Computer Science Master's Project Report
Rensselaer Polytechnic Institute
Troy, NY 12180

Development of A Two-Level Iterative Computational Method for Solution of the Franklin Approximation Algorithm for the Interpolation of Large Contour Line Data Sets

Submitted by

John Childs

In partial fulfillment of the requirements for the degree of Master of Science in Computer Science

May, 2003

Approved by:

Project Advisor

_____ Date: _____

Table of Contents

Abstract

A method was developed to solve the over determined Laplacian approximation algorithm (Franklin algorithm) for the interpolation of contour lines to terrain surfaces using desktop type computers. The method is based on the Saunders-Paige LSQR iterative Conjugate Gradient solver provided with a very good initial estimate. The initial estimate is computed from the input elevation grid using a fast solver producing a lower quality interpolation of the input grid. The fast solver uses a four-nearest-neighbor second order central difference approximation of the Laplacian heat equation. The solution vector produced by the fast interpolation is used as the initial estimate for the LSQR solver. LSQR is then used to solve a sparse matrix representation of the over determined system of equations resulting from the Franklin formulation of the Laplacian heat equation interpolation. The result is a high quality interpolation characteristic of the Franklin approximation in less time and requiring much less storage than either a direct solution (using factorization or row reduction) or a solution computed by the LSQR solver without a good initial estimate.

This is useful because solution of the over determined system of linear equations associated with the Franklin algorithm by direct means is computationally expensive from both a time and space complexity perspective. The method developed as part of this project significantly reduces the in-process storage requirements to compute the solution vector for large systems of equations, for example one million equations with one million unknowns. The method accomplishes this by use of an iterative least squares solver, the well known LSQR program developed by Saunders and Paige.

The iterative LSQR solver makes more effective use of limited RAM resources as compared to the MATLAB direct solver for example, allowing larger problems to be solved than with previous methods. In addition, solution times were reduced by supplying LSQR with a high quality initial estimate. The fast central difference Laplacian algorithm exploits the problem structure to supply this estimate cheaply. The fast solver produces a solution of unacceptable quality as a final interpolation but produces an excellent initial estimate for LSQR.

Acknowledgements

I would like to thank my project advisor, Dr. William Randolph Franklin, RPI Troy, for generating the idea for this project and for his help, advice and support. I feel extremely grateful for the opportunity to spend a semester working on a project in a field of great personal interest under his direction. Dr. Franklin is a true expert in the field of computational cartography, applied mathematics, computational geometry as well as being a devoted teacher to his students.

I would also like to thank Dr. M. A. Saunders of Stanford University for his key advice concerning application of his iterative Conjugate Gradient solver. He took time from his busy academic schedule to respond to my enquiry in a very helpful and detailed fashion. This advice proved essential to whatever small success this project may have achieved.

Dr. Gilbert Strang's Online Linear Algebra lectures at the MIT web site were very useful in getting me up to speed with the mathematics required for this project.

I would also like to thank Jackie Bassett for proofreading this document and for offering editorial advice.

1.0 Introduction.

Background

Interpolating information from known data to unknown data points is one of the classical problems of computational cartography. This technique is used widely in many diverse GIS (Geographical Information Systems) applications. One example is the derivation of DEMs from topographical contour maps for areas where DEMs are not available. In order to do this, DEMs are often reverse-engineered from contour map data. In this multi step process, contour maps are scanned into digital format (if they are not already archived that way). Then the contour line elevation data layer is separated from the other information on the map. The contour line data is then conditioned (to the highest degree possible) to correct bleeding, breaks, spurs bridges and other anomalies[1].

The contours must then be tagged with their elevation values in a machine-readable format. This is typically done by vectorizing the contour lines using a line following method[2] and tagging the vectors with their elevations either automatically, or more commonly using a semi-automated labor-intensive process. The tagged vector data is then rasterized and transferred to a grid using an interpolation algorithm. Finally, the gridded elevation values are written to some type of GIS format that can be used by other applications, such as the USGS ASCII DEM format. The interpolative step in the process is the focus of this proposal. Often, the interpolation algorithms used to produce terrain surfaces from the gridded elevation values are somewhat primitive, for example the four-nearest neighbor algorithm derived from the Laplacian Heat Equation PDE. This often results in markedly inferior terrain surfaces that exhibit a variety of unnatural features, for example terracing.

One algorithm known to be an improvement over commonly used algorithms for the interpolation (or rather the approximation) of known elevations to a regular grid is described by Dr. William Randolph Franklin paper entitled *Applications of Analytical Cartography* [3].

The algorithm is based on a novel application of the Laplacian PDE (partial differential equation) whereby a system of over determined linear equations is formulated and solved by performing a linear least squares fit to the known data and unknown grid nodes.

The algorithm has several advantages as compared to previous algorithms:

1) It doesn't require continuous contours; i.e. it can deal with breaks, such as those that commonly occur when contours are too closely spaced.

2) It can use isolated point elevations if known, such as mountain tops.

3) If continuous contours with many points are available, it can use all the points (without needing to sub sample them).

4) Information flows across the contours, causing the slope to be continuous and the contours not to be visible in the generated surface.

5) If local maxima, such as mountain tops, are not given, it interpolates a reasonable bulge. It does not just put a plateau at the highest contour.

6) Unlike some other methods, it works even near kidney-shaped contours, where the closest contour in all four cardinal directions might be the same contour.

7) It has the advantage of producing a surface virtually free of the negative artifacts associated with other interpolation algorithms, for example terracing and ringing.

8) It also offers the advantage of allowing the surface smoothness and fit to be adjusted by selective weighting of the equations corresponding to the known elevations, thereby allowing grid adjustments to be made on a node-by-node basis if desired.

Although the algorithm presents advantages, it also presents computational challenges. If the input elevation grid is of size N by N, the solution is formulated in terms of a N^2 by N^2 coefficient matrix. Since the number of flops for row reduction for an N by N matrix is approximately $2N^3/3$ flops, the time complexity for solving an N^2 by N^2 matrix by row reduction is $(N^2)^3 = N^6$. Even more significantly, the initial storage requirements are large for large input files and these can become even larger during processing due to fill-in during computation.

The Franklin algorithm was previously demonstrated on data sets as big as 257 X 257 nodes using sparse matrix techniques. Although this was sufficient to demonstrate the advantages of the algorithm, real data sets are larger. Unfortunately, it has not been possible previously to apply this beneficial algorithm to realistic data sets using desktop computers because of the large computational complexity of solving the linear least squares system of equations by row reduction.

The goal of this project is to develop a computational technique that would allow the algorithm to be used for realistic data sets using small computers. The target size is 1201 by 1201 grid postings, the size of a USGS 30 minute Level 2 Digital Elevation Model. The task is made difficult by the rapid increase in problem size as a function of the number of input elevation nodes. For example, the full matrix for the Franklin formulation of a 1201 by 1201 elevation node input file contains approximately 2e+12 data elements!

A further goal was to compare the quality of the grids produced by the algorithm (once a computational technique was developed) with a commercially implemented algorithm. The hope is thus to apply Dr. Franklin's algorithm to large data sets using a standard desktop computer and to further demonstrate its utility to the cartographic community.

Objectives

The objective of this research project are as follows:

- Develop a computational method for solution of the algorithm described in the Dr. William Randolph Franklin paper cited above.

- Demonstrate the computational method using a desktop-type computer solving datasets of sizes associated with commonly used digital elevation model (DEM) formats (for example input files of 1201X1201 elevation grid nodes). The developmental hardware platform is a desktop computer running the Windows XP OS with 256MB of memory. (An HP 1.1GHz laptop was used for some tests.)

- Compare the results of the algorithm and computational method with a commercial contour-to-grid interpolator: R2V from Able Software, Inc. and with a research grade hydrologic application: CachmentSIM-GIS.

Although a fair amount of code was written during the course of this project, the objective was not to produce a production grade software application. Rather, the goal was to develop a computational method that was capable of solving this problem for large input data sets through the use of code modules using several developmental environments, including the research environment MATLAB. The integration of these experimental modules into a robust application provides an interesting opportunity for future work. The main tasks required for achieving this future goal are discussed in Section 14 of this report.

Development Environments Used

Several development environments were used during the course of this work. The first was the well-known scientific math package MATLAB. MATLAB is an interactive developmental and research programming platform designed specifically for matrix processing that offers a FORTRAN-like procedural programming language, a library of matrix functions, and a wide variety of graphical visualization tools. Although MATLAB is not a general-purpose development environment, demonstrating the algorithm using MATLAB modules immediately suggests corresponding implementations using production-grade software tools such as C++, FORTRAN or Java.

Although MATLAB is a wonderfully convenient development environment because of its rich library of specialized matrix processing, matrix solving and display methods, it is definitely not an all-purpose tool. Its main disadvantage is that user-defined code modules are interpreted rather than compiled, and typically run several orders of magnitude more slowly than identical algorithms implemented in C++ or Java, for example. This required the use of alternative development environments for some of the fast pre-processing tasks required for this project, particularly for the module used to prepare the initial estimate for the iterative solver used in this project and for the module that prepared the sparse index file from the raw elevation matrix.

2.0 Theoretical and Mathematical Background of the Problem

A brief discussion of the theoretical basis for the Franklin approximation algorithm and the method of problem formulation and solution is presented in this section. The Laplace heat equation:

$$\delta^2u/\delta x^2 + \delta^2u/\delta y^2 = 0$$

describes many time-independent physical phenomenon including the steady state distribution of temperature in two dimensions. This equation can be solved in closed form for simple systems subject to boundary conditions consisting of for example initial temperatures at the edges of a rectangular plate.

The Laplace equation has also been used to model equilibrium displacements in a membrane, gravitational and electrostatic potentials and certain fluid flows. It is not surprising therefore that it is also applied to topographical systems. In this case elevation is considered a potential analogous to temperature. The boundary conditions consist of the known elevations at the elevation contours (not necessarily at the edges as is common in a heat conduction model) and the equation models the "flow" of elevations from the fixed contours to the rest of the system in the same way that heat flows in the plate example[7].

Central Difference Approximation of the Laplace Equation

Numerical solution methods are often used in systems where closed form solutions of differential equations are difficult or impractical. The central difference formulas can be used to approximate the second derivative of a function f(x)[4] as follows. Consider the second order Taylor series approximation of a function f(x) around the point 'a':

$$T(x) = f(a) + f'(a)(x-a) + \tfrac{1}{2} f''(a)(x-a)^2 + O(n^3)$$

Where $O(n^3)$ is an error term resulting from the truncation of the series. Consider a point slightly greater than x so that $x-a=\Delta x$ or $a=x-\Delta x$. Substituting into the Taylor expansion around this point yields:

$$T(x) = f(x-\Delta x) + f'(x-\Delta x)(x-(x-\Delta x)) + \tfrac{1}{2} f''(x-\Delta x)(x-(x-\Delta x))^2 + O(x^3)$$

$$= f(x-\Delta x) + f'(x-\Delta x)(\Delta x) + \tfrac{1}{2} f''(x-\Delta x)(\Delta x)^2 + O(x^3)$$

$$T(x+\Delta x) = f(x+\Delta x -\Delta x) + f'(x+\Delta x -\Delta x)(\Delta x) + \tfrac{1}{2} f''(x+\Delta x -\Delta x)(\Delta x)^2 + O(x^3)$$

$$= f(x) + f'(x)(\Delta x) + \tfrac{1}{2} f''(x)(\Delta x)^2 + O(x^3)$$

A similar substitution for a Taylor approximation around a point a little less than x where $x-a = -\Delta x$ or $\Delta x = x+a$ yields:

$$T(x-\Delta x) = f(x) - f'(\Delta x)\,\Delta x + \tfrac{1}{2} f''(x)(\Delta x)^2 + O(x^3)$$

Adding the two equations together gives:

$$T(x+\Delta x) + T(x-\Delta x) = 2f(x) + (\Delta x)^2 f'''(x) + O(x^3)$$

If Δx is small the $O(x^3)$ error term can be ignored, and T(x) is very close to f(x). Substituting $T(x) \approx f(x)$ and Solving for f''(x):

$$f''(x) = f(x+\Delta x) + f(x-\Delta x) - 2f(x)/ (\Delta x)^2$$

If f is a continuously differentiable function mapping two dimensional coordinates x,y to elevation f(x,y)=z then

$$\delta f(x,y)/\delta x^2 \approx f(x+\Delta x,y) + f(x-\Delta x,y) - 2f(x,y)/ (\Delta x)^2$$

A similar expansion in the y direction yields:

$$\delta f(x,y)/\delta y^2 \approx f(x, y+\Delta y) + f(x, y-\Delta y) - 2f(x,y)/ (\Delta y)^2$$

If $\Delta x = \Delta y$ then both of these terms can be replaced with Δ. Substituting these two equations into the Laplace equation yields:

$$f(x+\Delta,y) + f(x-\Delta,y) - 2f(x,y)/ (\Delta)^2 + f(x, y+\Delta) + f(x, y-\Delta) - 2f(x,y)/ (\Delta)^2 = 0$$

Multiplying by Δ^2 and collecting terms yields the central difference equation:

$$f(x+\Delta,y) + f(x-\Delta,y) + f(x, y+\Delta) + f(x, y-\Delta) - 4f(x,y) = 0$$

The continuous function f is often approximated by a set of points on a grid or mesh. In this case the central difference equation is defined at the grid points. The discrete central difference equation can be written:

$$f(i+1,j) + f(i-1, j) + f(i, j+1) + f(i, j-1) - 4f(i,j) = 0$$

If f(i,j) maps grid points to an elevation z then the discrete central difference equation can be written in the more concise notation:

$$z_l + z_r + z_u + z_{ld} - 4\, z_{ij} = 0$$

Where the indices refer to the grid nodes to the immediate left, right, up and down grid positions relative to the central node.

Applications to Contour Interpolation

Franklin[1] uses the central difference approximation to formulate this problem by constructing a system of linear equations:

$$z_{i,j-1} + z_{i,j+1} + z_{i+1,j} + z_{i-1,j} - 4\, z_{ij} = 0$$

for each node in the input elevation grid. (This is sometimes referred to as the stencil equation). Since there are N^2 grid points this yields an equal number of equations, each with N^2 (mostly zero) coefficients. This in turn yields a coefficient matrix with N^4 total coefficients. Since each row is characterized by an equation with exactly five non zero coefficients as in the equation above, each row of this sparse matrix has N^2 entries, of which (N^2–5) are zeros. The application of a linear equation solving algorithm yields the elevations directly, as long as N is not too large.

However, solving this system produces terrain surfaces exhibiting excessive terracing between the known contours in areas where the lower contour lines are longer than the higher ones, because the lower elevation contour lines will contribute more toward the new center node elevation than the higher elevations. Also, many of the grid points corresponding to contours have known elevations. If the solution vector is constrained to maintain these fixed elevations, the surface will not be continuous across the contours and considerable terracing will result. If the solution vector is not so constrained, inaccurate elevations can occur.

The central difference approximation of the Laplace equation as formulated above can also be solved by iterative methods. For example the central difference approximation can be applied to each unknown node in the elevation grid in successive computational passes using Jacobi or Gauss-Seidel iterative solvers. At each iteration a new elevation is computed for the central node. The greater the number of iterations, the closer the approximation becomes to the exact solution (if the method converges). Although fast, efficient and compact, the result is no better than that produced by direct solution as the underlying formulation of the problem is the same. However, this technique subsequently proved valuable in the multi-level solution approach that was used to solve the system.

A Better Formulation of the Problem

Franklin suggests a different formulation of the system of equations, as follows:

1) Pretend that all the N^2=M points have unknown elevations z_{ij} .
2) Create an equation for each z_{ij} setting it to the average of its neighbors as in the equation above.
3) For each of the K points whose elevation e_i are known create an additional equation $z_i = e_i$.

This results in the system of equations:

$$Az=b$$

where A is an (M+K) by M coefficient matrix, z is an M by 1 vector, and b is a (M+K) by 1 vector of zeros or known elevation values.

This system is over determined, so a solution exists only when b happens to be in the column space of A, which is obviously unlikely.

So rather than attempting to solve the system exactly (and unsuccessfully) a linear least squares solution can be applied instead. This technique minimizes the vector:

$$(b-Az)^T \ (b-Az)$$

Where Az is the (projection of b) = b_{proj} on the column space of the coefficient matrix A and $(b-Az)^T$ is the transpose of the vector quantity $(b-Az)$. The solution z to the matrix equation $Az = b_{proj}$ is the vector in the column space of A that is closest to b, and is the "best" solution to the (probably unsolvable) system Az=b. Since $(b-Az) = b - b_{proj} = e$ is minimized precisely when e is orthogonal to A, then, in matrix terms:

$$A^T(b-Az)\ z=0.$$

minimizes e, which is equivalent to:

$$A^TAz=A^Tb$$

where A^TA is a square, symmetric (M^2+K by M^2+K) matrix[5] . This is called the linear least squares solution of the problem because it minimizes the sum of the squares of the error vector e. This system has been solved by Franklin for data sets of up to 257 by 257 nodes using MATLAB sparse matrix utilities and solvers. However, this problem is considerably larger than the direct solution described above by row reduction because of the additional matrix multiplications required. More importantly, the increase in space complexity due to the need to store more large data arrays leads to significant computational difficulties.

Franklin's work demonstrated that this formulation can virtually eliminate the disadvantages associated with a direct application of the central difference equation. The Franklin formulation does not constrain the solution vector to maintain the known elevations but does in fact constrain the grid points corresponding to the known elevations to be the average of their four neighbors. In addition, since the method minimizes the squares of the error components, scaling the coefficient and the known elevation results in a weighting of that elevation. This aspect, considered a problem in statistical regression application is an advantage here as it allows the user to constrain or relax the solution vector's conformance to the known elevations, thereby balancing surface accuracy and smoothness.

The main problem with this approach is the time and space complexity for computation of the solution vector z, particularly on desktop (vector) systems. Successfully overcoming the computational obstacles was the main focus of this project.

3.0 Preliminary Investigations

MATLAB Code Modules and their Limitations

A series of MATLAB code modules was written and used throughout this study to investigate various solution options. The MATLAB function sparseA( ) was written to formulate the sparse coefficient matrix 'A' from the input elevation node matrix using the Franklin formulation described above. The function makeB( ) formulated the right hand side (RHS) vector 'b' from the same input elevation node matrix. The function repack( ) converted the N^2 by 1 solution vector z into a N by N output elevation grid. Once formulated, the system could be solved using the MATLAB native direct solver by invoking z=A\b. An over determined case like the Franklin formulation is detected automatically by MATLAB and a least squares solution is formulated and computed. Code listings for these routines are attached in Appendix 1.

Experiments with these utilities and the native MATLAB least squares solver indicated that the largest problem that could be solved on an eMachines T4200 2.0GHz processor with 256MB RAM running the Microsoft XP operating system was about 400 by 400 elevation nodes. This yielded a system of 165685 equations and unknowns. An input elevation matrix of 500 by 500 nodes caused an out of memory processor error.

Experiments with Alternative MATLAB Solvers

The first tests associated with this project examined some alternative MATLAB tools for solving systems of linear equations to see of they offered any computational efficiencies. Straightforward solution of the linear least squares equation solve the equation:

$$A^T A z = A^T b$$

by computing $A^T A$, then $A^T b$, and then solving for z using elimination. Alternatively, the system can be solved by first performing a QR factorization of A. The classical solution approach[6] uses the following steps:

1) Compute Q
2) Compute R using $R=Q^T A$
3) Solve for z using $Rz = Q^T b$

The MATLAB documentation states that for sparse matrices, Q is mostly full. Since Q is an m by n matrix, this would be unfavorable from a computational perspective. The MATLAB documentation suggests computing

$$[C,R] = qr(A,B)$$

for sparse matrices, applying the orthogonal transformations to B, producing $C = Q^{-1}*B$ without computing Q. For sparse matrices, the Q-less QR factorization allows the solution of sparse least squares problems with two steps:

$$[C,R] = qr(A,b)$$
$$x = R\backslash c$$

This method was used to solve an experimental system of equations. The tests indicated that the computational time and complexity was identical to that produced by applying the MATLAB command z=A\b. It is considered likely that the above step wise method is used by MATLAB when A\b is invoked.

4.0 MATLAB Sparse Matrix Reordering Experiments

When certain operations such as matrix multiplication and particularly the factorizations used to solve linear systems are conducted on sparse matrices the density of the matrices and thus the storage requirements can increase dramatically. This is called fill-in. It has been known for some time that the row and column order of a sparse matrix can have a large effect on fill-in during these computations. It has been proven that the computation of optimal ordering is NP-hard. However, there are several heuristics commonly used to improve ordering for this purpose.[7]

An investigation was conducted to determine if the computation of the over determined Laplacian approximation algorithm (Franklin algorithm) could be improved by using alternative row or column orderings of the large sparse coefficient matrix. The hope was that the improvement would be enough to allow processing of files of the target size. A small test system was built and experiments with the reordering schemes provided by MATLAB were conducted and noted.

Permutation Vectors

Consider an n by n square matrix A and a 1 by n permutation vector p such that

$$p=[i_1, i_2, ...i_n] \ , \ i \in I$$

The MATLAB matrix permutation operator can produce a row or column permuted matrix P:

$$P_{row}=A(p, :)$$

and

$$P_{col} = A(: ,p)$$

where P_{row} is the matrix with rows permuted according to the encoding in the permutation vector p, i.e.

$$P_{row} = \begin{bmatrix} row_{i1} \\ row_{i2} \\ \cdot \\ \cdot \\ \cdot \\ row_{in} \end{bmatrix}$$

and

$$P_{col}=[\ col_{i1} \quad col_{i2} \quad \ldots \quad col_{in}\ ]$$

The inverse operation can be performed with the vector q computed from the MATLAB command:

$$q(p) = 1:n$$

The inverse operation is then:

$A = P_{row}(q, :)$

for the row permutation or

$$A=P_{col}(:, q)$$

for the column permutation.

It is also possible to permute the linear system

$$Az = b$$

as long as the permutations are consistent. That is, it is possible to perform row or column permutations on an n by m matrix A as long as the corresponding permutation is performed on either vector z or vector b. For example, to perform a column permutation defined by 1 by n permutation vector p on matrix A, first permute the matrix:

$$P_{col}=A(:, p)$$

Then solve the linear system:

$$P_{col}\, z = b$$

Then compute the inverse permutation vector q:

$$q(p) = 1:n$$

and then compute z by re-applying the permutation vector:

$$z = z(q, :)$$

This technique is valid for an over determined system as well.

Test System

Various reordering schemes were applied to a 200 by 200 node input elevation grid consisting of straight rows of contour lines running the width of the map. A small Java utility called InputMatrix.java was used to build this and similar input files. The grid was imported into MATLAB as elevation matrix 'A'.

Test 1 – Baseline

The first test established the processing time for an unordered MATLAB sparse elevation matrix. Actually, as a result of the way that sparseA.m builds the input matrix and due to the structure inherent in the problem, the unordered sparse matrix has a very high degree of structure. This structure tends to place the non zero elements on the diagonal, which in fact is one of the heuristics used for certain reordering schemes (see below). As

a result, it is expected that the unordered sparse matrix may actually be closer to optimal than to extremely sub-optimal.

The utilities sparseA.m, makeB.m, and repack.m were used to prepare the input matrices. The program sparseA.m computed a 43,800 by 40,000 sparse coefficient matrix. The characteristics of the computed sparseMatrix can be seen from the MATLAB spy plot in Figure 1. below.

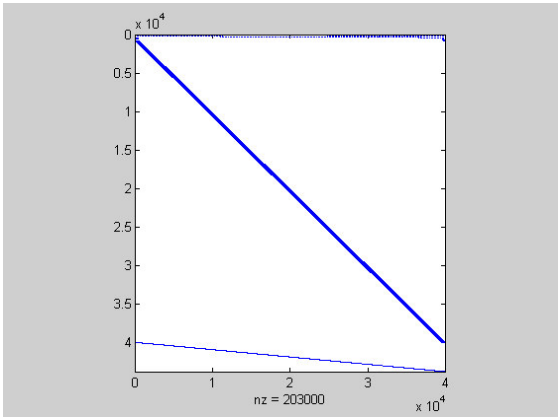

Figure 1. Unordered Sparse Matrix

The program makeB.m computed a 40,000 by 1 right hand side (RHS) vector. The system was then solved using the MATLAB command:

t1=clock; z=sparseMatrix\b; t2=clock; e=etime(t2, t1)

The elapsed time e was 197.86s on a Hewlett Packard 1.1MHz laptop.

Test 2 – colmmd

colmmd is a MATLAB column reordering function returning a permutation vector

p = colmmd(S)

such that p is a minimum degree permutation vector for S.

The permutation vector 'p' was calculated for sparseMatrix and then used to permute sparseMatrix to colmmdMatrix. The MATLAB spy plot for colmmdMatrix is shown in Figure 2.

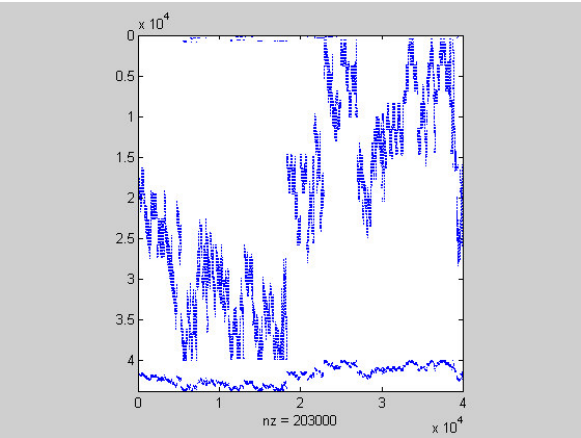

Figure 2. colmmd Reordered Sparse Matrix

The MATLAB colmmdMatrix\b command was then used to solve z. The elapsed time was 221.87s, definitely not an improvement.

Test 3 - Non Zero Vector Sort

The next test was a column sort by number of non zero elements. The permutation vector was computed by:

p = colperm(sparseMatrix)

Then sparseMatrix was permuted as before. The MATLAB spy for the matrix colpermMatrix is shown in Figure 3 below.

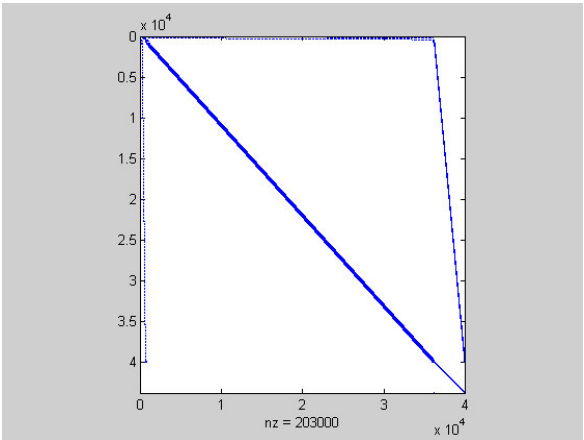

Figure 3. colperm Reordered Sparse Matrix

The computation elapsed time in this case was 174.27s, a slight improvement.

Test 4 – colamd

The MATLAB function colamd returns a permutation based on an alternative minimum degree ordering algorithm. As before, a permutation vector was computed by:

p = colamd(sparseMatrix)

and as before a matrix colamdMatrix was computed by permuting sparseMatrix with p. The MATLAB spy for this reordering is shown in Figure 4.

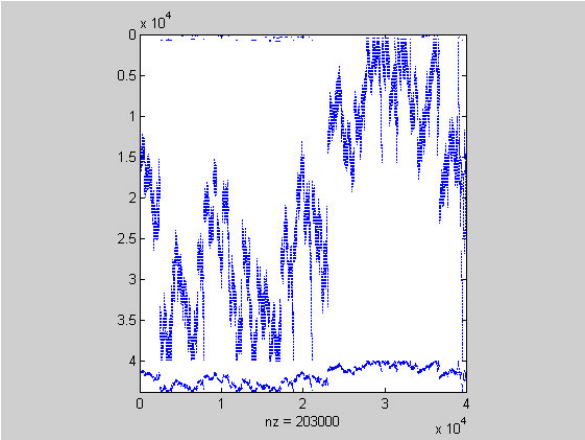

Figure 4. colamd Reordered Sparse Matrix

The elapsed time for the computation of the solution vector showed a significant improvement this time, dropping to 136.12s.

Test 5 – colamd with spparms Parameter Adjustment

MATLAB provides the function spparms( ) that allows ten minimum degree ordering algorithm parameters to be adjusted. There is a recommended grouping that can be invoked using the command:

spaarms('tight')

that optimizes the algorithm to provide less fill-in at the expense of increased computational time for the reordering. The colamd experiment was rerun after setting the parameters 'tight'. The MATLAB spy for the matrix spaarmsTightMatrix is shown in Figure 5.

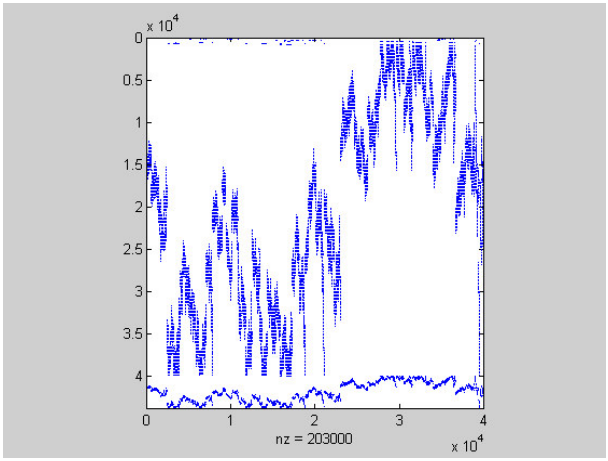

Figure 5. spaarms TightMatrix Reordered Sparse Matrix

This yielded the best processing time of all, 128.735s. However, inspection of the spys for the two colamd matrices shows that they appear to be identical, so any difference in processing time might be hardware related. A replicate run of the 'tight' configuration showed a processing time of 132.06s, possibly confirming this analysis.

Test 6. – Verification of Solution Recovery using Inverse Permutation.

A test was conducted to determine if the solution vector z could be recovered from the permuted solution vector z according to the formula specified above. This was done for the colamd permutation vector p. The inverse vector q was computed and applied to z . The solution vector z was repacked. The expected contour plot, shown below in Figure 6, was produced.

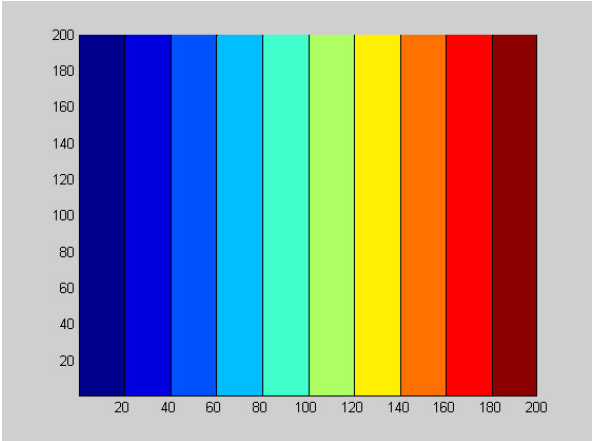

Figure 6. Recovered Sparse Matrix

The colamd permutation took only 66% of the run time as compared to the unordered matrix, indicating that some improvement as a result of column reordering may be possible for larger non-artificial input matrices

Unfortunately, while significant, the improvements demonstrated in this test were judged

not enough of an improvement to approach the goal of processing a 1201 by 1201 node elevation grid.

5.0 METIS Reordering Experiments

The search for productive matrix reordering algorithms was next extended beyond MABLAB. This search indicated that the developmental and research software package METIS might be a good candidate for producing useful reordering schemes.

METIS was obtained by downloading from the site http://www-users.cs.umn.edu/~karypis/metis/. METIS is a family of multilevel partitioning algorithms developed at the University of Minnesota. The package contains several utilities for graph partitioning, mesh conversion and, of particular interest: sparse matrix ordering. The accompanying documentation was clear and well-written but lacked theoretical depth.

The first problem to be solved was understanding how the program worked. The METIS input file format is called a graph file and is composed of connectivity and edge and vertex weight information such that each line of the graph file corresponds to one graph node. It was apparent that this program was aimed at graph applications primarily. The seven node graph example included in the documentation was used to help generate the correct graph file format from the node and edge information provided in the example.

The adjacency matrix corresponding to the example graph was then generated and represented in MATLAB sparse matrix format. The MATLAB function called makeGraph.m was written to convert the MATLAB sparse representation of the adjacency matrix to METIS graph file format. This utility produced a graph file called outfile.graph. The METIS utility graphchk was then run on outfile.graph to check the correctness of the generated file. This worked satisfactorily. The reordering programs oemetis.exe and onmetis.exe were run on outfile.graph. This produced two versions of outfile.graph.iperm, the permutation vectors. These vectors were of the correct dimension and contained no obvious mistakes.

A problem arose when running graphchk.exe on a METIS graph file generated from a MATLAB sparse matrix generated from a small (4 by 4) test elevation contour file. Apparently, METIS can only process symmetric matrices, i.e. only valid adjacency matrices where if coefficient c_1 indicating an edge between node i and node j is present, a corresponding coefficient $c_2 = c_1$ exists indicating that the same edge between node j and i is also present. Unfortunately, this is not the case for sparse matrices derived from elevation contours which are generally asymmetric. METIS also does not accept edges that start and end at the same node, meaning that no non zero diagonal matrix entries are permitted.

The MATLAB minimum degree ordering utilities are able to somehow handle unsymmetrical matrices. Lacking any information from the MATLAB documentation as to exactly how this was done, it was therefore decided to try to force METIS to suggest a permutation vector by direct means. The strategy was to modify the input sparse matrix so that METIS would accept it. METIS was then asked to compute a permutation vector. This permutation vector was itself used to reorder the columns or the rows or both for the

original (unmodified) sparse matrix. (Since the permutation vector is 1 by n the columns can be permuted directly. The permutation vector must be augmented in order to perform the corresponding permutation on the rows. See below.)

In order to accomplish this, the sparse input matrix needed the following:

1) It must be converted to a square matrix;
2) it must not have any non-zeros on the diagonal;
3) it must be symmetric.

Condition (1) was accomplished by removing the code from sparseA.m that concatenated the equations resulting from the known elevations onto the square sparse Matrix from containing the unknown equation coefficients. Condition (2) was accomplished by extracting the diagonal matrix, multiplying it by (-1) and then adding it to the sparse matrix to zero out any non zero coefficients on the diagonal. Then (3) was accomplished by converting all non zero elements to ones and then reflecting every non-zero element across the diagonal. These were all accomplished in makeGraph.m.

After debugging, the permutation vector for a 50 by 50 elevation contour test file was computed using METIS utilities oemetis.exe and onmetis.exe. Both the columns and the rows were permuted using these vectors. The column permutation was straightforward because the dimensions matched. In order to permute the rows, extra permutation indices were concatenated to just copy the rows corresponding to the known elevation equations to the permuted matrix unchanged.

The MATLAB spy plot for the column permutation is shown below in Figure 7 .

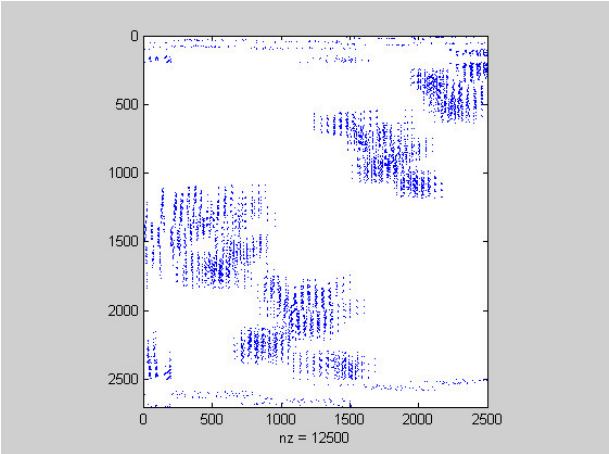

Figure 7. METIS Column permutation Sparse Matrix

The plot is interesting in that the algorithm seems to "clump" the coefficients into regions of high density and low density, the column permutation matrix seemingly blocked along the anti-diagonal. The MATLAB solver was run on the column permuted matrix and also the unpermuted sparseMatrix as the control. The results are in Table 1.

Table 1 Column Permutations

Permutation	METIS Algorithm	Input File Size	Sparse Matrix Size	Solution Time
none	none	50 by 50	2700 by 2500	0.4530
none	none	50 by 50	2700 by 2500	0.4850
none	none	50 by 50	2700 by 2500	0.4840
column	oemetis	50 by 50	2700 by 2500	0.5310
column	onmetis	50 by 50	2700 by 2500	0.5310
column	onmetis	50 by 50	2700 by 2500	0.5000
column	onmetis	50 by 50	2700 by 2500	0.4840
column	onmetis	50 by 50	2700 by 2500	0.4850
column	oemetis	50 by 50	2700 by 2500	0.4840

The first round of testing showed that the column permutations did not outperform the unpermuted sparse matrix. A second round of tests was run for a row permuted matrix permuted with the augmented permutation vector. These results are shown in Table 2.

Table 2 . Row Permutations.

Permutation	METIS Algorithm	Input File Size	Sparse Matrix Size	Solution Time
none	none	50 by 50	2700 by 2500	0.5470
none	none	50 by 50	2700 by 2500	0.5620
row	onmetis	50 by 50	2700 by 2500	0.5160
row	onmetis	50 by 50	2700 by 2500	0.5150

(Note: the run times for both the control and the test cases were slower . However, the test case ran faster than the control in the row permutation test and slower or equal to the control for the column permutation test.)

The tests on the 50 by 50 elevation grid were not encouraging but not entirely conclusive either. The problem was that the total run time for the MATLAB solution was less than a second, typically less than half a second. Small hardware variations unrelated to the problem introduced a standard error that was significant when compared to the measured elapsed time. As a result a larger problem was run that was known to have a longer computational time.

Larger Elevation Grid Input File

The procedure was the same as used for the 50 by 50 node test: a square sparseMatrix was computed, and then the METIS graph file was computed from it using makeGraph.m. The METIS utility onmetis.exe was then used to compute outfile.graph.iperm, the permutation vector.

After incrementing each element of the permutation vector by one, the rectangular sparseMatrix was recomputed. The column reordered permutationMatrix was computed from it using the permutation vector q. The system was then solved using MATLAB:

permutationMatrix\b.

The elapsed time for this computation was 125.3120s. The corresponding time for the control calculation was 120.030s, indicating no improvement as a result of the permutation. The spy plot for the permutationMatrix is shown below in Figure 8.

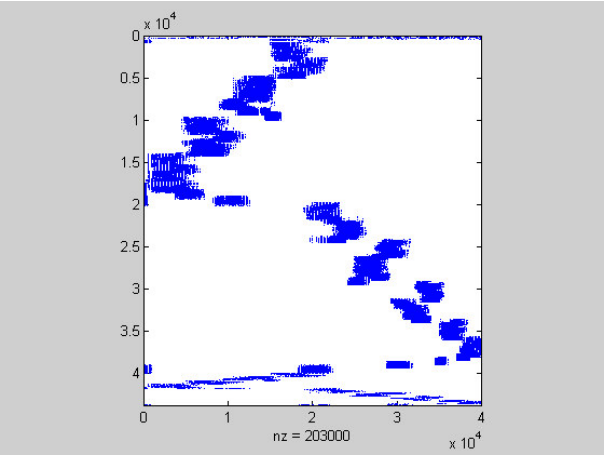

Figure 8. METIS Column Permutation Sparse Matrix

Next a row permutation was attempted using the same technique as for the 50 by 50 node test. A new permutation vector was constructed by augmenting the old permutation vector so that its row dimension matched sparseMatrix. It was then used to compute a row permuted permutationMatrix and solved the system

permutationMatrix\b.

The solution time was 103.5320s, an improvement of about 16%. As a final check, the control was rerun. Virtually the same computation time of 120.4370s was observed. The data is summarized in Table 3. The spy plot for the row permutation is shown below.

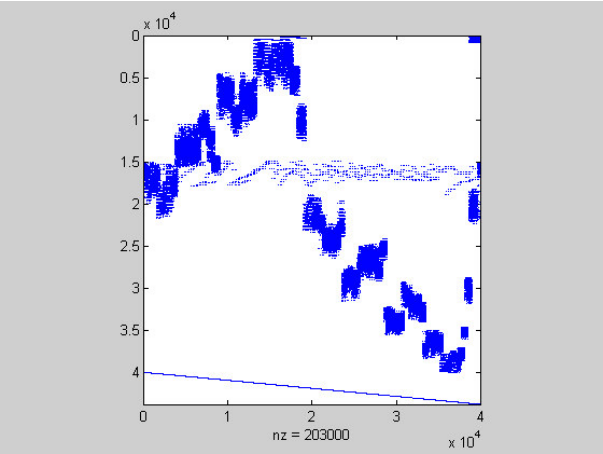

Figure 9. METIS Column Permutation Sparse Matrix

Table 3 . Column and Row Permutations: 200 by 200 Node Input File.

Permutation	**METIS Algorithm**	**Input File Size**	**Sparse Matrix Size**	**Solution Time**
none	none	200 by 200	43800 by 40000	120.0310
column	onmetis	200 by 200	43800 by 40000	125.3120
none	none	200 by 200	43800 by 40000	120.4370
row	onmetis	200 by 200	43800 by 40000	103.5320

Although the METIS onmetis.exe row permutations consistently produced faster computation times than the unordered MATALB control, the improvement was not better than the best MATLAB permutation.

Although it is possible that an ordering scheme that outperforms the MATLAB unsymmetrical minimum degree ordering algorithm exists, it was considered more likely that a point of diminishing returns was realized for this approach and that the limited time could be better spent working on alternative strategies to reduce computational time.

This part of the study was concluded with an understanding of row reduction techniques that can reduce the computational time for the over determined Laplacian system by 15 to 33 percent through use of the reordering algorithms described in this section. While this improvement is considered quite substantial, it was not considered of sufficient magnitude to warrant further investigation.

It must be remembered that the reason for this study was to find computational methods that would allow the solution of larger systems of over determined linear equations formulated by the Franklin algorithm. The main reason for the failure to compute such systems was not excessive computational time, but was in fact excessive in-process memory requirements.

It might seem prudent at this point to determine if the best reordering scheme, as applied to solution of systems generated by the Franklin Algorithm, would significantly reduce memory requirements commensurate or in excess of the reduction in computational time.

This was not done because the size of the reduction in computational time (used as a screening method) was not sufficient to justify further study. It is expected that the problem size grows at least as a polynomial function of the number of input nodes. Therefore, reduction in the solution times better than those observed in this study are required if the target problem (1201 by 1201 elevation nodes) is to be successfully solved.

As a result further study was not conducted and a new approach to the general problem was started.

6.0 Saunders-Paige LSQR Experiments using Well-Behaved Experimental Input Files

After achieving only limited success toward the goal of solving large input data sets using the MATLAB linear least squares solver, a search was conduced for alternative solvers. One of the most interesting candidates was the LSQR package offered by the Software Optimization Library of the department of MS and E at Stanford University. The software was written by Dr. C.C. Paige and Dr. M.R. Saunders. The software is implemented in C, Fortran77 and MATLAB. It solves Ax=b for consistent systems or minimizes:

$$\| Ax - b \| \text{ or}$$
$$\text{sqrt}(\| Ax - b \|^2 + d\|x\|^2)$$

for over determined systems, where d is a damping parameter. The solver uses a method based on Golub-Kahan bidiagonalization. It is algebraically equivalent to applying the symmetric conjugate gradient (iterative) method to the normal equations:

$$(A'A + d^2 I) x = A'b$$

but according to the authors[8] has better numerical properties, especially if A is ill-conditioned.

This package was interesting for two reasons. Experiments with iterative solvers during the proposal stage of this project indicated that this type was very efficient both from a computational and memory use standpoint for the solution of consistent systems using the Laplacian PDE formulation. Iterative solvers also allow a certain degree of flexibility because solution accuracy may be traded off for decreased run time if necessary. The LSQR solver also met another criterion: it was ported to non MATLAB platforms, allowing a migration path from the developmental environment if the subsequent tests proved successful.

Initial Tests

The first approach was to use the MATLAB lsqr.m and other files offered on Dr. Saunder's download page to solve a small problem. It was then intended to investigate the C implementation once the MATLAB implementation was understood adequately.

Two files are required: lsqr.m and aprob.m, where lsqr.m is the solver and aprob.m defines the matrix A through its multiplication algorithm. (The iterative technique is based on a recursive application of multiplication of a matrix $Nx^{(k-1)}$ where $x^{(k-1)}$ is the $(k-1)^{th}$ iterative approximation of the solution vector. Therefore, efficient matrix multiplication algorithms are essential for the rapid solution of large problems. (It is common to define the matrix multiplication algorithm, rather than the matrix itself in the case of large sparse matrices, where the latter would be cumbersome or impossible to

handle explicitly.)

After some initial problems understanding the modules, Dr. Saunders was contacted at Stanford by email with some questions. Dr. Saunders graciously provided a detailed reply with much of helpful advice. The most helpful piece of information was that LSQR was implemented in MATLAB as a native routine, obviating the need for the functions provided on the website, at least for research purposes.

The subsequent LSQR tests described in this report were all run on the native MATLAB implementation. InputMatrix.java, included in the Java Code Appendix 2 was used to prepare many of the test input files used for this section.

Native MATLAB LSQR( ) Implementation Tests: 50 by 50 Node Input Elevation Grid

A first test was conducted on a 50 by 50 node input elevation matrix to ensure that the function call syntax was understood. Input was of the form:

```
input5;
sparseMatrix=sparseA(A);
b=makeB(A);
z = LSQR(sparseMatrix, b, 10e-06, 100)
```

Where input5 is the InputMatrix.java generated 50 by 50 input grid; sparseMatrix = sparseA(A) is the MATLAB sparse representation of the Laplacian PDE matrix; b is the over determined Laplacian RHS vector; z is the solution vector; 1-e-06 is the convergence tolerance and 100 is the maximum number of iterations.

The method computed the solution to a relative residual of 0.051 using 0.7110 seconds. Computing 200 iterations returned a relative residual of 0.0072 in a time of 1.3520 seconds, indicating that solution time was linear with the number of iterations specified, as expected.

Repacking z using repack( ) and viewing using contourf( ) produced the Figure 10 plot.

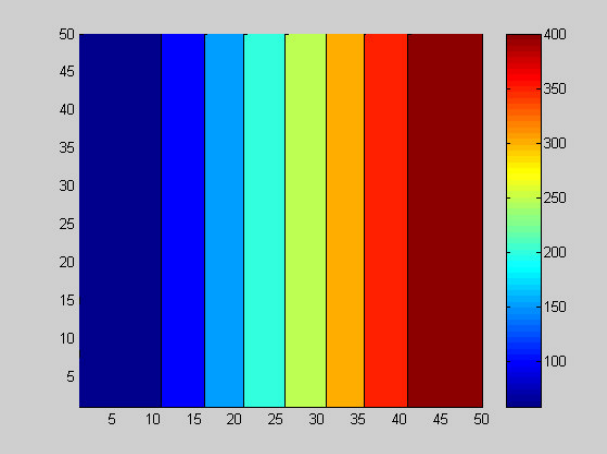

Figure 10. LSQR 50 X 50 Repacked Solution Vector

This plot is indistinguishable from the corresponding plot produced by the MATLAB direct solver.

Native MATLAB LSQR Implementation Tests: 200 by 200 Node Input Elevation Grid

Next, the solution of a 200 by 200 elevation node input file was attempted using operations similar to those described above. The MATLAB command:

z = LSQR(sparseMatrix, b, 10e-06, 50)

where sparseMatrix is the MATLAB sparse representation of the Laplacian PDE matrix derived from the 200 by 200 input file solved the system in 6.669s to a relative residual of 0.048s.

Repeating the test with 200 iterations LSQR solved the system in an elapsed time of 25.937s to a relative residual of 0.0064.

The MATLAB direct solver processing files generated by sparseMatrix and b as a control required an elapsed time of 224.3s.

Native MATLAB LSQR( ) Implementation Tests: 500 by 500 Node Input Elevation Grid

The next test was on a 500 by 500 node input grid. This was not solvable using the MATLAB direct solver on the test hardware platform. The MATLAB input command:

z = LSQR(sparseMatrix, b, 10e-06, 200)

where sparseMatrix is the MATLAB sparse representation of the Laplacian PDE matrix derived from the 500 by 500 node input file solved the system in an elapsed time of 108.07s to a relative residual of 0.004.

Repeating the test for 400 iterations solved the system in 216.1720s to a relative residual of 0.0002.

This result was encouraging in two respects. Firstly, LSQR was able to easily handle an input file larger than that handled by the MATLAB direct solver. Also, the solution time seemed to grow relatively slowly with input file size.

Native MATLAB LSQR( ) Implementation Tests: 800 by 800 Node Input Elevation Grid

The final test was an 800 by 800 elevation node input grid. This was larger than the largest file previously solved by the MATLAB direct solver using a computer with 1024MB of RAM. The following MATLAB commands were input:

```
input80;
sparseMatrix=sparseA(A);
b=makeB(A);
size(sparseMatrix)
703,200, 640,000
z = LSQR(sparseMatrix, b, 10e-06, 200)
```

where sparseMatrix is the MATLAB sparse representation of the 800 by 800 node input file. LSQR solved the system in an elapsed time of 276.0940s to a relative residual of 0.0031.

Repeating the test for 400 iterations solved the system in 551.9380s to a relative residual of 9.8e-05.

The contour plot of the repacked solution vector is shown below in Figure 11.

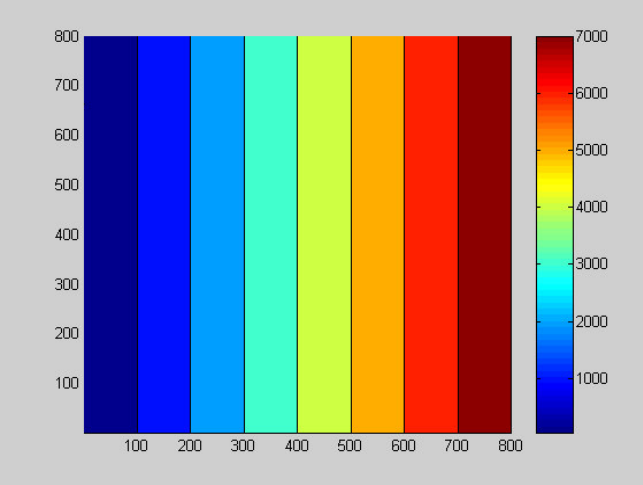

Figure 11. LSQR 800 X 800 Repacked Solution Vector

The plot shows that the solution is at least reasonably close to the expected result. Inspection of the output file showed a maximum computed solution value of 7936.1 fitted to an input file with a maximum contour line value of 7900.0.

These initial tests of the LSQR algorithm were encouraging for the reasons previously stated:

1) Problems close to the target problem size could be easily handled by LSQR.
2) The computation time grew very slowly with respect to problem size as compared to the MATLAB direct solver.
3) The solutions produced by the LSQR method appeared to compare favorably with those produced by the direct solver. For example, the difference between a 400

iteration LSQR computation of a 50 by 50 input elevation grid and the corresponding direct solution was essentially nil (~2.71e-08 for a 436.2622 computed elevation value.)

7.0 Saunders-Paige LSQR Experiments using Input Files Selected to Exercise LSQR more Fully

The initial tests with LSQR were very encouraging, with the program processing files in much less time than the MATLAB direct solver. More importantly, processor memory requirements were also much lower. However, there were several deficiencies in the initial test series that needed to be addressed.

No quantitative estimate of the agreement between the LSQR solution and the direct solution was computed. In addition, the largest file processed was 800 by 800 elevation grid nodes. Since the target was 1201 by 1201 grid nodes, it was necessary to attempt this larger input file size.

Moreover, the tests were conducted with very simple input files with a constant elevation gradient from one end of the grid to the other. The initial tests were run on test grids produced by the Java program InputMatrix. This program was written to output a mock contour line grid. The grid contour lines were evenly spaced at 10 unit increments, they were all parallel, their elevation values increased linearly across the grid, and they were parallel or perpendicular to the grid borders. This format was chosen because a correct interpolation could be determined at a glance at the 2D contour plot output. It was suspected that performance might be degraded if more realistic files were processed.

In order to determine this, a series of increasingly more complex input files were prepared and used to exercise LSQR. The first test run, however was designed to obtain a more quantitative comparison between the LSQR solution and the direct solution.

Test 1. Comparison of LSQR Solution of 200 by 200 Node Input Grid Produced by InputMatrix with MATLAB Direct Solution

The 200 by 200 grid simple input file computed by LSQR in the last section was recomputed and compared to the corresponding MATLAB direct solution. A difference matrix was computed:

$$\text{difference} = c - cc$$

where c is the repacked solution vector from the direct solution and cc is the repacked solution from the LSQR solution. The MATLAB Statistic (max(max)) (which computes the maximum value in the matrix) was computed on the difference matrix and found to be 6.6734. Then a contour plot of 'difference' was computed and inspected for the 200 by 200 node solution. This showed visually the differences between the two solutions across the height and width of the grid. The results showed generally good agreement between the two solvers. However, there was a deviation at the "lowest" edge of the difference plot that was difficult to explain at the time but the reason for which subsequently became clearer. A contour plot of the difference matrix is shown below.

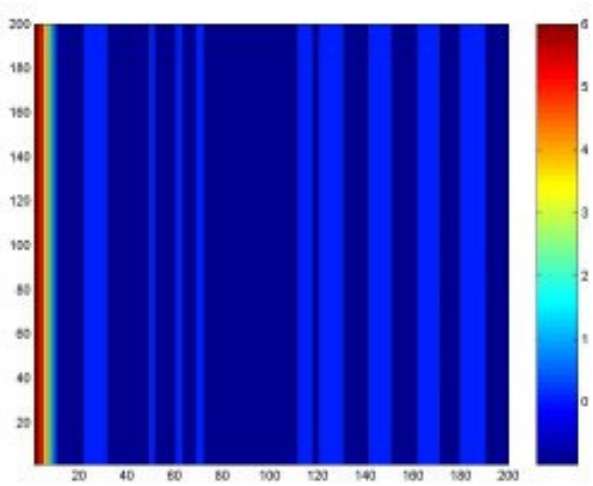

Figure 12. LSQR 200 X 200 Difference Plot

Test 2. 1201 by 1201 Input Grids

As an extension of the above testing, a 1201 by 1201 node input grid was run to see if LSQR could handle a file this big. A suitable input file was prepared using InputMatrix and a solution was attempted. However, this approach did not work for the 1201 by 1201 grid as the routine never got past the sparse matrix preparation code after 24 hours of trying. The reason, as discussed before, is the extremely poor performance of MATLAB user-defined code modules.

As a consequence, a C++ program called Sparse was written to prepare a MATLAB sparse matrix index file from an input elevation grid. (The code listing is included in Appendix 3.) MATLAB allows importation of sparse matrices using such an index file format. Although the input files can be quite large, despite the sparse data structure, formulating the sparse matrix using Sparse was much faster than the MATLAB method, requiring only a few minutes of processing time.

Using this technique, LSQR was able to solve the system in 1.586e+04 seconds using 400 iterations. Although no direct solution was available to compare it with, (because the file was too big for the MATLAB direct solver) the contour plot was reasonable and the computed values were as expected. The output contour plot for the 1201 by 1201 test file is shown below in Figure 13.

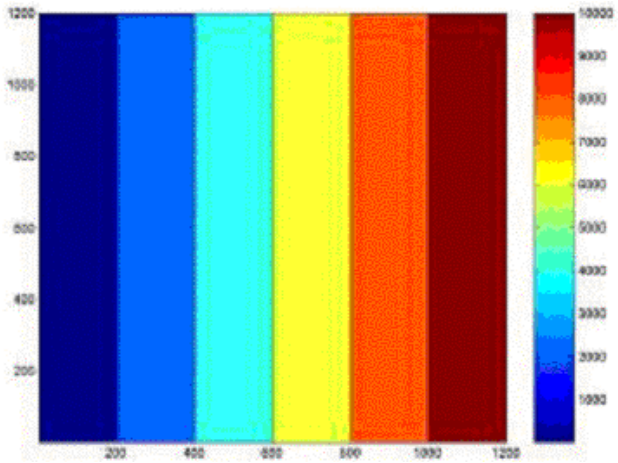

Figure 13. LSQR 1201 X 1201 Repacked Solution Vector

Test 3 Wavy.dat

The next test was designed to investigate the performance of the LSQR solver when used on irregular input grids. InputMatrix was modified to produce a more irregular input grid. Instead of increasing regularly from minimum to maximum, the contour lines generally increased, but in an irregular "wavy" pattern. The contour lines were no longer evenly spaced. However, the contour lines were still parallel and still column aligned in the input grid.

Figure 14. LSQR wavy.dat Solution Vector

Figure 15. MATLAB direct Solution Vector

The results of the LSQR computation are shown in the mesh plot above left (Figure 14), while the direct solution is to the right (Figure 15). The solution tracked the input contour lines closely, fitting a smooth surface to the contour lines in 200 iterations.

The Figure 16 plot shows the difference contour for the LSQR and direct solutions. Differences between the two plots were negligible.

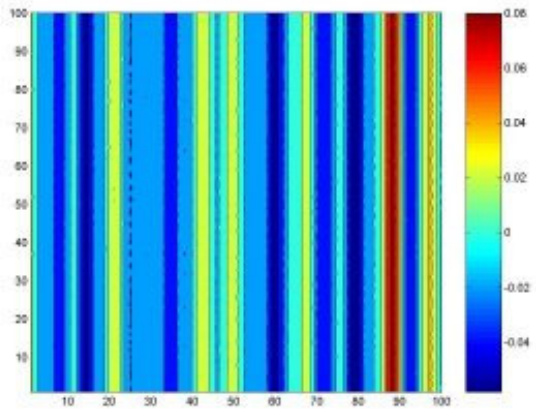

Figure 16. wavy.dat Difference Plot

Test 4 square.dat

As a second test of an irregular input grid the instructor-supplied 'square.dat' was used. This grid was specially constructed to present problems to interpolation algorithms. The file is characterized by square, concentric contour lines. Poor interpolators produce a stepped pyramid, while good ones produce a smoother surface. The LSQR solver produced a solution that was very close to that produced by the direct solver. Mesh plots of the solutions produced by the two solvers are shown below in Figures 17 and 18.

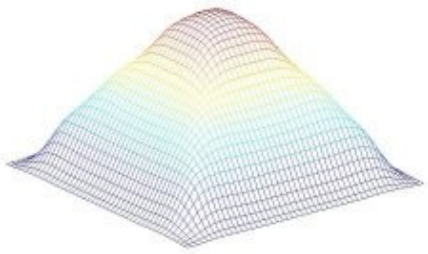

Figure 17. LSQR square.dat Repacked Solution Vector

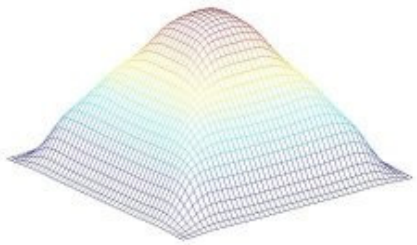

Figure 18. Direct Solver Repacked Solution Vector

The difference contour for the two plots is shown below in Figure 19.

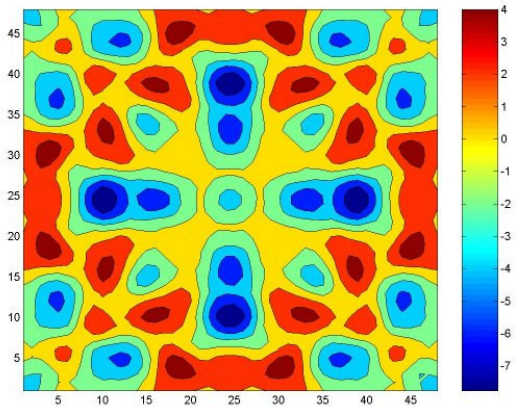

Figure 19. square.dat Difference Matrix

Test 5. Tuckermans.dat

The next test was run on the instructor-supplied input grid of Mt. Washington, NH derived from an actual USGS contour map and called tuckermans.dat. This was an 800 node by 800 node file, well within LSQR's demonstrated solution capability. A Sparse matrix index file and a RHS vector was prepared using Sparse as with the 1201 by 1201 grid described above.

The input file was solved by LSQR using 200 iterations in about 250 seconds. Unfortunately, the solution computed by LSQR was grossly in error, with many large negative elevations resulting. The surface and contour plots for this run are shown below in Figures 20 and 21.

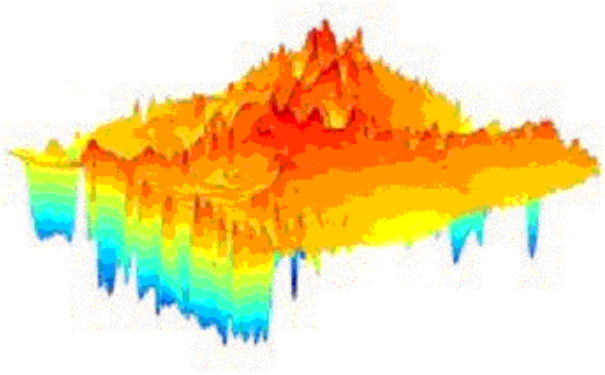

Figure 20. LSQR tuckermans.dat Repacked Solution Vector

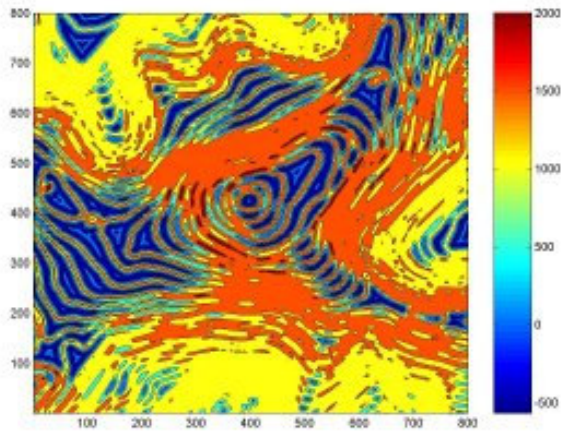

Figure 21. LSQR tuckermans.dat contourf plot

Test 6 tuckermans.dat Subset Diagnostic Tests

It was not obvious why the solution failed so badly after the previous successes. The full tuckermans.dat file was too large for convenient diagnostic testing so it was subsetted and a series of LSQR solutions was run on the subset file. A parallel direct solution was run on each of the subsetted files to help see where the problem was occurring.

The first test was with a 50 by 50 node subset. This computed correctly using LSQR in 500 iterations, producing results that were very similar to those produced by the direct solver. If less than 500 iterations were used the results were poor.

Next a 100 by 100 node grid was computed using LSQR. In this case 500 iterations did not converge but 1000 iterations did.

Finally, a 400 node by 400 node subset was run. This solution produced persistent negative elevation "islands" that required many iterations to overcome. Ultimately, 3000 iterations were required before a satisfactory solution was produced. This indicated that, at least for this input file the solution was initially unstable but did not explode and ultimately did converge. The larger the file, the more iterations seemed necessary for convergence.

The elevation contours below show the result of the direct solution (Figure 22) , the LSQR solution after 2000 iterations (Figure 23), and the LSQR solution after 3000 iterations (Figure 24) for the 400 by 400 node input file.

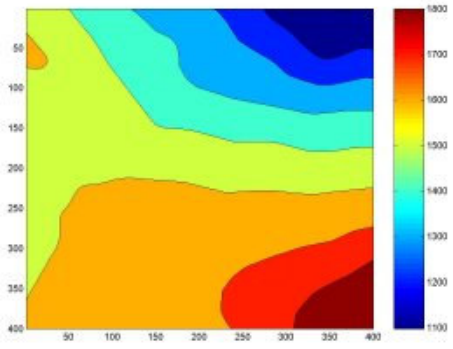

Figure 22. 400 X 400 Direct Solution

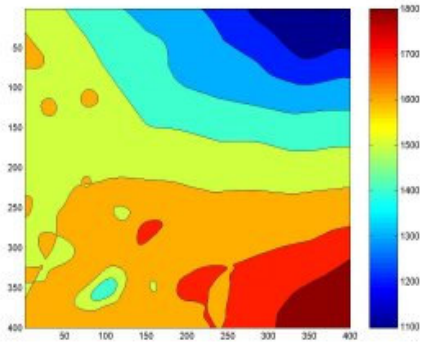

Figure 23. LSQR 400 X 400 2000 Iterations

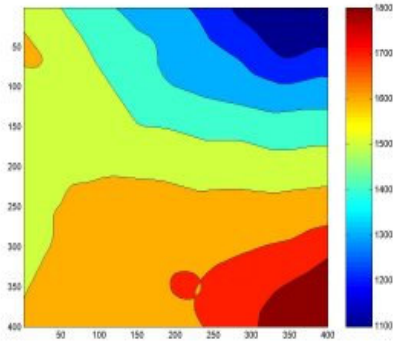

Figure 24. LSQR 400 X 400 3000 Iterations

Another test was conducted to determine the effect of the initial guess on convergence for LSQR. The LSQR program allows an initial guess vector to be supplied as a parameter when the function is called.

The 50 by 50 node subset of tuckermans.dat was again used as input. The system was

solved using LSQR and it was determined that it took approximately 500 iterations to converge with the default initial guess of x0=0.

The same solution was attempted again but with an initial guess of x0=ones(2500, 1) * 1000, this vector being chosen because 1000 was estimated to be the approximate average elevation. This choice reduced the number of iterations to converge to 400.

A third test was attempted, this time setting x0=ones(2500, 1) * max(max(A)) where A is the elevation grid. This allowed the solution to converge after 300 iterations, cutting 200 from the default estimate solution.

Tuckermans.dat Test Rerun

The tuckermans.dat file was again solved by LSQR, this time making a better initial guess and using more iterations. Previous experience indicated that 5000 iterations could be accommodated in an overnight run.

The initial guess was x0=ones(640000, 1) * max(max(tuckermans)), i.e. a column vector of the maximum elevation values in the input file. (The maximum elevation was the best of the three initial guesses tried earlier, the others being the average elevation and all zeros). The MATLAB command:

z=LSQR(sparseMatrix, tuckermans_b, 10e-06, 5000, [ ], [ ], x0)

was invoked. The elapsed time for the run was 6.7486e+03 s, with a relative residual of 0.00052. This run was more successful and appeared to converge properly. A contour plot and a mesh plot corresponding to this run are shown below in Figures 25 and 26.

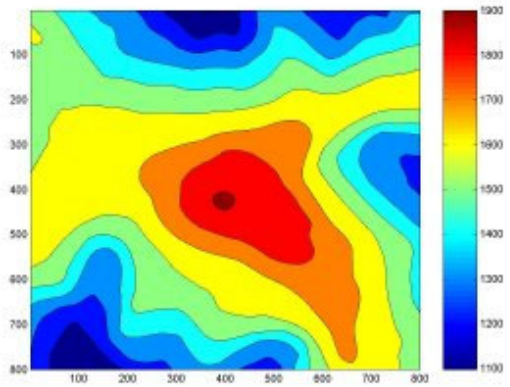

Figure 25. LSQR tuckermas.dat Repacked Solution

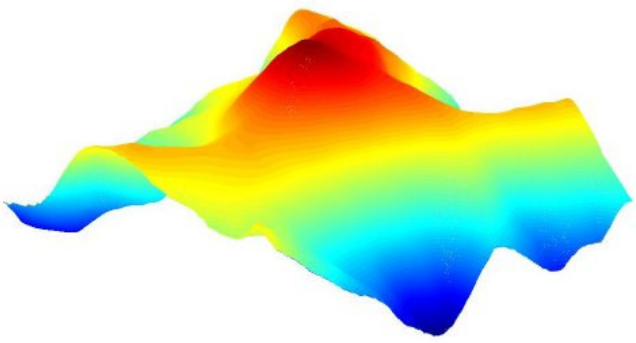

Figure 26. LSQR tuckermas.dat Repacked Solution Vector 5000 Iterations mesh Plot

This series of tests indicated the following:

1) The LSQR solutions agreed well with MATLAB direct solutions for input files specially constructed to exercise interpolation algorithms.

2) A very serious LSQR solution problem was introduced when a real input grid was attempted.

3) The problem seemed to be characterized by an initial instability that developed relatively quickly, subsequently requiring many iterations to converge. It seemed that the larger the input file, the greater the initial instability and the greater the number of iterations required to recover.

4) The problem seemed to be highly correlated to the orientation of the contour lines in the grid, with parallel or perpendicular contour lines behaving well, even if they were irregularly spaced and of uneven elevation. (This type of file may converge faster than a smooth file due to the nature of the iterative processor.)

5) A better initial guess led to significantly better convergence in a small test file.

6) A better guess and more iterations yielded a successful result for the tuckermans.dat file.

8.0 Interpolating Difficult Input Terrains using LSQR

The instructor-supplied crater.dat input file is a somewhat larger (900 by 900 elevation node) input file. This elevation contour file was chosen to be particularly difficult to interpolate because it contains the top of a steep mountain terminating in a crater falling off even more steeply into a lake. An attempt was made to solve it using the Saunders-Paige LSQR iterative solver and the techniques described in the previous report as a further test of those techniques. They were inadequate for this file and initially, very poor results were obtained using them.

The initial guess was a column vector of 810,000 maximum elevation values (7680 feet). The LSQR solver was run using 1000, 2000, 3000 and 5000 iterations without success. The problem seemed to be convergence again. The maximum run time was 8.49e+03 seconds with a relative residual of 0.0013.

A 200 by 200 node subset of the larger file was again used to investigate the problem. The subset included both the rim of the crater, the steep slope, and the lake. Tests indicated that the subset converged slowly to a correct-looking solution with the lower left corner of the map (the lake section) converging last. (Better convergence was apparently obtained because of the smaller size of the input file.)

Next, the subset input file was run with an initial guess of the minimum elevation instead of the maximum. This performed much better, converging after only 500 iterations. Finally, an initial guess of intermediate value, 6800 feet was attempted. This did not converge as quickly as the minimum. (It became apparent after subsequent tests that the minimum elevation worked best in this case because of the problematic lake area of the map. An initial guess close to the lake elevation helped this troublesome area the most.)

It was apparent at this point that the nature of the terrain, i.e. the high ridge dropping off a sharp cliff into a flat lake was a particularly difficult interpolation problem. The exercise also confirmed the importance of file size and initial guess on solver performance.

Test 2. Second Trial of crater.dat

The full crater.dat file was rerun using the minimum elevation as the initial guess (based on the previous test results) in the hope that this would improve convergence. Unfortunately, this did not occur as the solution did not converge after 5000 iterations.

The crater.dat file was next run using the minimum elevation as the initial estimate and 10,000 iterations of LSQR. This took 1.69e+04 seconds and returned a relative residual of 0.00029. The resulting mesh plot looked much better but there were still minor but obvious areas of non convergence. (Note: the relative residual has proven to be a quite useless indicator of convergence or correctness of solution for interpolating terrain. These problems are characterized by relatively good convergence for the majority of the

map and minor but significant errors in small sections of the map. This results in a low relative residual (which is a combined error term like a norm) but an unacceptable map.)

Finally, crater.dat was run using LSQR and 20,000 iterations. This ran in 3.396e+04 seconds with a relative residual of 0.00029. This plot looked better except for a very obvious undershoot problem at the foot of the cliff (in the lake) and some less obvious "phantom" gullies (i.e. gullies with no corresponding contours) at the edges of the map. The result of this computation is shown in the figure below in Figure 27.

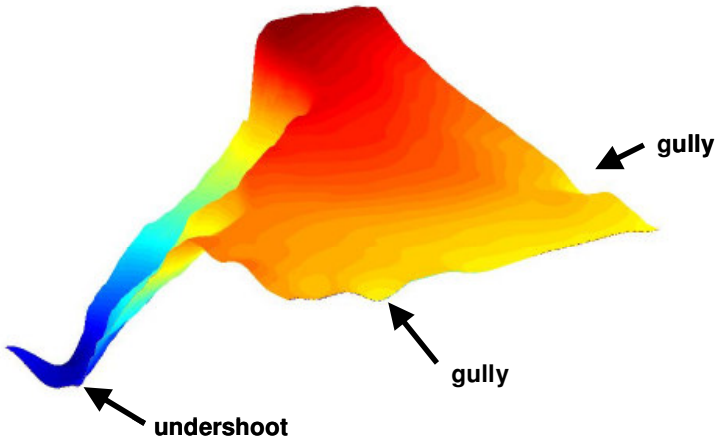

Figure 27. LSQR crater.dat Repacked Solution Vector 20,00 Iterations

Test 4. Matrix Preconditioning

Dr. Saunders, in his email in response to an email enquiry suggested the following advice:

"Also, be sure to note the recommendation about scaling the columns of 'A' to have the same Euclidean norm. This is diagonal preconditioning. The hope is that LSQR will converge quickly enough. If you're very lucky, it might need only a few hundred iterations."

As a result, the MATLAB function 'scale.m' was written. This program returned a sparse diagonal matrix such that when used to post multiply 'A' produced a normalized matrix (all the column vectors having length one). The math for this preconditioning is as follows:

For the system:

$[A]\,[x] = [b]$

$([A][D])\,([D]^{-1}\,[x]) = [b]$

The second equation says that if A is post multiplied by D, then solution of the system returns ($[D]^{-1}$ [x]) and not x. It should be possible to recover x by pre-multiplying by D after computation.

Several tests were run using a 100 by 100 node subset of crater.dat, running both unpreconditioned and preconditioned sparse matrices. Although the preconditioning changed the convergence characteristics of the solution, it was not for the better in this test. The preconditioned matrix converged either much more slowly or not at all, with large errors at the corners of the map and odd errors elsewhere.

Test 5. Elevation Clipping

Processing of crater.dat and other files using LSQR indicated that the iterative solutions were characterized by elevation overshooting and undershooting. As a result, a MATLAB program called clipper.m was written. The idea was to produce an intermediate solution, use clipper.m to remove the overshoot and undershoot elevations, and then use the conditioned intermediate solution vector as an initial estimate for LSQR.

The approach was tested using a 200 by 200 node subset of crater.dat. This did not prove to be a success either. The "bad" elevations persisted and kept reappearing in the solution vector even after they were clipped out of the intermediate solution vector. No application of the clipper.m program had any useful effect.

Test 6. Multiple Level Solutions

During the course of attempting solutions of crater.dat using LSQR, it seemed that the convergence was related to the initial estimate. Sometimes setting the initial estimate to the maximum elevation worked well, and with other files the minimum elevation worked better, depending on the nature of the particular file.

What was needed to speed convergence was an initial estimate that was small where the solution vector was small and large where it was large. The problem is that prior knowledge or prediction of the solution was needed to accomplish this. Various ways of performing a single pass interpolation on the input file to produce a better initial estimate were considered. The problem is that the first pass interpolation should ideally condition the zeros in the grid to something more closely approaching the adjacent contour elevations in order to speed convergence.

It was considered that the (not over determined) iterative Laplacian solver described in Section 3 might be the ideal generator of initial estimates for LSQR. Because the estimated system was not over determined, a least squares solution was not required, only Gauss Seidel iteration of the full rank matrix. The iterative solver was possibly ideal for the application because although this solver produced a lower quality interpolation, only a coarse or "good enough" solution was required for the purpose of initial estimate.

The zero elevations in the input file would be interpolated relatively rapidly to values

close to their contour line neighbors. (With an iterative solver, initial convergence is rapid and the final degrees of convergence can take most of the computational time.) Such a solver was adapted from code written during the proposal phase of the project. The Java interpolation program (accessed through the InterpolationUI.java module) has a four nearest neighbor (Laplacian) iterative solver, a Thin Plate solver and is capable of matrix row reduction. The code listings for this program are included in Appendix 2.

The first test of this idea was conducted using the full (900 by 900 node) crater.dat input file. This file had been "solved" using 20,000 iterations of LSQR with an initial estimate of a 810000 by 1 initial estimate vector of all minimum elevation values. The result was a mesh plot that had a significant undershoot in the lake and small "gullies" near the edges where no gullies were supposed to be as described above.

The fast iterative Laplacian solver running the crater.dat input file exhibited similar convergence problems. (It was becoming apparent that convergence is a major concern with iterative solvers.) It took 50,000 iterations to produce a reasonable initial solution as evidenced by inspection of the mesh plot. However, because the iterative solver was relatively fast 50,000 iterations took only 1445 seconds, a tolerable amount of time.

Then crater.dat was rerun using the LSQR solver with the solution vector produced by the fast iterative Laplacian solver as the initial estimate for LSQR. Experiments showed that it took 20,000 iterations using the LSQR solver to produce a considerably improved result. The undershoot was now absent from the solution, as were the unwanted gullies. However the lake area was still not perfect because instead of the lake being perfectly flat it was "filleted" at the base of the cliff, even after adding an additional elevation node in the lake.

The results are shown in the figures below. The figure on the left (Figure 28) is the solution computed by the fast iterative Laplacian solver. The figure on the right (Figure 29) is the solution computed by LSQR using the solution produced by the fast iterative Laplacian solver as the initial estimate.

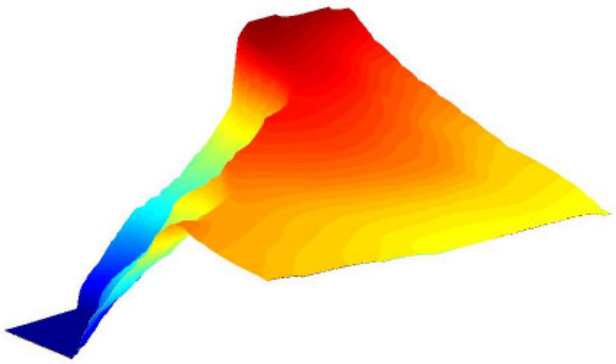

Figure 28. Laplacian crater.dat Repacked Solution Vector 50,000 Iterations (Initial Estimate Vector).

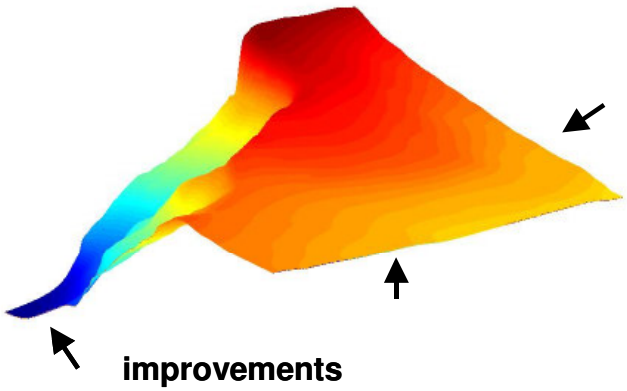

Figure 29. LSQR crater.dat Repacked Solution Vector 20,000 Iterations using Initial Estimate Shown In Figure 28.

It was apparent that crater.dat was a particularly difficult file to interpolate. It is considered that cliffs represent a special class of interpolations worthy of a study of their own. At this point, sufficient technique had been developed to attempt the solution of the goal of this project, a reasonably well-behaved 1201 by 1201 elevation node input file.

Solving 1201 by 1201 bountiful.dat

In order to produce a suitable input file the 2100 by 2100 node instructor-supplied bountiful.dat file was subsetted to the target 1201 by 1201 elevation node size. A small program called writeMatrix.m was written to convert the subset file into a format that Sparse.exe could handle.

A coarse solution was run using the fast iterative Laplacian solver, the 5000 iterations processing in 313.312 seconds. This solution was then used as the initial estimate for LSQR, running for 500 iterations. Because of the large size of the file (around 250MB), considerable memory paging was required for the first time for MATLAB's LSQR to run the solution, slowing down the solution very considerably. The 500 iterations took 3.0433e+04 seconds (8.45 hours) to compute.

A mesh plot of the repacked solution vector is shown below in Figure 30. The maximum difference between the initial estimate and the LSQR solution after 500 iterations was 18.8 feet (the elevations range from about 6800 minimum to 7800 maximum feet.) This is a considerable difference for a 20 foot contour interval file given the relatively high quality of the initial estimate. However, the actual quality of the computed solution is not known because the input file was of course too large for solution using the direct MATLAB solver.

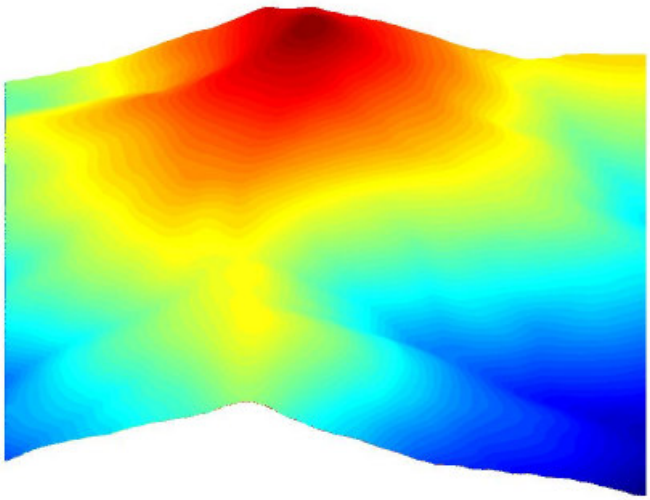

Figure 30. LSQR bountiful_subset.dat 1201 X 1201 Repacked Solution Vector 500 Iterations using Initial Estimate. (Note: the apparent contour lines visible in the plot are an artifact of the MATLAB surface color interpolation and are not residuals of the elevation interpolation.)

9.0 Comparison of the Quality of Solutions and Convergence Rates of the Two-Level Laplacian/LSQR Solver and the MATLAB Direct Solver

Investigation Solution Quality

It was considered necessary to compare the quality of the two level iterative solution technique developed in the previous section with the solution produced by the MATLAB direct solver. In order to investigate this, a series of tests was conducted using square.dat. This file was designed specifically to be difficult to interpolate. In order for the proposed two-level technique to be useful, it should produce a very different solution from the initial estimate and a solution indistinguishable from that produced by application of the MATLAB direct solver to the same input file.

The square.dat input file was interpolated by applying the iterative Laplacian solver using 5000 iterations. The result was a particularly ugly (tent-like) mesh plot with the expected drooping between the contour lines and characteristic sharp corners. (The Laplacian iterative solver is written to "lock" the contour lines at their original values and to not let them change regardless of the number of iterations, thus exaggerating the bad qualities of the four nearest neighbor interpolation algorithm.)

This intermediate solution was then imported into MATLAB and converted into an initial estimate vector using the MATLAB commands:

```
transpose=intermediate';
x0=transpose(:);
```

A sparseMatrix and RHS vector 'b' were computed using the following MATLAB commands:

```
sparseMatrix=sparseA(square);
b=makeB(square);
```

The system was then solved using the commands:

```
z=LSQR(sparseMatrix, b, 10e-06, 500, [ ], [ ], x0);
```

Next the control was computed:

```
zc=LSQR(sparseMatrix, b, 10e-06, 500);
```

Both these solutions converged to the same solution vector. Mesh plots of the initial estimate vector, along with the two identical solution vectors are shown in below.

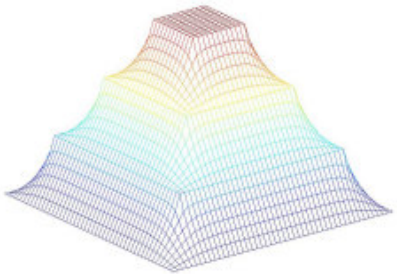

Figure 31. Laplacian Solution

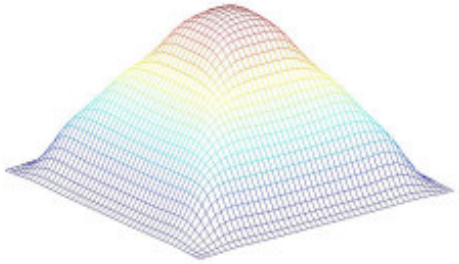

Figure 32. Direct Solution

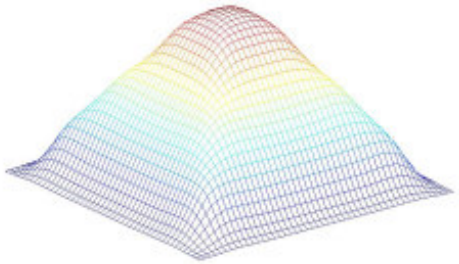

Figure 33. LSQR Solution

This test indicated that the very coarse initial estimate was iterated by LSQR to a solution virtually identical to that produced by the MATLAB direct solver. There was no evidence in this test to indicate that the two-level solution approach was any worse than the direct solution (which took much more processing time and much more storage).

Comparison of the Rate of Convergence of LSQR without an Initial Estimate and the Two-Level Laplacian/LSQR Solver with an Initial Estimate

The preceding test demonstrated that the LSQR solution of square.dat converged to the same solution as the MATLAB direct solver. It was next considered necessary to determine quantitatively the effect on LSQR convergence of using a high quality initial estimate (such as that produced by the fast Laplacian iterative solver) as compared to LSQR using the default initial estimate.

The square.dat input file was again chosen as the test file. In the first series, MATLAB was used to solve the equation:

z=LSQR(sparseMatrix, b, 10e-06, <iterations>);

where , <iterations> was a the number of iterations requested, ranging from 10 to 90.

Contour plots of the resulting mesh files are shown in the figures below. The file at the upper left of Figure 34 was run for ten iterations, top center twenty iterations, etc.

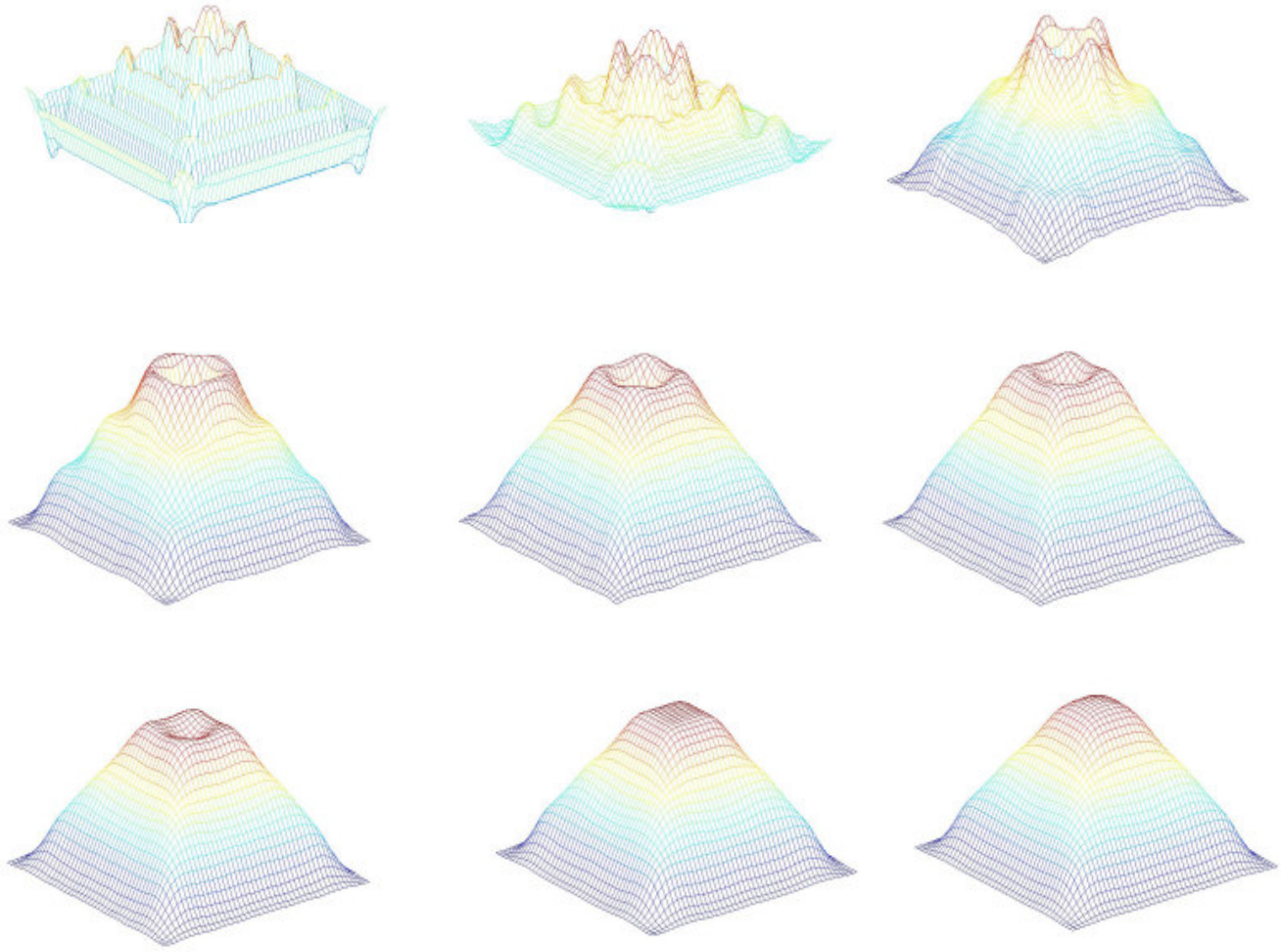

Figure 34. LSQR Solution of square.dat: Solution Progression with Default Initial Estimate, 10-90 Iterations.

Next, a second series of solutions was run, this time supplying an initial estimate.

MATLAB was used to solve the equation

z=LSQR(sparseMatrix, b, 10e-06, <iterations>, [ ], [ ], x0);

where x0 was the initial estimate computed by the fast Laplacian solver and <iterations> again ranged from 10 to 90. This series of mesh plots is shown below in the series of images in Figure 35.

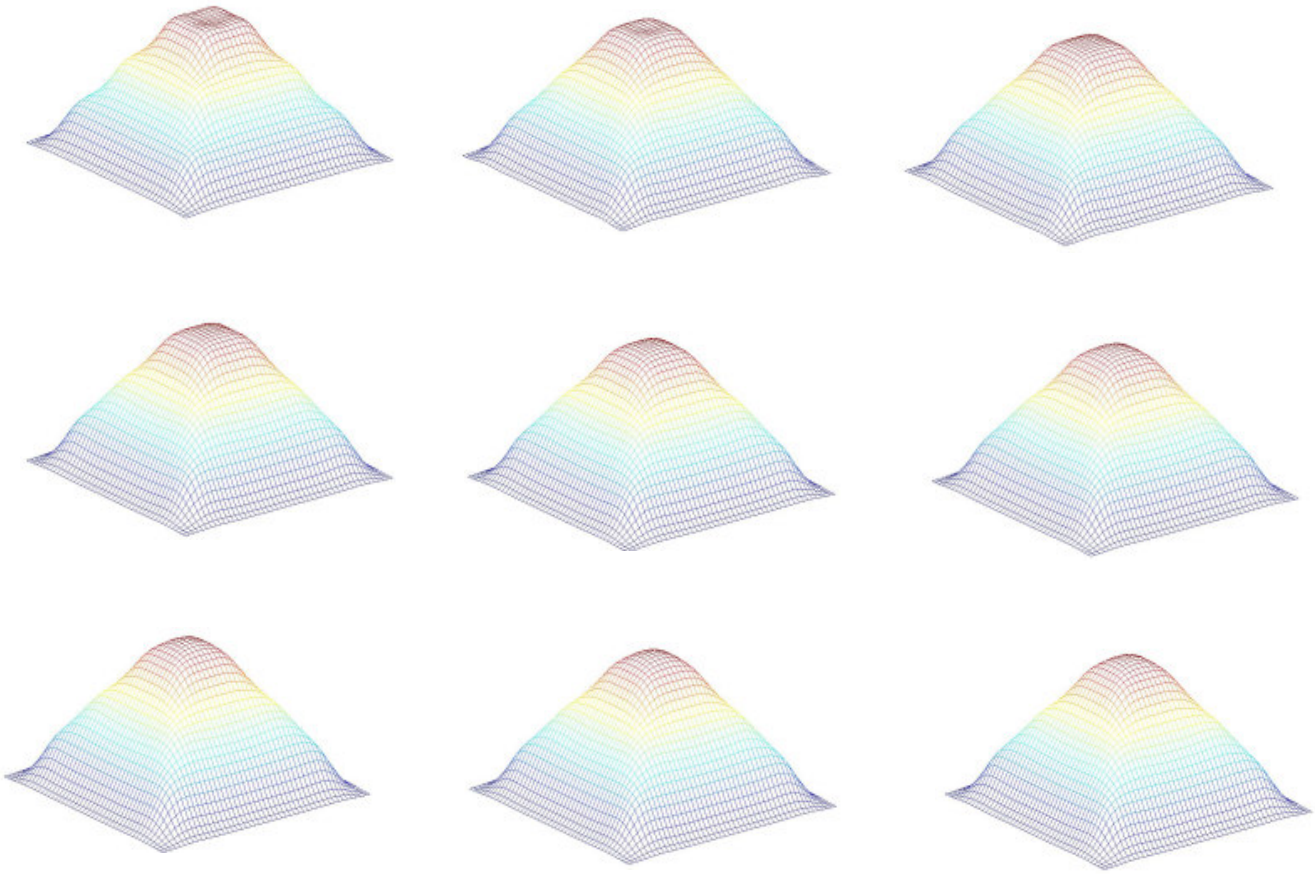

Figure 35. LSQR Solution of square.dat: Solution Progression with Laplacian Solver Initial Estimate, 10-90 Iterations.

Examination of the convergence performance of these two test groups is interesting and highlights many of the convergence characteristics observed during the previous test using large input files. The most important feature is the very marked improvement of the approximation after only ten iterations of the very coarse initial estimate (which is shown as the first of three figures in Section 9). This indicates that LSQR computing the Franklin algorithm with a fast Laplacian initial estimate would be expected to converge to a well conditioned surface relatively quickly.

The obvious difference between the two groups is the much more rapid early convergence of the second (two-level solution) group, the one that was computed by LSQR with the coarse Laplacian initial estimate. The convergence after ten iterations of this group (the first figure at the upper left) is better than the convergence of the second group after seventy iterations (the figure at the lower left corner of the first group of figures).

It is also apparent however that the difference between each successive iteration is less than that of the first group. That is, the degree of improvement after each successive iteration decreases as the current approximation converges toward the solution vector.

10.0 Comparison of the Solution Vector produced by the MATLAB Direct Solver with the LSQR Solution for a Large Input File

The final test in this series was designed to investigate the quality of the solution vector produced by three solvers. The standard was the solution vector produced by the MATLAB direct solver. The first test output was the solution vector produced by LSQR without an initial estimate and the second test output was the two-level solver running LSQR and an initial estimate computed by the fast Laplacian solver.

It was necessary to determine if the results observed in the tests run on square.dat would hold true for a larger input file. The largest file that could be processed to completion by the MATLAB direct solver on the test hardware platform was about 400 by 400 elevation nodes. The large bountiful.dat was subsetted to this size. The input file was converted to sparse representation using sparseA.m and the RHS vector was computed using makeB.m. Then the solution was computed by the MATLAB direct solver.

Next, LSQR computed a solution vector using no initial estimate and 5000 iterations.

Finally, LSQR computed a solution vector using an initial estimate prepared by the Laplacian solver and run for 500, 1000 and 1500 iterations.

Several techniques were used to determine the quality of the solution vectors. The first was direct inspection of the mesh plots. This turned out to be a difficult indicator of solution quality, as careful inspection of these plots was required to spot the differences. The mesh plots for the coarse initial estimate and the MATLAB direct solution are shown side by side in the figures below. The figure on the left (Figure 36) is the coarse initial estimate produced by the Laplacian solver. The figure on the right (Figure 37) is the control produced by MATLAB running the direct solver.

Careful inspection of the Laplacian plot in Figure 36 shows pronounced scalloping at the edges that appear to be an artifact of the contour lines. These features are indicated by the arrows in the plot.

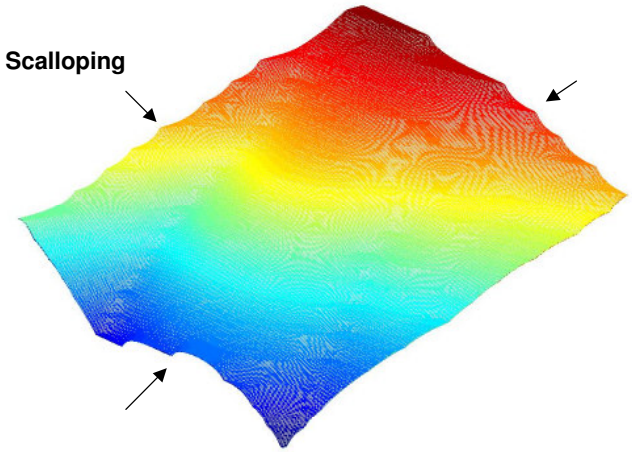

Figure 36. Laplacian Fast Solver Solution of bountiful_subset.dat.

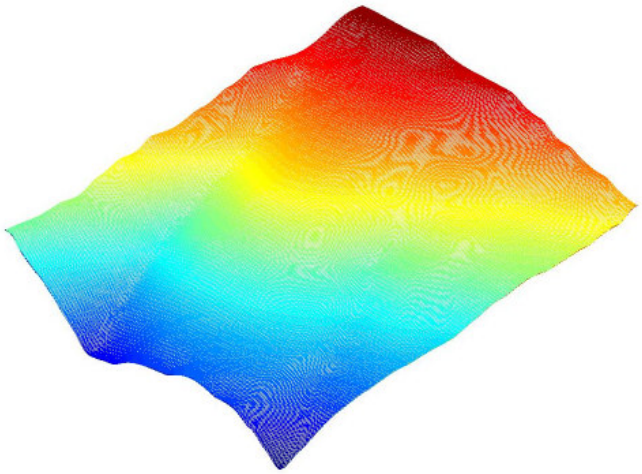

Figure 37. MATLAB Direct Solution of bountiful_subset.dat

The next two plots are the solution vectors produced by LSQR run for 500 iterations using the Laplacian solution as the initial estimate and LSQR run for 5000 iterations using no initial estimate. The laplace/LSQR initial estimate is on the left and the LSQR with the default initial estimate is on the right.

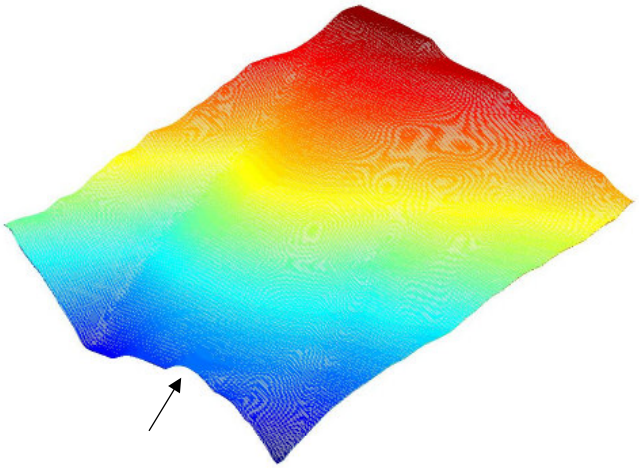

Figure 38. LSQR Solution of bountiful_subset.dat. 500 Iterations with Laplacian Initial Estimate. Note the edge scalloping where indicated.

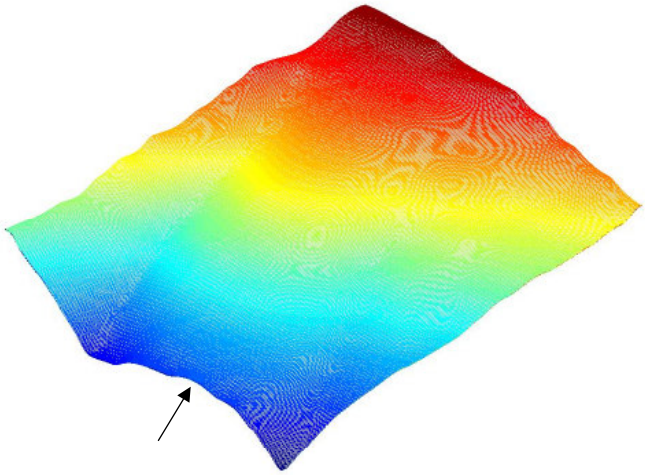

Figure 39. LSQR Solution of bountiful.dat, 5000 Iterations using default initial estimate. The scalloping is reduced due to the higher iteration count.

Note the decrease in scalloping of the initial estimate plot (Figure 38) on the left compared to the straight initial estimate plot in Figure 39. However, the fast Laplacian result is not as smooth as the straight LSQR plot (5000 iterations) on the right.

The final two plots show the result of LSQR running for 1000 iterations and 1500 iterations using the Laplacian (fast) initial estimate. The 1000 iteration result is on the left (Figure 40) and the 1500 iteration result is on the right (Figure 41). The quality of the latter plot is very close to both the 5000 iteration LSQR (default initial estimate) and MATLAB direct solutions.

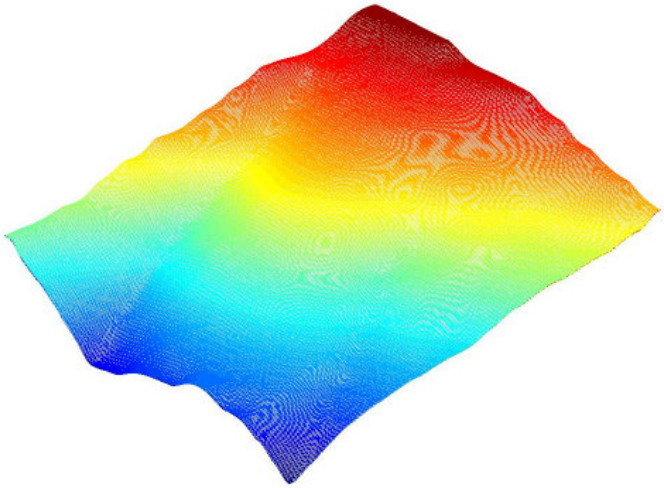

Figure 40. LSQR Solution of bountiful.dat, 1000 Iterations using Laplacian initial estimate.

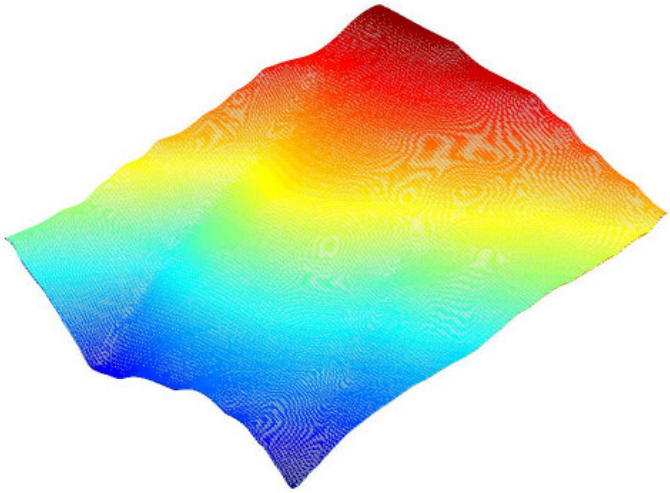

Figure 41. LSQR Solution of bountiful.dat, 1500 Iterations using Laplacian initial estimate.

In order to compare the solutions more carefully, it is useful to compute difference matrices. The plot in Figure 42 is the difference matrix produced by subtracting the MATLAB direct solution from the default LSQR (5000 iterations). The plot in Figure 43 is the difference between the direct MATLAB solution and the LSQR/laplacian 1500 iterations.

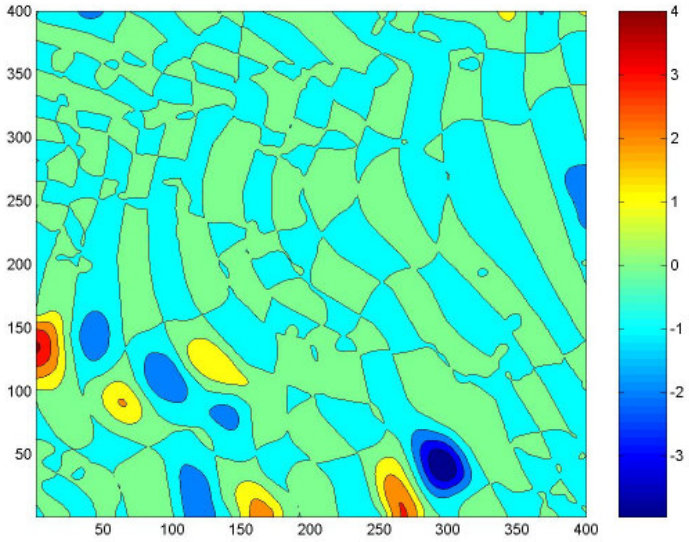

Figure 42. LSQR Solution of bountiful.dat, 5000 Iterations using default initial estimate. MATLAB contourf Plot of difference matrix.

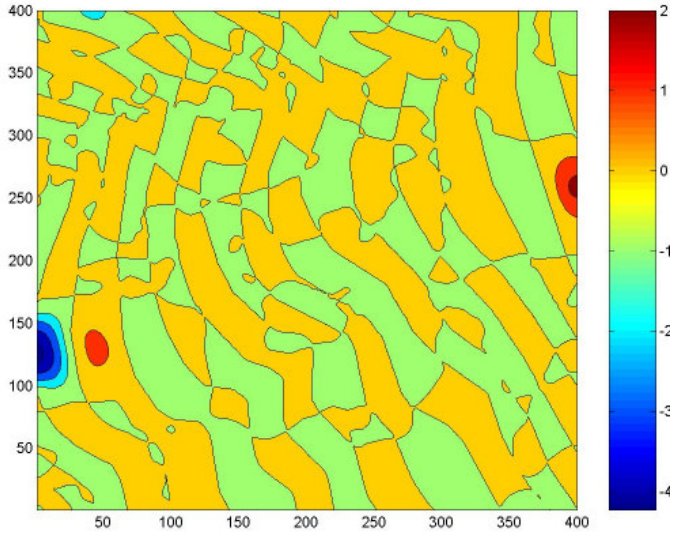

Figure 43. LSQR Solution of bountiful.dat, 1500 Iterations using Laplacian initial estimate. MATLAB contourf Plot of difference matrix.

The plots show that the 1500 iteration (with initial estimate) plot is slightly better overall than the corresponding 5000 iteration LSQR without the initial estimate. The areas of localized divergence in the two difference plots are characteristic signatures of iterated solutions with insufficient iterations.

Next, an LSQR computation with an initial estimate was run for 3000 LSQR iterations. The difference plot (using the MATLAB direct solution as a standard) is shown in the Figure 44 below. It shows that the LSQR computation run for 3000 iterations using an initial estimate far exceeds the quality of the LSQR computation run for 5000 iterations without the initial estimate (see figure above). The maximum deviation from the direct solution is –0.2 feet. Although convergence is not complete, agreement with the direct solution is very good and far better than the same LSQR computation without the initial estimate. This is consistent with all earlier observations that indicate the advantage of more rapid convergence of the LSQR solution when it is supplied with a coarse but high quality initial estimate.

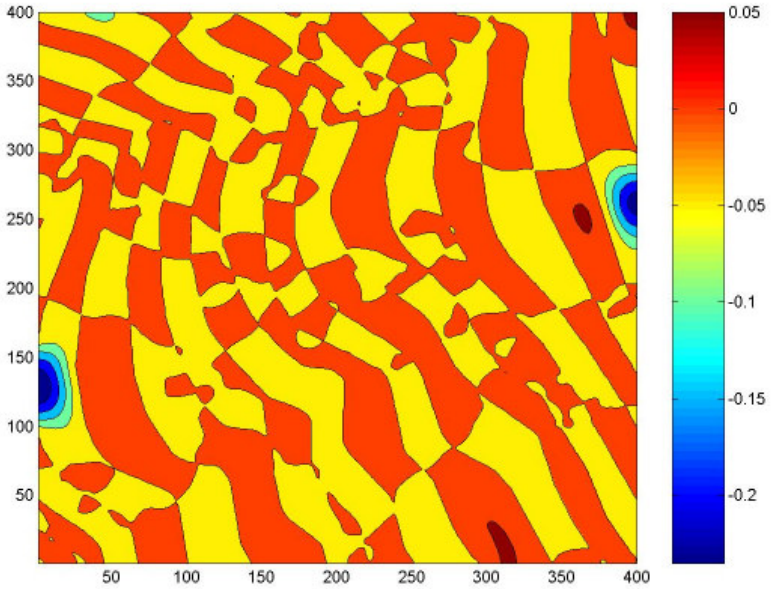

Figure 44. LSQR Solution of bountiful.dat, 3000 Iterations using Laplacian initial estimate. MATLAB contourf Plot of difference matrix.

A final test was conducted to determine if LSQR would converge to the same solution as the MATLAB direct solver if run for sufficient iterations. In this test LSQR without an initial estimate was run for 10,000 iterations. At the end of the run, the maximum deviation from the direct solution was –1.076e-004 feet, indicating that excellent agreement with the direct solution ultimately.

Although computation times were not reported in this section, the cost of the Laplacian iterative computation for 5000 iterations was negligible: 59 seconds of CPU time. The cost of the 2000 LSQR iterations was about one hour of CPU time on the test hardware platform.

The series of tests described in this section indicate the following:

1) Large elevation grids containing cliffs ending in lakes are difficult to interpolate. If the tuckermans.dat file had been 1201 by 1201 nodes it might have been impossible to process on the test computer due to the large number of refinement iterations required. Additional techniques are probably required to handle the special case of cliffs.

2) The technique developed in this report allows solution of the target 1201 by 1201 system, albeit after eight hours of processing time. Upgrading memory from 256MB to 512MB would probably improve this performance considerably, as extensive memory paging was required to handle the large problem. This memory paging was absent during solution of the 900 by 900 node crater.dat file.

3) The fast iterative Laplacian solver seems to be a useful tool for preparation of initial

estimates for LSQR solving the Franklin algorithm. It seems clear that using LSQR running the Franklin algorithm with no initial estimate is a waste of computational effort. A less elegant but much faster algorithm can do the initial part of the computation just as well as the better but slower Franklin algorithm solved by LSQR. The Franklin algorithm can then be used to finish the computation, producing a result as good as if no initial estimate had been supplied to LSQR running a greater number of iterations. This solution was also as good as the direct solution in all cases studied.

5) Two fundamental problems with iterative solvers present obstacles to the technique developed in this report for solving the Franklin interpolation algorithm: the slow spread of updated information across the solution vector in general and a slowing of the spread of updated information as the current iteration approaches the ultimate solution vector.

This can mean that although a tremendous computational time savings results in using a fast solver to generate an initial estimate, the final fractional improvements can still take a great deal of time (i.e. the final 5% of the solution takes 95% of the time.) An added problem is that it is not clear when the ultimate solution has been achieved, as statistics like relative residual are not very useful for this application. Nor is the quality of the solution (that is the difference between the present iteration and the correct answer) readily available.

6) Follow up tests using the small test file square.dat seem to validate the two-level solution method proposed.

7) Tests showed that an LSQR system that was supplied the initial estimate converged to a solution closely approximating the MATLAB direct solution vector much more quickly than a similar LSQR system using the default initial estimate.

8) Comparison of the MATLAB direct solver, LSQR without an initial estimate, and LSQR with an initial estimate supplied by the fast Laplacian solver showed that the LSQR solution agreed well with the direct solution, the agreement dependent on the number of iterations run. Supplying a good initial estimate increased convergence significantly, decreasing the number of iterations to achieve a given degree of convergence by about half.

11.0 Comparison of the Franklin Algorithm with the Able Software R2V Native Interpolation Algorithm

Although it was known that the Franklin approximation algorithm produced better results than theoretical alternatives, it was not known how this algorithm would compare to those implemented in commercial software packages. The details of such algorithms are generally not publicized.

The commercial package R2V from Able Software, Inc. is a multipurpose application capable of all of the operations described in Section 1 necessary to convert a raster contour line map to a DEM. A demonstration version of this software is available for free download. Export files are limited to 256 by 256 elevation nodes but this restriction did not pose an obstacle as a large input file was not needed in order to compare the quality of the surfaces generated.

The first test compared the ability of R2V to handle the interpolation of the difficult square.dat input file with the Franklin approximation algorithm solved using the two-level technique developed as part of this project. R2V is a commercial software package that runs on the Windows OS. The application costs around $1495 at the time of this writing. One of the main applications is the extraction of digital elevation models (DEMs) from contour maps. In order to accomplish this, R2V reads a contour map (for example a USGS topo map contour layer) as a TIF file. The program automatically vectorizes the raster contours using a line following vectorization method [2]. The operator is then required to manually or semi-manually edit the vectors and then tag them with their elevation values. When this is complete, the application performs a vector to raster conversion using the elevation vector attributes to create a raster elevation input file. Finally the program interpolates the elevation input file to a raster DEM.

In order to test R2V, square.dat (which is a simple ASCII text file) had to be converted into a corresponding TIF file that R2V could import. In order to do this, the student-supplied C++ program ‘ascii2tga’ which converts such a text file to TGA format was used. (TGA was chosen as an intermediate graphic format because the student already possessed a TGA writer that was modified to read a plain ASCII file instead of the USGS DEM format that it was originally written to convert.) This program is attached in Appendix 3. The square.dat file was slightly modified such that the elevation values were scaled to span the 8-bit grayscale color space and a border of zero padding elevations were added to solve an R2V clipping problem. The result was a 50 by 50 elevation node input file.

After suitable image processing in the image processing application Paintshop Pro (principally binarization) the file was imported into R2V, vectorized, tagged, and interpolated.

As a means of comparison, the same input file was interpolated using the Franklin approximation algorithm and the solution technique described in previous reports. The results of these efforts are shown in the following two figures. Figure 45 shows the R2V

interpolation and the one on the right (Figure 45) is the Franklin approximation.

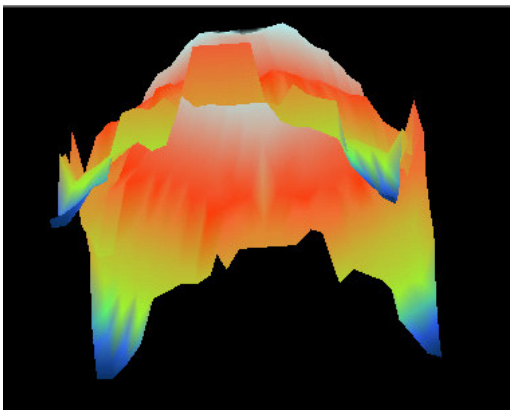

Figure 45. R2V Solution of square.dat

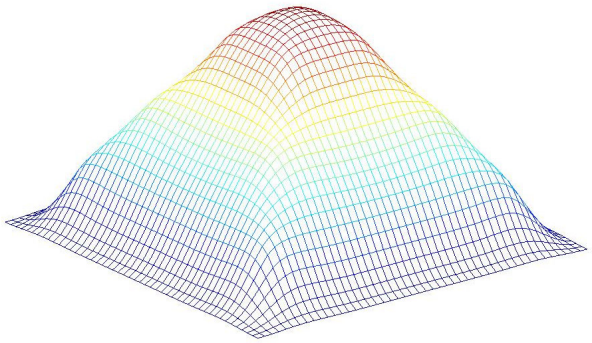

Figure 46. LSQR Solution of square.dat using Laplacian initial estimate.

The differences are striking. R2V was not able to handle square.dat appropriately. A very poor (distorted) surface resulted, the appearance suggestive of an iterative interpolation used without sufficient iterations. On the other hand, the Franklin method handled the interpolation as expected, producing the usual smooth surface.

Test 2. Comparison of R2V and the Franklin Algorithm Solving tuckermans_subset.dat

A less severe a test was prepared from tuckermans_subset.dat. This was also a 50 by 50 elevation node input file representative of a more "normal" terrain. A TGA file was prepared using ascii2tga.exe as before and imported into R2V. The same sequence of vectorization, elevation tagging and interpolation described above was again used.

The results of this interpolation are shown in the following two figures. Figure 47 shows the R2V interpolation and Figure 48 on the right is the Franklin interpolation of the same input file.

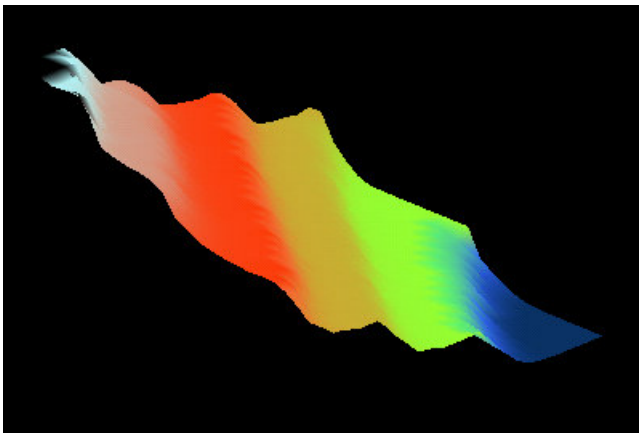

Figure 47. R2V Solution of bountiful_subset.dat.

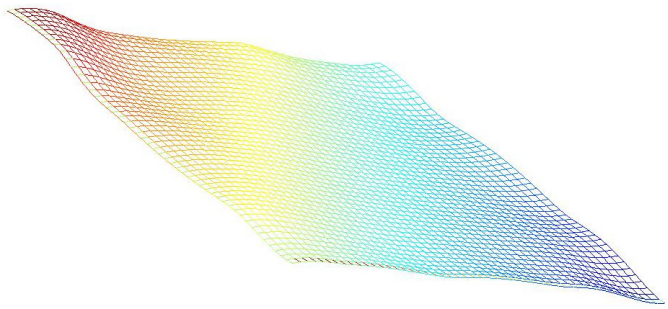

Figure 48. LSQR Solution of bountiful_subset.dat using Laplacian initial estimate.

This time R2V did a little better, at least producing a surface. However, it is obviously deeply terraced. Alternatively, the Franklin algorithm has yielded a much smoother surface with little or no terracing.

This commercial comparison test indicated the following:

1) The commercial program R2V from Able software produced results inferior to those produced by the Franklin algorithm using the two-level solution method developed during the course of this project. The algorithm used by R2V probably employs no better than a Laplacian (nearest four neighbor) interpolation algorithm. The results from the square.dat test indicate that the implementation of even this inferior algorithm is flawed, because the result of the R2V interpolation attempt was a very badly distorted surface that did not resemble the expected surface very much at all.

2) It is likely that the Franklin algorithm would out-perform the R2V algorithm in almost every interpolation application.

3) An interesting artifact of Franklin algorithm and of interpolation algorithms in general was discovered during this investigation. The initial approach to creation of an input file was to simply draw elevation contours using the well known graphic application Paintshop Pro and then inport them into R2V and into the Laplacian interpolator as TIF and GIF files, respectively. In order to produce the base map for LSQR, the Laplacian solver was run for zero iterations. This resulted in a contour map with the contours commonly four or five elevation values thick.

The Franklin algorithm had trouble with this base contour map, typically converging to a much more terraced result than if the contour lines were only one elevation value thick.

Investigation indicated that this is a well-known problem with all interpolation algorithms multiple pixel line widths cause additional equations of the form:

$$z_i = c_i$$

to be generated and added to the coefficient matrix. This was equivalent to creation of a small terrace or plateau. The sensitivity of the algorithm to this effect must be kept in mind when preparing input files and may be a good subject for further investigation.

12.0 Examination of the CatchmentSIM-GIS 8-32 Ray Interpolation Algorithm

CatchmentSIM-GIS is a topographic parameterization and hydrologic analysis tool available for free download at www.uow.edu.au~cjf03/index.htm. The program incorporates a contour line to DEM interpolation algorithm.

The interpolation algorithm is based on a distance weighted average of a series of linear interpolations along a set number of cross-sections taken through a single elevation node. For example, a 180 degree arc may be divided into 8 increments by the algorithm. Interpolation rays are then initiated at the appropriate angles, and all rays are paired with a mirror ray which travels in the opposite direction. Once an interpolation ray and its corresponding mirror ray both intersect pixels with assigned elevations, linear interpolation is applied to determine the pixel elevation for that particular interpolation and mirror ray combination.

The final value for the pixel is based on a weighted average of all the cross-section interpolations. The basis for weighting each derived elevation is the distance between the intersected pixels that form each end of the cross-section relative to the total distance of all 16 cross-sections. The program allows the user to designate the number of interpolation rays (and mirror rays) that are used to interpolate the pixel elevation. The algorithm is shown schematically in Figure 49.

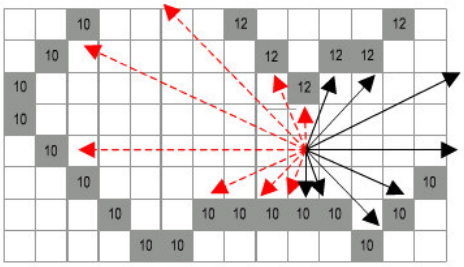

Figure 49. CachmentSIM Interpolation Algorithm (graphic: CachmentSIM-GIS)

The intent was to conduct the same tests used in the previous section in order to compare the results of this algorithm with the Franklin algorithm. Unfortunately, this application was discovered too late in the course of the project to accomplish this.

The problem was that CatchmentSIM required a MapInfo Interchange File (MIF) format for its input elevation file, which actually required a MID/MIF file pair. Although the MIF file format specification was obtained, the student was not able to understand it well enough to create test CatchmentSIM input files in time to include the test in this project.

As an alternative, the test contour line file included with the application was interpolated and the results of the interpolation were examined in as much detail as possible for quality of interpolation. Besides being unable to run square.dat or to compare the result directly with the Franklin algorithm, the comparison was hampered by the limited

rendering capabilities of CatchmentSIM. CatchmentSIM is only capable of rendering the DEM using a 2D contour plot, which experience has shown to be an inadequate indicator of terrain surface quality. However, sufficient indicators of interpolation quality could be obtained or inferred from analysis of the rendered results to make the analysis worthwhile and revealing.

The figure on the left (Figure 50) shows the 700 by 1000 pixel test input elevation contour plot. The MIF file from which this was derived consists of one polyline object for each contour line. Each polyline object has an elevation value listed in the MID file indexed to the contour lines.

The figure on the right (Figure 51) shows a 2D contour plot of the interpolated DEM using 8 rays per elevation node.

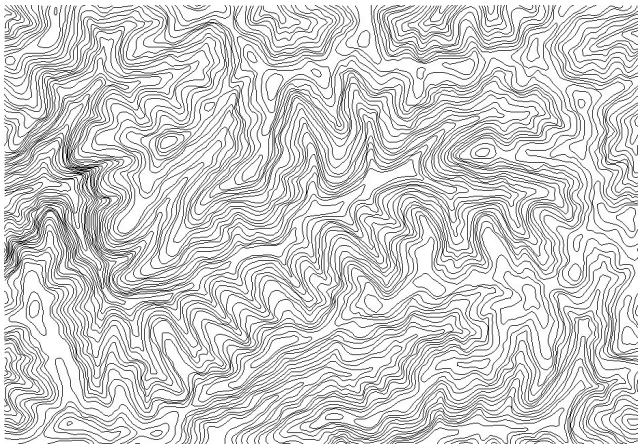

Figure 50. Contour Line Input Plot

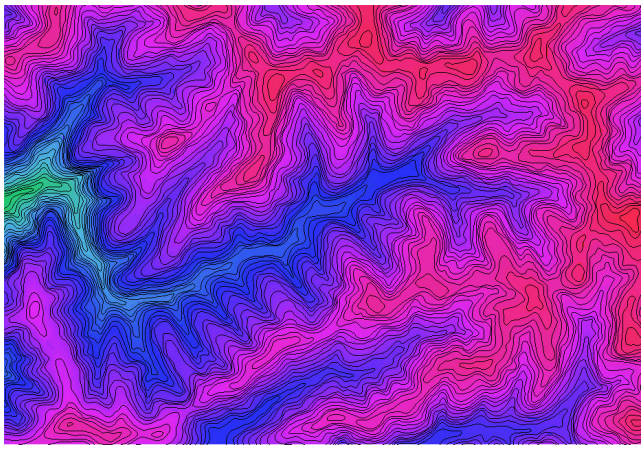

Figure 51. CachmentSIM Interpolated DEM with Superimposed Contour Lines

Although CachmentSIM does not have 3D rendering capability, it was possible to infer some information about the quality of the terrain surface by dragging the cursor across the map and inspecting the current pixel elevations reported on the screen. This analysis indicated that CachmentSIM did a very good job avoiding terracing. Elevations increased smoothly up or down the fall line with no drooping between contour lines apparent in any part of the map tested.

CachmentSIM did not do as well in any area where the contours were enclosed or almost enclosed. Enclosed contours (for example at the tops of hills) were interpolated to a constant elevation (plateau) as were many valley areas. CachmentSIM has a utility for finding flat areas and another one for fixing them. Figure 52 shows a section of the DEM where the flats have been detected. The solid black areas of the map are plateaus.

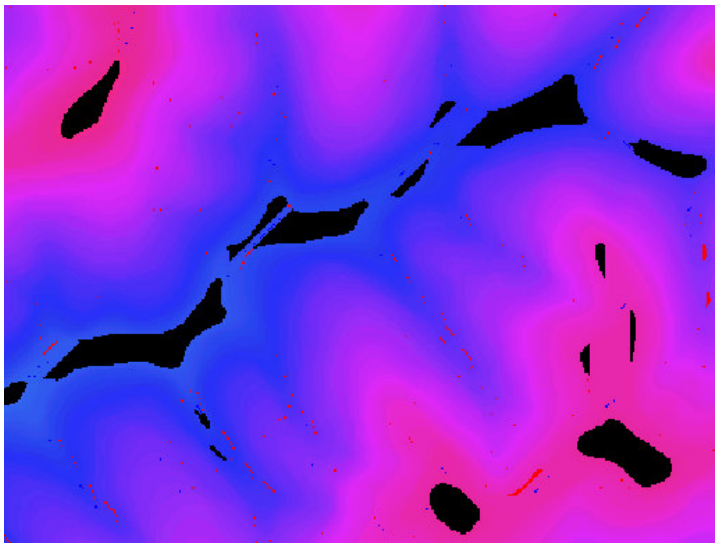

Figure 52. DEM Detail Showing Flat Areas (in black) and pits.

It is apparent that the interpolation algorithm used by CachmentSIM is very good at producing terrace-free slopes and is also very fast to compute. However, it suffers from plateaus as a result of its algorithm. The algorithm also produces small pits that seem to be distributed randomly across the map. The reason for these pits is not known, but is apparently an artifact of the interpolation algorithm used.

13.0 Analysis of the Results

Any success that this project achieved is a result of the early decision to abandon refined direct solution techniques in favor of an iterative solution approach. Although significant improvements in computational time were achieved by more optimal matrix reordering, these improvements were judged insufficient to accomplish the objective.

It is possible that this decision was hasty, as the reordering tests examined the improvements in computational time rather than improvements in storage requirements as a result of reduced fill-in. (Although computational times for direct solution of the Franklin algorithm are lengthy, the real problem is excessive storage requirements.) However, these tests would have been quite time consuming. Progressive computations increasing the size of test input files until out of memory errors caused failure would be required. It was assumed that the modest success in decreasing computational time would have been accompanied by similarly modest improvements in storage requirements.

The decision to investigate iterative solvers was influenced by prototyping work that was done during the proposal phase of the project, when the simple Laplacian iterative Java code was written. Iterative solvers can better exploit the matrix structure characteristic of the Franklin formulation, which can result in more compact data storage requirements. As a result they can address the main obstacle that prevented the solution of large input files using the Franklin algorithm: out-of-memory processing errors due to the large size of in-process data arrays.

As discussed in Section 2, direct solvers are presented with an approximately N^2 by N^2 matrix whereby each matrix row corresponds to the stencil equation of an elevation node with one stencil equation for each elevation node in the input file. The direct solver cannot attack this data structure directly because of its large size. Its optimization strategy is limited to transformation of the full matrix to (indexed) sparse matrix representation. This is in fact essential as the full N^2 by N^2 matrix for a 1201 by 1201 input grid exceeds 2.08e+12 data elements and cannot possibly be handled by any solver directly. It is only by indexing the relatively few non-zero coefficients that the system can be handled by the solver. This still results in a very large data structure for a desktop environment. (The ASCII index array for the 1201 by 1201 node input file exceeded 250MB in size.)

When this efficiency is exploited, the direct solver has little more to say about data structure. Its only remaining strategy is defensive: an educated guess to reorder the sparse matrix to (hopefully) prevent storage requirements from exploding to a hopelessly unmanageable size due to fill-in during computation. As discussed in Section 4, this must be done heuristically as the computation of the optimal ordering is NP-hard, and the development of useful heuristics is still an area of active research.

In particular, the MATLAB's direct linear least square attempts to solve the system

$$A^T A z = A^T b$$

by Q-less QR factorization as described in Section 3. The factorization ultimately requires row reduction, which probably causes excessive fill-in and leads ultimately to the out-of-memory computational failures observed for input grids larger than 400 by 400 nodes on the test hardware platform. This occurs despite MATLAB's native reordering scheme. As indicated in Section 4, computation time can be reduced by about 35% but the underlying problem probably cannot be eliminated or even managed to the degree necessary to solve the target problem.

The LSQR solver is also presented with the N^2 by N^2 sparse matrix as its input. However, an iterative solver solves systems of the general form

$$Mx^{(k+1)} = Nx^k + b$$

and as a result can define its computations in terms of matrix multiplications instead of row reductions. Thus fill-in does not occur and the memory use does not grow explosively due to in-process storage requirements.

Moreover, iterative solvers have more to say about data structure and are often able to exploit the coefficient matrix structure much more effectively than a direct solver. For example, the non-native MATLAB (i.e. C, FORTRAN and Saunders-defined LSQR.m) LSQR implementations do not require and in fact will not even accept a coefficient matrix as an argument directly. Instead, they require that the user define the function y = aprod(mode, m, n, x, iw, rw) were, depending on the value of 'mode' returns the either the vector y = Ax or y = A'x. In cases where the coefficient matrix is highly structured, this product can be defined in terms of a rule rather than an explicit matrix multiplication, precluding storing the coefficient matrix at all (see Saunders program lsqr.m and aprod.m user notes for an example of such a matrix).

The simple Laplacian solver used to generate the initial estimate for LSQR represents an even more efficient implementation. It does not even formulate the sparse matrix but instead formulates the equations that would have appeared in each row of the sparse matrix using a stencil equation that relates the next value of a given elevation to the current value of its four nearest neighbors. This can be done because the stencil is the same for each node and can therefore be expressed in terms of indices rather than explicitly. Because it is a Gauss-Seidel iterative solver it does not even need a temporary data structure but instead simply updates the input elevation grid with each successive calculation. As a result of this scheme, its data storage requirements are $O(N^2)$ and it is very fast.

It may in fact be possible to apply this very efficient approach to the over determined system as well, as both A and A^T have only two types of equations: the five element nearest neighbor stencil and the one element known elevation stencil. Writing a least squares solver to exploit this structure would preclude LSQR completely and could result in a very fast and straightforward computation of the Franklin algorithm.

Despite these considerable advantages, there are important problems exhibited by iterative solvers, several of which were encountered in this study. The first problem is convergence. Although iterative solvers hopefully converge to the correct solution (for example the solution vector produced by the direct solver) this is not guaranteed.

This could arguably be overlooked if the class of systems being solved was well understood and known to converge to the correct solution in many cases. Even if this was accepted, the second major problem associated with iterative solvers would still exist. That is, even if the iterative solver converges to the correct solution, it can not be known how close a given iteration solution vector is to the correct solution vector without knowledge of the correct solution ahead of time. This is obviously unavailable.

Conventional statistics such as residuals are of little use for terrain interpolations. This is because, as the solution approaches convergence, many of the computed elevation values may be nearly correct but the terrain may be very incorrect locally. This occurred in processing crater.dat (see Section 8). Another example occurred in the Section 10 analysis.

Another method commonly used to test for convergence is to compute an overall change statistic between successive iterations. The idea is, when there is insignificant change between the successive iterations, the computation is complete.

However, this can fail for the same reason. Since the solution may only be incorrect locally, even large local changes may be "washed out" when aggregated with the vast majority of unchanging elevation values.

There are other interfering factors. Information typically propagates very slowly across the solution vector as the iteration number increases. A well known characteristic[9] of iterative solvers is their tendency to "stall out" far from convergence after the high frequency components of the solution vector are removed.

As a result, although the solution of input files of the target size is demonstrably possible, the techniques developed during this project fall somewhat short of a fully practical method. For example, although the solution of a 1201 by 1201 node bountiful.dat subset was demonstrated, it is not known how close that solution is to the correct solution. Prospective users of the technique are left with the following choices to judge solution quality:

1) Use qualitative methods such as examining the solution vector for characteristic or signature features indicating non-convergence (scalloping along the edges of the solution plot, suspicious mounds or sinks where none are expected based on inspection of the contours, etc.);

2) Comparison of run statistics to those of similar known solutions (that is, if a particular combination of coarse solver and fine solver iterations worked well for a previous input

grid, the same combination might work well for a new similar elevation grid);

3) "Bombing" the problem by running a huge number of iterations judged (again based on experience) to vastly exceed the number required for the degree of convergence desired. This is practical as long as the cost of particularly the fine solver iterations is not too great.

14.0 Conclusions

It is considered that the main objective of this project, the development of a method to compute the Franklin approximation for DEMs as large as 1201 by 1201 elevation nodes, using an unsophisticated desktop computer, was technically accomplished. Prior to starting this work, Dr. Franklin demonstrated the inability of the MATLAB direct solver, running on a laptop computer with 1024MB RAM to process an (over determined) 800 by 800 elevation node system (crater.dat) to completion.

The method developed as a result of this project, the two-level iterative computation, has been demonstrated to compute input files of 800 by 800, 900 by 900, and 1201 by 1201 elevation nodes to completion. The test hardware platform with 256MB RAM was only marginally adequate for the 1201 by 1201 input file. However, it is considered likely that a similar desktop platform with 512MB RAM would easily handle the 1201 by 1201 input file and also significantly larger files. This memory requirement to quickly compute the target input file size is not unreasonable considering the ever decreasing cost of RAM.

Follow up studies have demonstrated that the quality of the solution vector for small problems is identical to that produced by direct solution of the system of equations. A further study demonstrated the general effectiveness of the method used to speed convergence, i.e. the generation of a coarse initial estimate using a fast solver running a low-quality algorithm, again for small problems.

A third study using a larger (400 by 400 elevation node) input file confirmed the general findings of the studies of smaller files. This analysis confirmed the utility of the LSQR iterative solver and the benefit in terms of convergence improvement of computing an initial estimate using a fast solver.

A final study demonstrated that the Franklin algorithm is markedly superior to an interpolative algorithm offered by a commercial DEM extraction software package that is considered a leading product in its field. This study also indicated the superiority of the DEM surfaces computed by the Franklin algorithm as compared to surfaces computed by the hydrologic application CachmentSIM-GIS. Although the interpolation algorithm used in CachmentSIM is clearly superior to that used in R2V, it is inferior to the Franklin algorithm in its interpretation of hilltops and valleys.

15.0 Future Work

Several areas for future work are suggested by this study.

1) The development of better methods to estimate the degree of convergence, as discussed above. These would ideally take into account the localized nature of errors and other convergence characteristics unique or common to iteratively interpolating terrains.

2) Porting of the program to a C or FORTRAN implementation of LSQR would probably result in faster computational times and thus reduce the cost of LSQR iterative passes. Lowering the cost of iterative passes makes accurate techniques for the estimation of convergence less critical, as simply running excessive numbers of iterations would be cheap.

3) Ultimately, an integrated solver incorporating the Laplacian routines for preparation of the initial estimate, the LSQR solver, and methods for estimating the quality of the solution would possibly yield a very useful software product.

4) Development of techniques for handling special cases, such as cliffs would result in the most broadly applicable implementation of the Franklin algorithm.

5) Writing a specialized least squares solver that exploits the structure inherent in the problem as discussed in Section 12 is probably the most interesting area for future investigation of all. If this could be done, much of the other future work identified would be unnecessary. The cost of each computational iteration would be so small such that convergence issues would be largely irrelevant, since excessive iterations could be run much more cheaply even for large problems.

16.0 References

[1] P Arrighi, P Soille 'From Scanned Topographic Maps to Digital Elevation Models'. In Proceedings of Geovision '99, International Symposium on Imaging Applications in Geology (ICA) University of Liege, Belgium, Page 1.

[2] D. L. Powers, "Machine Interpretation of Line Drawing Images" , Springer-Verlag, 2000

[3] W. R. Franklin.: 'Applications of Analytical Cartography' *Cartography and Geographic Information Systems 27(3),* 2000, pp. 225-237

[4] J.H. Matthews Numerical Methods for Mathematics, Science and Engineering Prentice-Hall 1992.

[5] M. K. Gousie and WR Franklin. 'Converting Elevation Contours to a Grid' In *Eighth International Symposium on Spatial Data Handling (SDH)*, Vancouver BC Canada, July 1998. Dept of Geography, Simon Fraser University, Burnaby, BC, Canada.

[6] D. C. Lay Linear Algebra and its Applications Addison Wesley 2000.

[7] G. Karypis and V. Kumar, LSQR: "Metis: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices", University of Minnesota, Department of Computer Science System Documentation (1998).

[8] C. C. Paige and M. A. Saunders, "LSQR An Algorithm for Sparse Linear Equations and Sparse Least Squares", ACM TOMS 8(1), 43-71 (1982).

[9] W. L. Briggs, C. E. Hensen and S. F. McCormick, "A Multigrid Tutorial", Society for Industrial and Applied Mathematics (2000).

[10] W. R. Franklin and M. K. Gousie 'Terrain Elevation Data Structure Operations'. In *19th International Cartographic Conference & 11th General Assembly of the International Cartographic Association (ICA)* Ottawa Canada, August 1999

[11] M. K. Gousie, W. R. Franklin 'Contour to DEM' , Presentation Notes 1998.

[12] D. L. Powers, "Boundary Value Problems" , Academic Press 1972 pp. 113-117.

[13] D. R. Kincaid and L. J. Hayes, "Iterative Methods for Large Linear Systems" Academic Press (1990).

[14] G. H. Golub and C. F. Van Loan, "Matrix Computations", The John Hopkins University Press (1996).

Appendix 1 MATLAB Developmental Code Modules

```
function sparseMatrix = sparseA(inputMatrix)
%
%   inputMatrix: rectangular input matrix
%   returns: sparse banded matrix sparseMatrix
%   of size M+K by M
%   where M is m X n
%   and K is k by 1.
%
%   where k is the number of non-zero
%   (known elevation) input array elements.
%
%   The expanded and awkward code
%   is designed to preserve the symmetry
%   of the mapping from the input matrix
%   'inputMatrix' to the output matrix
%   'sparseMatrix.'
%
%   The mapping is a little tricky and
%   writing it this way helps me to keep
%   the bookkeeping straight.

SIZE=size(inputMatrix);
rows=SIZE(1,1);
columns=SIZE(1,2);
sparseMatrix=spalloc(rows*columns, rows*columns, 5*rows*columns);
sparseRows=0;

%-----------------------------------------------
%   The first section of the code sets up the
%   (mxn) by (mxn) upper block of the sparseMatrix.
%   This is
%   the section corresponding to equations of the
%   form: Zl + Zr + Zu + Zd - 4Zi.  Most of the
%   code is for handling the edge nodes.
%-----------------------------------------------
%handles the upper left node

for i=1:1
    for j=1:1
        sparseRows=sparseRows+1;
        keyColumn=((i-1)*rows + j);
        rightColumn=keyColumn+1;
        lowerColumn=((i)*columns + j);
        sparseMatrix(sparseRows, keyColumn)=-2;
        sparseMatrix(sparseRows, rightColumn)=1;
        sparseMatrix(sparseRows, lowerColumn)=1;
    end
end

%-----------------------------------------------
%handles the upper right node
for i=1:1
    for j=columns:columns
        sparseRows=sparseRows+1;
        keyColumn=((i-1)*rows + j);
        leftColumn=keyColumn-1;
        lowerColumn=((i)*columns + j);
        sparseMatrix(sparseRows, keyColumn)=-2;
        sparseMatrix(sparseRows, leftColumn)=1;
        sparseMatrix(sparseRows, lowerColumn)=1;
    end
end

%-----------------------------------------------
%handles the lower left node

for i=rows:rows
```

```
    for j=1:1
        sparseRows=sparseRows+1;
        keyColumn=((i-1)*columns + j);
        rightColumn=keyColumn+1;
        upperColumn=((i-2)*columns + j);
        sparseMatrix(sparseRows, keyColumn)=-2;
        sparseMatrix(sparseRows, rightColumn)=1;
        sparseMatrix(sparseRows, upperColumn)=1;
    end
end

%-----------------------------------------------
%handles the lower right node

for i=rows:rows
    for j=columns:columns
        sparseRows=sparseRows+1;
        keyColumn=((i-1)*columns + j);%
        leftColumn=keyColumn-1;
        upperColumn=((i-2)*columns + j);
        sparseMatrix(sparseRows, keyColumn)=-2;
        sparseMatrix(sparseRows, leftColumn)=1;

        sparseMatrix(sparseRows, upperColumn)=1;

    end
end

%-----------------------------------------------
%handles the left boundary

for i=2:rows-1
    for j=1:1
        sparseRows=sparseRows+1;
        keyColumn=((i-1)*columns + j);
        rightColumn=keyColumn+1;
        upperColumn=((i-2)*columns + j);
        lowerColumn=((i)*columns + j);
        sparseMatrix(sparseRows, keyColumn)=-3;
        sparseMatrix(sparseRows, rightColumn)=1;
        sparseMatrix(sparseRows, upperColumn)=1;
        sparseMatrix(sparseRows, lowerColumn)=1;
    end
end

%-----------------------------------------------
%handles the right boundary

for i=2:rows-1
    for j=columns:columns
        sparseRows=sparseRows+1;
        keyColumn=((i-1)*columns + j);
        leftColumn=keyColumn-1;
        upperColumn=((i-2)*columns + j);
        lowerColumn=((i)*columns + j);
        sparseMatrix(sparseRows, keyColumn)=-3;
        sparseMatrix(sparseRows, leftColumn)=1;
        sparseMatrix(sparseRows, upperColumn)=1;
        sparseMatrix(sparseRows, lowerColumn)=1;
    end
end


%-----------------------------------------------
%handles the upper boundary

for i=1:1
    for j=2:columns-1
        sparseRows=sparseRows+1;
        keyColumn=((i-1)*rows + j);
        leftColumn=keyColumn-1;
```

```
        rightColumn=keyColumn+1;
        lowerColumn=((i)*columns + j);
        sparseMatrix(sparseRows, keyColumn)=-3;
        sparseMatrix(sparseRows, leftColumn)=1;
        sparseMatrix(sparseRows, rightColumn)=1;
        sparseMatrix(sparseRows, lowerColumn)=1;
    end
end

%-----------------------------------------------
%handles the lower boundary

for i=rows:rows
    for j=2:columns-1
        sparseRows=sparseRows+1;
        keyColumn=((i-1)*columns + j);
        leftColumn=keyColumn-1;
        rightColumn=keyColumn+1;
        upperColumn=((i-2)*columns + j);
        sparseMatrix(sparseRows, keyColumn)=-3;
        sparseMatrix(sparseRows, leftColumn)=1;
        sparseMatrix(sparseRows, rightColumn)=1;
        sparseMatrix(sparseRows, upperColumn)=1;
    end
end

%-----------------------------------------------
%handles the inside nodes

for i=2:rows-1
    for j=2:columns-1
        sparseRows=sparseRows+1;
        keyColumn=((i-1)*columns + j);
        leftColumn=keyColumn-1;
        rightColumn=keyColumn+1;
        upperColumn=((i-2)*columns + j);
        lowerColumn=((i)*columns + j);
        sparseMatrix(sparseRows, keyColumn)=-4;
        sparseMatrix(sparseRows, leftColumn)=1;
        sparseMatrix(sparseRows, rightColumn)=1;
        sparseMatrix(sparseRows, upperColumn)=1;
        sparseMatrix(sparseRows, lowerColumn)=1;
    end
end

%-----------------------------------------------
%   The next section of the code sets up the
%   mxk lower block of the sparseMatrix.  This is
%   the zection corresponding to equations of the
%   form: Zi = ei.
%-----------------------------------------------
%concatenates the K matrix

for i=1:rows
    for j=1:columns
        if inputMatrix(i,j) ~= 0.0
            sparseRows=sparseRows+1;
            sparseMatrix(sparseRows,:)=0;
            keyColumn=((i-1)*columns + j);
            sparseMatrix(sparseRows, keyColumn)=1;
        end
    end
end
```

```
function bVector = makeB(inputMatrix)
%   inputMatrix: input matrix
%   returns: constraint vector bVector
%   of size mxn+K X 1
%
%   K is the number of non-zero
%   (known elevation) input array elements.

SIZE=size(inputMatrix);
rows=SIZE(1,1);
columns=SIZE(1,2);
bVector=zeros(rows*columns,1);
sparseRows=rows*columns;

%-------------------------------------------------
%Makes the 'b' vector from the known elevations
for i=1:rows
    for j=1:columns
        if inputMatrix(i,j) ~= 0.0
            sparseRows=sparseRows+1;
            bVector(sparseRows)=inputMatrix(i,j);
        end
    end
end
```

```
function solutionMap = repack(inputVector, rows, columns)
%   inputVector: NX1 input vector
%   returns: NXN output matrix
%
%   This function maps the M+KX1 solution
%   vector resulting from solving the
%   M+K X N sparse matrix equations.
%
%
%   Note: input rows, columns where
%   rows = sparseMatrix precurser matrix rows;
%   columns = sparseMatrix precurser matrix columns

index=1;

%-------------------------------------------------
%Repacks the inputVector and builds the solutionMap
for i=1:rows
    for j=1:columns
        solutionMap(i,j)=inputVector(index,1);
        index=index+1;
    end
end
```

```
function writeMatrix(rows, columns, inputMatrix, fileName)
%
% rows - the number of rows of the inputMatrix
% columns - the number of columns of the inputMatrix
% inputMatrix - the input matrix to be written to file.
% fileName - a string file name, with suffix

%-------------------------------------------------
% open the output stream and write data to file.
fid=fopen(fileName, 'w');
for i=1:rows
   for j=1:columns
       fprintf(fid, '%s', ' ');
       fprintf(fid, '%d', inputMatrix(i,j));
   end
   fprintf(fid, '\n');
end
fclose(fid);
```

```
function metisGraph = makeGraph(sparseMatrix)
%
%   sparseMatrix: square input matrix
%   returns: METIS formatted graph matrix
%   of size sparseMatrixRows+1 by 13
%   where sparseMatrixRows is the number of nonzero
%   sparseMatrix elements
%   Note: use with squareSparseA.m to produce square sparse
%   matrix only.


%-----------------------------------------------
%   Compute some necessary parameters.
%-----------------------------------------------

[sparseMatrixRows, sparseMatrixColumns]=size(sparseMatrix);
metisGraphRows=sparseMatrixRows;
metisGraphColumns=10;
metisGraph=zeros(metisGraphRows, metisGraphColumns);
numberEdges=0;

%-----------------------------------------------
%   This section creates a sparse symmetric analog
%   to sparseMatrix with zeros on the diagonals.
%-----------------------------------------------

onesMatrix=spones(sparseMatrix);
diagonalMatrix=diag(diag(onesMatrix));
zeroDiagonalMatrix=((-1*diagonalMatrix) + onesMatrix);
[iZeroIndex, jZeroIndex]=find(zeroDiagonalMatrix);

%disp('zeroDiagonalMatrix is:  ')
%disp(zeroDiagonalMatrix)

for k=1:nnz(zeroDiagonalMatrix)
    zeroDiagonalMatrix(jZeroIndex(k),     iZeroIndex(k))=zeroDiagonalMatrix(iZeroIndex(k),
jZeroIndex(k));
  %  disp(zeroDiagonalMatrix(jZeroIndex(k), iZeroIndex(k)))
  %  disp(jZeroIndex(k))
  %  disp(iZeroIndex(k))
  %  disp(zeroDiagonalMatrix(iZeroIndex(k), jZeroIndex(k)))
  %  disp('  ')
end
nonzeroElements=nnz(zeroDiagonalMatrix);
numberVertices=nonzeroElements/2;
[iZeroIndex, jZeroIndex]=find(zeroDiagonalMatrix);

%disp('zeroDiagonalMatrix is:  ')
%disp(full(zeroDiagonalMatrix))

%-----------------------------------------------
%   This section builds the METIS graph matrix
%   from the MATLAB (modified)sparse matrix
%   representation.
%-----------------------------------------------

for i=1:sparseMatrixRows
    counter=1;
    for j=1: nonzeroElements
        if iZeroIndex(j)==i
            metisGraph(i, counter)=jZeroIndex(j);
            counter=counter+1;
            metisGraph(i, counter)=zeroDiagonalMatrix(iZeroIndex(j), jZeroIndex(j));
            counter=counter+1;
            numberEdges=numberEdges+1;
        end
    end
end

%-----------------------------------------------
%   This section builds the header for the METIS
```

```
%   file and concatenates it onto the first row.
%-----------------------------------------------

numberEdges=numberEdges/2;
[metisGraphRows, metisGraphColumns]=size(metisGraph);
numberNodes=metisGraphRows;
header=zeros(1,metisGraphColumns);
header(1,1)=numberNodes;
header(1,2)=numberEdges;
header(1,3)=1;
metisGraph = cat(1,header,metisGraph);
disp('number of nodes is'); disp(numberNodes)
disp('number of vertices is'); disp(numberVertices)
dlmwrite('outfile.graph', metisGraph, '\t');

end
```

```
function sparseMatrix = scale(sparseMatrix)
%   inputMatrix: a m by n sparse matrix
%   returns: a normalized sparse matrix
%   of size m by n
%

SIZE=size(sparseMatrix);
columns=SIZE(1,2);

%-----------------------------------------------
%Divide each column by its 2-norm.
for i=1:columns
    normalized=(1/norm(sparseMatrix(:,i)));
    sparseMatrix(:,i)=(sparseMatrix(:,i)) * normalized;
end
```

```
function c = clipper(sparseMatrix, b, x0, iterations, cycles, grid_size, zmin, zmax)
%   sparseMatrix: m by n sparse matrix
%   b: n by 1 RHS vector
%   x0 m by 1 initial guess vector
%   iterations: number of iterations per cycle
%   cycles: number of cycles to run.
%   grid_size: the size of the elevation grid
%   zmin: minimum elevation
%
%

SIZE=size(sparseMatrix);
rows=SIZE(1,1);
columns=SIZE(1,2);


%-------------------------------------------------
%Run LSQR through 'iterations' iterations.
%Then stop and remove any elevation excursions.
%Resume and repeat for 'cycle' computations.

for i=1:cycles
    %z=LSQR(sparseMatrix, b, 10e-06, iterations, [], [], x0);
    %x0=z;
    for j=1:columns
        if x0(j, 1) < zmin
           x0(j,1)=zmin;
        end
        if x0(j, 1) > zmax
            x0(j,1)=zmax;
        end
    end
    z=LSQR(sparseMatrix, b, 10e-06, iterations, [], [], x0);
    x0=z;
end
c=repack(z, grid_size, grid_size);
```

Appendix 2 Java Code Modules

```
/*
 * InputMatrix Class
 *
 *
 ************************************************************************
 *
 * Course:        ECSE 6980
 * Instructor:    William Randolph Franklin
 * Date:          January 26, 2003
 * Student:       John Childs
 *
 ************************************************************************
 *
 * Create a test input matrix for Matlab
 *
 ************************************************************************/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class InputMatrix extends JFrame
{

   private JTextField text1;
   private JTextField text2;
   private JTextField text3;
   private String s="";
   private String rows=null;
   private String columns=null;
   private String outputFileBuffer=null;


   private InputMatrix()
   {
      super("Input Parameters Box");

      Container c=getContentPane();
      c.setLayout(new FlowLayout());

      text1=new JTextField(20);
      text1.setToolTipText("Key in the rows hit Enter.");
      c.add(text1);

      text2=new JTextField(20);
      text2.setToolTipText("Key in the columns hit Enter");
      c.add(text2);

      TextFieldHandler handler = new TextFieldHandler();
      text1.addActionListener(handler);
      text2.addActionListener(handler);

      setSize(325, 200);
      show();
   }

   public static void main(String args[])
   {
      InputMatrix inputMatrix=new InputMatrix();

      inputMatrix.addWindowListener(
                        new WindowAdapter()
                        {
                           public void windowClosing(WindowEvent e)
                           {
                              System.exit(0);
                           }
```

```
                    }
                );
    }

    private class TextFieldHandler implements ActionListener
    {

        private FileInputStream fis;
        private FileOutputStream fos;

        public void actionPerformed(ActionEvent e)
        {
            if(e.getSource()==text1){
                s="input rows: "+ e.getActionCommand();
                rows=(String)e.getActionCommand();
                text2.requestFocus();
            }
            else if(e.getSource()==text2){
                s="input columns: " + e.getActionCommand();
                columns=(String)e.getActionCommand();
                text1.requestFocus();
            }

            if((rows != null) && ( columns!=null)){
                JOptionPane.showMessageDialog(null, rows + "  " + columns);
                createMatrix();
            }
        }

        private void createMatrix()
        {
            outputFileBuffer=JOptionPane.showInputDialog(null,"Enter Output File Name");
            try{
                FileWriter fw=new FileWriter(outputFileBuffer);
                BufferedWriter bw=new BufferedWriter(fw);
                PrintWriter outFile=new PrintWriter(bw);
                int counter=0;
                int contour=0;

                for(int i=1; i<=Integer.parseInt(rows); i++)
                {
                    counter=0;
                    contour=0;
                    outFile.print("\n");
                    for(int j=1; j<=Integer.parseInt(columns); j++)
                    {
                        if(counter % 10 ==0){
                            outFile.print(Integer.toString(contour));
                            outFile.print(" ");
                            contour=contour+100;
                        }
                        else
                            outFile.print("0 ");
                        counter=counter+1;
                    }
                }

                outFile.close();
            }
            catch(IOException ioe)
            {

            }
        }
    }
}
```

```
/*
 * InterpolatorUI Class
 *
 *
 ************************************************************************
 *
 * Course:        ECSE 6990
 * Instructor:    Dr. William Randolph Franklin
 * Date:          December 24, 2002
 * Student:       John Childs
 *
 ************************************************************************
 *
 * This class provides all UI services and represents the view layer
 * for the Linear, Laplacian and Thin Plate Equation Solver System.
 *
 ************************************************************************/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.StringTokenizer;
import graphics.*;
import java.awt.image.*;


public class InterpolatorUI extends JFrame
{
//---------------------------------------------------------------------------
//    class variables

   private JTextArea t1;
   private JTextField text1;
   private Box b=Box.createHorizontalBox();
   private String inputFileBuffer;
   private String outputFileBuffer;
   private JMenuItem openItem;
   private JMenuItem openOutputItem;
   private JMenuItem linearItem;
   private JMenuItem laplacianItem;
   private JMenuItem thinPlateItem;
   private JMenuItem setIterationItem;
   private JMenuItem setGraphicsItem;
   private JMenuItem openGraphicsItem;
   private InterpolatorIO interpolatorIO=new InterpolatorIO();
   private int iterations;
   private Painter painter;
   private boolean linearLock=true;
   private int height, width;
   private Image img;
   private double startTime, stopTime;

//---------------------------------------------------------------------------

   ////////////////////////////////////////////////////////////////////////
   // No argument constructor for InterpolatorUI
   ////////////////////////////////////////////////////////////////////////

   private InterpolatorUI()
   {
      super("Linear, Laplacian and Thin Plate Solver");
      setMenuBar();
      Container c=getContentPane();
      setTextArea(b);
      c.add(b);
      setSize(450, 450);
      show();
   }
```

```
//////////////////////////////////////////////////////////////////////
// This method sets up the menu bar and determines actionPerformed()
// Certain menu selections are initially disabled.
// Some reporting to the TextArea occurs as well.
//////////////////////////////////////////////////////////////////////

private void setMenuBar()
{
   JMenuBar bar = new JMenuBar();
   this.setJMenuBar(bar);

   JMenu fileMenu=new JMenu("File");
   fileMenu.setMnemonic('F');

   ////////////////////////////////////////////////////////////////////
   // Code for getting the input file name and opening input text file
   // by selecting File|Input Text File.
   ////////////////////////////////////////////////////////////////////

   openItem=new JMenuItem("Open Input Text File");
   openItem.setMnemonic('I');
   openItem.addActionListener(
      new ActionListener()//inline class instantiation and definition
      {
         public void actionPerformed(ActionEvent e)
         {
            try{inputFileBuffer=JOptionPane.showInputDialog(
               InterpolatorUI.this,
               "Enter Input Text File Name");
               interpolatorIO.openInputTextFile(inputFileBuffer);
               t1.append("\nInput file name:  " + inputFileBuffer);
               interpolatorIO.getTextInputData();

               try{interpolatorIO.closeInputTextFile();
               }
               catch(IOException ioe){JOptionPane.showMessageDialog(
                  null,
                  "Could not close input file.");
               }


               openItem.setEnabled(false);
               openOutputItem.setEnabled(true);
               openGraphicsItem.setEnabled(false);
            }
            catch(IOException ioe){
               JOptionPane.showMessageDialog(
               null,
               "Could not open input file or file does not exist.");
            }
         }
      }
   );
   fileMenu.add(openItem);


   ////////////////////////////////////////////////////////////////////
   // Code for getting the input file name and opening an input graphics
   // file
   // by selecting File|Input Graphics File.
   ////////////////////////////////////////////////////////////////////

   openGraphicsItem=new JMenuItem("Open Input Graphics File");
   openGraphicsItem.setMnemonic('I');
   openGraphicsItem.addActionListener(
      new ActionListener()//inline class instantiation and definition
      {
         public void actionPerformed(ActionEvent e)
         {
            try{inputFileBuffer=JOptionPane.showInputDialog(
               InterpolatorUI.this,
```

```
            "Enter Input Graphics File Name");
            openInputGraphicsFile(inputFileBuffer);
            t1.append("\nInput file name:  " + inputFileBuffer);
            openGraphicsItem.setEnabled(false);
            openItem.setEnabled(false);
            openOutputItem.setEnabled(true);
            linearLock=false;
         }
         catch(IOException ioe){
            JOptionPane.showMessageDialog(
            null,
            "Could not open input file or file does not exist.");
         }
      }
   }
);
fileMenu.add(openGraphicsItem);

//////////////////////////////////////////////////////////////////////
// Code for getting the output file name by selecting File|Input.
//////////////////////////////////////////////////////////////////////

openOutputItem=new JMenuItem("Open Output");
      openOutputItem.setMnemonic('I');
      openOutputItem.addActionListener(
         new ActionListener()
         {
            public void actionPerformed(ActionEvent e)
            {
               try{outputFileBuffer=JOptionPane.showInputDialog(
                  InterpolatorUI.this,
                  "Enter Output File Name");
                  interpolatorIO.openOutputTextFile(outputFileBuffer);
                  t1.append("\nOutput file name:  " + outputFileBuffer);
                  openOutputItem.setEnabled(false);
                  setIterationItem.setEnabled(true);
                  if(linearLock)linearItem.setEnabled(true);
               }
               catch(IOException ioe){
                  JOptionPane.showMessageDialog(
                  null,
                  "Could not open output file or file does not exist.");
               }
            }
         }
      );
      fileMenu.add(openOutputItem);
      openOutputItem.setEnabled(false);

//////////////////////////////////////////////////////////////////////
// Code for exiting.
//////////////////////////////////////////////////////////////////////

JMenuItem exitItem=new JMenuItem("Exit");
exitItem.setMnemonic('x');
exitItem.addActionListener
(  new ActionListener()
   {
      public void actionPerformed(ActionEvent e)
      {
         System.exit(0);
      }
   }
);
fileMenu.add(exitItem);
bar.add(fileMenu);


//////////////////////////////////////////////////////////////////////
// Code for getting the number of iterations.
//////////////////////////////////////////////////////////////////////
```

```
JMenu iterationMenu=new JMenu("Iterations");
setIterationItem=new JMenuItem("Set Iterations");
setIterationItem.setMnemonic('S');
setIterationItem.addActionListener(
   new ActionListener()
   {
      public void actionPerformed(ActionEvent e)
      {
         try{iterations=Integer.parseInt(JOptionPane.showInputDialog(
            InterpolatorUI.this,
            "Enter the Number of Iterations"));
            openOutputItem.setEnabled(false);
            linearItem.setEnabled(true);
            setIterationItem.setEnabled(false);
            linearItem.setEnabled(false);
            laplacianItem.setEnabled(true);
            thinPlateItem.setEnabled(true);
         }
         catch(NumberFormatException nfe){
            JOptionPane.showMessageDialog(
            null,
            "You must enter an integer.");
         }
         t1.append("\nRunning:  "+iterations+" iterations.\n");
      }
   }
);
iterationMenu.add(setIterationItem);
setIterationItem.setEnabled(false);
bar.add(iterationMenu);


////////////////////////////////////////////////////////////////////
// Code for running the linear equation solver.
//
// Note: some fancy footwork here with re-opening the input file.
// The problem is that the linear solver needs some special error
// checking on its input matrix that the Laplacian and Thin Plate
// solvers do not need.  So their file reading code is called from
// the 'openInputTextFile()' method in the text file open
// action event handler.  After opening and reading the file, it
// is immediately closed.
//
// Since this file has not been error checked, it will not do for
// the linear solver.  So the file is opened again so the file
// pointer can be re-zeroed for the special code in the
// interpolatorIO.linearSolver() method.
////////////////////////////////////////////////////////////////////

JMenu runMenu=new JMenu("Run");
runMenu.setMnemonic('R');
linearItem=new JMenuItem("Linear Equation Solver");
linearItem.setMnemonic('r');
linearItem.addActionListener(
   new ActionListener()
   {
      public void actionPerformed(ActionEvent e)
      {
         interpolatorIO.setRows(0);
         interpolatorIO.setCols(0);
         try{interpolatorIO.openInputTextFile(inputFileBuffer);

      }
      catch(IOException ioe){JOptionPane.showMessageDialog(
         null,
         "Could not open input file or file does not exist.");
      }
         t1.append("\n\nrunning solver...\n\n");
         startTime=java.lang.System.currentTimeMillis();
         t1.append(interpolatorIO.linearSolver());
```

```
         stopTime=java.lang.System.currentTimeMillis();
         t1.append("\nElapsed time = "+ (stopTime-startTime)/1000.0 + " seconds.");
         t1.append("\n\nEnd of successful run.");
         try{interpolatorIO.closeInputTextFile();
         }
         catch(IOException ioe){JOptionPane.showMessageDialog(
            null,
            "Could not close input file.");
         }

         try{interpolatorIO.closeOutputTextFile();
         }
         catch(IOException ioe){JOptionPane.showMessageDialog(
            null,
            "Could not close output file.");
          }

         linearItem.setEnabled(false);
         setGraphicsItem.setEnabled(false);
         setIterationItem.setEnabled(false);
         thinPlateItem.setEnabled(false);
         laplacianItem.setEnabled(false);
      }
   }
);
runMenu.add(linearItem);
linearItem.setEnabled(false);
bar.add(runMenu);

//////////////////////////////////////////////////////////////////////
// Code for running the Laplacian equation solver.
//////////////////////////////////////////////////////////////////////

laplacianItem=new JMenuItem("Laplacian Equation Solver");
laplacianItem.setMnemonic('r');
laplacianItem.addActionListener(
   new ActionListener()
   {
      public void actionPerformed(ActionEvent e)
      {
         t1.append("\nrunning Laplacian solver...\n");
         startTime=java.lang.System.currentTimeMillis();
         t1.append(interpolatorIO.laplacianSolver(iterations));
         stopTime=java.lang.System.currentTimeMillis();
         t1.append(  "\nInput matrix size:  "+
                  interpolatorIO.getRows()+
                  " rows by "+
                  interpolatorIO.getCols()+" columns.");
         t1.append("\nEnd of successful run.");
         t1.append("\nElapsed time = "+ (stopTime-startTime)/1000.0 + " seconds.");
         try{interpolatorIO.closeOutputTextFile();
         }
         catch(IOException ioe){JOptionPane.showMessageDialog(
            null,
            "Could not close output file.");
         }
         laplacianItem.setEnabled(false);
         thinPlateItem.setEnabled(false);
         setGraphicsItem.setEnabled(true);
      }
   }
);
runMenu.add(laplacianItem);
laplacianItem.setEnabled(false);


//////////////////////////////////////////////////////////////////////
// Code for running the Thin Plate equation solver.
//////////////////////////////////////////////////////////////////////

thinPlateItem=new JMenuItem("Thin Plate Equation Solver");
```

```
thinPlateItem.setMnemonic('r');
thinPlateItem.addActionListener(
   new ActionListener()
   {
      public void actionPerformed(ActionEvent e)
      {
         t1.append("\nrunning Thin Plate solver...\n");
         startTime=java.lang.System.currentTimeMillis();
         t1.append(interpolatorIO.thinPlateSolver(iterations));
         stopTime=java.lang.System.currentTimeMillis();
         t1.append(  "\nInput matrix size:  "+
                  interpolatorIO.getRows()+
                  " rows by "+interpolatorIO.getCols()+
                  " columns.");
                  t1.append("\nEnd of successful run.");
         t1.append("\nElapsed time = "+ (stopTime-startTime)/1000.0 + " seconds.");
         try{interpolatorIO.closeOutputTextFile();
         }
         catch(IOException ioe){JOptionPane.showMessageDialog(
            null,
            "Could not close output file.");
         }
         thinPlateItem.setEnabled(false);
         laplacianItem.setEnabled(false);
         setGraphicsItem.setEnabled(true);
      }
   }
);
runMenu.add(thinPlateItem);
thinPlateItem.setEnabled(false);

//////////////////////////////////////////////////////////////////////
// Code for the 'About' menu.
//////////////////////////////////////////////////////////////////////

JMenu aboutMenu=new JMenu("About");
aboutMenu.setMnemonic('A');
JMenuItem aboutItem=new JMenuItem("About...");
aboutItem.setMnemonic('a');
aboutItem.addActionListener(
   new ActionListener()
   {
      public void actionPerformed(ActionEvent e)
      {
         JOptionPane.showMessageDialog(
            InterpolatorUI.this,
            aboutString(),
            "About",
            JOptionPane.PLAIN_MESSAGE);
      }
   }
);
aboutMenu.add(aboutItem);
bar.add(aboutMenu);

//////////////////////////////////////////////////////////////////////
// Code for pop up graphics window.
//////////////////////////////////////////////////////////////////////

JMenu graphicsMenu=new JMenu("Graphics");
setGraphicsItem=new JMenuItem("Generate Graphics");
setGraphicsItem.setMnemonic('G');
setGraphicsItem.addActionListener(
   new ActionListener()
   {
      public void actionPerformed(ActionEvent e)
      {
         painter=new Painter(interpolatorIO.getRows(),
                        interpolatorIO.getCols(),
                        interpolatorIO.getN());
         painter.addWindowListener(
```

```
                new WindowAdapter(){
                   public void windowClosing(WindowEvent e){
                      //default close focus window.
                   }
                }
             );
          }
       }
   );
   graphicsMenu.add(setGraphicsItem);
   setGraphicsItem.setEnabled(false);
   bar.add(graphicsMenu);
}

////////////////////////////////////////////////////////////////////////
// This method sets up the text area.
////////////////////////////////////////////////////////////////////////

private void setTextArea(Box b)
{

         String s=   "Welcome to the Linear Equation Solver/Laplacian"+
                  "and Thin PlateSolver.\n"+
                  "Select File | Open Input to get started.\n";
         t1=new JTextArea(s, 100,150);
         t1.setEditable(false);
         b.add(new JScrollPane(t1));
}

////////////////////////////////////////////////////////////////////////
// This method returns the About Message string.
////////////////////////////////////////////////////////////////////////

private String aboutString()
{
   String messageString=
      "Gauss-Jordan Linear Equation Solver and " +
      "Laplacian/Thin Plate Equation Solver v1.01 \n"+
      "Copyright 2002 John Childs\n\n" +
      "This program solves simultaneous linear equations\n" +
      "using the Gauss-Jordan elimination method\n" +
      "and interpolates free data to fixed data using\n" +
      "the Laplacian Heat Equation and the\n"+
      "Thin Plate Equation.\n\n"+

      "If the input file is a text file it must be a space-separated\n" +
      "ASCII file of decimal numbers.  The program\n"+
      "is looking for a (rows x (cols=rows+1)) input data matrix\n" +
      "for the Linear solver and a rows x columns input data matrix\n" +
      "for the Laplacian and Thin Plater solvers." +
      "if the matrix is oversized, it will truncate.  If undersized\n" +
      "an exception will be thrown.  The program reads the rows and\n"+
      "columns automatically from the input file.\n\n" +

      "Alternatively, the input file can be a .gif or .jpg file\n" +
      "with the elevation data encoded in the RGB values of the\n"+
      "contour line pixels.\n\n" +

      "For example, a contour line elevation of 255 would be encoded as\n" +
      "0x0000ff in an RGB color model, i.e. a blue line on a black\n"+
      "background.\n\n" +

      "To use the program, first enter the input file name.\n" +
      "Then you will be able to enter the output file name.\n\n" +

      "If you are running the Laplace solver or the Thin Plate\n"+
      "you must enter the \n number of iterations you wish to run.\n\n"+

      "After this is done, click on \"run\" \n\n" +

      "For Linear Equations, the solution vector will be output as a \n"+
```

```
      "row vector into the output file you have specified.\n\n"+

      "For the Laplace and Thin Plate solvers, the entire solution  \n"+
      "matrix will be written into the output file you \n"+
      "have specified.\n\n"+

      "You have the option of viewing the results of the Laplacian\n"+
      "or Thin Plate results graphically.\n\n";

   return(messageString);
}

//////////////////////////////////////////////////////////////////////////
// This method gets data from the input graphics file and displays
// the input image.
//
// It is arguable whether this is a display or control function.  There
// are probably elements of both.
//
// However, some of the functions will not work without an active
// window so this is an easier place to implement it than the
// interpolatorIO class.
//////////////////////////////////////////////////////////////////////////

public void openInputGraphicsFile(String fileName) throws IOException
{

   try{img = Toolkit.getDefaultToolkit().getImage(fileName);
   }
   catch(Exception e){JOptionPane.showMessageDialog(
               null,
               " error opening input file.");
   }

   try {
      MediaTracker tracker = new MediaTracker(this);
      tracker.addImage(img, 0);
         tracker.waitForID(0);
   }
   catch (Exception e) {JOptionPane.showMessageDialog(
               null,
               "MediaTracker problem.");
   }
   width = img.getWidth(this);
   height = img.getHeight(this);
   if(width==-1 || height==-1){
      JOptionPane.showMessageDialog(
                        null,
               "Graphics file not found in specified directory.");
               System.exit(0);
   }

   int[] pixels = new int[width * height];
   PixelGrabber pg=new PixelGrabber(img,0,0,width,height,pixels,0,width);
   try{
   pg.grabPixels();
   }
   catch (InterruptedException e) {JOptionPane.showMessageDialog(
               null,
               "Error grabbing pixels..");
   }
   interpolatorIO.initializeArray(width, height, pixels);

   MemoryImageSource mis = new MemoryImageSource(width, height, pixels, 0, width);
   Image testImage = this.createImage(mis);
   t1.append(  "\nwidth:  "+width);
   t1.append(  "\nheight:  "+height);
   InputPainter inputPainter=new InputPainter(testImage, width, height);
}
```

```
   ////////////////////////////////////////////////////////////////////////
   // Root method
   ////////////////////////////////////////////////////////////////////////

   public static void main(String args[])
   {
      InterpolatorUI app=new InterpolatorUI();

      app.addWindowListener(
         new WindowAdapter(){
            public void windowClosing(WindowEvent e){
               System.exit(0);
            }
         }
      );
   }
}
```

```
/*
 * InterpolatorIO Class
 *
 *
 ************************************************************************
 *
 * Course:        ECSE 6990
 * Instructor:    Dr. William Randolph Franklin
 * Date:          December 24, 2002
 * Student:       John Childs
 *
 ************************************************************************
 *
 * This class provides IO services for the Interpolator classes.
 *
 ************************************************************************/
import java.util.StringTokenizer;
import java.text.DecimalFormat;
import java.io.*;
import javax.swing.*;
import java.awt.*;
import solvers.*;

////////////////////////////////////////////////////////////////////////
// This class encapsulates i/o utilities for a UI
////////////////////////////////////////////////////////////////////////

public class InterpolatorIO
{
   private FileWriter fw;
   private BufferedWriter bw;
   private PrintWriter outFile;
   private LinearEquationSolver linearSolver;
   private ThinPlateSolver thinPlateSolver;
   private LaplacianSolver laplacianSolver;
   private LaplacianSolver tempSolver;
   private ThinPlateSolver otherTempSolver;
   private LinearEquationSolver linearTempSolver;
   private FileReader fr;
   private BufferedReader inFile;
   private StringTokenizer tokenizer;
   private int rows, cols;
   private double rowArray[];
   private  double M[][];
   private  double N[][];
   private boolean lineCheck=false;
   private int I[][];
   private int lineCounter;


   ////////////////////////////////////////////////////////////////////////
   // Constructor.
   ////////////////////////////////////////////////////////////////////////

    public InterpolatorIO()
    {

    }

   ////////////////////////////////////////////////////////////////////////
   // This method creates a DecimalFormat object for formatting the file
   // output string.  It returns a string formatted to the specified decimal
   // precision.
   ////////////////////////////////////////////////////////////////////////
   public String customFormat(String pattern, double value )
   {
         DecimalFormat myFormatter = new DecimalFormat(pattern);
         String output = myFormatter.format(value);
      return(output);
      }
```

```
////////////////////////////////////////////////////////////////////////
// This method opens a text file for output.
////////////////////////////////////////////////////////////////////////
public void openOutputTextFile(String fileName) throws IOException
{
   fw=new FileWriter(fileName);
   bw=new BufferedWriter(fw);
   outFile=new PrintWriter(bw);
   }


////////////////////////////////////////////////////////////////////////
// This method writes the specified string to the specified output file.
////////////////////////////////////////////////////////////////////////
private void writeString(String outputString)
{
   outFile.print(outputString);
   }


////////////////////////////////////////////////////////////////////////
// This method closes the specified output file.
////////////////////////////////////////////////////////////////////////
public void closeOutputTextFile() throws IOException
{
   outFile.close();
   }

////////////////////////////////////////////////////////////////////////
// This method closes the specified input file.
////////////////////////////////////////////////////////////////////////
public void closeInputTextFile() throws IOException
{

   try{inFile.close();
   }
   catch(IOException ioe){ JOptionPane.showMessageDialog(
                     null,
                     "Could not close input file.");
   }
   }

////////////////////////////////////////////////////////////////////////
// This method opens a text file for input.  I have to open stream,
// read all of the lines to get the row count, close the stream,
// and then re open it.  This was the only way I cound to re-zero
// the file position pointer.
////////////////////////////////////////////////////////////////////////
public void openInputTextFile(String fileName) throws IOException
{
   fr=new FileReader(fileName);
   inFile=new BufferedReader(fr);//////////////////////////////////////////////////

   while(inFile.readLine() !=null){
            lineCounter=lineCounter+1;
   }
   rows=lineCounter;
   lineCounter=0;
   inFile.close();

   fr=new FileReader(fileName);
   inFile=new BufferedReader(fr);//////////////////////////////////////////////////
   }


////////////////////////////////////////////////////////////////////////
// This method reads a line from the specified input file.
////////////////////////////////////////////////////////////////////////
public String readOneLine(BufferedReader readerObject)
{
   String line=null;
```

```
   try{line=readerObject.readLine();
   }
   catch(IOException ioe){
      JOptionPane.showMessageDialog(null,
      "Error reading line of input file.");
   }
   return line;
 }

////////////////////////////////////////////////////////////////////////
// This method parses one input line turning a line of text into an array
// of doubles.
////////////////////////////////////////////////////////////////////////
private double[] parseLine(String line)
{
   tokenizer=new StringTokenizer(line);
   double[] lineArray = new double[line.length()];
   int i=0;
   if(tokenizer.countTokens() != cols){
      JOptionPane.showMessageDialog(null,
         "\n\n***Warning: number of elements does not match"+
           "column dimension." +
         "\n***Matrix is probably corrupt.***" +
         "\n***Please check your input data file.***\n" );
   }
   while(tokenizer.hasMoreTokens()){
      lineArray[i]=Double.parseDouble(tokenizer.nextToken());
      i=i+1;
   }
   return(lineArray);
}

////////////////////////////////////////////////////////////////////////
// This method parses first input line.  The first line is handled
// as a special case so that the row and column information can be
// extracted and calculated once.
////////////////////////////////////////////////////////////////////////
private double[] parseFirstLine(String line) throws IOException
{
   tokenizer=new StringTokenizer(line);
   double[] lineArray = new double[line.length()];
   int i=0;
   cols=tokenizer.countTokens();


   ////////////////////////////////////////////////////////////////////////////
   //    System.out.println("parseFirstLine rows is:  " + rows + "\n");
   //    System.out.println("parseFirstLine cols is:  " + cols + "\n");
   ////////////////////////////////////////////////////////////////////////////

   while(tokenizer.hasMoreTokens()){
      lineArray[i]=Double.parseDouble(tokenizer.nextToken());
      i=i+1;
   }
   return(lineArray);
}

////////////////////////////////////////////////////////////////////////
// This method handles the first line of the input file.
////////////////////////////////////////////////////////////////////////
public void readFirstLine() throws IOException
{
   rowArray=parseFirstLine(readOneLine(inFile));
   M=new double[rows][cols];
   M[0]=rowArray;

   for(int j=0; j<cols-1; j++){
      if(lineCheck==false){
         if(rowArray[j]!=0.0){
            lineCheck=true;
         }
```

```
      }
   }

   if(lineCheck==false){
      if(rowArray[cols-1]==0.0){
         lineCheck=true;
      }
   }

}

////////////////////////////////////////////////////////////////////////
// This method handles the rest of the lines of the input file.
////////////////////////////////////////////////////////////////////////
public void readOtherLines() throws IOException
{

   for(int i=1; i<rows; i++){
      rowArray=parseLine(readOneLine(inFile));
      for(int j=0; j<cols; j++){
         M[i][j]=rowArray[j];
      }
   }

   for(int j=0; j<cols-1; j++){
      if(lineCheck==false){
         if(rowArray[j]!=0.0){
            lineCheck=true;
         }
      }
   }

   if(lineCheck==false){
      if(rowArray[cols-1]==0.0){
         lineCheck=true;
      }
   }
}

////////////////////////////////////////////////////////////////////////
// This method solves the matrix of linear equations.
////////////////////////////////////////////////////////////////////////
public String linearSolver()
{
   try{readFirstLine();
   }
   catch(IOException ioe){
      JOptionPane.showMessageDialog(
      null,
      "Error reading the first line of the file.");
   }

   if(lineCheck==false){
      return(noSolutionMessage());
   }
   else{try{readOtherLines();
      }
      catch(IOException ioe){
         JOptionPane.showMessageDialog(
         null,
         "Error reading a line of the file.");
      }
      if(lineCheck==false){
         return(noSolutionMessage());
      }
      else{return(solveLinearEquations());
      }
   }

}
```

```
////////////////////////////////////////////////////////////////////////
// This method calls the solver and calculates the solution vector.
////////////////////////////////////////////////////////////////////////
private String solveLinearEquations()
{
   String returnString=null;
   linearSolver = new LinearEquationSolver(rows, cols, M);
   linearTempSolver=linearSolver.gaussJord();
   N=linearTempSolver.M;
   returnString=linearTempSolver.toString();
     for(int i=0; i<rows; i++){
      for(int j=0; j<cols; j++){
      writeString(customFormat("000.00  ",
         N[i][j]));
      }
   }
   writeString("\n");
   return(returnString);
}

////////////////////////////////////////////////////////////////////////
// This method solves the matrix of elevations using the Laplacian
// equation.
//
// For the time being, I have deleted the lineCheck test but allowed
// readFirstLine() and readOtherLines() to continue making the comarisons.
//
// For large files, it may be advisable to write alternative versions
// of these methods to be called by laplacianSolver() to eliminate
// this checking, which is necessary for the linear equation solver
// but unnecessary for the laplacian solver.
////////////////////////////////////////////////////////////////////////
public String laplacianSolver(int iterations)
{
   return(solveLaplacianEquations(iterations));
}

////////////////////////////////////////////////////////////////////////
// This method calls the solver and relaxes the matrix using the
// Laplacian equation, then writes the result to the output file.
////////////////////////////////////////////////////////////////////////
private String solveLaplacianEquations(int iterations)
{
   String returnString=null;
   laplacianSolver = new LaplacianSolver(rows, cols, M, iterations);
   tempSolver=laplacianSolver.laplacian();
   N=tempSolver.M;
   returnString=(tempSolver.toString());
     for(int i=0; i<rows; i++){
      for(int j=0; j<cols; j++){
         writeString(customFormat("000.00  ",
            N[i][j] ));
         }
      writeString("\n");
   }
   return(returnString);
}

////////////////////////////////////////////////////////////////////////
// This method solves the matrix of elevations using the Thin Plate
// equation.
//
// For the time being, I have deleted the lineCheck test but allowed
// readFirstLine() and readOtherLines() to continue making the comarisons.
//
// For large files, it may be advisable to write alternative versions
// of these methods to be called by thinPlateSolver() to eliminate
// this checking, which is necessary for the linear equation solver
// but unnecessary for the laplacian solver.
////////////////////////////////////////////////////////////////////////
public String thinPlateSolver(int iterations)
```

```
{
   return(solveThinPlateEquations(iterations));
}

////////////////////////////////////////////////////////////////////////
// This method calls the solver and relaxes the matrix using the
// Thin Plate equation, then writes the result to the output file.
////////////////////////////////////////////////////////////////////////
private String solveThinPlateEquations(int iterations)
{
   String returnString=null;
   thinPlateSolver = new ThinPlateSolver(rows, cols, M, iterations);
   otherTempSolver=thinPlateSolver.thinPlate();
   N=otherTempSolver.M;
   returnString=(otherTempSolver.toString());
    for(int i=0; i<rows; i++){
      for(int j=0; j<cols; j++){
            writeString(customFormat("000.00  ",
            N[i][j] ));
      }
      writeString("\n");
   }
   return(returnString);
}

////////////////////////////////////////////////////////////////////////
// This method returns a "no solution" error message.
////////////////////////////////////////////////////////////////////////
private String noSolutionMessage()
{
   return("\n\n***Warning: no solution possible" +
   "\n***One row of the input matrix is of the form: ***" +
     "\n***CONSTANT=0***" +
     "\n***Please check your input data file.***\n" );
}

////////////////////////////////////////////////////////////////////////
// This method returns the matrix N
////////////////////////////////////////////////////////////////////////
public double[][] getN()
{
   return(N);
}

////////////////////////////////////////////////////////////////////////
// This method returns the rows
////////////////////////////////////////////////////////////////////////
public int getRows()
{
   return(rows);
}

////////////////////////////////////////////////////////////////////////
// This method sets the rows
////////////////////////////////////////////////////////////////////////

public void setRows(int rowValue)
{
   rows=rowValue;
}

////////////////////////////////////////////////////////////////////////
// This method returns the columns
////////////////////////////////////////////////////////////////////////
public int getCols()
{
   return(cols);
}
```

```
   ////////////////////////////////////////////////////////////////////////
   // This method sets the cols
   ////////////////////////////////////////////////////////////////////////
   public void setCols(int colValue)
   {
      cols=colValue;
   }

   ////////////////////////////////////////////////////////////////////////
   // This method gets the input data for text files.
   ////////////////////////////////////////////////////////////////////////
   public void getTextInputData()
   {
      try{readFirstLine();
      }
      catch(IOException ioe){
         JOptionPane.showMessageDialog(
         null,
         "Error reading the first line of the file.");
      }

      try{readOtherLines();
      }
      catch(IOException ioe){
         JOptionPane.showMessageDialog(
         null,
         "Error reading a line of the file.");
      }
   }
   ////////////////////////////////////////////////////////////////////////
   // This method transfers the graphics information stored in the 1D
   // pixels[] file into the 2D integer I[][] file.  The data is then
   // transferred to the double M[][] matrix.
   ////////////////////////////////////////////////////////////////////////
   public void initializeArray(int rows, int cols, int pixels[])
   {
      this.rows=rows;
      this.cols=cols;

      I=new int[rows][cols];
      M=new double[rows][cols];
      int index=0;
      for(int i=0; i<cols; i++){
         for(int j=0; j<rows; j++){
            I[j][i]=pixels[index]& 0x00ffffff;
            M[j][i]=(double)I[j][i];
            index=index+1;
         }
      }
   }
}
```

```
/*
 * LaplacianSolver Class
 *
 *
 ************************************************************************
 *
 * Course:        ECSE 6990
 * Instructor:    Dr. William Randolph Franklin
 * Date:          December 21, 2002
 * Student:       John Childs
 *
 ************************************************************************
 *
 * This class encapsulates Laplacian solver utilities.
 *
 *
 ************************************************************************/
package solvers;

import java.text.DecimalFormat;
//////////////////////////////////////////////////////////////////////////
// This class encapsulates an augmented matrix of linear equation
//  coefficients.
//////////////////////////////////////////////////////////////////////////

public class LaplacianSolver
{
    private int rows, cols, iterations;
    public double M[][];
    public double N[][];


   ////////////////////////////////////////////////////////////////////////
   // Constructor for args rows, columns and a 2D matrix of type double.
   ////////////////////////////////////////////////////////////////////////

    public LaplacianSolver(int tRows, int tCols, double T[][], int iterations)
    {
        M = new double[tRows][tCols];
        N = new double[tRows][tCols];
        rows = tRows;
        cols = tCols;
        this.iterations=iterations;

        for(int i=0; i<rows; i++)
            for(int j=0; j<cols; j++){
                M[i][j] = T[i][j];
                N[i][j] = T[i][j];
         }
    }

   ////////////////////////////////////////////////////////////////////////
   // Perform multiple iteration Laplacian calculation on Matrix M of
   // size rows, cols.
   //
   // Procedure:
   //
   // 1) The calculation is run on a subregion of the input matrix.  The
   //    subregion is offset inwards from the edge of the map by one
   //    index unit.  This allows the Laplacian equation to be run
   //    without any special cases for the edges.
   //
   // 2) The edges are assigned elevations by mapping the offset rows
   //    or columns to the edges before each iteration.
   ////////////////////////////////////////////////////////////////////////
    public LaplacianSolver laplacian()
    {
  //      LaplacianSolver solver = new LaplacianSolver(rows,cols,M, iterations);


      for(int k=0; k<iterations; k++){
```

```
      ////////////////////////////////////////////////////////////////
      // This code transfers elevations across the offset boundaries
      // to the edge of the map.
      ////////////////////////////////////////////////////////////////
      for(int i=1; i<rows-1; i++){
         M[i][0]=M[i][1];
         M[i][cols-1]=M[i][cols-2];
      }
      for(int j=1; j<cols-1; j++){
         M[0][j]=M[1][j];
         M[rows-1][j]=M[rows-2][j];
      }
      M[0][0]=M[1][1];
      M[rows-1][cols-1]=M[rows-2][cols-2];
      M[0][cols-1]=M[1][cols-2];
      M[rows-1][0]=M[rows-2][1];

      ////////////////////////////////////////////////////////////////
      // This code uses the Laplacian equation to calculate the new
      // elevations for the offset region.
      ////////////////////////////////////////////////////////////////
      for(int i=1; i<rows-1; i++){
         for(int j=1; j<cols-1; j++){
            if(N[i][j]==0.0){
               M[i][j]= (M[i-1][j] + M[i+1][j] +
                           M[i][j-1] + M[i][j+1])/4;
            }
         }
      }

      ////////////////////////////////////////////////////////////////
      // This code transfers elevations across the offset boundaries
      // to the edge of the map.
      ////////////////////////////////////////////////////////////////
      for(int i=1; i<rows-1; i++){
         M[i][0]=M[i][1];
         M[i][cols-1]=M[i][cols-2];
      }
      for(int j=1; j<cols-1; j++){
         M[0][j]=M[1][j];
         M[rows-1][j]=M[rows-2][j];
      }
      M[0][0]=M[1][1];
      M[rows-1][cols-1]=M[rows-2][cols-2];
      M[0][cols-1]=M[1][cols-2];
      M[rows-1][0]=M[rows-2][1];

      ////////////////////////////////////////////////////////////////
      // This code computes the edge nodes as the average of its
      // three neighbors.
      ////////////////////////////////////////////////////////////////
      for(int i=1; i<rows-1; i++){
         M[i][0]=(M[i][1] + M[i-1][0] + M[i+1][0])/3;
         M[i][cols-1]=(M[i][cols-2] + M[i-1][cols-1] + M[i+1][cols-1])/3;
      }
      for(int j=1; j<cols-1; j++){
         M[0][j]=(M[1][j] + M[0][j-1] + M[0][j+1])/3;
         M[rows-1][j]=(M[rows-2][j] + M[rows-1][j-1] + M[rows-1][j+1])/3;
      }
      M[0][0]=(M[0][1] + M[1][0])/2;
      M[rows-1][cols-1]=(M[rows-1][cols-2] + M[rows-2][cols-1])/2;
      M[0][cols-1]=(M[0][cols-2] + M[1][cols-1])/2;
      M[rows-1][0]=(M[rows-2][0] + M[rows-1][1])/2;
   }
     return this;
 }

////////////////////////////////////////////////////////////////////////
// Output the entire matrix as a String object.
////////////////////////////////////////////////////////////////////////
```

```
    public String toString()
    {
        String string = "";
      if(rows<40){
         for(int i=0;i<rows;i++){
                  for(int j=0; j<cols; j++){
                  string += customFormat("000.00", M[i][j]) + "     ";
               }
               string += "\n";
            }
            return string;
      }
      else{
         return("Output matrix is too large to display in text box.\n"+
               "See output file for data.\n\n");
      }
    }
   ////////////////////////////////////////////////////////////////////////
   // This method creates a DecimalFormat object for formatting the file
   // output string.  It returns a string formatted to the specified decimal
   // precision.
   ////////////////////////////////////////////////////////////////////////
   public String customFormat(String pattern, double value )
   {
         DecimalFormat myFormatter = new DecimalFormat(pattern);
         String output = myFormatter.format(value);
      return(output);
      }
}
```

```
/*
```

```
 * ThinPlateSolver Class
 *
 *
 ************************************************************************
 *
 * Course:        ECSE 6990
 * Instructor:    Dr. William Randolph Franklin
 * Date:          December 21, 2002
 * Student:       John Childs
 *
 ************************************************************************
 *
 * This class encapsulates a matrix of elevations and the utilities
 * needed to relax elevations using the Thin Plate equation.
 *
 * Note: because the Thin Plate equation requires data from positions
 * offset by two index positions, the edges and the nodes offset
 * one position from the edge constitute special cases that must
 * be handled separately.
 *
 * The nodes offset two index positions are calculated using the
 * Laplace equation while the edge nodes are merely copied from
 * the doubly offset nodes.
 *
 *
 ************************************************************************/
package solvers;

import java.text.DecimalFormat;
//////////////////////////////////////////////////////////////////////////
// This class encapsulates an augmented matrix of linear equation
//  coefficients.
//////////////////////////////////////////////////////////////////////////

public class ThinPlateSolver
{
    private int rows, cols, iterations;
    public double M[][];
    public double N[][];


   //////////////////////////////////////////////////////////////////////////
   // Constructor for args rows, columns and a 2D matrix of type double.
   //////////////////////////////////////////////////////////////////////////

    public ThinPlateSolver(int tRows, int tCols, double T[][], int iterations)
    {
        M = new double[tRows][tCols];
        N = new double[tRows][tCols];
        rows = tRows;
        cols = tCols;
        this.iterations=iterations;

        for(int i=0; i<rows; i++)
            for(int j=0; j<cols; j++){
                M[i][j] = T[i][j];
                N[i][j] = T[i][j];
         }
    }

   //////////////////////////////////////////////////////////////////////////
   // Perform multiple iteration Thin Plate calculation on Matrix M of
   // size rows, cols.
   //
   // Procedure:
   //
   // 1) The calculation is run on a subregion of the input matrix.  The
   //    subregion is offset inwards from the edge of the map by one
   //    index unit.  This allows the Thin Plate equation to be run
   //    without any special cases for the edges.
   //
```

```
// 2) The edges are assigned elevations by mapping the offset rows
//    or columns to the edges before each iteration.
////////////////////////////////////////////////////////////////////////
 public ThinPlateSolver thinPlate()
 {

   for(int k=0; k<iterations; k++){

      ////////////////////////////////////////////////////////////////
      // This code uses the Laplacian equation to calculate the new
      // elevations for the rows and columns offset one position
      // from the edge.
      ////////////////////////////////////////////////////////////////
      for(int i=1; i<rows-1; i++){
         if(N[i][cols-2]==0.0){
            M[i][cols-2]=(M[i-1][cols-2] + M[i+1][cols-2] +
                        M[i][cols-3] + M[i][cols-1])/4;
         }
      }

      for(int i=1; i<rows-1; i++){
         if(N[i][1]==0.0){
            M[i][1]= (M[i-1][1] + M[i+1][1] +
                        M[i][0] + M[i][2])/4;
         }
      }

      for(int j=1; j<cols-1; j++){
         if(N[1][j]==0.0){
            M[1][j]= (M[1][j-1] + M[1][j+1] +
                        M[0][j] + M[2][j])/4;
         }
      }

      for(int j=1; j<cols-1; j++){
         if(N[rows-2][j]==0.0){
            M[rows-2][j]=  (M[rows-2][j-1] + M[rows-2][j+1] +
                        M[rows-3][j] + M[rows-1][j])/4;
         }
      }

      ////////////////////////////////////////////////////////////////
      // This code calculates the elevations for the edge rows.
      ////////////////////////////////////////////////////////////////
      for(int i=1; i<rows-1; i++){
         M[i][0]=M[i][1];
         M[i][cols-1]=M[i][cols-2];
      }
      for(int j=1; j<cols-1; j++){
         M[0][j]=M[1][j];
         M[rows-1][j]=M[rows-2][j];
      }
      M[0][0]=M[1][1];
      M[rows-1][cols-1]=M[rows-2][cols-2];
      M[0][cols-1]=M[1][cols-2];
      M[rows-1][0]=M[rows-2][1];


      ////////////////////////////////////////////////////////////////
      // This code uses the Thin Plate equation to calculate the new
      // elevations for the offset region using the Thin Plate
      // equation.
      ////////////////////////////////////////////////////////////////
      for(int i=2; i<rows-2; i++){
         for(int j=2; j<cols-2; j++){
            if(N[i][j]==0.0){
               M[i][j]= (8*(M[i-1][j] + M[i+1][j] +
                           M[i][j-1] + M[i][j+1]) -
                           2*(M[i-1][j-1] + M[i-1][j+1] +
                           M[i+1][j-1] + M[i+1][j+1]) -
                           (M[i-2][j] + M[i+2][j] +
```

```
                              M[i][j-2] + M[i][j+2]))/20;
                }
             }
          }
       }

//    System.out.println("rows is:  "+ rows);
//          System.out.println("cols is:  "+ cols);
//          System.out.println("iterations is:  "+ iterations);
//                JOptionPane.showMessageDialog(
//                                      null,
//                        "returning ");
         return this;
     }

     ////////////////////////////////////////////////////////////////////////
    // Output the entire matrix as a String object.
    ////////////////////////////////////////////////////////////////////////

     public String toString()
     {
         String string = "";
       if(rows<40){
          for(int i=0;i<rows;i++){
                   for(int j=0; j<cols; j++){
                   string += customFormat("00.00", M[i][j]) + "     ";////////////bad, bad
line of code.
                }
                string += "\n";
             }
             return string;
       }
       else{
          return("Output matrix is too large to display in text box.\n"+
                "See output file for data.\n\n");
       }
     }

    ////////////////////////////////////////////////////////////////////////
    // This method creates a DecimalFormat object for formatting the file
    // output string.  It returns a string formatted to the specified decimal
    // precision.
    ////////////////////////////////////////////////////////////////////////
    private String customFormat(String pattern, double value )
    {
          DecimalFormat myFormatter = new DecimalFormat(pattern);
          String output = myFormatter.format(value);
       return(output);
       }

}
```

```
/*
 * LinearEquationSolver Class
 *
 *
 ************************************************************************
 *
 * Course:        ECSE 6990
 * Instructor:    Dr. William Randolph Franklin
 * Date:          December 21, 2002
 * Student:       John Childs
 *
 ************************************************************************
 *
 * This class encapsulates an augmented matrix of linear equation
 * coefficients along with the methods required to produce the
 * solution vector.
 *
 ************************************************************************/

package solvers;

//////////////////////////////////////////////////////////////////////////
// This class encapsulates an augmented matrix of linear equation
//  coefficients.
//////////////////////////////////////////////////////////////////////////
import java.text.DecimalFormat;

public class LinearEquationSolver
{
    private int rows, cols;
    public double M[][];

   ////////////////////////////////////////////////////////////////////////
   // Constructor for afgs rows, columns and a 2D matrix of type double.
   ////////////////////////////////////////////////////////////////////////

    public LinearEquationSolver(int tRows, int tCols, double T[][])
    {
        M = new double[tRows][tCols];
        rows = tRows;
        cols = tCols;

        for(int i=0; i<rows; i++)
            for(int j=0; j<cols; j++)
                M[i][j] = T[i][j];
    }

   ////////////////////////////////////////////////////////////////////////
   // Swap row r1 with row r2 for the matrix M.
   // Note: the matrix M in question is the one associated with the
   // LinearEquationSolver object calling the method, not necessarily
   // this.M.
   ////////////////////////////////////////////////////////////////////////

    public LinearEquationSolver swapRow(int r1, int r2)
    {
        double tempRow[] = new double[rows];
        LinearEquationSolver returnLinearEquationSolver =
         new LinearEquationSolver(rows,cols,M);

        if( (r1 >= rows) | (r2 >= rows) | (r1 < 0) | (r2 < 0) )
            throw new ArithmeticException(
            "LinearEquationSolver.swapRow: r1 or r2 not within matrix: "
            + r1 + ","+ r2 );

        tempRow = returnLinearEquationSolver.M[r1];
        returnLinearEquationSolver.M[r1] = returnLinearEquationSolver.M[r2];
        returnLinearEquationSolver.M[r2] = tempRow;

        return returnLinearEquationSolver;
```

```
 }


////////////////////////////////////////////////////////////////////////
// Multiply all the coefficients in row r1 by the scalar 'scalar'.
////////////////////////////////////////////////////////////////////////

 public LinearEquationSolver mulRow(int r1,double scalar)
 {
     LinearEquationSolver returnLinearEquationSolver =
      new LinearEquationSolver(rows,cols,M);

     if( (r1 >= rows) || (r1 < 0) )
         throw new ArithmeticException(
         "LinearEquationSolver.mulRow: r1 not within matrix: " + r1);

     for(int i=0; i<cols; i++)
         returnLinearEquationSolver.M[r1][i] = M[r1][i] * scalar;

     return returnLinearEquationSolver;
 }

////////////////////////////////////////////////////////////////////////
// Multiply each coefficient in row r1 by scalar.
// Then add each coefficient in r1 to each coeffiecient in r2 and
// place the result in r2.
//
// Return a new LinearEquationSolver object with the modified rows.
////////////////////////////////////////////////////////////////////////

 public LinearEquationSolver addMulRow(int r1, int r2, double scalar)
 {
     LinearEquationSolver returnLinearEquationSolver =
      new LinearEquationSolver(rows,cols,M);
     LinearEquationSolver tempLinearEquationSolver =
      new LinearEquationSolver(rows,cols,M);

     if( (r1 >= rows) | (r2 >= rows) | (r1 < 0) | (r2 < 0) )
         throw new ArithmeticException(
         "LinearEquationSolver.addMulRow: r1 or r2 not within matrix: "
         + r1 + ","+r2);

     tempLinearEquationSolver = tempLinearEquationSolver.mulRow(r1,scalar);

     for(int i=0; i<cols; i++)
         returnLinearEquationSolver.M[r2][i] =
            returnLinearEquationSolver.M[r2][i] +
            tempLinearEquationSolver.M[r1][i];

     return returnLinearEquationSolver;
 }

////////////////////////////////////////////////////////////////////////
// Perform a Gauss-Jordan elimination on Matrix M of size rows, cols.
//
// Procedure:
//
// 1) Find the first non-zero coefficient.
// 2) Check to see if the row is all zeros.
// 3) Set the first non-zero cofficient to 1 by multiplying the entire
//    row containing that coefficient by 1/coefficient.
// 4) Set the all the coefficients directly above the pivot to
//    zero by successively multiplying the pivot equation by
//    (-1) * coefficient above and then adding the two rows together.
// 5) Set all the coefficients directly below the pivot to
//    zero by the same method.
// 6) Repeat until each row contains only a '1' in the pivot
//    location and zeros everywhere else, except for the last
//    column.
////////////////////////////////////////////////////////////////////////
```

```
    public LinearEquationSolver gaussJord()
    {
        LinearEquationSolver ret = new LinearEquationSolver(rows,cols,M);

        for(int i=0;i<rows;i++){

            // first find first non-zero coefficient
            int j=0;
            while(ret.M[i][j] == (double)0){
                j++;
                if(j==cols)
                    break;
            }

            // if this row is all zeros just skip on to next row
            if(j==cols)
                continue;

            // set leading one
            ret = ret.mulRow( i, 1.0/ret.M[i][j]);

            //set zeros above leading one
            if(i!=0)
                for(int k=i-1; k>=0; k--)
                    ret=ret.addMulRow(i,k,-1.0*ret.M[k][j]);

            //set zeros below leading one
            for(int k=i+1; k<rows; k++)
                ret=ret.addMulRow(i,k,-1.0*ret.M[k][j]);

        } // end for loop
        return ret;
    }

    ////////////////////////////////////////////////////////////////////////
   // Output the entire matrix as a String object.
   ////////////////////////////////////////////////////////////////////////
    public String toString()
    {
        String string = "";
      if(rows<40){
         for(int i=0;i<rows;i++){
                   for(int j=0; j<cols; j++){
                   string += customFormat("00.00", M[i][j]) + "     ";
                }
               string += "\n";
            }
            return string;
      }
      else{
         return("Output matrix is too large to display in text box.\n"+
               "See output file for data.\n\n");
      }
    }

   ////////////////////////////////////////////////////////////////////////
   // This method creates a DecimalFormat object for formatting the file
   // output string.  It returns a string formatted to the specified decimal
   // precision.
   ////////////////////////////////////////////////////////////////////////
   private String customFormat(String pattern, double value )
   {
         DecimalFormat myFormatter = new DecimalFormat(pattern);
         String output = myFormatter.format(value);
      return(output);
      }
}
```

```
/*
 * Painter Class
 *
 *
 ************************************************************************
 *
 *
 * Course:        ECSE 6990
 * Instructor:    Dr. William Randolph Franklin
 * Date:          December 24, 2002
 * Student:       John Childs
 *
 ************************************************************************
 *
 * Popup Graphics Window Class
 *
 ************************************************************************/
package graphics;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

//-------------------------------------------------------------------------
public class Painter extends JFrame
{
   private int rows, cols;
   private double[][] N;
   Color col[]={  Color.blue,
               Color.green,
               Color.lightGray,
               Color.magenta,
               Color.pink,
               Color.red,
               Color.orange,
               Color.yellow};

//-------------------------------------------------------------------------
   public Painter(int rows, int cols, double[][]N)
   {
      super("Contour Plot Window");
      setBackground(Color.white);
      this.rows=rows;
      this.cols=cols;
      this.N=N;
      setSize(rows, cols);
      repaint();
      show();
   }

//-------------------------------------------------------------------------
   public void paint(Graphics g)

   {
      double max=getMax();
      double min=getMin();
      double interval=(max-min)/7.0;

      if(min<0.0){
         for(int i=0; i<rows; i++){
            for(int j=0; j<cols; j++){
               N[i][j]=N[i][j]+(0.0-min);
            }
         }
      }
      if(rows<100){
         for(int i=0; i<rows; i++){
            for(int j=0; j<cols; j++){
               g.setColor(col[((int)((N[i][j]-min)/interval))]);
               g.drawLine(i,j,i,j);
            }
```

```
         }
      }
      else{
         for(int i=0; i<rows; i++){
            for(int j=0; j<cols; j++){
               g.setColor(col[((int)((N[i][j]-min)/interval))]);
               g.drawLine(i,j,i,j);
            }
         }
      }

   }
//---------------------------------------------------------------------------
   private double getMax()

   {
      double max=N[0][0];
      for(int i=0; i<rows; i++){
         for(int j=0; j<cols; j++){
            if(N[i][j]>max){max=N[i][j];
            }
         }
      }
      return(max);
   }
//---------------------------------------------------------------------------
   private double getMin()

   {
      double min=N[0][0];
      for(int i=0; i<rows; i++){
         for(int j=0; j<cols; j++){
            if(N[i][j]<min){min=N[i][j];
            }
         }
      }
      return(min);
   }
//---------------------------------------------------------------------------
}
```

```
/*
 * InputPainter Class
 *
 *
 ***************************************************************************
 *
 *
 * Course:        ECSE 6990
 * Instructor:    Dr. William Randolph Franklin
 * Date:          December 24, 2002
 * Student:       John Childs
 *
 ***************************************************************************
 *
 * Popup Graphics Window Class
 *
 ***************************************************************************/
package graphics;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

//-----------------------------------------------------------------------------
public class InputPainter extends JFrame
{
   private Image img;

//-----------------------------------------------------------------------------
   public InputPainter(Image img, int width,int height)
   {
      super("Image Window");
      setBackground(Color.white);
      setSize(width, height);
      this.img=img;
      repaint();
      show();
   }

//-----------------------------------------------------------------------------
   public void paint(Graphics g)

   {
      g.drawImage(img, 0, 0, null);
   }
}
```

```
/*
```

Appendix 3 C++ Code Modules

```
/* Course:        CSCI6980
* Instructor:     Dr. Franklin
* Date:        February 14, 2003
* Student:        John Childs
*
**************************************************************************
*
* Master's Project:
* IODataManager
*
* This is the root function for the SparseSystem.
 *************************************************************************/

#include "SparseSystem.h"

int main( )
{
   SparseSystem sparseSystem;
   return (0);
}
```

```
/*
 * Declarations for SparseSystem.h.
 *
 *
 ***************************************************************************
 *
 * Course:        CSCI6980
 * Instructor:    Dr. Franklin
 * Date:       February 14, 2003
 * Student:       John Childs
 *
 ***************************************************************************
 *
 * Master's Project:
 * SparseSystem
 *
 * This class is the main C++ class for converting an elevation grid
 * file to a MATLAB sparse index file.
 *
 * Note:  it is not possible to instantiate IODataManager from the heap.
 * If this is done (by declaring a pointer and then instantiating with
 * the 'new' operator) it is not possible to write to the output file
 * using the overloaded stream injection operator '<<', as in
 * outfile<<somenumber.  This appears to be a bug because the extraction
 * operator '>>' works OK in this case, and both work when the object
 * is instantiated from the stack instead of the heap.
 ***************************************************************************/

#ifndef sparse_system_h
#define sparse_system_h
#include<iostream>
#include<stdio.h>
#include "IODataManager.h"
#include "AllocationManager.h"
#include "ComputationManager.h"
#include<time.h>

using std::endl;

class SparseSystem

{

public:
        SparseSystem();
        ~SparseSystem();

private:
      AllocationManager *allocationManager;
      ComputationManager *computationManager;
      void containerMethod(void);
      int** indexHandle;
      double* vectorHandle;
      int** inputHandle;
      int elevationRows, elevationColumns;
      int nzn;
      int indexRows;
      int indexColumns;
      int vectorColumns;
      clock_t start, end;
};
#endif
```

```
#include "SparseSystem.h"
IODataManager ioDataManager;


//////////////////////////////////////////////////////////////////////////
// Constructor                                                          //
//////////////////////////////////////////////////////////////////////////

SparseSystem::SparseSystem(){
   cout<<"in the SparseSystem constructor"<<endl;
   allocationManager=new AllocationManager;
   computationManager=new ComputationManager;
   containerMethod();
};

//////////////////////////////////////////////////////////////////////////
// Destructor                                                           //
//////////////////////////////////////////////////////////////////////////

SparseSystem::~SparseSystem(){
   allocationManager->deAllocate(inputHandle, elevationRows);
   allocationManager->deAllocateRHS(vectorHandle);
};

//////////////////////////////////////////////////////////////////////////
// This method calls the other methods.                                 //
//////////////////////////////////////////////////////////////////////////

void SparseSystem::containerMethod(void){

   ////////////////////////////////////////////////////////////////////
   // Write a greeting banner.                                       //
   ////////////////////////////////////////////////////////////////////

   ioDataManager.writeBanner();

   ////////////////////////////////////////////////////////////////////
   // Get the elevaton grid rows and columns.                        //
   ////////////////////////////////////////////////////////////////////

   elevationRows=ioDataManager.getRows();
   elevationColumns=ioDataManager.getColumns();


   ////////////////////////////////////////////////////////////////////
   // Open an input file.  The file to be opened is determined by    //
   // the first argument.  In this case getInputFile() returns a     //
   // reference to the input file declared in IODataManager          //
   // ioDataManager.                                                 //
   ////////////////////////////////////////////////////////////////////

   ioDataManager.openInputFile(  ioDataManager.getInputFile(),
                           ioDataManager.getInputFileBuffer());


   ////////////////////////////////////////////////////////////////////
   // Open an output file.  The file to be opened is determined by   //
   // the first argument.  In this case getIndexFile() returns a     //
   // reference to the index file declared in IODataManager          //
   // ioDataManager.                                                 //
   ////////////////////////////////////////////////////////////////////

   ioDataManager.openIndexFile(  ioDataManager.getIndexFile(),
                           ioDataManager.getIndexFileBuffer());


   ////////////////////////////////////////////////////////////////////
   // Open another output file.  The file to be opened is determined by //
   // the first argument.  In this case getVectorFile() returns a    //
   // reference to the b vector file declared in IODataManager       //
   // ioDataManager.                                                 //
```

```
//////////////////////////////////////////////////////////////////////


ioDataManager.openVectorFile( ioDataManager.getVectorFile(),
                        ioDataManager.getVectorFileBuffer());

//////////////////////////////////////////////////////////////////////
// Allocate an rows row by columns column array of ints from the     //
// heap. The array pointer is declared in allocationManager.  The    //
// particular array pointer is specified by the first argument.  In  //
// this case it is the pointer returned by getInputArray, i.e. the   //
// pointer to the array that will hold the input elevation data.     //
//////////////////////////////////////////////////////////////////////

inputHandle=allocationManager->allocate(allocationManager->getInputArray(),
                              elevationRows, elevationColumns);

//////////////////////////////////////////////////////////////////////
// Read the input file into the input array.                         //
//////////////////////////////////////////////////////////////////////

ioDataManager.readArray(ioDataManager.getInputFile(), inputHandle,
                  elevationRows, elevationColumns);

//////////////////////////////////////////////////////////////////////
// Count the number of non zero nodes in the input file.             //
//////////////////////////////////////////////////////////////////////

nzn=computationManager->nonzeroNodes(  inputHandle,
                             elevationRows, elevationColumns);

//////////////////////////////////////////////////////////////////////
// Compute the number of rows in the index array.                    //
//////////////////////////////////////////////////////////////////////

indexRows=computationManager->numberIndexRows(  elevationRows,
                                   elevationColumns, nzn);

//////////////////////////////////////////////////////////////////////
// Compute the number of columns in the index array.                 //
//////////////////////////////////////////////////////////////////////

indexColumns=computationManager->numberIndexColumns();

//////////////////////////////////////////////////////////////////////
// Allocate a rows row by columns column array of ints from the      //
// heap. The array pointer is declared in allocationManager.  The    //
// particular array pointer is specified by the first argument.  In  //
// this case it is the pointer returned by getIndexArray, i.e. the   //
// pointer to the array that will hold the MATLAB sparse index.      //
// The function returns the pointer (and not the array I hope)       //
// because it is needed for subsequent array access.                 //
//////////////////////////////////////////////////////////////////////

indexHandle=allocationManager->allocate(allocationManager->getIndexArray(),
                             indexRows, indexColumns);

//////////////////////////////////////////////////////////////////////
// Fill the indexArray with all zeros.                               //
//////////////////////////////////////////////////////////////////////

indexHandle=computationManager->allZeros( indexHandle,
                                indexRows, indexColumns);

//////////////////////////////////////////////////////////////////////
// Compute the index file coefficients and place them in the         //
// indexArray.                                                       //
//////////////////////////////////////////////////////////////////////

start=clock();
```

```
indexHandle=computationManager->computeUpperLeft(  indexHandle,
                                    elevationRows,
                                    elevationColumns);

indexHandle=computationManager->computeUpperRight( indexHandle,
                                    elevationRows,
                                    elevationColumns);

indexHandle=computationManager->computeLowerLeft(  indexHandle,
                                    elevationRows,
                                    elevationColumns);

indexHandle=computationManager->computeLowerRight( indexHandle,
                                    elevationRows,
                                    elevationColumns);

indexHandle=computationManager->computeLeft(    indexHandle,
                                    elevationRows,
                                    elevationColumns);

indexHandle=computationManager->computeRight(      indexHandle,
                                    elevationRows,
                                    elevationColumns);

indexHandle=computationManager->computeUpper(      indexHandle,
                                    elevationRows,
                                    elevationColumns);

indexHandle=computationManager->computeLower(      indexHandle,
                                    elevationRows,
                                    elevationColumns);

indexHandle=computationManager->computeInside(     indexHandle,
                                    elevationRows,
                                    elevationColumns);

indexHandle=computationManager->computeLowerBlock( inputHandle,
                                    indexHandle,
                                    elevationRows,
                                    elevationColumns);
end=clock();


//////////////////////////////////////////////////////////////////////
// Write run statistics to console.                                 //
//////////////////////////////////////////////////////////////////////

ioDataManager.runStatistics(  nzn, indexRows, indexColumns, start, end,
                        computationManager->getEquations());


//////////////////////////////////////////////////////////////////////
// Write indexArray to file                                         //
//////////////////////////////////////////////////////////////////////

ioDataManager.writeArray(  ioDataManager.getIndexFile(),
                     indexHandle, indexRows, indexColumns);


//////////////////////////////////////////////////////////////////////
// Deallocate the big index array because it is no longer needed.   //
//////////////////////////////////////////////////////////////////////

allocationManager->deAllocate(indexHandle, indexRows);

//////////////////////////////////////////////////////////////////////
// Allocate a rows row by columns column array of ints from the     //
// heap. The array pointer is declared in allocationManager.  The   //
// particular array pointer is specified by the first argument.  In //
// this case it is the pointer returned by getVectorArray, i.e. the //
// pointer to the array that will hold the MATLAB RHS vector.       //
```

```
   // Note: the number of RHS vector rows is the same as the index      //
   // matrix rows.                                                      //
   ///////////////////////////////////////////////////////////////////////

   vectorHandle=allocationManager->allocateRHS(allocationManager->
                                  getVectorArray(),
                                  indexRows);

   ///////////////////////////////////////////////////////////////////////
   // Compute RHS vector.                                               //
   ///////////////////////////////////////////////////////////////////////

   vectorHandle=computationManager->zeroRHS( vectorHandle,
                                  ((elevationRows *
                                  elevationColumns)
                                  + nzn));

   vectorHandle= computationManager->computeRHS(   vectorHandle,
                                     inputHandle,
                                     elevationRows,
                                     elevationColumns,
                                     computationManager->
                                     computeSparseRows
                                     (elevationRows,
                                     elevationColumns));

   ///////////////////////////////////////////////////////////////////////
   // And finally, write it to file.  Destructor handles remaining      //
   // deallocation chores.                                              //
   ///////////////////////////////////////////////////////////////////////

   ioDataManager.writeRHSArray(ioDataManager.getVectorFile(),
                       vectorHandle, ((elevationRows *
                       elevationColumns) + nzn));
}
```

```
/* Course:        CSCI6980
 * Instructor:    Dr. Franklin
 * Date:          February 14, 2003
 * Student:       John Childs
 *
 ********************************************************************************
 *
 * Master's Project:
 * IODataManager
 *
 * This class is the IO helper class for SparseSystem.
 ********************************************************************************/

#ifndef io_data_manager_h
#define io_data_manager_h
#include<iostream>
#include<conio.h>
#include <fstream.h>
#include<iomanip.h>
#include<time.h>


using std::endl;

class IODataManager

{
private:
      ifstream inputFile;
      ofstream indexFile;
      ofstream bFile;
      char inputFileBuffer[40];
      char indexFileBuffer[40];
      char bFileBuffer[40];

public:
        IODataManager();
        ~IODataManager();
      ifstream& openInputFile(ifstream&, char*);
      ofstream& openIndexFile(ofstream&, char*);
      ofstream& openVectorFile(ofstream&, char*);


      void writeBanner(void);
      ifstream& getInputFile(void);
      ofstream& getIndexFile(void);
      ofstream& getVectorFile(void);

      char* getInputFileBuffer(void);
      char* getIndexFileBuffer(void);
      char* getVectorFileBuffer(void);


      int getOneint(ifstream&);
      void writeOneint(int, ofstream&);
      void writeArray(ofstream&, int**, int, int);
      void writeRHSArray(  ofstream&, double*, int);

      void dumpArray(   int**, int, int);
      void readArray(   ifstream&, int**, int, int);

      int getRows(void);
      int getColumns(void);

      void runStatistics(int, int, int, clock_t, clock_t, int);

};
#endif
```

```
#include "IODataManager.h"

////////////////////////////////////////////////////////////////////////
// Constructor                                                        //
////////////////////////////////////////////////////////////////////////

IODataManager::IODataManager(){
   cout<<"in IODataManager constructor."<<endl;
};

////////////////////////////////////////////////////////////////////////
// Destructor.  Closes all files.                                     //
////////////////////////////////////////////////////////////////////////

IODataManager::~IODataManager(){
   indexFile.close();
   inputFile.close();
   bFile.close();
};

////////////////////////////////////////////////////////////////////////
// Function writes the greeting banner.                               //
////////////////////////////////////////////////////////////////////////

void IODataManager::writeBanner(void)
{
   cout<<endl;
   cout<<"  ************************************************"<<endl;
   cout<<"  *                                              *"<<endl;
   cout<<"  *           Welcome to SparseSystem.           *"<<endl;
   cout<<"  *                                              *"<<endl;
   cout<<"  *                                              *"<<endl;
   cout<<"  *   The program accepts as input an ASCII tab   *"<<endl;
   cout<<"  *   separated elevation contour grid and        *"<<endl;
   cout<<"  *   computes the MATLAB sparse index file and   *"<<endl;
   cout<<"  *   the corresponding RHS vector for the        *"<<endl;
   cout<<"  *   Franklin over-determined Laplacian          *"<<endl;
   cout<<"  *   algorithm.                                  *"<<endl;
   cout<<"  *                                              *"<<endl;
   cout<<"  *   Note: you must specify the number of rows   *"<<endl;
   cout<<"  *   and columns for the elevation grid file     *"<<endl;
   cout<<"  *   when prompted.  If you make a mistake in    *"<<endl;
   cout<<"  *   specifying them, the program will at best   *"<<endl;
   cout<<"  *   produce a bad output file or at worst       *"<<endl;
   cout<<"  *   overflow an array and cause a segment       *"<<endl;
   cout<<"  *   violation error.                            *"<<endl;
   cout<<"  *                                              *"<<endl;
   cout<<"  *   If you are lucky, this will only cause      *"<<endl;
   cout<<"  *   the program to terminate.                   *"<<endl;
   cout<<"  *                                              *"<<endl;
   cout<<"  *                                              *"<<endl;
   cout<<"  ************************************************"<<endl<<endl;
   cout<<"Press any key to continue"<<endl;
   getch();
}

////////////////////////////////////////////////////////////////
// This function opens the elevation file for input in binary  //
// mode.  Returns a reference that can be forwarded to other    //
// helper classes                                               //
////////////////////////////////////////////////////////////////

ifstream& IODataManager::openInputFile(   ifstream& someFile,
                              char inputFileBuffer[])
{

LABL: cout<<endl<<"******************************************************"<<endl;
      cout<<"\nInput the elevation grid file name.\n";
      cout<<"\nThe correct file will look like '<filename>.txt'"<<endl;
      cout<<"Please type in the complete file name including";
      cout<<"the directory path\n";
```

```
      cout<<"for example: c:\\inputfile.txt\n";
      cout<<"*******************************************************"<<endl<<endl;

      cin.get(inputFileBuffer,40);  //extra read to work
      cin.ignore(100, '\n');        //around console bug.

      cin.get(inputFileBuffer,40);
      cin.ignore(100, '\n');
      someFile.open(inputFileBuffer, ios::binary);
      if(!someFile)
      {
         cout<<"**Cannot open"<<inputFileBuffer<<"**"<<endl;
         cout<<"Please try again."<<endl<<endl;
         goto LABL;
      }
      return(someFile);
}


//////////////////////////////////////////////////////////////////
// This function opens the index file for output in binary      //
//mode.                                                         //
//////////////////////////////////////////////////////////////////

ofstream& IODataManager::openIndexFile(   ofstream& otherFile,
                            char outputFileBuffer[])
{

LABL:    cout<<endl<<"*******************************************************"<<endl;
         cout<<"\nInput the name of the output index file\n";
         cout<<"\nPlease type in the complete file name including";
         cout<<" the directory path\n";
         cout<<"for example: c:\\outfile.dat\n";
         cout<<"*******************************************************"<<endl<<endl;

         cin.get(outputFileBuffer,40);
         cin.ignore(100, '\n');
         otherFile.open(outputFileBuffer, ios::binary);
         if(!otherFile)
         {
            cout<<"**Cannot open"<<outputFileBuffer<<"**"<<endl;
            cout<<"Please try again."<<endl<<endl;
            goto LABL;
         }
         return(otherFile);
}


//////////////////////////////////////////////////////////////////
// This function opens the b vector file for output in binary   //
// mode.                                                        //
//////////////////////////////////////////////////////////////////

ofstream& IODataManager::openVectorFile(  ofstream& otherFile,
                              char outputFileBuffer[])
{

LABL:    cout<<endl<<"*******************************************************"<<endl;
         cout<<"\nInput the name of the output b vector file\n";
         cout<<"\nPlease type in the complete file name including";
         cout<<" the directory path\n";
         cout<<"for example: c:\\bfile.m\n";
         cout<<"*******************************************************"<<endl<<endl;

         cin.get(outputFileBuffer,40);
         cin.ignore(100, '\n');
         otherFile.open(outputFileBuffer, ios::binary);
         if(!otherFile)
         {
            cout<<"**Cannot open"<<outputFileBuffer<<"**"<<endl;
            cout<<"Please try again."<<endl<<endl;
```

```
            goto LABL;
         }
         return(otherFile);
}

////////////////////////////////////////////////////////////////
// This function gets the number of rows in the input elevation //
// matrix.  Some people would frown on this code, for no        //
// logical reason.  It works.                                   //
////////////////////////////////////////////////////////////////

int IODataManager::getRows(void)
{
int rows;

LABL: cout<<"\n\nInput the number of rows in the elevation file\n\n";
         cin>>rows;
         if(rows<=0)
         {
            cout<<"**rows must be > 0**"<<endl;
            cout<<"Please try again."<<endl<<endl;
            goto LABL;
         }
         return(rows);
}

////////////////////////////////////////////////////////////////
// This function gets the number of columns in the input        //
// elevation matrix.                                            //
////////////////////////////////////////////////////////////////

int IODataManager::getColumns(void)
{
int columns;

LABL: cout<<"\n\nInput the number of columns in the elevation file\n\n";
         cin>>columns;
         if(columns<=0)
         {
            cout<<"**columns must be > 0**"<<endl;
            cout<<"Please try again."<<endl<<endl;
            goto LABL;
         }
         return(columns);
}

////////////////////////////////////////////////////////////////////////
// Function gets a reference to the inputFile                         //
////////////////////////////////////////////////////////////////////////

ifstream& IODataManager::getInputFile(void)
{
   return(inputFile);
}

////////////////////////////////////////////////////////////////////////
// Function gets a reference to the indexFile                         //
////////////////////////////////////////////////////////////////////////

ofstream& IODataManager::getIndexFile(void)
{
   return(indexFile);
}

////////////////////////////////////////////////////////////////////////
// Function gets a reference to the bFile                             //
////////////////////////////////////////////////////////////////////////

ofstream& IODataManager::getVectorFile(void)
{
   return(bFile);
```

```
}
////////////////////////////////////////////////////////////////////////////
// Function gets a pointer to the inputFileBuffer                         //
////////////////////////////////////////////////////////////////////////////

char* IODataManager::getInputFileBuffer(void)
{
   return(inputFileBuffer);
}

////////////////////////////////////////////////////////////////////////////
// Function gets a pointer to the outputFileBuffer                        //
////////////////////////////////////////////////////////////////////////////

char* IODataManager::getIndexFileBuffer(void)
{
   return(indexFileBuffer);
}

////////////////////////////////////////////////////////////////////////////
// Function gets a pointer to the vectorFileBuffer                        //
////////////////////////////////////////////////////////////////////////////

char* IODataManager::getVectorFileBuffer(void)
{
   return(bFileBuffer);
}

////////////////////////////////////////////////////////////////////////////
// Function reads a int from the input file stream                        //
////////////////////////////////////////////////////////////////////////////

int IODataManager::getOneint(ifstream& inputFile)
{
   int tempint;
   inputFile>>tempint;
   return(tempint);
}


////////////////////////////////////////////////////////////////////////////
// Function the contents of a file into an array                          //
////////////////////////////////////////////////////////////////////////////

void IODataManager::readArray(   ifstream& inputFile, int** inputArray,
                           int rows, int columns)
{
   for(int i=0; i<rows; i++)
   {
      for(int j=0; j<columns; j++)
      {
         inputFile>>inputArray[i][j];
      }
   }
}

////////////////////////////////////////////////////////////////////////////
// Function writes a int to the output file stream                        //
////////////////////////////////////////////////////////////////////////////

void IODataManager::writeOneint(int outint, ofstream& outputFile)
{
   cout<<"writing "<<outint<<" to output file"<<endl;
   outputFile<<outint;
}

////////////////////////////////////////////////////////////////////////////
// Function writes the entire contents of an array to the specified   //
// output file in MATLAB sparse matrix index file format.  This fomat    //
// is suitable for conversion to a MATLAB sparse matrix using the        //
// command spconvert().  A newline (\n) is inserted at the end of every  //
```

```
// row. Formatting is field width=10, precision=0 (no decimals since all //
// elements are expected to be integers.                                  //
//                                                                        //
// This method must be redefined in order to output the dat in other      //
// formats.                                                               //
////////////////////////////////////////////////////////////////////////////

void IODataManager::writeArray(  ofstream& outputFile, int** outputArray,
                        int rows, int columns)
{
   cout<<"writing sparse matrix coefficient index file"<<endl<<endl;
   for(int i=0; i<rows; i++)
   {
      for(int j=0; j<columns; j++)
      {
         outputFile<<setw(10)<<setprecision(0)<<setiosflags(ios::fixed)
                                    <<outputArray[i][j];
      }
      outputFile<<"\n";
   }
}


////////////////////////////////////////////////////////////////////////////
// Function writes the RHS array to the specified file.                   //
////////////////////////////////////////////////////////////////////////////

void IODataManager::writeRHSArray(  ofstream& outputFile, double* outputArray,
                        int rows)
{
   cout<<"writing RHS vector file"<<endl<<endl;
   outputFile<<"b=[\n";
   for(int i=0; i<rows; i++)
   {

      outputFile<<setw(10)<<setprecision(5)<<setiosflags(ios::fixed)
                                    <<outputArray[i];
      outputFile<<"\n";
   }
      outputFile<<"]";

}


////////////////////////////////////////////////////////////////////////////
// Function writes the entirre contents of an array to console.           //
// For diagnostic purposes.                                               //
////////////////////////////////////////////////////////////////////////////

void IODataManager::dumpArray(   int** outputArray,
                        int rows, int columns)
{
   cout<<"writing to output file"<<endl;
   for(int i=0; i<rows; i++)
   {
      for(int j=0; j<columns; j++)
      {
         std::cout<<outputArray[i][j];
         std::cout<<"  ";
      }
      std::cout<<"\n";
   }
}

////////////////////////////////////////////////////////////////////////////
// Function writes statistics for the run.                                //
////////////////////////////////////////////////////////////////////////////

void IODataManager::runStatistics(  int nzn,
                           int indexRows,
                           int indexColumns,
```

```
                          clock_t start,
                          clock_t end,
                          int equations)
{
   cout<<endl;
   cout<<"*****************************************************"<<endl;
   cout<<"non zero nodes is:  "<<nzn<<endl;
   cout<<"index rows is:  "<<indexRows<<endl;
   cout<<"index columns is:  "<<indexColumns<<endl;
   cout<<endl<<"Total CPU time is:  "<<end-start<<" CPU units"<<endl;
   cout<<endl<<"Total CPU time is:  "<<setw(10)<<setprecision(5)<<
            (float)((end-start)/CLK_TCK)<<" seconds"<<endl<<endl;
   cout<<"The number of equations computed is:  "<<
                        equations<<endl;
   cout<<"*****************************************************"<<endl<<endl;
}
```

```
/*
 * Declarations for AllocationManager.h.
 *
 *
 ***************************************************************************
 *
 * Course:        CSCI6980
 * Instructor:    Dr. Franklin
 * Date:       February 14, 2003
 * Student:       John Childs
 *
 ***************************************************************************
 *
 * Master's Project:
 * AllocationManager
 *
 * This class is the C++ helper class for allocating and
 * otherwise managing the large arrays.
 ***************************************************************************/

#ifndef allocation_manager_h
#define allocation_manager_h
#include<iostream>

using std::endl;

class AllocationManager

{

public:
        AllocationManager();
        ~AllocationManager(){};
      int** allocate( int **, int rows, int columns);
      void deAllocate( int **, int);

      int** getIndexArray(void);
      double* getVectorArray(void);
      int** getInputArray(void);


      int getIndexArrayElement(int**,int, int);
      void setIndexArrayElement(int**, int, int, int);
      void setVectorArrayElement(   double*, int, int);
      double* allocateRHS( double*, int);
      void deAllocateRHS( double*);



private:
      int** indexArray;
        double* vectorArray;
      int** inputArray;


};
#endif
```

```
#include "AllocationManager.h"

////////////////////////////////////////////////////////////////////////
// Constructor                                                        //
////////////////////////////////////////////////////////////////////////

AllocationManager::AllocationManager(){
   std::cout<<"in AllocationManager constructor"<<endl;

};

////////////////////////////////////////////////////////////////////////
// Function allocates int arrays for row X column sized arrays.      //
////////////////////////////////////////////////////////////////////////

int** AllocationManager::allocate(  int **inputArray,
                                    int rows, int columns)
{
   inputArray=new int*[rows];
   for (int n = 0; n < rows; n++)
   {
      inputArray[n] = new int[columns];
   }
   return(inputArray);
}

////////////////////////////////////////////////////////////////////////
// Function allocates int arrays for row X 1 sized arrays.            //
////////////////////////////////////////////////////////////////////////

double* AllocationManager::allocateRHS(   double *inputArray,
                                    int rows)
{
   inputArray=new double[rows];
   return(inputArray);
}

////////////////////////////////////////////////////////////////////////
// Function deallocates int arrays for row X column sized arrays.     //
////////////////////////////////////////////////////////////////////////

void AllocationManager::deAllocate( int **inputArray, int rows)
{
   for (int n = 0; n < rows; n++)
   {
      delete[] inputArray[n];
   }
   delete[]inputArray;
}

////////////////////////////////////////////////////////////////////////
// Function deallocates int arrays for row X column sized arrays.     //
////////////////////////////////////////////////////////////////////////

void AllocationManager::deAllocateRHS( double *inputArray)
{
   delete[]inputArray;
}

////////////////////////////////////////////////////////////////////////
// Function returns a pointer to the indexArray.                      //
////////////////////////////////////////////////////////////////////////

int** AllocationManager::getIndexArray(void)
{
   return(indexArray);
}

////////////////////////////////////////////////////////////////////////
// Function returns a pointer to the vectoArray.                      //
```

```
////////////////////////////////////////////////////////////////////////

double* AllocationManager::getVectorArray(void)
{
   return(vectorArray);
}

////////////////////////////////////////////////////////////////////////
// Function returns a pointer to the inputArray.                      //
////////////////////////////////////////////////////////////////////////

int** AllocationManager::getInputArray(void)
{
   return(inputArray);
}

////////////////////////////////////////////////////////////////////////
// Function sets the i,j index of the indexArray                      //
////////////////////////////////////////////////////////////////////////

void AllocationManager::setIndexArrayElement(   int** indexArray,
                                    int i, int j, int value)
{
   indexArray[i][j]=value;
}

////////////////////////////////////////////////////////////////////////
// Function sets the i,j index of the vector Array                    //
////////////////////////////////////////////////////////////////////////

void AllocationManager::setVectorArrayElement(  double* vectorArray,
                                    int i, int value)
{
   vectorArray[i]=value;
}


////////////////////////////////////////////////////////////////////////
// Function returns the value at the i,j index of the indexArray      //
////////////////////////////////////////////////////////////////////////

int AllocationManager::getIndexArrayElement( int** indexArray,
                                    int i, int j)
{
   return(indexArray[i][j]);
}
```

```
/*
 * Declarations for ComputationManager.h.
 *
 *
 ***************************************************************************
 *
 * Course:        CSCI6980
 * Instructor:    Dr. Franklin
 * Date:       February 14, 2003
 * Student:       John Childs
 *
 ***************************************************************************
 *
 * Master's Project:
 * ComputationManager
 *
 * This class is the C++ helper class for computing the sparse index matrix
 * array and the b vector array.
 ***************************************************************************/

#ifndef computation_manager_h
#define computation_manager_h
#include<iostream>

#define INDEXCOLUMNS 3;
#define VECTORCOLUMNS 1;

//using std::cout;
using std::endl;

class ComputationManager
{

public:
        ComputationManager();
        ~ComputationManager(){};
      void computeEdgeNodes(void);
      int nonzeroNodes(int**, int, int);
      int numberIndexRows(int, int, int);
      int numberIndexColumns(void);
      int numberVectorColumns(void);
      int getEquations(void);

      int** computeUpperLeft(int**, int, int);
      int** computeUpperRight(int**, int, int);
      int** computeLowerLeft(int**, int, int);
      int** computeLowerRight(int**, int, int);
      int** allZeros(int**, int, int);
      int** computeLeft(int**, int, int);
      int** computeRight(int**, int, int);
      int** computeUpper(int**, int, int);
      int** computeLower(int**, int, int);
      int** computeInside(int**, int, int);
      int** computeLowerBlock(int**, int**, int, int);
      double*  computeRHS(double*, int**, int, int, int);
      int   computeSparseRows(int, int);
      double*   zeroRHS(double*, int);



private:
      int equationCounter;
      int sparseRows;

};
#endif
```

```
#include "ComputationManager.h"


////////////////////////////////////////////////////////////////////////////
// Constructor                                                            //
////////////////////////////////////////////////////////////////////////////

ComputationManager::ComputationManager():equationCounter(0), sparseRows(0){
   std::cout<<"in ComputationManager constructor"<<endl;
}

////////////////////////////////////////////////////////////////////////////
// Zero out the index matrix.                                             //
////////////////////////////////////////////////////////////////////////////

int** ComputationManager::allZeros(int** inputArray, int rows, int columns){
   for(int i=0; i<rows; i++)
   {
      for(int j=0; j<columns; j++)
      {
         inputArray[i][j]=0;
      }
   }
   return(inputArray);
}


////////////////////////////////////////////////////////////////////////////
// Find the number of non zero nodes                                      //
////////////////////////////////////////////////////////////////////////////

int ComputationManager::nonzeroNodes(int** inputArray, int rows, int columns){
   int nonZeroCounter=0;
   for(int i=0; i<rows; i++)
   {
      for(int j=0; j<columns; j++)
      {
         if(inputArray[i][j]>10e-06)
            nonZeroCounter=nonZeroCounter+1;
      }
   }

   return(nonZeroCounter);
}

////////////////////////////////////////////////////////////////////////////
// Compute indexRows, the number of rows in the MATLAB index array.       //
////////////////////////////////////////////////////////////////////////////

int ComputationManager::numberIndexRows(int rows, int columns, int nonZeroElevations){
   int indexRows=0;
   indexRows=(((4*3) + ((rows-2)*4)*2 + ((columns-2)*4)*2 + ((rows-2)*(columns-2))*5)) +
nonZeroElevations;
   std::cout<<"nonZeroElevations is:  "<<nonZeroElevations<<endl;

   return(indexRows);
}

////////////////////////////////////////////////////////////////////////////
// Compute indexColumns, the number of columns in the MATLAB index array.//
////////////////////////////////////////////////////////////////////////////

int ComputationManager::numberIndexColumns(void){
   int indexColumns;
   indexColumns=INDEXCOLUMNS;
   return(indexColumns);
}
```

```
////////////////////////////////////////////////////////////////////////////
// Get the number of equations inserted into indexArray                   //
////////////////////////////////////////////////////////////////////////////

int ComputationManager::getEquations(void){
   return(equationCounter);
}

////////////////////////////////////////////////////////////////////////////
// Compute indexColumns, the number of columns in the MATLAB RHS vector  //
// array.                                                                 //
////////////////////////////////////////////////////////////////////////////

int ComputationManager::numberVectorColumns(void){
   int vectorColumns;
   vectorColumns=VECTORCOLUMNS;
   return(vectorColumns);
}
////////////////////////////////////////////////////////////////////////////
// Compute the matrix coefficients for the upper left node in the        //
// elevation grid.                                                        //
////////////////////////////////////////////////////////////////////////////


int** ComputationManager::computeUpperLeft(int** indexMatrix, int rows, int columns){

   int keyColumn;
   int rightColumn;
   int lowerColumn;
   for(int i=1; i<=1; i++)
   {
      for(int j=1; j<=1; j++)
      {
         sparseRows=sparseRows+1;
         keyColumn=((i-1)*rows + j);
         rightColumn=keyColumn+1;
         lowerColumn=((i)*columns + j);

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=keyColumn;
         indexMatrix[equationCounter][2]=-2;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=rightColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=lowerColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;
      }
   }

   return(indexMatrix);
}

////////////////////////////////////////////////////////////////////////////
// Compute the matrix coefficients for the upper right node in the       //
// elevation grid.                                                        //
////////////////////////////////////////////////////////////////////////////


int** ComputationManager::computeUpperRight(int** indexMatrix, int rows, int columns){
   int keyColumn;
   int lowerColumn;
   int leftColumn;

   for(int i=1; i<=1; i++)
   {
```

```
      for(int j=columns; j<=columns; j++)
      {
         sparseRows=sparseRows+1;
         keyColumn=((i-1)*rows + j);
         leftColumn=keyColumn-1;
         lowerColumn=((i)*columns + j);

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=keyColumn;
         indexMatrix[equationCounter][2]=-2;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=leftColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=lowerColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;
      }
   }
   return(indexMatrix);
}


////////////////////////////////////////////////////////////////////////
// Compute the matrix coefficients for the lower left node in the        //
// elevation grid.                                                       //
////////////////////////////////////////////////////////////////////////


int** ComputationManager::computeLowerLeft(int** indexMatrix, int rows, int columns){
   int keyColumn;
   int rightColumn;
   int upperColumn;

   for(int i=rows; i<=rows; i++)
   {
      for(int j=1; j<=1; j++)
      {
         sparseRows=sparseRows+1;
         keyColumn=((i-1)*columns + j);
         rightColumn=keyColumn+1;
         upperColumn=((i-2)*columns + j);

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=keyColumn;
         indexMatrix[equationCounter][2]=-2;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=rightColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=upperColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;
      }
   }
   return(indexMatrix);
}

////////////////////////////////////////////////////////////////////////
// Compute the matrix coefficients for the lower right node in the       //
// elevation grid.                                                       //
////////////////////////////////////////////////////////////////////////
```

```
int** ComputationManager::computeLowerRight(int** indexMatrix, int rows, int columns){
   int keyColumn;
   int leftColumn;
   int upperColumn;

   for(int i=rows; i<=rows; i++)
   {
      for(int j=columns; j<=columns; j++)
      {
         sparseRows=sparseRows+1;
         keyColumn=((i-1)*columns + j);
         leftColumn=keyColumn-1;
         upperColumn=((i-2)*columns + j);

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=keyColumn;
         indexMatrix[equationCounter][2]=-2;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=leftColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=upperColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;
      }
   }
   return(indexMatrix);
}

////////////////////////////////////////////////////////////////////////////
// Compute the matrix coefficients for the left boundary.                 //
////////////////////////////////////////////////////////////////////////////

int** ComputationManager::computeLeft(int** indexMatrix, int rows, int columns){
   int keyColumn;
   int upperColumn;
   int lowerColumn;
   int rightColumn;

   for(int i=2; i<=rows-1; i++)
   {
      for(int j=1; j<=1; j++)
      {
         sparseRows=sparseRows+1;
         keyColumn=((i-1)*columns + j);
         rightColumn=keyColumn+1;
         upperColumn=((i-2)*columns + j);
         lowerColumn=((i)*columns + j);

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=keyColumn;
         indexMatrix[equationCounter][2]=-3;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=rightColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=upperColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=lowerColumn;
         indexMatrix[equationCounter][2]=1;
```

```
         equationCounter=equationCounter+1;
      }
   }
   return(indexMatrix);
}

////////////////////////////////////////////////////////////////////////
// Compute the matrix coefficients for the right boundary.            //
////////////////////////////////////////////////////////////////////////

int** ComputationManager::computeRight(int** indexMatrix, int rows, int columns){
   int keyColumn;
   int upperColumn;
   int lowerColumn;
   int leftColumn;

   for(int i=2; i<=rows-1; i++)
   {
      for(int j=columns; j<=columns; j++)
      {
         sparseRows=sparseRows+1;
         keyColumn=((i-1)*columns + j);
         leftColumn=keyColumn-1;
         upperColumn=((i-2)*columns + j);
         lowerColumn=((i)*columns + j);

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=keyColumn;
         indexMatrix[equationCounter][2]=-3;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=leftColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=upperColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=lowerColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;
      }
   }
   return(indexMatrix);
}

////////////////////////////////////////////////////////////////////////
// Compute the matrix coefficients for the upper boundary.            //
////////////////////////////////////////////////////////////////////////

int** ComputationManager::computeUpper(int** indexMatrix, int rows, int columns){
   int keyColumn;
   int rightColumn;
   int lowerColumn;
   int leftColumn;

   for(int i=1; i<=1; i++)
   {
      for(int j=2; j<=columns-1; j++)
      {
         sparseRows=sparseRows+1;
         keyColumn=((i-1)*rows + j);
         leftColumn=keyColumn-1;
         rightColumn=keyColumn+1;
         lowerColumn=((i)*columns + j);

         indexMatrix[equationCounter][0]=sparseRows;
```

```
         indexMatrix[equationCounter][1]=keyColumn;
         indexMatrix[equationCounter][2]=-3;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=leftColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=rightColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=lowerColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;
      }
   }
   return(indexMatrix);
}

//////////////////////////////////////////////////////////////////////////
// Compute the matrix coefficients for the lower boundary.              //
//////////////////////////////////////////////////////////////////////////

int** ComputationManager::computeLower(int** indexMatrix, int rows, int columns){
   int keyColumn;
   int rightColumn;
   int upperColumn;
   int leftColumn;

   for(int i=rows; i<=rows; i++)
   {
      for(int j=2; j<=columns-1; j++)
      {
         sparseRows=sparseRows+1;
         keyColumn=((i-1)*columns + j);
         leftColumn=keyColumn-1;
         rightColumn=keyColumn+1;
         upperColumn=((i-2)*columns + j);

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=keyColumn;
         indexMatrix[equationCounter][2]=-3;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=leftColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=rightColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=upperColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;
      }
   }
   return(indexMatrix);
}

//////////////////////////////////////////////////////////////////////////
// Compute the matrix coefficients for the inside nodes.                //
//////////////////////////////////////////////////////////////////////////
```

```
int** ComputationManager::computeInside(int** indexMatrix, int rows, int columns){
   int keyColumn;
   int rightColumn;
   int upperColumn;
   int leftColumn;
   int lowerColumn;

   for(int i=2; i<=rows-1; i++)
   {
      for(int j=2; j<=columns-1; j++)
      {
         sparseRows=sparseRows+1;
         keyColumn=((i-1)*columns + j);
         leftColumn=keyColumn-1;
         rightColumn=keyColumn+1;
         upperColumn=((i-2)*columns + j);
         lowerColumn=((i)*columns + j);

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=keyColumn;
         indexMatrix[equationCounter][2]=-4;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=leftColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=rightColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=upperColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;

         indexMatrix[equationCounter][0]=sparseRows;
         indexMatrix[equationCounter][1]=lowerColumn;
         indexMatrix[equationCounter][2]=1;
         equationCounter=equationCounter+1;
      }
   }
   return(indexMatrix);
}

//////////////////////////////////////////////////////////////////////////
// Compute the matrix coefficients for the lower block of the           //
// sparse matrix.  This section corresponds to the equations of the form://
// zi=ei.                                                               //
//////////////////////////////////////////////////////////////////////////


int**  ComputationManager::computeLowerBlock(int**  inputMatrix,  int**indexMatrix,  int
rows, int columns){
   int keyColumn;
   for(int i=0; i<rows; i++)
      for(int j=0; j<columns; j++)
      {
         if(inputMatrix[i][j] != 0)
         {
            sparseRows=sparseRows+1;
            keyColumn=((i-1+1)*columns + j+1);
            indexMatrix[equationCounter][0]=sparseRows;
            indexMatrix[equationCounter][1]=keyColumn;
            indexMatrix[equationCounter][2]=1;
            equationCounter=equationCounter+1;
         }
      }
   return(indexMatrix);
```

```
}

////////////////////////////////////////////////////////////////////////
// Zero out the RHS array.                                            //
////////////////////////////////////////////////////////////////////////

double* ComputationManager::zeroRHS(double*vectorMatrix, int rows){

   for(int i=0; i<rows; i++)
      vectorMatrix[i]=0;
   return(vectorMatrix);
}



////////////////////////////////////////////////////////////////////////
// Compute the sparseRows: input to computeRHS.                       //
////////////////////////////////////////////////////////////////////////

int ComputationManager::computeSparseRows(int rows, int columns){

   int sparseRows=rows * columns;
   return(sparseRows);
}

////////////////////////////////////////////////////////////////////////
// Compute the RHS vector.                                            //
////////////////////////////////////////////////////////////////////////

double* ComputationManager::computeRHS(double* vectorMatrix, int**inputMatrix, int rows,
int columns, int sparseRows){

   for(int i=0; i<rows; i++)
      for(int j=0; j<columns; j++)
      {
         if(inputMatrix[i][j] != 0)
         {
            vectorMatrix[sparseRows]=inputMatrix[i][j];
            sparseRows=sparseRows+1;
         }
      }
   return(vectorMatrix);
}
```

```
/*
 * Declarations for ASCII2TGA.cpp
 *
 *
 ************************************************************************
 *
 * Course:           CSCI6980
 * Instructor:    Dr. Franklin
 * Date:          April 4, 2003
 * Student:       John Childs
 *
 ************************************************************************
 *
 * Master's Project:
 * ASCII2TGA
 *
 * This class converts a plain ASCII elevtion file to TGA format.
 ************************************************************************/


#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <fstream.h>

class ascii2tga
{
   private:

    typedef unsigned char BYTE;
   void OpenForOutput(ofstream& , char[40]);
   void OpenForInput(ifstream&, char[40]);
   void Write_Banner(void);
   void allocate(unsigned char**, int , int);
   void de_allocate(unsigned char**, int);

   void get_data(int, int);
   int getRows(void);
   int getColumns(void);
   void setup_array(int, int);
   void readArray(   ifstream& inputFile, unsigned char** inputArray,
                          int rows, int columns);

   void WriteHeaderData(ofstream&, int, int);
   void Write_to_File(ofstream& outfile1,
                           int r, int c,
                           unsigned char**some_array);


   void ContainerFunction(void);

   unsigned char** raw_array;
   char infileBuffer[40];

   char outfileBuffer[40];
    int rows, columns;
   BYTE lowByte;
   BYTE highByte;
   BYTE leader_id, tempchar;
   unsigned int integer1;

   typedef struct _TgaHeader
   {
      BYTE IDLengthByte;
      BYTE ColorMapTypeByte;
      BYTE ImageTypeByte;
      BYTE CMapStartByte1;
      BYTE CMapStartByte2;
      BYTE CMapLengthByte1;
      BYTE CMapLengthByte2;
```

```
      BYTE CMapDepthByte;
      BYTE XOffsetByte1;
      BYTE XOffsetByte2;
      BYTE YOffsetByte1;
      BYTE YOffsetByte2;
      BYTE WidthByte1;
      BYTE WidthByte2;
      BYTE HeightByte1;
      BYTE HeightByte2;
      BYTE PixelDepthByte;
      BYTE ImageDescriptorByte;
   }  TGAHEADER;

   ifstream infile1;
   ofstream outfile1;
   public:

   ascii2tga();
   ~ascii2tga();
};

////////////////////////////////////////////////////////////////////////////
// Function definition section.
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Constructor at the present is the area where the test functions are   //
// called.                                                               //
////////////////////////////////////////////////////////////////////////////

ascii2tga::ascii2tga(void)

{
   highByte = 0;
   lowByte = 0;
   integer1 = 0;
   rows = 0;
   columns = 0;
   Write_Banner();
   ContainerFunction();
}

////////////////////////////////////////////////////////////////
// Destructor cleans up and shuts down.                       //
////////////////////////////////////////////////////////////////

ascii2tga::~ascii2tga(void)
{
   infile1.close();
   outfile1.close();
   if(raw_array) de_allocate(raw_array, rows);
// if(processed_array) de_allocate2 (processed_array, rows);
// if(mirror_array) de_allocate2 (mirror_array, rows);
   cout<<endl<<"Press any key to continue"<<endl;
   getch();
   cout<<endl<<"Done"<<endl;
}

////////////////////////////////////////////////////////////////
// This function opens a file for output in binary mode.    //
////////////////////////////////////////////////////////////////

void ascii2tga::OpenForOutput(ofstream& otherFile, char outfileBuffer[])
{

LABL3:   cout<<"\n\nInput the name of the output TGA file\n\n";
         cout<<"Please type in the complete file name including";
         cout<<" the directory path\n";
         cout<<"for example: a:\\outfile.tga\n\n";
         cin.get(outfileBuffer,40);
         cin.ignore(100, '\n');
```

```
         otherFile.open(outfileBuffer, ios::binary);
         if(!otherFile)
         {
            cout<<"**Cannot open"<<outfileBuffer<<"**"<<endl;
            cout<<"Please try again."<<endl<<endl;
            goto LABL3;
         }
}

////////////////////////////////////////////////////////////////
// This function opens the ascii file for input in binary mode.//
// After checking that the file has opened OK, it closes.   //
// The file must be opened and closed as needed using          //
// 'infileBuffer[]'.                                          //
////////////////////////////////////////////////////////////////

void ascii2tga::OpenForInput( ifstream& someFile, char somefileBuffer[])
{

LABL3:   cout<<"\n\nInput the ASCII file.\n\n";

         cout<<"Please type in the complete file name including";
         cout<<"the directory path\n";
         cout<<"for example: c:\\infile.txt\n\n";
         cin.get(infileBuffer,40);
         cin.ignore(100, '\n');
         cin.get(infileBuffer,40);  //extra read to work
         cin.ignore(100, '\n');        //around console bug.
         someFile.open(somefileBuffer, ios::binary);
         if(!someFile)
         {
            cout<<"**Cannot open"<<somefileBuffer<<"**"<<endl;
            cout<<"Please try again."<<endl<<endl;
            goto LABL3;
         }
}

////////////////////////////////////////////////////////////////////
// Function allocates integer arrays for row X column sized arrays.  //
////////////////////////////////////////////////////////////////////

void ascii2tga::allocate( unsigned char **tempArray, int rows, int columns)
{
   for (int j = 0; j <= rows; j++)
      tempArray[j] = new unsigned char[columns];
}


////////////////////////////////////////////////////////////////////
// Function de-allocates integer arrays size row X column.           //
////////////////////////////////////////////////////////////////////

void ascii2tga::de_allocate(unsigned char **tempArray, int rows)
{
   for (int i = 0; i <= (rows);  i++)
       delete[] tempArray[i];
   delete[] tempArray;
}

////////////////////////////////////////////////////////////////////////
// Function writes the greeting banner.                                //
////////////////////////////////////////////////////////////////////////

void ascii2tga::Write_Banner(void)
{
   cout<<"  ***********************************************"<<endl;
   cout<<"  *                                             *"<<endl;
   cout<<"  *           Welcome to SDTSTGA2.              *"<<endl;
   cout<<"  *                                             *"<<endl;
   cout<<"  *   This program copyright 2003 John Childs   *"<<endl;
   cout<<"  *            Granby, CT USA                   *"<<endl;
```

```
   cout<<"  *          www.terrainmap.com                  *"<<endl;
   cout<<"  *                                              *"<<endl;
   cout<<"  *   The program accepts as input an ASCII text  *"<<endl;
   cout<<"  *   file and converts them to                  *"<<endl;
   cout<<"        *   8-bit TGA files suitable for input to       *"<<endl;
   cout<<"        *   many GIS programs.                          *"<<endl;
   cout<<"  *                                              *"<<endl;
   cout<<"  ************************************************"<<endl<<endl;
   cout<<"  Press any key to continue"<<endl;
   getch();
}


//////////////////////////////////////////////////////////////////////////
// Function the contents of a file into an array                        //
//////////////////////////////////////////////////////////////////////////

void ascii2tga::readArray( ifstream& inputFile, unsigned char** inputArray,
                           int rows, int columns)
{
   int temp;
   for(int i=0; i<rows; i++)
   {
      for(int j=0; j<columns; j++)
      {
         inputFile>>temp;
         inputArray[i][j]=(unsigned char)temp;;
      }
   }
}


//////////////////////////////////////////////////////////////////////////
// Function gets the rows from the console.                             //
//                                                                      //
//////////////////////////////////////////////////////////////////////////

int ascii2tga::getRows(void)
{

   int rows;
   cout<<"\n\nInput the rows\n";
    cin>>rows;
   return(rows);

}

//////////////////////////////////////////////////////////////////////////
// Function gets the columns from the console.                          //
//                                                                      //
//////////////////////////////////////////////////////////////////////////

int ascii2tga::getColumns(void)
{

   int columns;
   cout<<"\n\nInput the columns\n";
   cin>>columns;
   return(columns);
}

//////////////////////////////////////////////////////////////////////////
// Function sets up the arrays.                                         //
//////////////////////////////////////////////////////////////////////////

void ascii2tga::setup_array(int number_rows, int number_columns)
{
   raw_array = new unsigned char*[number_rows+1];
   allocate(raw_array, number_rows, number_columns);
// processed_array = new  unsigned char*[number_rows+1];
// allocate2(processed_array, number_rows, number_columns);
```

```
// mirror_array = new  unsigned char*[number_rows+1];
// allocate2(mirror_array, number_rows, number_columns);
}



////////////////////////////////////////////////////////////////
// This function writes a header section of the file.         //
////////////////////////////////////////////////////////////////

void ascii2tga::WriteHeaderData(ofstream& outfile1, int r, int c)
{
   TGAHEADER Header =
   {
      0x00,
      0x01,
      0x01,
      0x00,
      0x00,
      0x00,
      0x01,
      0x24,
      0x00,
      0x00,
      0x00,
      0x00,
      0x00,
      0x00,
      0x00,
      0x00,
      0x08,
      0x00
   };

   Header.WidthByte1 = (unsigned char)(c & 0x00ff);
   Header.WidthByte2 = (unsigned char)((c & 0xff00) >> 8);
   Header.HeightByte1 = (unsigned char)(r & 0x00ff);
   Header.HeightByte2 = (unsigned char)((r & 0xff00) >> 8);

   outfile1<<Header.IDLengthByte;      //start eighteen bytes of header data
   outfile1<<Header.ColorMapTypeByte;
   outfile1<<Header.ImageTypeByte;
   outfile1<<Header.CMapStartByte1;
   outfile1<<Header.CMapStartByte2;
   outfile1<<Header.CMapLengthByte1;
   outfile1<<Header.CMapLengthByte2;
   outfile1<<Header.CMapDepthByte;
   outfile1<<Header.XOffsetByte1;
   outfile1<<Header.XOffsetByte2;
   outfile1<<Header.YOffsetByte1;
   outfile1<<Header.YOffsetByte2;
   outfile1<<Header.WidthByte1;
   outfile1<<Header.WidthByte2;
   outfile1<<Header.HeightByte1;
   outfile1<<Header.HeightByte2;
   outfile1<<Header.PixelDepthByte;
   outfile1<<Header.ImageDescriptorByte;


   for(int i=0; i<=255; i++)  // Palette section of 256 ordered triples
   {
      char temp = (char)(i & 0x00ff);
      outfile1.put(temp);
      outfile1.put(temp);
      outfile1.put(temp);
   }
}
////////////////////////////////////////////////////////////////
// This function maps a 16-bit unsigned integer (0-65,536)    //
// to 8-bit TGA format.                                       //
//                                                            //
```

```
// Note: a glitch occurs when a pixel equals the minimum    //
// elevation value in the dataset.  This was fixed by          //
// detecting these values and incrementing them from 0x00 to   //
// 0x01.  The function also writes the processed data to the   //
// output file.                                              //
////////////////////////////////////////////////////////////////

void ascii2tga::Write_to_File(ofstream& outfile1,
                              int r, int c,
                              unsigned char**some_array)
{

    for (int i=0; i<r; i++)
      {
         for(int j=0; j<c; j++){
            outfile1.put(some_array[i][j]);
         }
      }
}


////////////////////////////////////////////////////////////////
// This function contains all the calls.                    //
////////////////////////////////////////////////////////////////

void ascii2tga::ContainerFunction(void)
{

   rows=getRows();
   columns=getColumns();
    OpenForInput( infile1, infileBuffer);
   setup_array(rows, columns);
   readArray(infile1, raw_array, rows, columns);
   infile1.close();
   infile1.close();
   OpenForOutput(outfile1, outfileBuffer);
   WriteHeaderData(outfile1, rows, columns);
   cout<<"rows is:  "<<rows<<endl;
   cout<<"columns is:  "<<columns<<endl;
   Write_to_File(outfile1, rows, columns, raw_array);
   outfile1.close();

}

////////////////////////////////////////////////////////////////
// Main part of the program.                                  //
////////////////////////////////////////////////////////////////

int main()
{
   ascii2tga Mydem;
   return(0);
}
```