp{i})
        jj = tbp{i}{j};
        sim = tri(jj, :);
        A = [nodesx(sim(1)), nodesy(sim(1)), 1; ...
            nodesx(sim(2)), nodesy(sim(2)), 1; ...
            nodesx(sim(3)), nodesy(sim(3)), 1];
        b = zeros(3,1);
        for k = 1:length(b)
            ...
        end
    end
end
```
The tent functions $\varphi_j$ themselves are easily computed at a point $(x, y)$ as the dot product of the relevant coefficient vector and the 3-vector $(x, y, 1)$. However, determining which coefficient vector to use requires knowing which triangle a given point lies in. MATLAB provides a triangle search routine (tsearch), but it makes false assumptions about the triangulation which causes it to fail. David Martin of Boston College has written a replacement routine, mytsearch, which works for any triangulation [6].

This implementation unfortunately is not vectorized and as a result performs slowly when evaluated at many points. The performance of the for-loop is so poor that evaluating the solution can take longer than solving for its parameters. Vectorization of this routine would be the first step in an extension of this study.

Given the piecewise definitions for all $\varphi_j$, the stiffness matrix $A$ can be generated according to 3. The first optimization is to note that the stiffness matrix is symmetric, so only half of the elements need be solved for. Also, given two distinct interior nodes $i$ and $j$, at most two triangles can have a vertex at both nodes. The integrand is constant over each triangle, so the integral is simply the integrand multiplied by the area of the triangle.
This implementation does not fully exploit the sparseness of the stiffness matrix, as it still makes $ij/2$ comparisons to determine which pairs of interior nodes have triangles in common. However, as the matrix depends only on the mesh and the boundary conditions and not on the charge distribution, it need be calculated only once, and intermediate data structures providing nodes with knowledge of adjacent nodes would simply replace one lengthy one-time computation with another.

The first calculation dependent on the charge distribution is generating the vector $u$ via equation 4. A high-precision method for generating $u$ would require numerical integration in two dimensions for each interior node. However, this approach is far too computationally expensive for fine meshes. Alternatively, the integral can be approximated assuming that the charge distribution described by $q(x, y)$ is constant over each region of the mesh. For a fine enough mesh and reasonably smooth charge distributions, this approximation is reasonable. Furthermore, three of the distributions being considered involve locally constant charge densities, making this approximation exact except at the edges of counties.

Assuming $q$ is constant over the areas of integration, the integral reduces to the value of $q$ at the region’s interior node multiplied by the volume of the tent function $\varphi_j$ over that region. Because each tent function is a union of tetrahedra of height 1, their volume is simply $1/3$ times the area of the region they cover.

function \texttt{v = getApxQVec(nodesx, nodesy, inidx, tri, tpbp, q)}
\begin{verbatim}
%GETAPXQVEC Approximate q vector for finite element method
% GETAPXQVEC(NODESX,NODESY,INIDX,TRI,TBP,Q) returns a vector equal in
% length to INIDX – the indicies of interior nodes into the node vectors
% NODESX and NODESY: The exact values in this vector would be found by
% integrating the function Q multiplied by the tent function for the
% interior node over the region surrounding that node. This function
% approximates that integral by assuming that the function Q is constant
% over the region.

% AUTHOR: Curran D. Muhlberger <cdmuhlb@umd.edu>
% DATE: 2006-12-10

if (nargin \^= 6)
  error('cdmuhlb:getApxQVec:NotEnoughInputs', 'Needs 6 inputs.');
end
if (\texttt{size(nodesx, size(nodesy)))}
  error('cdmuhlb:getApxQVec:InputSizeMismatch', ...
        'NOODESX and NODESY must be the same size.');
end
if (\texttt{size(inidx, size(tpbp)))}
  error('cdmuhlb:getApxQVec:InputSizeMismatch', ...
        'INIDX and TBP must be the same size.');
end
v = \texttt{zeros(length(inidx), 1)};
for i = 1:length(inidx)
  qval = q(nodesx(inidx(i)), nodesy(inidx(i)));
  a = 0;
  c = tpbp(i);
end
end
end
end
end

dif \texttt{\textbackslash{}isequal(size(nodesx), size(nodesy)))}
  error('cdmuhlb:getStiffness:InputSizeMismatch', ...
        'NODESX and NODESY must be the same size.');
end

n = \texttt{length(tpbp)};
a = \texttt{sparse(n, n)};
for i = 1:n
  for j = i
    \texttt{c = intersect(tpbp{i}, tpbp{j});
    for k = 1:length(c)}
      \texttt{t = c(k);
        polyx = nodesx(tri(t, :));
        polyy = nodesy(tri(t, :));
        a(i, j) = a(i, j) + dot(coefs{i, t}(1:2), coefs{j, t}(1:2)) * polyarea(polyx, polyy);}
  end
  a(j, i) = a(i, j);
end
end
for k = 1:length(c)
    t = c(k);
    polyx = nodesx(tri(t, :));
    polyy = nodesy(tri(t, :));
    a = a + polyarea(polyx, polyy);
end

v = qval * a * 1/3;
end
end

Once a solution vector $u$ is obtained, it can be used to calculate the solution to Poisson’s equation at any point $(x, y)$ via equation 2. Most of the terms in this sum are zero, so it is more efficient to search for the non-zero terms and sum only them. Unfortunately, parts of this search are not vectorized, so performance will be hindered by MATLAB’s for-loop. The use of a for-loop either here or in the tentfunc function called here makes evaluating the solution the slowest part of the method. Both functions would need to be vectorized to see a performance increase. Parallelism would also help in this situation, as the region to be plotted could be divided into sections, and each section would then be plotted independently on separate processor cores.

function f = uh(nodesx, nodesy, tri, inidx, coefs, uvec, xi, yi)
%UH Solution function from FEM
% UH(NODESX,NODESY,TRI,INIDX,COEFS,UVEC,XI,YI) evaluates the solution to
% the differential equation as represented by UVEC at the points (XI,YI).
% AUTHOR: Curran D. Muhlberger <cdmuhl@umd.edu>
% DATE: 2006-12-13

if (nargin ~= 8)
    error('cdmuhl:uh:NotEnoughInputs', 'Needs 8 inputs.);
end
if (~isequal(size(nodesx), size(nodesy)))
    error('cdmuhl:uh:InputSizeMismatch', ...
    'NODESX and NODESY must be the same size.');
end
if (~isequal(size(xi), size(yi)))
    error('cdmuhl:uh:InputSizeMismatch', ...
    'XI and YI must be the same size.');
end
f = zeros(length(xi), 1);
t = mytsearch(nodesx, nodesy, tri, xi, yi);
for i = 1:length(xi)
    if (~isnum(t(i)))
        v = tri(t(i), :);
        [tf, loc] = ismember(v, inidx);
        locs = loc(tf);
        for j = 1:length(locs)
            f(i) = f(i) + uvec(locs(j)) * tentfunc(nodesx, nodesy, tri, ...
            locs(j), coefs, xi(i), yi(i));
        end
    end
end
end
end

IV. PROCEDURES

Boundary data for the state of Maryland was obtained in Shapefile format from the National Atlas [3]. All islands belonging to the state were ignored, leaving a single polygon. The 7,744 coordinates for this polygon were exported to a .poly file representing Maryland’s Planar Straight Line Graph (PSLG). Jonathan Shewchuk’s Triangle program [7] was then used to generate a conforming Delaunay triangulation for Maryland with a minimum angle of 20° and a maximum area of 0.001 (MATLAB’s built-in triangulation routine requires that the PSLG be convex and does not allow for such quality constraints). The resulting mesh is shown in figure 1. It is composed of 17,153 triangles and 4,545 interior nodes (out of 12,610 total nodes).
The mesh was then loaded into MATLAB, and the TrianglesByPoints data structure, the tent function coefficients, and the stiffness matrix were all generated, as they depend only on the boundary conditions and the mesh structure and not on the charge density function.

The first charge density considered was one inversely proportional to the distance between a point and my house, located in Ellicott City, Maryland. This distance is measured in degrees and is proportional to the linear distance (with a proportionality constant of $\pi R/180$, where $R$ is the radius of the Earth), as all angles being considered were small ($< 3^\circ$). An approximate $\mathbf{q}$ vector was computed for this function, and the solution vector $\mathbf{u}$ was solved for using the stiffness matrix $A$. This solution vector was then used to compute the solution function on a mesh covering the state. A plot of the solution is shown in figure 2.

All other charge densities considered were based on county statistics. The density functions were constant over each county, equal to the value of the statistic in question for that county. The boundary coordinates for Maryland’s counties (and Baltimore City) were obtained from the National Atlas in Shapefile format and converted to vectors of $x$ and $y$ coordinates to be imported into MATLAB. Once again, all islands were ignored, so each county consisted of a single connected polygon. These regions are shown in figure 3.

The first statistic considered was county population (2005 estimate) as reported by the U.S. Census Bureau. Note that the charge density is not proportional to the population density in this scenario. Rather, it is proportional
FIG. 3: Polygons representing Maryland’s counties (and Baltimore City), ignoring all islands.

FIG. 4: Colored contour plot of the solution to Poisson’s equation when charge density is constant across each county, proportional to each county’s population.

to the total population of each county across the entire county. The voltage resulting from such a charge density is plotted in figure 4.

Next, charge density was made proportional to each county’s 1999 per capita money income, also reported by the U.S. Census Bureau [4]. The resulting voltage is plotted in figure 5.

Finally, the results of the 2006 senatorial election were considered, as reported by CNN [5]. For each county, the percentage of voters who voted for Ben Cardin was taken and subtracted by 50%. This effectively represents the extent to which each county leans Democratic (positive values) or Republican (negative values). Since several counties tend to lean Republican, this charge distribution is the only one among those considered to include both positive and negative charges. A plot of the resulting voltage is shown in figure 6.
FIG. 5: Colored contour plot of the solution to Poisson’s equation when charge density is constant across each county, proportional to each county’s mean income.

FIG. 6: Colored contour plot of the solution to Poisson’s equation when charge density is constant across each county, proportional to each county’s Democratic or Republican leanings.

V. ANALYSIS

There are striking similarities between the solutions to all four charge distributions, but it is difficult to determine whether these similarities are due to restrictions imposed by the boundary conditions or rather are simply reflecting socio-economic correlations in the state of Maryland. In all cases the voltage peaks in the center of the state, near Howard and Montgomery counties. On the one hand, this region is least affected by boundary conditions, since it is contained in the largest uninterrupted area in the state. Other local maxima and minima also occur in regions of large uninterrupted area, such as in the northwest and southeast. In terms of the locations of extrema, this seems to indicate that the boundary conditions have a greater impact on the solution than the charge density function.

However, analysis of central Maryland is confounded by correlations between the statistics analyzed. The fact that the first voltage considered peaks in Howard County is coincidental, since that is where the author’s house happens
to be. Had the house been located on the Eastern Shore, the voltage likely would not have peaked in Howard County. The similarity between the three other distributions is more socio-economic in nature. Baltimore County, Baltimore City, Montgomery County, and Prince George’s County have by far the largest populations in the state, and all are located in central Maryland. In terms of income, Howard County and Montgomery County are significantly higher than the rest of the state. Finally, Baltimore City, Montgomery County, and Prince George’s County are the most Democratic counties in the state. The fact that all of these statistics peak in central Maryland makes the relative effects of boundary conditions and charge densities on the solution difficult to discern.

VI. CONCLUSIONS

This study has demonstrated that it is possible to rapidly implement the finite element method for solving Poisson’s equation for arbitrary boundaries. The results obtained are continuous functions whose features can be traced back to both the boundary conditions and the charge distributions. While the solutions obtained are piecewise linear approximations, for fine enough meshes they are reasonably accurate. Furthermore, even for very fine meshes such as the one used in this study, the computation time is quite reasonable.

For future work, the requirement that the boundary be grounded could be relaxed, resulting in a different choice of basis functions $\varphi_j$. Also, the method could be re-implemented in a lower-level language to take advantage of the parallel processing capabilities of modern computers. Alternately, the existing code could be used to study other charge distributions over Maryland or any other state. In fact, the current implementation would probably work even for disconnected regions (though this has not been tested). This would allow Maryland’s many islands to be included in the analysis as well.

Finally, one could calculate the electric field produced from these voltages, which would point along the normal vectors to the contours shown above. The trajectories of test charges could then be modeled and plotted.

APPENDIX A: TECHNICAL NOTES

All MATLAB commands were evaluated in 64-bit MATLAB R2006a for x86_64 Linux. This was run on a 2.2 GHz AMD Athlon64 3500+ CPU with 4 GB of RAM operating under Gentoo Linux with a 2.6.18 kernel. ESRI Shapefiles were imported using OpenJUMP 1.0.1. Triangulation meshes were generated using Triangle 1.6 compiled from source.

APPENDIX B: CHARGE DENSITY FUNCTIONS

function q = qHomeOverR(x, y)
    %QHOMEOVERR 1/r function centered at home
    % q = QHOMEOVERR(X,Y) returns the value of 1/r, where r is the distance
    % between my house and the point (X,Y). The distance is computed in the
    % approximation that latitude and longitude form a locally Cartesian
    % coordinate system.
    % AUTHOR: Curran D. Muhlberger <cdmuhlb@umd.edu>
    % DATE: 2006-12-10

    if (nargin ~= 2)
        error('cdmuhlb:qHomeOverR:NotEnoughInputs', 'Needs 2 inputs.');
    end
    if (~isequal(size(x), size(y)))
        error('cdmuhlb:qHomeOverR:InputSizeMismatch', 'X and Y must be the same size.');
    end
    q = 1 ./ sqrt((x + 76.840464).^2 + (y - 39.276235).^2);
end

function q = countyPop(x, y, counties)
    %COUNTYPOP Population by county
    % q = COUNTYPOP(X,Y,COUNTIES) returns a vector containing populations
    % (2005 estimate) for the Maryland counties containing each point (X,Y)
    % where X is a vector of longitudes and Y is a vector of latitudes. If
    % any point (X,Y) is not located within a Maryland county, the function
% returns zero at that point.

% AUTHOR: Curran D. Muhlberger <cdmuhl@umd.edu>
% DATE: 2006-12-10

if nargin ~= 3
    error('cdmuhl:countyPop:NotEnoughInputs', 'Needs 3 inputs.');
end
if ~isequal(size(x), size(y))
    error('cdmuhl:countyPop:InputSizeMismatch', 'X and Y must be the same size.');
end
q = zeros(size(x));
q( inpolygon(x, y, counties.allegany_x, counties.allegany_y) ) = 73639;
q( inpolygon(x, y, counties.anne_arundel_x, counties.anne_arundel_y) ) = 510878;
q( inpolygon(x, y, counties.baltimore_x, counties.baltimore_y) ) = 786113;
q( inpolygon(x, y, counties.baltimore_city_x, counties.baltimore_city_y) ) = 628670;
q( inpolygon(x, y, counties.calvert_x, counties.calvert_y) ) = 87925;
q( inpolygon(x, y, counties.caroline_x, counties.caroline_y) ) = 31822;
q( inpolygon(x, y, counties.carroll_x, counties.carroll_y) ) = 168541;
q( inpolygon(x, y, counties.cecil_x, counties.cecil_y) ) = 97796;
q( inpolygon(x, y, counties.charles_x, counties.charles_y) ) = 138822;
q( inpolygon(x, y, counties.dorchester_x, counties.dorchester_y) ) = 31401;
q( inpolygon(x, y, counties.frederick_x, counties.frederick_y) ) = 220701;
q( inpolygon(x, y, counties.garrett_x, counties.garrett_y) ) = 29099;
q( inpolygon(x, y, counties.harford_x, counties.harford_y) ) = 239259;
q( inpolygon(x, y, counties.howard_x, counties.howard_y) ) = 269457;
q( inpolygon(x, y, counties.kent_x, counties.kent_y) ) = 19899;
q( inpolygon(x, y, counties.montgomery_x, counties.montgomery_y) ) = 927583;
q( inpolygon(x, y, counties.prince_georges_x, counties.prince_georges_y) ) = 846123;
q( inpolygon(x, y, counties.queen_x, counties.queen_y) ) = 45612;
q( inpolygon(x, y, counties.somersett_x, counties.somersett_y) ) = 25845;
q( inpolygon(x, y, counties.st_marys_x, counties.st_marys_y) ) = 96518;
q( inpolygon(x, y, counties.talbot_x, counties.talbot_y) ) = 35683;
q( inpolygon(x, y, counties.washington_x, counties.washington_y) ) = 141895;
q( inpolygon(x, y, counties.wicomico_x, counties.wicomico_y) ) = 90402;
q( inpolygon(x, y, counties.worcester_x, counties.worcester_y) ) = 48750;
end

function q = countyIncome(x, y, counties)
% COUNTYINCOME Per capita income by county
% q = COUNTYINCOME(X,Y,COUNTIES) returns a vector containing the per capita money incomes (1999) for the Maryland counties containing each point (X,Y) where X is a vector of longitudes and Y is a vector of per capita income by county.
% The function returns zero at that point.

% AUTHOR: Curran D. Muhlberger <cdmuhl@umd.edu>
% DATE: 2006-12-10

if nargin ~= 3
    error('cdmuhl:countyIncome:NotEnoughInputs', 'Needs 3 inputs.');
end
if ~isequal(size(x), size(y))
    error('cdmuhl:countyIncome:InputSizeMismatch', 'X and Y must be the same size.');
end
q = zeros(size(x));
q( inpolygon(x, y, counties.allegany_x, counties.allegany_y) ) = 16780;
q( inpolygon(x, y, counties.anne_arundel_x, counties.anne_arundel_y) ) = 27578;
q( inpolygon(x, y, counties.baltimore_x, counties.baltimore_y) ) = 26167;
q( inpolygon(x, y, counties.baltimore_city_x, counties.baltimore_city_y) ) = 16978;
q( inpolygon(x, y, counties.calvert_x, counties.calvert_y) ) = 25410;
q( inpolygon(x, y, counties.caroline_x, counties.caroline_y) ) = 17279;
q( inpolygon(x, y, counties.carroll_x, counties.carroll_y) ) = 23829;
q( inpolygon(x, y, counties.cecil_x, counties.cecil_y) ) = 21384;
q( inpolygon(x, y, counties.charles_x, counties.charles_y) ) = 24285;
q( inpolygon(x, y, counties.dorchester_x, counties.dorchester_y) ) = 18929;
q( inpolygon(x, y, counties.frederick_x, counties.frederick_y) ) = 25040;
q( inpolygon(x, y, counties.garrett_x, counties.garrett_y) ) = 16219;
function q = countySenate(x, y, counties)
% COUNTYSENATE Senate election results by county
% q = COUNTYSENATE(X,Y,COUNTIES) returns a vector containing percentage
% above 50 of voters who voted for the Democratic party in the 2006
% senate election in the Maryland counties containing each point (X,Y)
% where X is a vector of longitudes and Y is a vector of latitudes. If
% any point (X,Y) is not located within a Maryland county, the function
% returns zero at that point.

% AUTHOR: Curran D. Muhlberger <cdmuhlb@umd.edu>
% DATE: 2006-12-10

if nargin ~= 3
    error('cdmuhlb:countySenate:NotEnoughInputs', 'Needs 3 inputs.');
end

if ('isequal(size(x), size(y))')
    error('cdmuhlb:countySenate:InputSizeMismatch', 'X and Y must be the same size.');
end

q = zeros(size(x));
q = inpolygon(x, y, counties.harford_x, counties.harford_y));
q = inpolygon(x, y, counties.howard_x, counties.howard_y));
q = inpolygon(x, y, counties.kent_x, counties.kent_y));
q = inpolygon(x, y, counties.montgomery_x, counties.montgomery_y));
q = inpolygon(x, y, counties.prince_georges_x, counties.prince_georges_y));
q = inpolygon(x, y, counties.queen_annes_x, counties.queen_annes_y));
q = inpolygon(x, y, counties.somerset_x, counties.somerset_y));
q = inpolygon(x, y, counties.st_marys_x, counties.st_marys_y));
q = inpolygon(x, y, counties.talbot_x, counties.talbot_y));
q = inpolygon(x, y, counties.washington_x, counties.washington_y));
q = inpolygon(x, y, counties.wicomico_x, counties.wicomico_y));
q = inpolygon(x, y, counties.worcester_x, counties.worcester_y));

end


