MASSACHUSETTS INSTITUTE OF TECHNOLOGY

DEPARTMENT OF MECHANICAL ENGINEERING

COURSE: 18.086

Mathematical Methods for Engineers II

Project 1

Experimental Analysis of the Two Dimensional Laplacian Matrix (K2D): Applications and Reordering Algorithms

Project 1
Course: 18.086

Experimental Analysis of the Two Dimensional Laplacian Matrix (K2D): Applications and Reordering Algorithms

Section 1: Introduction

This project focuses on the analysis and experimentation of the two dimensional (2D) Laplacian matrix (K2D). Special emphasis is put on solving large systems, particularly for problems related to digital image processing and optical imaging. We discuss a typical application of the K2D matrix in finding the edges of an image. In the forward problem, $F = K2D \cdot U$, the input is the image we want to process by a Laplacian based edge detection algorithm. As will be discussed in the next section, the input image is preprocessed by a raster scanning algorithm to form the vector *U*. The output vector *F* is also processed by the inverse raster scanning algorithm to form the 2D Laplacian of the image. This new matrix is used in the edge detection algorithm to find all the edges.

In the inverse problem, $U = K2D^{-1} \cdot F$, the Laplacian of the image is the input and we try to recover the original object. This is the case for certain optical systems as discussed in Section 2. In order to solve this problem efficiently, we discuss the properties of K2D and experiment several ways to speed up the elimination and also reduce the storage during computations.

In Section 3, we analyze three popular reordering algorithms: minimum degree, red-black and graph separator orderings. We developed several experiments to explore their behavior under conditions such as variable matrix sizes. In addition, experiments are developed to estimate their required computational time and efficiency under LU and Cholesky decompositions.

Section 2: Application of the Laplacian Matrix in Digital Image Processing

In this section, the implementation of the two dimensional (2D) Laplacian Matrix (**K2D**) on a digital image processing problem is discussed. In this problem, the second derivative of the image (i.e. the Laplacian) is used to find discontinuities by implementing an **edge detection algorithm**. Edge detection algorithms are widely used in applications such as object detection and recognition, shape measurement and profilometry, image analysis, digital holographic imaging and robotic vision.

Edges in images appear as regions with strong intensity variations. In the case of images obtained with a conventional camera, edges typically represent the contour and/or morphology of the object(s) contained in the field of view (FOV) of the imaging system. From an optics perspective, edges represent the high spatial frequency information of the scattered wave coming from an illuminated object. If an object contains a sharp discontinuity, the information for this region will be mapped in a region of high frequency in the Fourier plane. Edge detecting an image is very important as it significantly reduces the amount of data and filters out useless information, while preserving structural properties of the image [1].

There are two main categories of edge detection algorithms: gradient and Laplacian based algorithms. In a typical gradient based algorithm, the first derivative of the image is computed in both dimensions (row and column wise). In this new image, the edges become apparent and the maximum and minimum gradients of the image are compared to a threshold. If there is a maximum with a value larger than the threshold, the spatial coordinate of that maximum is considered an edge. An example of gradient based algorithms is the **Sobel** edge detection algorithm. Figure 2.1 shows a 1D discontinuity centered at x=0. Figure 2.2 is a plot of the first derivative of the intensity function of Figure 2.1. It is important to note that the maximum is also centered at x=0. If we set the threshold of the edge detection algorithm equal to 0.2, an edge would be detected at x=0.

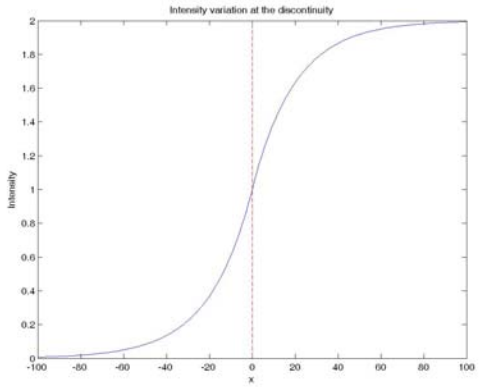

Figure 2.1: Example of 1D edge in an image

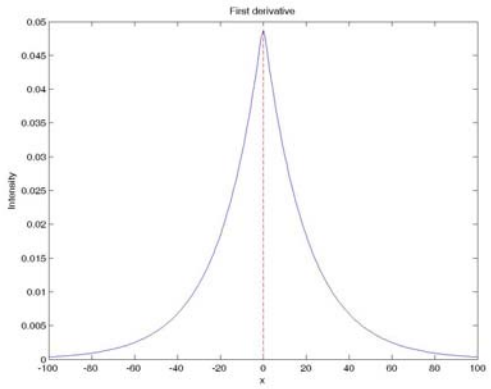

Figure 2.2: First derivative of the 1D edge of Figure 2.1.

For a Laplacian based algorithm, the second derivative of the original image is computed. The zero crossings of the resulted matrix are used to find the edges. Figure 2.3 shows the second derivative computed for the 1D example discussed above. The Matlab code used to generate Figures 2.1-2.3 is included in Appendix A.1.

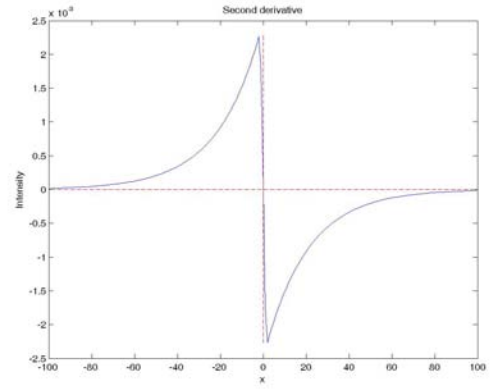

Figure 2.3: Second derivative of the 1D edge of Figure 2.1.

In the 2D case, the Laplacian is computed in both dimensions (row and column wise). This can be achieved by using the matrix K2D in the forward problem:

$$K2D \cdot U = F\,,$$

where U is a vector formed after raster scanning the original image. The measurement vector F is the raster scanned version of the 2D Laplacian of the image. The matrix K2D is an $N^2 \times N^2$ matrix formed by the addition of the Kronecker tensor products

$$K2D = kron(K, I) + kron(I, K).$$

Here, I is the $N \times N$ identity matrix and K is also $N \times N$ and tridiagonal of the form:

$$K = \begin{bmatrix} 2 & -1 & 0 & \cdots 0 \\ -1 & 2 & -1 & \cdots 0 \\ \vdots & \ddots & \ddots & \ddots \\ 0 & \cdots & -1 & 2 \end{bmatrix}.$$

To exemplify the 2D forward problem, consider the $500 \times 500\, pixels$ image of Figure 2.4. To form the vector *U*, we need to implement a raster scanning algorithm. In the raster scanning algorithm, the $N \times N$ matrix (i.e. the image) is decomposed into a vector of size N^2. This is achieved by extracting each column of the original matrix and placing it at the bottom of the vector. For example, a $3 \times 3\, pixels$ image would produce

$$\widehat{U} = \begin{bmatrix} u_1 & u_2 & u_3 \\ u_4 & u_5 & u_6 \\ u_7 & u_8 & u_9 \end{bmatrix} \rightarrow \begin{matrix} raster \\ scanning \end{matrix} \rightarrow U = \begin{bmatrix} u_1 \\ u_4 \\ u_7 \\ u_2 \\ u_5 \\ u_8 \\ u_3 \\ u_6 \\ u_9 \end{bmatrix}.$$

Figure 2.4: Photograph of Prof. Gilbert Strang ($500 \times 500\, pixels$).

The forward problem is solved by multiplying the *K2D* matrix, which in this case is 250000×250000, with the vector *U*. Since both the vectors and the matrix are large, it is necessary to implement the 'sparse' function available in Matlab. The 'sparse' function reduces the required storage by only considering the coordinates of the nonzero elements in the sparse matrix. The resultant matrix is obtained by the implementation of the inverse-raster scanning algorithm. Figure 2.5 shows the 2D Laplacian obtained after

solving the forward problem for the image of Figure 2.4. The code used to generate Figure 2.5 is included in Appendix A.2.

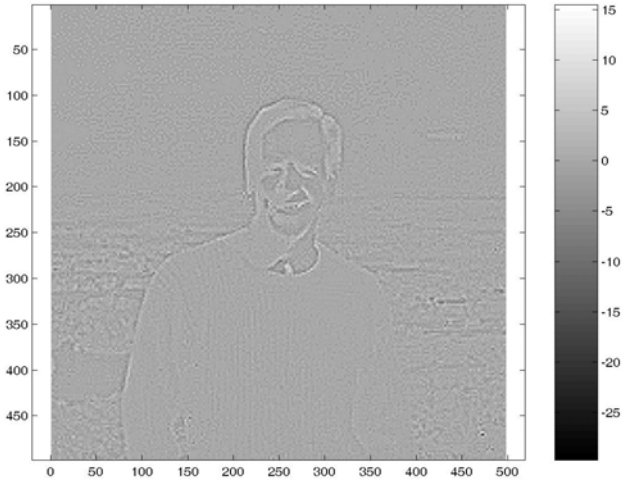

Figure 2.5: 2D Laplacian of the image of Figure 2.4.

For a Laplacian based algorithm, the information contained in Figure 2.5 is used to find the edges of the image. The result of this algorithm is a binary image such as the one shown in Figure 2.6. For more information about edge detection algorithms refer to [2].

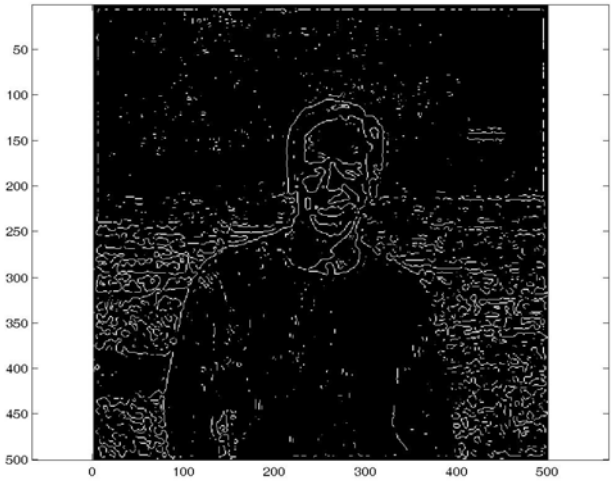

Figure 2.6: Result of applying the Laplacian edge detection algorithm on Figure 2.4.

Now we discuss the inverse problem. In the inverse problem, we are given the Laplacian of the image and our task is to find the original image

$$U = K2D^{-1}F\,.$$

This is a common problem in communications where we want to transmit as little information as possible. In an extreme case, we would like to transmit just the edges of the image (a sparse matrix) and from this recover the original image. Another example is an optical imaging system that has a high-pass Fourier filter in the Fourier plane, such as the one shown in Figure 2.7. In the optical configuration of Figure 2.7, an object is illuminated by a mutually-coherent monochromatic wave and the forward scatter light enters a 4f system. The 4f system is a combination of two positive lenses that are separated by a total distance of $d = f1 + f2$, where $f1$ and $f2$ are the focal lengths of both lenses. The first lens functions like an optical Fourier Transform operator. A high-pass Fourier filter is positioned in between both lenses (at the focal plane). This high-pass filter is designed to block the low spatial frequency content of the optical field and only let the high frequency information pass. The filtered spectrum is inverse Fourier transformed by the second lens and forms an image on a CCD detector. If the cut-off frequency is set relatively high (i.e. cutting a great part of the low frequency information), the edges of the object start being accentuated at the image plane.

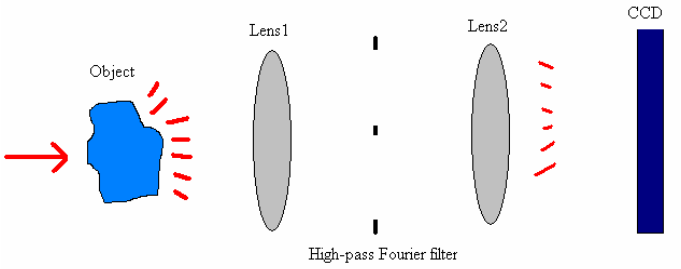

Figure 2.7: Optical realization of edge extraction

In the inverse problem, the input is an image similar to the one shown in Figure 2.5 and the output is another image like the one shown in Figure 2.4. The algorithm to compute this includes an elimination step (for the code included in Appendix A.3, the elimination is done using Matlab's '\' function) and a reordering. Different types of reordering such as minimum degree, red-black and nested dissection orderings will be analyzed and compared in Section 3. Choosing a suitable reordering algorithm is very important especially for large systems (for images with a high space-bandwidth product), as it reduces the number of fill-ins during elimination.

2.1 The K2D Matrix

The K2D matrix can be though of as the discrete Laplacian operator applied to vector U. As mentioned before, K2D can be formed from the addition of the Kronecker products of K and I. K2D has 4's in the main diagonal and -1's for the off-diagonal terms. For example, the 9×9 K2D matrix is given by

$$K2D = \begin{bmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{bmatrix}$$

The main characteristics of K2D is that it is symmetric and positive definite. From symmetry we know that $K2D = K2D^T$ and that all its eigenvalues are real and its eigenvectors are orthogonal. From positive defines we know that all the eigenvalues of K2D are positive and larger than zero; therefore, its determinant is larger than zero. The determinant for the 9×9 matrix shown above is $\det|K2D| = 100352$. The Gershgorin circle theorem applied to the matrix K2D indicates that all its eigenvalues should be contained inside a circle centered at 4 and with a radius of 4. Figure 2.8 shows the eigenvalues of the 9×9 K2D matrix in the complex plane. As predicted by the Gershgorin circle theorem, all the eigenvalues are between 0 and 8. The code to generate the plot of Figure 2.8 is included in Appendix A.4. Figure 2.9 shows the eigenvalues for a much larger K2D matrix (4900×4900). These 4900 eigenvalues also remain between 0 and 8.

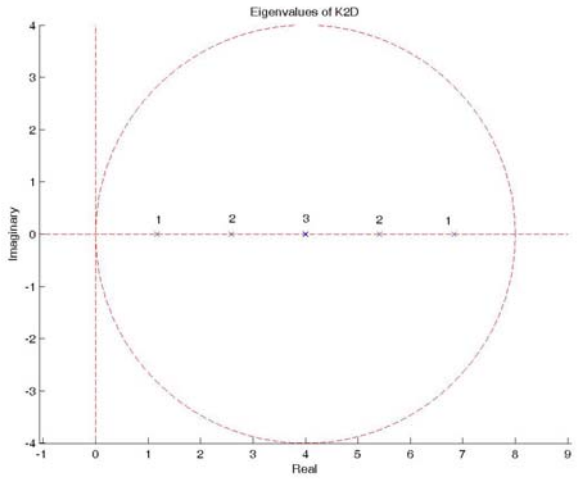

Figure 2.8: Eigenvalues of the 9×9 K2D matrix

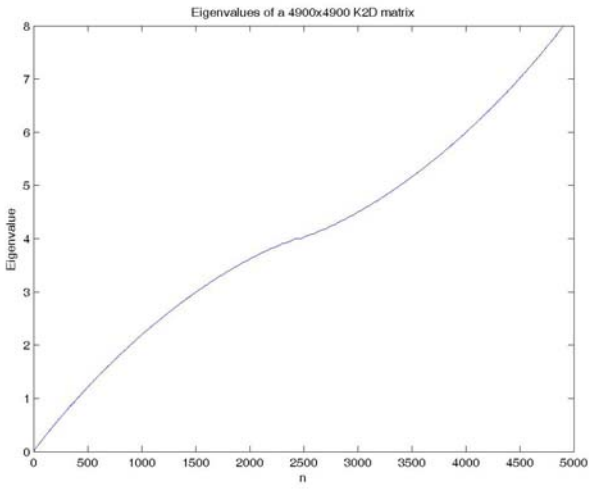

Figure 2.9: Eigenvalues of a 4900×4900 K2D matrix

Another interesting measure that describes the behavior of K2D is the condition number or in the context of information theory, the Rayleigh metric. The condition number is defined as

$$c = \frac{|\lambda_{\max}|}{|\lambda_{\min}|},$$

and gives information about how close the matrix is to ill-posedness. If c = 1, the matrix is said to be **well-posed**, but if the condition number is large ($c \to \infty$), the matrix gets close to becoming singular. In other words, we cannot completely trust the results obtained from an **ill-posed** matrix. Figure 2.10 shows the condition number for K2D of different sizes. The code used to generate this figure is included in Appendix A.5.

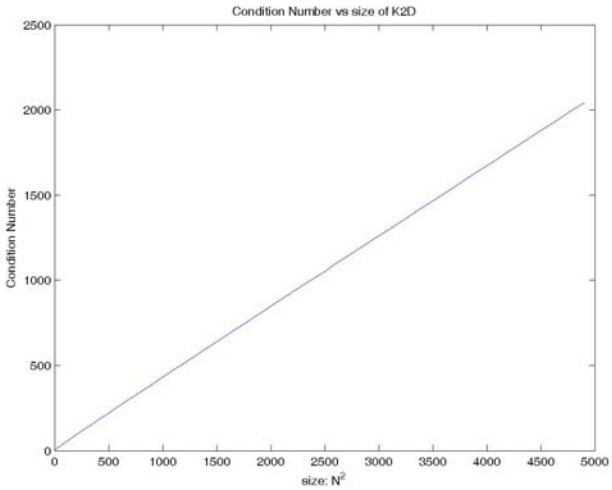

Figure 2.10: Condition number of K2D as a function of size

Section 3: Reordering Algorithms

In the previous section we described some of the properties of the K2D matrix along with related problems (forward and inverse problems) that are important for digital image processing. The main difficulty in solving such systems arises when the system is large. A large system would be required if we intend to process an image with a large number of pixels. If we do the elimination directly on K2D, several fill-ins would appear and the computational costs (memory and computational time) would increase. For this reason, in this section we discuss different reordering algorithms that help reduce the number of fill-ins during elimination.

3.1: Minimum Degree Algorithm

The minimum degree algorithms reorder *K2D* to form a new $\widehat{K2D} = P(K2D)P^T$ by eliminating the pivots that have the minimum degree. If we draw the graph corresponding to *K2D*, the minimum degree algorithm eliminates the edges (off-diagonal terms) from the nodes that have the minimum degree (i.e. the least number of edges connected to it). To describe the steps involved in this algorithm, we carry the reordering algorithm on the 9×9 K2D matrix described above. Figure 3.1 shows its corresponding graph. The steps of the algorithm are as follows:

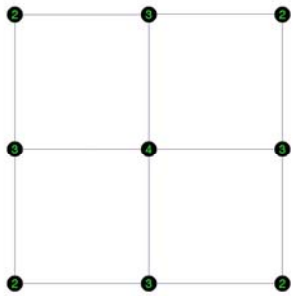

Figure 3.1: Graph corresponding to a 9×9 K2D matrix

1. We start by choosing the first element in the matrix as the pivot (in reality, you can choose any of the corner nodes of the graph, as all of them are degree 2). Using this pivot, carry out elimination as usual. This is equivalent to eliminating the edges connected to the pivot node. Two new fill-ins are generated in this step, and this is shown in the graph as a new edge as shown in Figure 3.2.
2. Update the permutation vector: $P = [1]$. The permutation vector keeps track of the order of the nodes we choose for elimination.
3. Go back to step one and choose one of the three remaining nodes that are degree 2. For example, if the next node we choose is node 3, after elimination, the permutation vector becomes: $P = [1 \quad 3]$.

Figure 3.3 shows the graph sequence for the remaining steps of the minimum degree algorithm. The final permutation vector is $P = [1 \quad 3 \quad 7 \quad 9 \quad 6 \quad 5 \quad 2 \quad 8 \quad 4]$. Figure 3.4

shows a comparison between the structures of the original and the reordered K2D matrices. Figure 3.5 shows the lower triangular matrices produced after LU decomposing the original and the reordered K2D matrices. As can be seen from this figure, reordering the matrix produces fewer numbers of nonzeros during the LU decomposition. The Matlab code used to generate Figures 3.1-3.5 was obtained at [3].

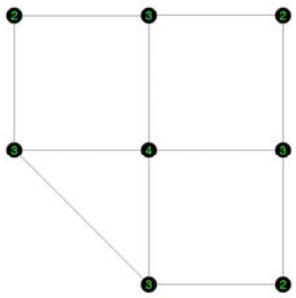

Figure 3.2: Graph of the 9×9 K2D matrix after step one of the minimum degree algorithm

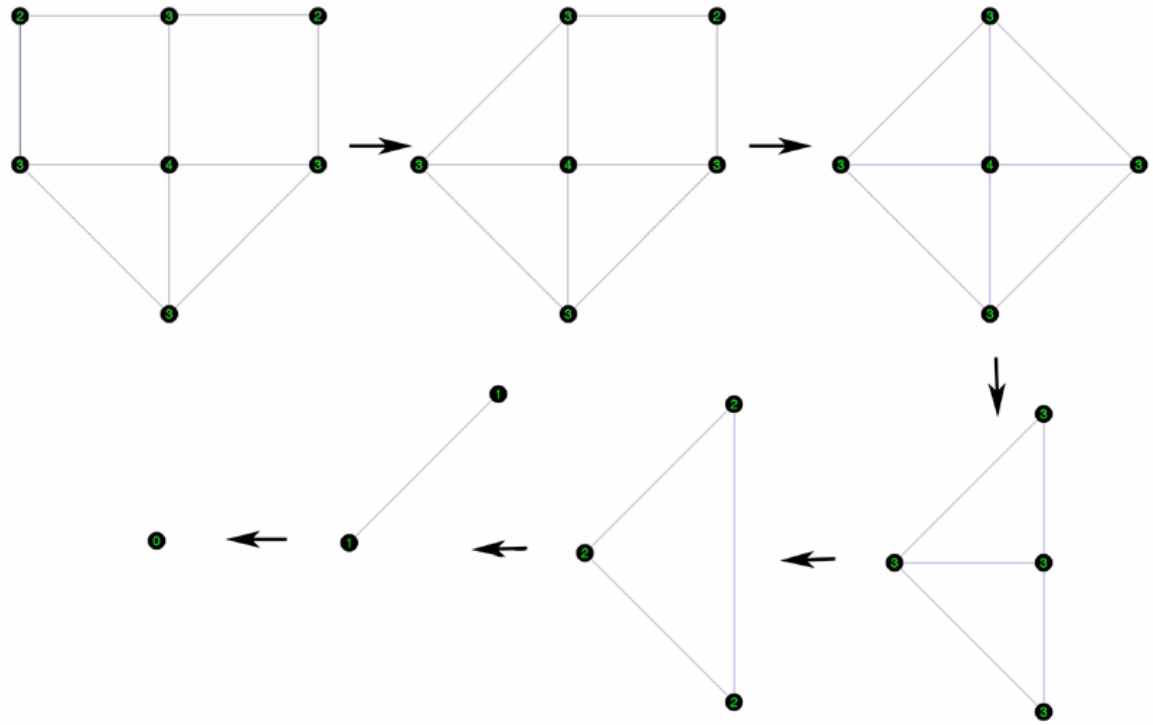

Figure 3.3: Sequence of graphs for the remaining steps in the minimum degree algorithm

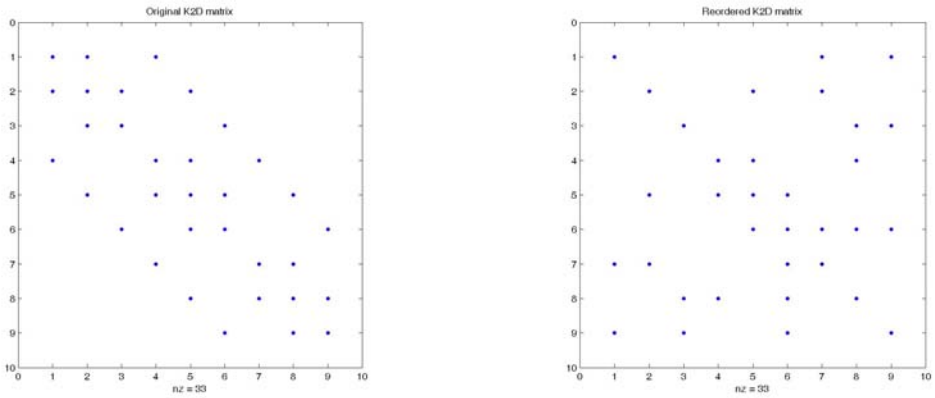

Figure 3.4: Structural comparison between the original and the reordered K2D matrices

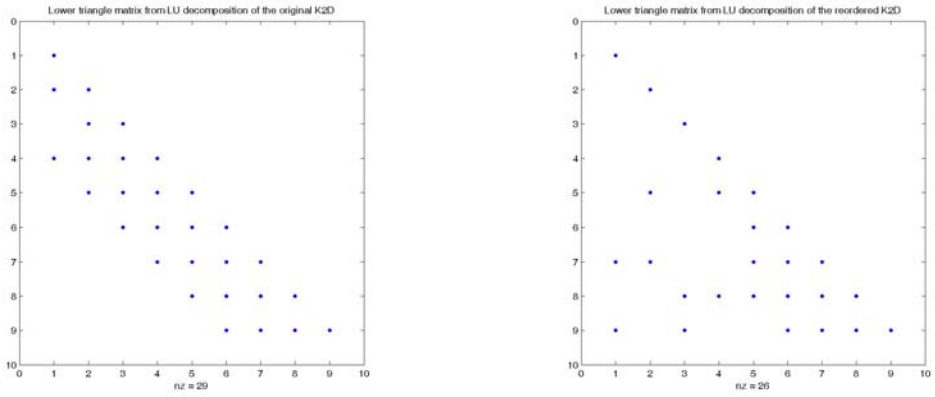

Figure 3.5: Lower triangular matrices produced after LU decomposition of the original (#nonzeros = 29) and the reordered (#nonzeros = 26) K2D matrices

Several minimum degree reordering algorithms are available. Each of these algorithms produces a different permutation vector. We now compare five of such algorithms:

1. Matlab's "**symamd**" algorithm: Symmetric approximate minimum degree permutation.
2. Matlab's "**symmmd**" algorithm: Symmetric minimum degree permutation.
3. Matlab's "**colamd**" algorithm: Column approximate minimum degree permutation.
4. Matlab's "**colmmd**" algorithm: Column minimum degree permutation.
5. Function "**realmmd**" algorithm [3]: Real minimum degree algorithm for symmetric matrices.

Figure 3.6 shows the structure of a 81×81 K2D matrix. Figure 3.7 shows a structural comparison of the reordered matrices using the five minimum degree algorithms mentioned above. Figure 3.8 shows their respective lower triangular matrix produced after LU decomposing the reordered K2D matrix. From this figure, it is evident that the real minimum degree algorithm produces the least number of nonzero elements (469) in

the decomposed lower triangular matrix. The code used to generate Figures 3.6-3.8 is included in Appendix B.1.

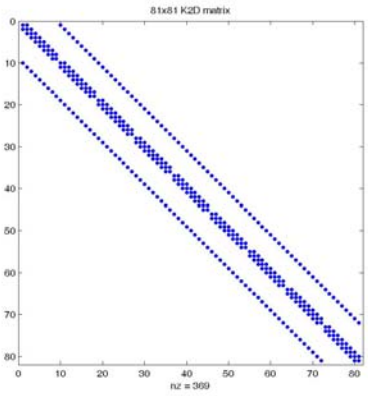

Figure 3.6: Structure of a 81×81 *K2D* matrix

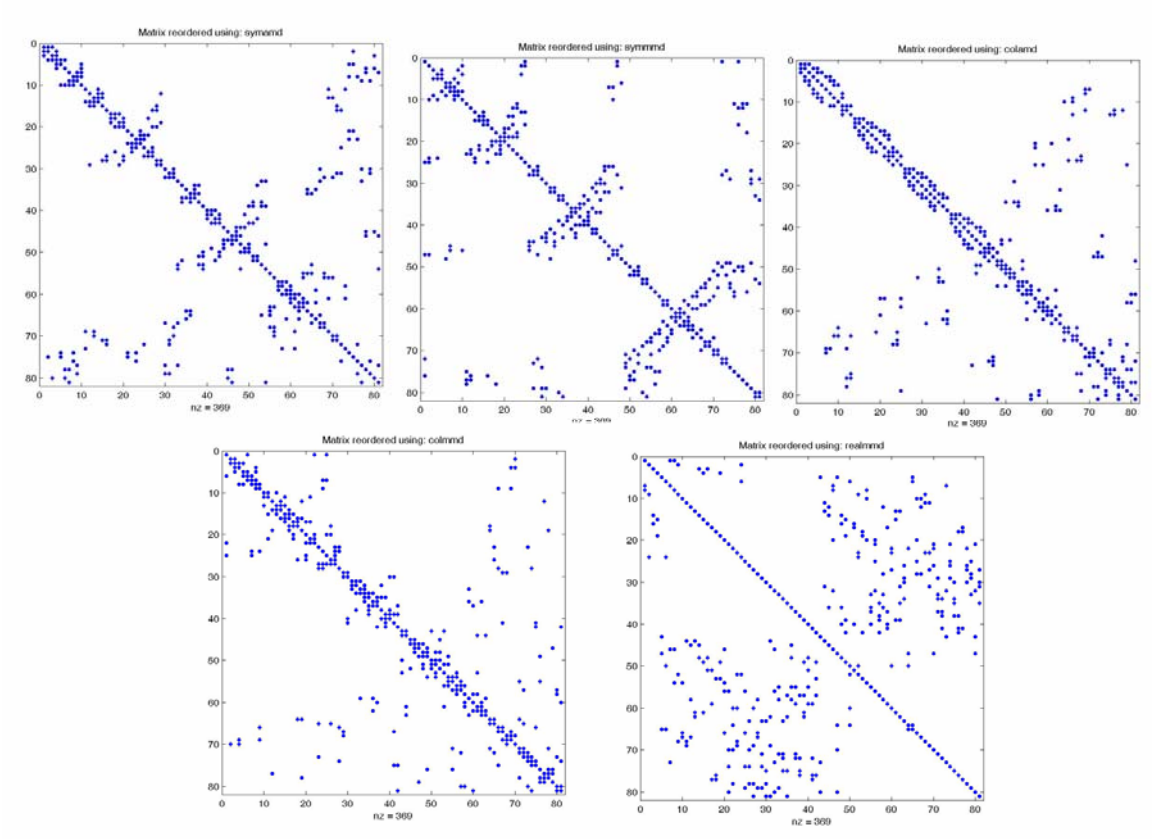

Figure 3.7: Structural comparison of reordered matrices using the five algorithms described above

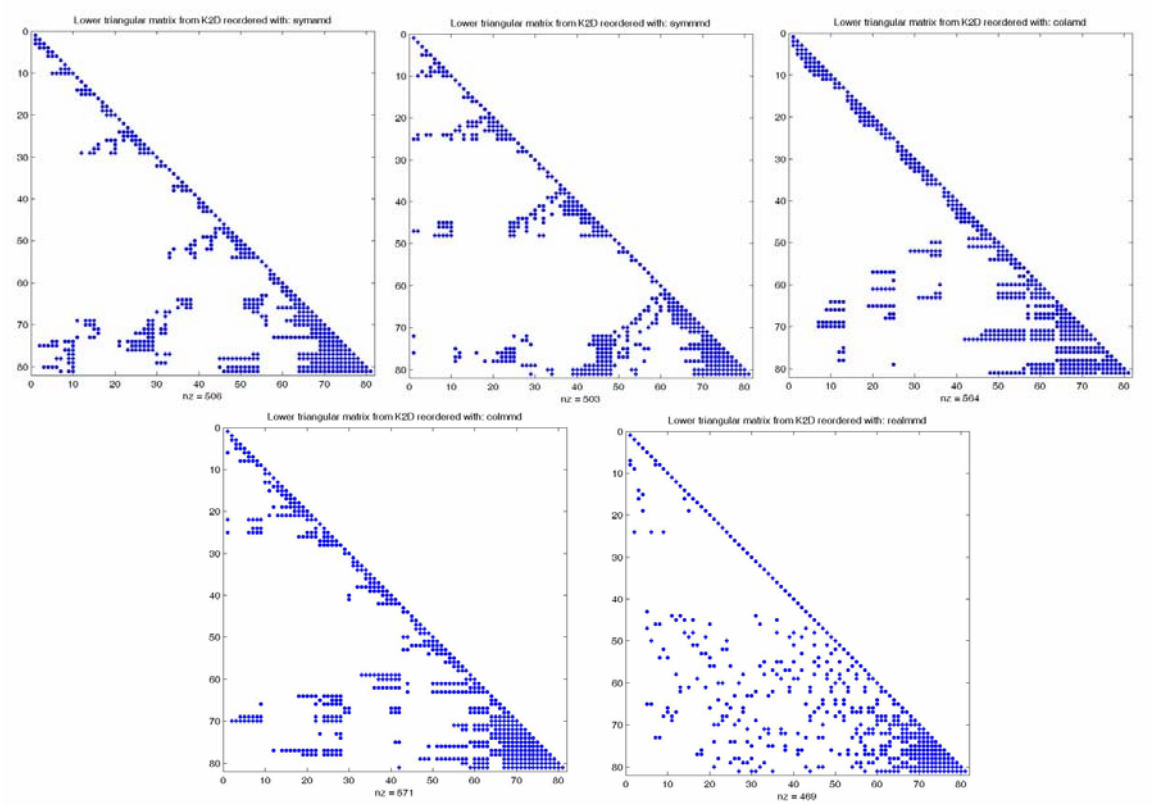

Figure 3.8: Structural comparison of the lower triangular matrices decomposed from the reordered matrices using the five algorithms.

Now we compare how the five minimum degree algorithms behave as a function of matrix size. In particular, we are interested to know the number of nonzeros in the lower triangular matrix, L, after LU decomposing the reordered K2D matrices for different matrix sizes. Figure 3.9 shows this comparison for matrices with sizes ranging from 4×4 to 3844×3844 entries. Figure 3.10 zooms into Figure 3.9 to show an interesting behavior of the 'realmmd' and the 'symamd' algorithms with relative small sized matrices. For small matrices, the 'realmmd' algorithm produces the least number of nonzeros in L; however, if the matrix size increases (especially for large matrices), the 'symamd' algorithm is the winner. The code used to generate Figure 3.9-3.10 is included in Appendix B.2.

In our next experiment, we compare four of these minimum degree methods for different matrix sizes after performing a Cholesky decomposition on the reordered matrix. The size of the matrices ranges from 4×4 to 120409×120409 in steps of 25. The results of this experiment are plotted in Figure 3.11. As in the previous experiment, we can see that the methods 'symamd' and 'symmmd' produce the least number of nonzeros entries in the factorized matrix. However, the question now is: are these methods (symamd and symmmd) fast to compute?

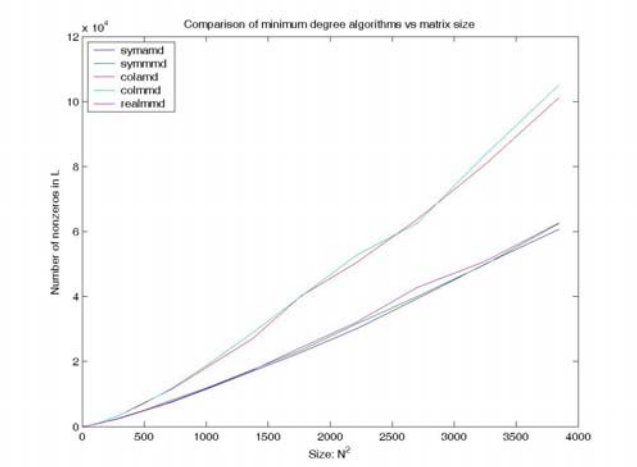

Figure 3.9: Comparison of the nonzeros generated by the five algorithms as a function of matrix size

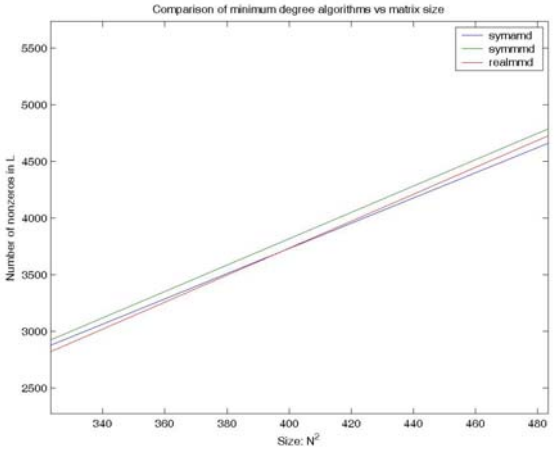

Figure 3.10: Zoomed in from Figure 3.9.

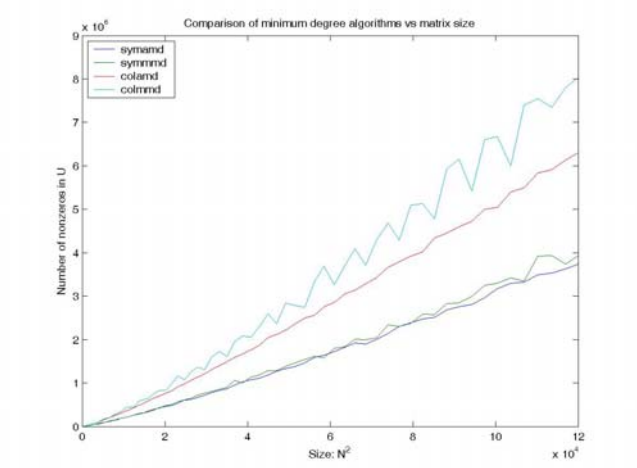

Figure 3.11: Nonzeros generated by the Cholesky factorized matrix, reordered with four different minimum degree algorithms

To answer the question stated above, we now turn to compare the required computational time for reordering K2D by these four minimum degree algorithms. The results are shown in Figure 3.12. From this figure we can see that although it is slightly faster to reorder a large matrix with 'colamd' rather than using 'symamd', the number of nonzeros generated in the decomposed matrix is significantly less for a matrix reordered by the 'sysmamd' algorithm. In conclusion, 'sysmamd' is our big winner.

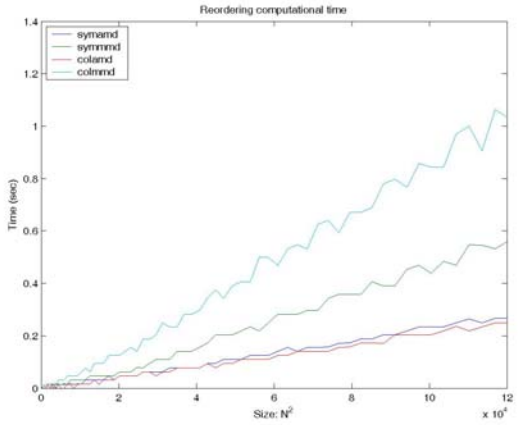

Figure 3.12: Computational time for reordering K2D of different sizes

The reason we didn't include the 'realmmd' algorithm in the comparison of Figure 3.13, is because this algorithm requires a long computational time (even for relatively small matrices) as shown in Figure 3.13.

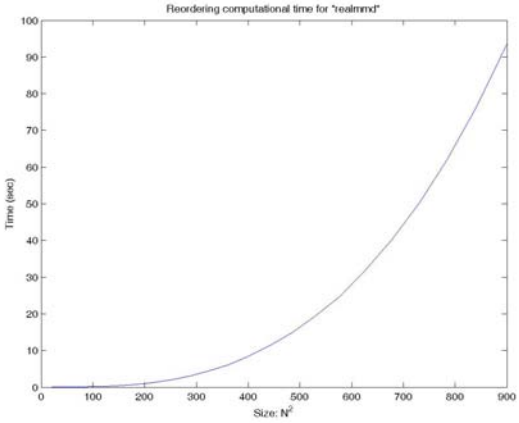

Figure 3.13: Computational time required by 'realmmd' as a function of matrix size

If we use Matlab's functions 'chol' and 'lu' to produce the Cholesky and LU decompositions respectively, it is fair to ask: what is the computational time required by both functions? Is the computational time dependent on the input matrix (i.e. if the matrix was generated by 'symamd', 'symmd', 'colamd' or 'colmmd')? How does the computational time behave as a function of the matrix size? To answer all of these questions, we generated the plots shown in Figures 3.14 and 3.15. The Matlab code used to generate Figures 3.11-3.15 is included in Appendix B.3.

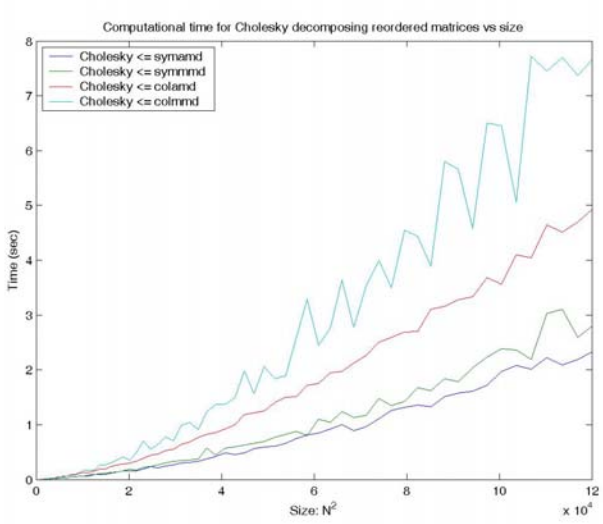

Figure 3.14: Computational time required by Matlab's Cholesky decomposition function 'chol'

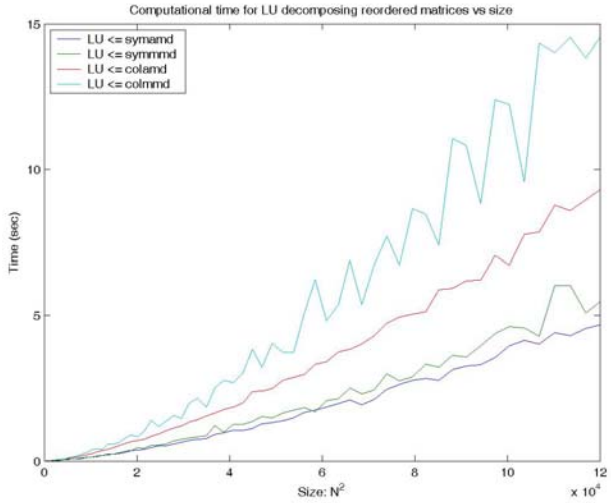

Figure 3.15: Computational time required by Matlab's LU decomposition function 'lu'

3.2: Red-Black Ordering Algorithm

The red-black ordering algorithm is an alternative reordering technique in which it is divided in a way similar to a checkerboard (red and black or odd and even squares). The permutation vector is simply generated by first selecting for elimination the odd nodes and then all the even nodes on the grid. The red-black permutation is given by

$$P(K2D)P^T = \begin{bmatrix} 4I_{red} & B \\ B^T & 4I_{red} \end{bmatrix},$$

where B is a matrix composed by -1s from the off-diagonal or black elements of K2D.

For example, Figure 3.16 shows a structural comparison between the original 9×9 K2D matrix and the its equivalent reordered using the red-black ordering algorithm. The permutation vector in this example is: $P=\begin{bmatrix}1 & 3 & 5 & 7 & 9 & 2 & 4 & 6 & 8\end{bmatrix}$. The reordered K2D matrix is given by

$$P(K2D)P^T=\begin{bmatrix}4 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0\\ 0 & 4 & 0 & 0 & 0 & -1 & 0 & -1 & 0\\ 0 & 0 & 4 & 0 & 0 & -1 & -1 & -1 & -1\\ 0 & 0 & 0 & 4 & 0 & 0 & -1 & 0 & -1\\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & -1 & -1\\ -1 & -1 & -1 & 0 & 0 & 4 & 0 & 0 & 0\\ -1 & 0 & -1 & -1 & 0 & 0 & 4 & 0 & 0\\ 0 & -1 & -1 & 0 & -1 & 0 & 0 & 4 & 0\\ 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 4\end{bmatrix}.$$

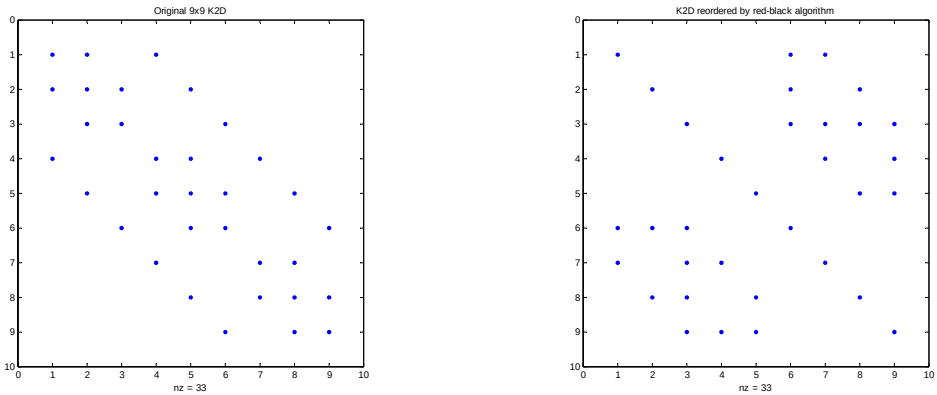

Figure 3.16: Structural comparison between the original K2D (to the left) and the reordered matrix using the red-black algorithm (to the right)

Figure 3.17 shows the sequence of eliminations that occur on the graph representation of the matrix described above. From this figure, we can see that the algorithm starts by eliminating all the edges of the odd/red nodes (remember that the graph is numbered row by row starting from the bottom left corner). After finishing with the odd nodes, the algorithm continues to eliminate the even nodes until all the edges have been eliminated. A comparison between the lower triangular matrices produced after elimination of the original and reorder matrices is shown in Figure 3.18. The reordered matrix produced less number of nonzero entries in L (27 nonzeros instead of 29). The Matlab code used to generate Figures 3.16-3.18 is included in Appendices B.4 and B.5. This code generates a movie of the sequence followed during elimination in the graph.

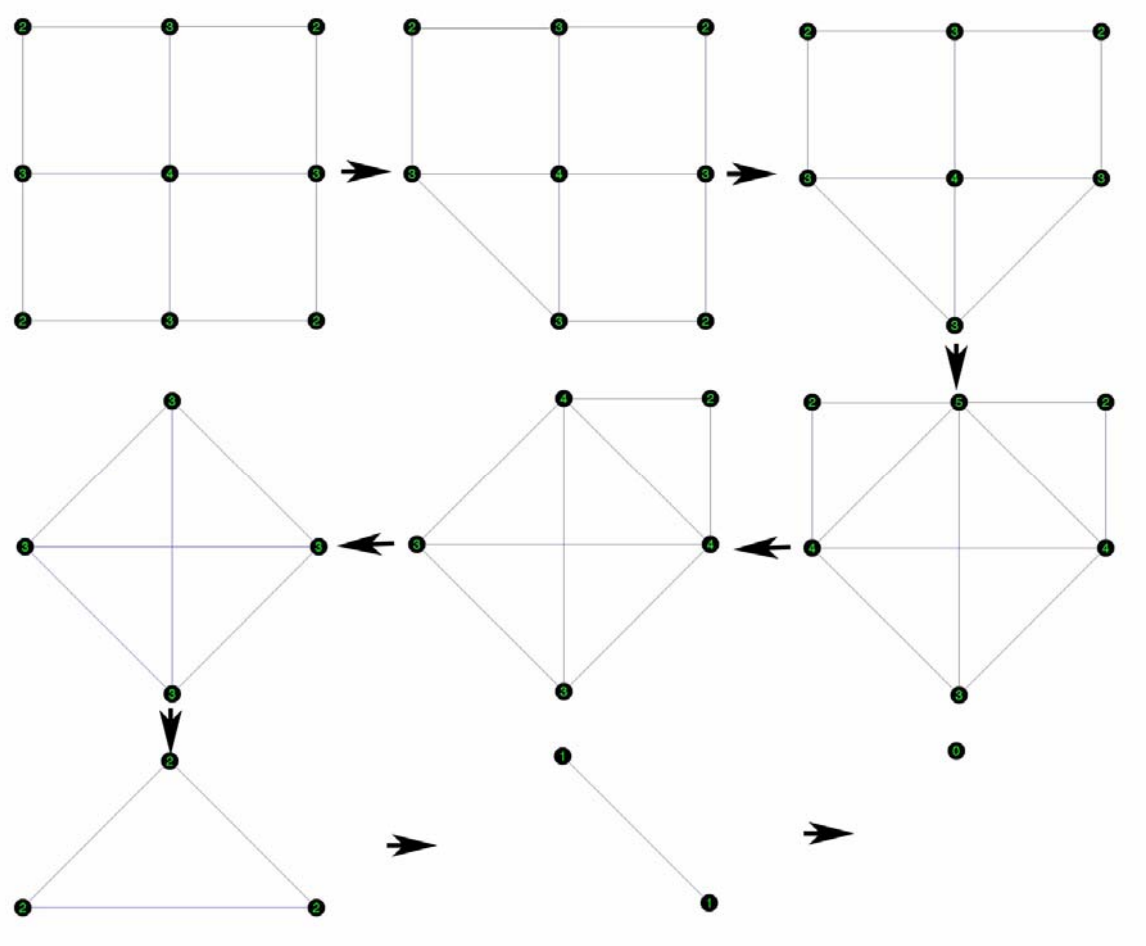

Figure 3.17: Sequence of eliminations for the red-black ordering algorithm

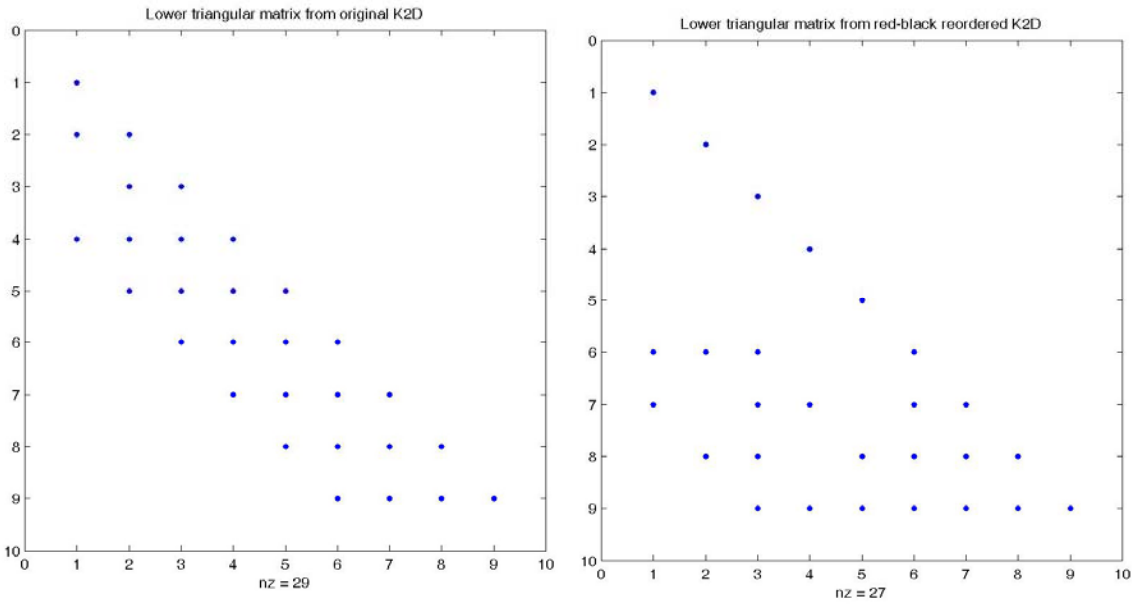

Figure 3.18: Comparison of lower triangular matrices produced after LU decomposing the original and the reordered K2D matrices

We now try to study the behavior of the red-black ordering algorithm under matrices of different sizes. Figure 3.19 shows a comparison between the original and the reordered matrices for the number of nonzeros generated in the lower triangular matrix during LU decomposition. Especially for large matrices, the red-black algorithm shows an important improvement due to the significant reduction of the number of nonzeros in L. From this figure we can see that the red-black algorithm performs worse than the minimum degree algorithms discussed earlier. However, the main advantage of the red-black algorithm resides in its simplicity and easy implementation.

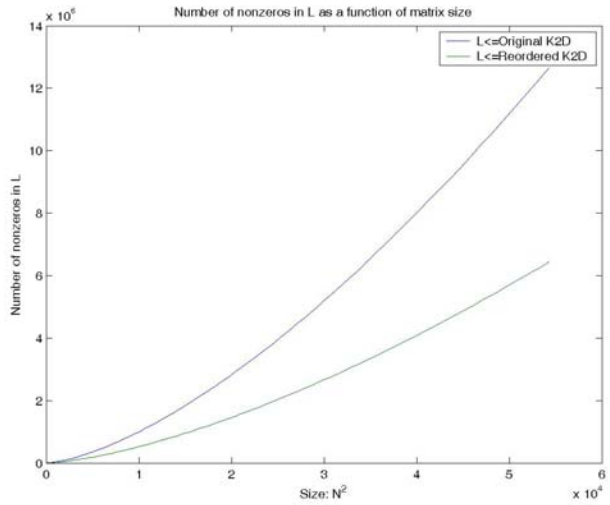

Figure 3.19: Number of nonzeros generated in L after LU decomposing the original and the reordered K2D matrices

Figure 3.20 shows the computational time required by the red-black algorithm as a function of matrix size.

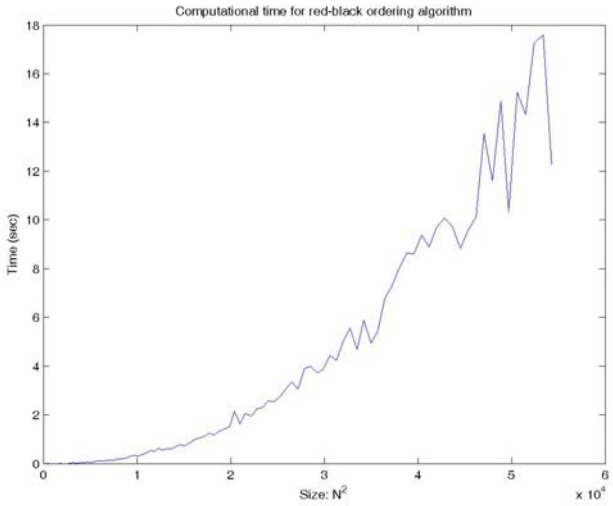

Figure 3.20: Computational time required by the red-black ordering algorithm as a function of matrix size

Finally, we want to compare the computational time required for the LU decomposition of the original and the reordered matrices as a function of matrix size. This is shown in Figure 3.21. As expected, the reordered matrix requires less time to produce the factors L and U. The Matlab code used to generate Figures 3.19-3.21 is included in Appendix B.6.

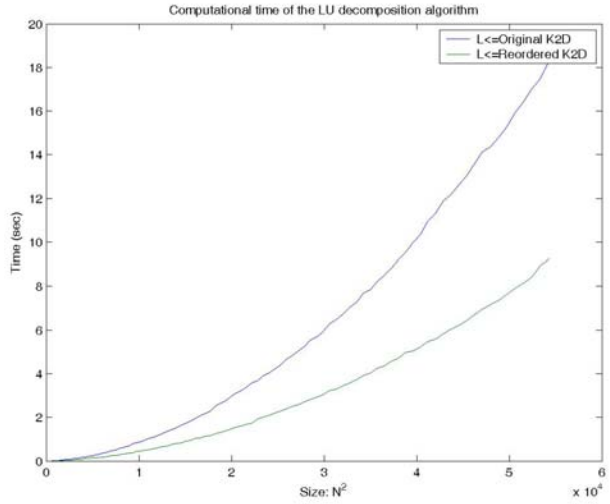

Figure 3.21: Computational time required by the LU decomposition of the original and the reordered matrices

3.3: Graph Separators Algorithm and Nested Dissection

In this subsection we discuss an alternative reordering algorithm called graph separators. Again, it is easier to understand the logic behind this algorithm by looking directly at the graph. The key idea is that a separator, S, is introduced to divide the graph in two parts P and Q (subsequent divisions of the graph would lead to nested dissection algorithms). The separator S is formed by a collection of nodes and it has typically a smaller or equal size than P and Q. To illustrate this algorithm, we go back to our 9×9 matrix. The graph of this matrix is shown in Figure 3.22. As shown in this figure, the graph is divided in two parts by S. In this case, P, Q and S have the same size. The graph was initially numbered row wise as before. In the graph separators algorithm, the graph is reordered starting from all nodes in P, then all nodes in Q and finally all nodes in S as shown in the same figure. For this example, the permutation vector becomes: $P=\begin{bmatrix}1 & 4 & 7 & 3 & 6 & 9 & 2 & 5 & 8\end{bmatrix}$. The elimination sequence is performed following this new order.

Figure 3.23 shows the structure of the reordered matrix and its corresponding lower triangular factor. The graph separator permutation is given by

$$P(K2D)P^T=\begin{bmatrix} K_P & 0 & K_{PS} \\ 0 & K_Q & K_{QS} \\ K_{SP} & K_{SQ} & K_S \end{bmatrix}.$$

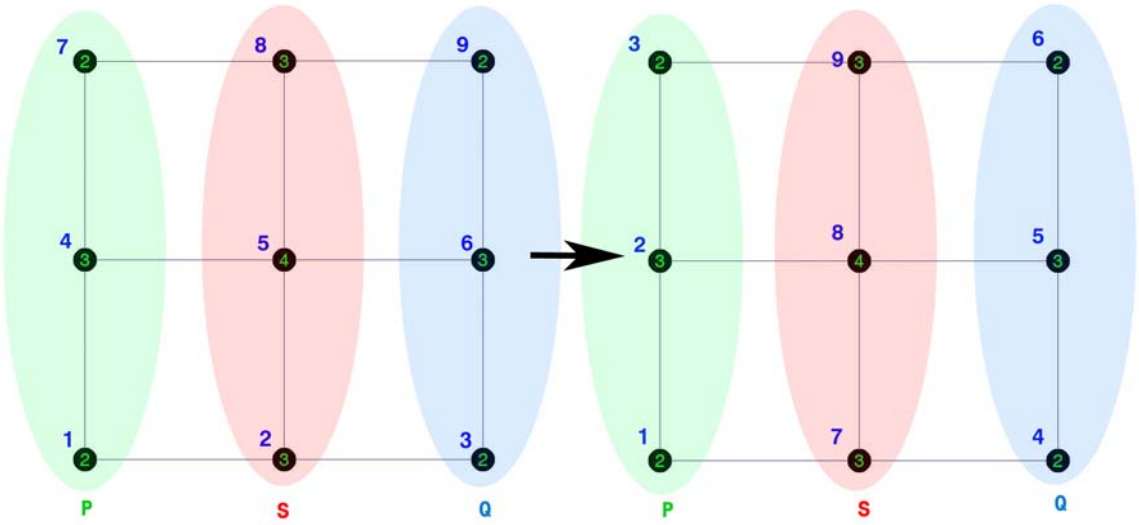

Figure 3.22: Reordering occurred in the graph separators algorithm

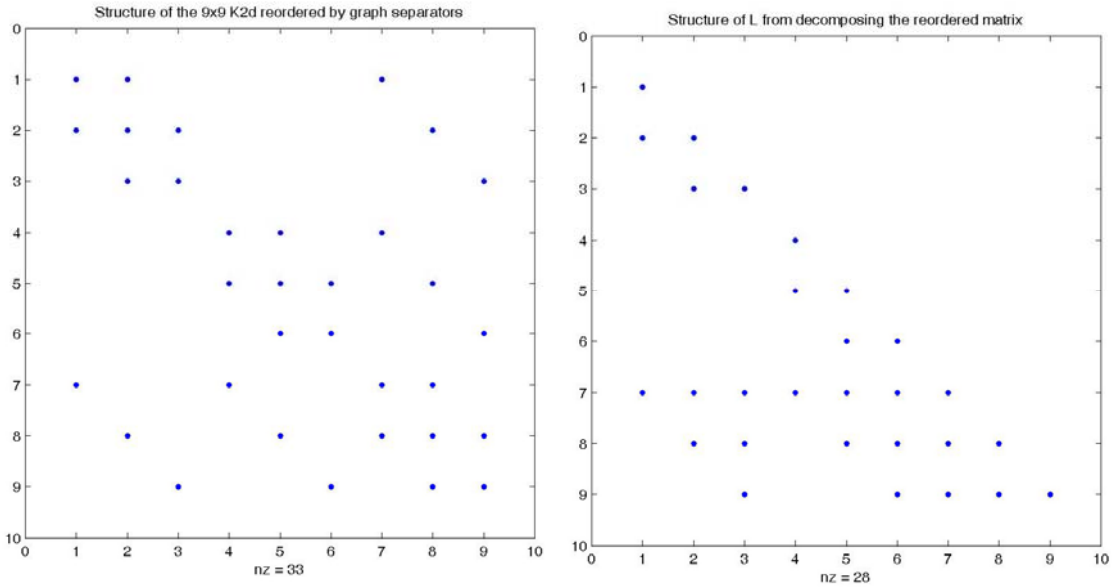

Figure 3.23: Structure of the reordered and the lower triangular factor of the 9×9 K2D matrices

As we mentioned before, the introduction of additional separators will produce a nested dissection algorithm. For the example described above, we can introduce a maximum of two additional separators as shown in Figure 3.24. With this ordering, the permutation vector becomes: $P = \begin{bmatrix} 1 & 7 & 4 & 3 & 9 & 6 & 2 & 5 & 8 \end{bmatrix}$. Figure 3.25 shows the new structure of the reordered matrix as well as the structure for L. The total number of nonzeros got reduced from 28 to 26 (remember that with out any reordering L has 29 nonzeros).

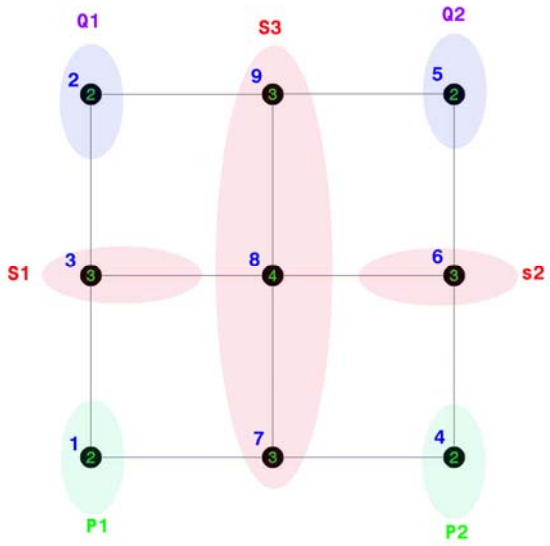

Figure 3.24: Nested dissection algorithm

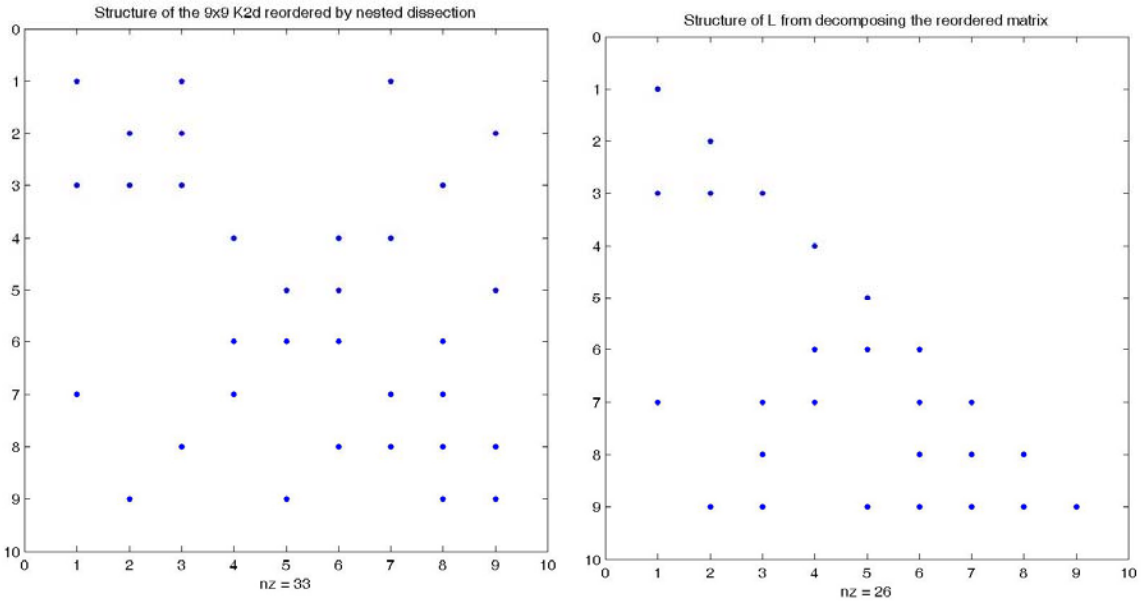

Figure 3.25: Structure of the reordered matrix and its lower triangular factor

We now turn to experiment with more sophisticated nested dissection algorithms available in **Matlab Mesh Partitioning and Graph Separator Toolbox** [4]. In particular, we will compare two algorithms:

1. "specnd": Spectral nested dissection ordering. This algorithm makes use of certain properties of the Laplacian matrix to compute the proper separators. For more information regarding spectral nested dissection algorithms refer to [7].
2. "gsnd": Geometric spectral nested dissection ordering.

Figure 3.26 compares the number of nonzero entries in L after LU decomposing the matrices reordered by both algorithms and the original K2D. From this figure we can see that the results produced by the spectral-based nested dissection algorithms get closer to those obtained by the minimum degree algorithms; however, some of the minimum degree algorithms such as the "symamd" algorithm perform much better.

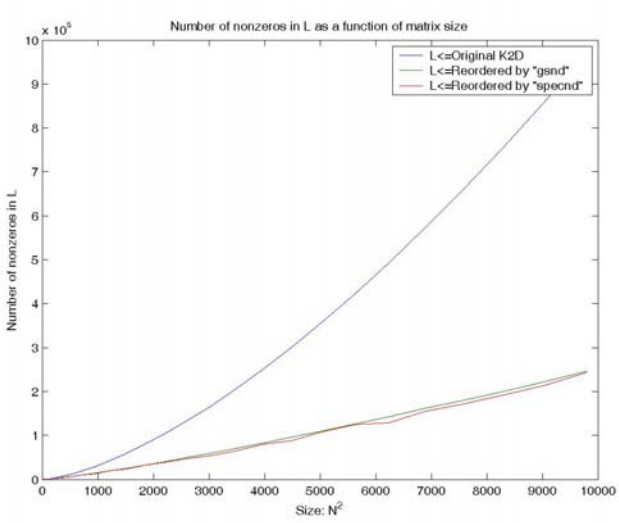

Figure 3.26: Number of nonzeros generated in L after LU decomposing the original and the reordered K2D matrices (by "gsnd" and "specnd")

A comparison of the computational time required by both algorithms is plotted in Figure 3.27. Again, we confirm that Matlab's minimum degree algorithms perform much faster on large matrices than the nested dissection algorithms.

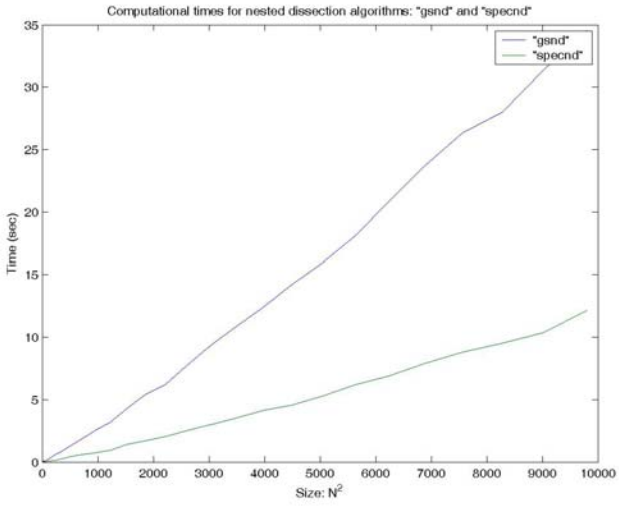

Figure 3.27: Computational time required by "gsnd" and "specnd" as a function of matrix size

Finally, we compare the computational time required by the LU factorization algorithm when the input matrix was reordered by both algorithms. The results are plotted in Figure 3.28. The Matlab codes used to generate Figures 3.26-3.28 is included in Appendix B.6.

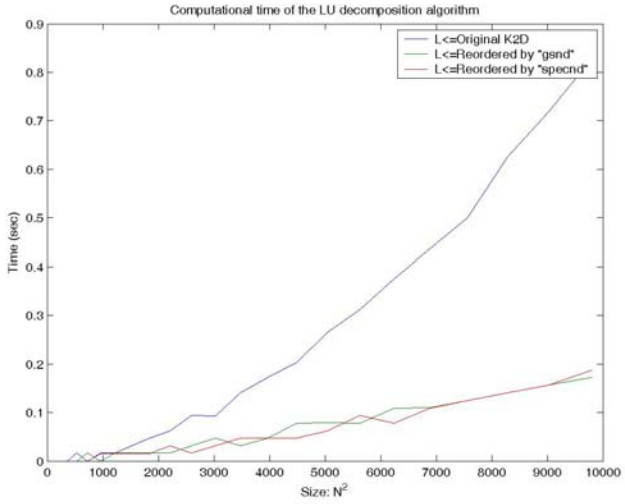

Figure 3.28: Computational time required by the LU decomposition of the original and the reordered matrices

Section 4: References

[1]: Internet access: http://www.pages.drexel.edu/~weg22/edge.html. Date accessed: 04/02/2006.

[2]: J. S. Lim, *Two-Dimensional Signal and Image Processing*, Prentice Hall, 1990.

[3]: Internet access: http://www.cerfacs.fr/algor/Softs/MESHPART/, *Matlab mesh partitioning and graph separator toolbox.* Date accessed: 04/04/06.

[4]: G. Strang, *Introduction to applied mathematics,* Wellesley, Cambridge Press.

[5]: A. Pothen, H.D. Simon, L. Wang, *Spectral Nested Dissection,* 1992.

Section 5: Appendices

A.1: This Appendix contains the Matlab code used to generate Figures 2.1-2.3.

```
%Computation of the first and second derivatives of an intensity function
%I(x) to be used to detect discontinuities using and edge detection
%algorithm

%General Parameters
m = 0:0.01:1;
K = 5;
x = -100:100;

%1D discontinuity
F1 = exp(-K*m);
F2 = -F1+2;
F = fliplr(F1);
```

```
F(length(F2)+1:length(F2)*2-1)=F2(2:end);

%Plot 1
figure;
plot(x,F,zeros(length(x),1)',[0:0.01:2],'--r',x,zeros(length(x),1)','--r')
title('Intensity variation at the discontinuity')
xlabel('x')
ylabel('Intensity')

%Gradient
delF = gradient(F);

%Plot 2
figure;
plot(x,delF,zeros(length(x),1)',[0: 2.4279e-004:0.0488],'--r',x,zeros(length(x),1)','--r')
title('First derivative')
xlabel('x')
ylabel('Intensity')

%Laplacian
del2F = gradient(gradient(F));

%Plot 2
figure;
plot(x,del2F,zeros(length(x),1)',[-0.0023:2.3e-005:0.0023],'--r',x,zeros(length(x),1)','--
r')
title('Second derivative')
xlabel('x')
ylabel('Intensity')
```

A.2: Matlab code used to generate Figure 2.4.

```
%Forward problem: given an image, find the Laplacian for edge detection

%General parameters
U = imread('g1','bmp');
U = double(U);
N = size(U,1);
h = 0.5;

%1D difference matrix
K = diag([2*ones(N,1)],0)+diag([-ones(N-1,1)],1)+diag([-ones(N-1,1)],-1);
K = sparse(K);
%Identity matrix
I = speye(N);
%2D Laplacian matrix
K2D = kron(K,I)+kron(I,K);

%Raster Scanning the image
count = 1;
for m = 1:N
    Ur(count:count+N-1,1) = U(:,m);
    count = count+N;
end

%Forward problem
F = K2D*Ur;

%Inverse-raster scanning for F:
count = 1;
for m = 1:N
    Fnew(:,m) = F(count:count+N-1,1);
    count = count+N;
end
Fnew = Fnew(2:499,2:499);
Fnew = -Fnew*h^2;
figure; imagesc(Fnew);
colorbar
colormap gray
axis equal

%Note: alternative solution is obtained using Matlab's del2 function:
% Fnew = del2(U);
```

A.3: Inverse Problem code

```
%Inverse Problem: Given the Laplacian of an image, find original image

%General parameters
```

```
F = imread('g1','bmp');
F = double(F);
N = size(F,1);
h = 0.5;

%Computing the 2D Laplacian
F = del2(F);

%1D difference matrix
K = diag([2*ones(N,1)],0)+diag([-ones(N-1,1)],1)+diag([-ones(N-1,1)],-1);
K = sparse(K);

%Identity matrix
I = speye(N);

%2D Laplacian matrix
K2D = kron(K,I)+kron(I,K);

%Raster Scanning Measurement:
count = 1;
for m = 1:N
    Fr(count:count+N-1,1) = F(:,m);
    count = count+N;
end

%Elimination
U = K2D\Fr;

%Undo raster scanning for U:
count = 1;
for m = 1:N
    Unew(:,m) = U(count:count+N-1,1);
    count = count+N;
end
Unew = -Unew/h^2;
figure; imagesc(Unew);
colorbar
colormap gray
axis equal
```

A.4: Plots the eigenvalues of a 9×9 K2D matrix

```
%Code to plot the eigenvalues of a 9X9 K2D matrix inside Gershgorin circle
%General Parameters
b=-1:0.0125:9;
c = -4:0.01:4;
N=3;
%1D difference matrix
K = diag([2*ones(N,1)],0)+diag([-ones(N-1,1)],1)+diag([-ones(N-1,1)],-1);
K = sparse(K);
%Identity matrix
I = speye(N);
%2D Laplacian matrix
K2D = kron(K,I)+kron(I,K);
%Eigenvalues of K2D
E = eig(K2D);
%Gershgorin Circle
x = 0:0.01:8;
y = sqrt(4^2-(x-4).^2);
%Plot
figure;
hold on
plot(x,y,'--r',x,-y,'--r',b,zeros(801,1),'--r',zeros(801,1),c,'--r')
plot(E,zeros(length(E),1),'X')
hold off
axis equal
xlabel('Real')
ylabel('Imaginary')
title('Eigenvalues of K2D')
```

A.5: Plots the condition number as a function of size for K2D

```
%Code to plot condition number as a function of size for matrix K2D
T=2:70;
for m = 1:length(T)
N = T(m);
%1D difference matrix
K = diag([2*ones(N,1)],0)+diag([-ones(N-1,1)],1)+diag([-ones(N-1,1)],-1);
K = sparse(K);
%Identity matrix
```

```
I = speye(N);
%2D Laplacian matrix
K2D = kron(K,I)+kron(I,K);
clear K; clear I; clear N;
%Compute Eigenvalues and condition number
E = eig(K2D);
condi(m) = max(E)/min(E);
clear E;
end
%Plot
plot(T.^2,condi)
xlabel('size: N^2')
ylabel('Condition Number')
title('Condition Number vs size of K2D')
```

B.1: Generates plots that compare different minimum degree algorithms

```
%Comparison of different minimum degree algorithms
N = 9;
%1D difference matrix
K = sparse(diag([2*ones(N,1)],0)+diag([-ones(N-1,1)],1)+diag([-ones(N-1,1)],-1));
%Identity matrix
I = speye(N);
%2D Laplacian matrix
K2D = kron(K,I)+kron(I,K);
clear I K;

for m = 1:5
   if m ==1
      p=symamd(K2D);
      s = 'symamd';
   end
   if m ==2
      p=symmmd(K2D);
      s = 'symmmd';
   end
   if m ==3
      p=colamd(K2D);
      s = 'colamd';
   end
   if m ==4
      p=colmmd(K2D);
      s = 'colmmd';
   end
   if m ==5
      p=realmmd(K2D);
      s = 'realmmd'
   end
   K2Dmod=K2D(p,p);
   figure;
   spy(K2Dmod)
   title(['Matrix reordered using: ',s]);
   [L,U]=lu(K2Dmod);
   figure;
   spy(L)
   title(['Lower triangular matrix from K2D reordered with: ',s]);
end
```

B.2: Comparison of different minimum degree algorithms as a function of matrix size

```
%Comparison of different minimum degree algorithms
T = 2:5:62;
for k = 1:length(T)
    N = T(k);
    %1D difference matrix
    K = sparse(diag([2*ones(N,1)],0)+diag([-ones(N-1,1)],1)+diag([-ones(N-1,1)],-1));
    %Identity matrix
    I = speye(N);
    %2D Laplacian matrix
    K2D = kron(K,I)+kron(I,K);
    clear I K;
    for m = 1:5
        if m ==1
           p=symamd(K2D);
        end
        if m ==2
           p=symmmd(K2D);
        end
        if m ==3
           p=colamd(K2D);
```

```
        end
        if m ==4
           p=colmmd(K2D);
        end
        if m ==5
           p=realmmd(K2D);
        end
        K2Dmod=K2D(p,p);
        [L,U]=lu(K2Dmod);
        NoN(m,k) = nnz(L);
       clear L K2Dmod;
    end
end
%Plot
figure;
plot(T.^2,NoN(1,:),T.^2,NoN(2,:),T.^2,NoN(3,:),T.^2,NoN(4,:),T.^2,NoN(5,:))
legend('symamd','symmmd','colamd','colmmd','realmmd')
title('Comparison of minimum degree algorithms vs matrix size')
ylabel('Number of nonzeros in L');
xlabel('Size: N^2')
figure;
plot(T(4:6).^2,NoN(1,4:6),T(4:6).^2,NoN(2,4:6),T(4:6).^2,NoN(5,4:6))
legend('symamd','symmmd','realmmd')
title('Comparison of minimum degree algorithms vs matrix size')
ylabel('Number of nonzeros in L');
xlabel('Size: N^2')
```

B.3: Comparison: nonzeros in Cholesky factor, computational time for LU and Cholesky decompositions, computational time in reordering

```
%Comparison of different minimum degree algorithms: Computational time,
%Cholesky and LU decomposition and number of nonzero elements
T = 2:5:347;
for k = 1:length(T)
    N = T(k);
    %1D difference matrix
    K = sparse(diag([2*ones(N,1)],0)+diag([-ones(N-1,1)],1)+diag([-ones(N-1,1)],-1));
    %Identity matrix
    I = speye(N);
    %2D Laplacian matrix
    K2D = kron(K,I)+kron(I,K);
    clear I K;
    for m = 1:4
        if m ==1
           tic
           p=symamd(K2D);
           time = toc;
           K2Dmod=K2D(p,p);
       end
        if m ==2
           tic
           p=symmmd(K2D);
           time = toc;
           K2Dmod=K2D(p,p);
        end
        if m ==3
           tic
           p=colamd(K2D);
           time =toc;
           K2Dmod=K2D(p,p);
       end
        if m ==4
           tic
           p=colmmd(K2D);
           time=toc;
           K2Dmod=K2D(p,p);
        end
        if m ==5
           tic
           p=realmmd(K2D);
           time=toc;
           K2Dmod=K2D(p,p);
        end
        tic
        [L]=chol(K2Dmod);
        time2 = toc;
        tic
        [L2,U2]=lu(K2Dmod);
        time4 = toc;
        NoN1(m,k) = nnz(L);
```

```
        NoN2(m,k) = nnz(L2);
        TiMe(m,k)=time;
        TiMeChol(m,k)=time2;
        TiMeLU(m,k)=time4;
       clear L K2Dmod time conn time2 time3 time4 time5;
    end
end
%Plot
figure;
plot(T.^2,NoN1(1,:),T.^2,NoN1(2,:),T.^2,NoN1(3,:),T.^2,NoN1(4,:))
legend('symamd','symmmd','colamd','colmmd')
title('Comparison of minimum degree algorithms vs matrix size')
ylabel('Number of nonzeros in U');
xlabel('Size: N^2')

figure;
plot(T.^2,TiMeChol(1,:),T.^2,TiMeChol(2,:),T.^2,TiMeChol(3,:),T.^2,TiMeChol(4,:))
legend('Cholesky <= symamd','Cholesky <= symmmd','Cholesky <= colamd','Cholesky <=
colmmd')
title('Computational time for Cholesky decomposing reordered matrices vs size')
ylabel('Time (sec)');
xlabel('Size: N^2')

figure;
plot(T.^2,TiMeLU(1,:),T.^2,TiMeLU(2,:),T.^2,TiMeLU(3,:),T.^2,TiMeLU(4,:))
legend('LU <= symamd','LU <= symmmd','LU <= colamd','LU <= colmmd')
title('Computational time for LU decomposing reordered matrices vs size')
ylabel('Time (sec)');
xlabel('Size: N^2')

figure;
plot(T.^2,TiMe(1,:),T.^2,TiMe(2,:),T.^2,TiMe(3,:),T.^2,TiMe(4,:))
legend('symamd','symmmd','colamd','colmmd')
title('Reordering computational time')
ylabel('Time (sec)');
xlabel('Size: N^2')
```

B.4: Movie for red-black ordering algorithm (based on code obtain at [3])

```
%Movie for red-black ordering algorithm
N=3;
A=sparse(toeplitz([2,-1,zeros(1,N-2)]));
I=speye(N,N);
B=kron(A,I)+kron(I,A);
[x,y]=ndgrid(1:N,1:N);
p=redblack(B);
B=B(p,p);
R=chol(B(p,p));
xy=[x(p);y(p)]';

n=size(B,1);
L=zeros(n);
lpars={'marker','.','linestyle','none','markersize',64};
tpars={'fontname','helvetica','fontsize',16,'horiz','center','col','g'};
for i=1:n
  clf
  gplot(B,xy);
  line(xy(i:N^2,1),xy(i:N^2,2),'color','k',lpars{:});
  for j=i:n
    degree=length(find(B(:,j)))-1;
    text(xy(j,1),xy(j,2),int2str(degree),tpars{:});
  end
  axis equal,axis off,axis([1,N,1,N])
  pause(0.3);

  line(xy(i,1),xy(i,2),'color','r',lpars{:});
  pause(0.3);

  L(i,i)=sqrt(B(i,i));
  L(i+1:n,i)=B(i+1:n,i)/L(i,i);
  B(i+1:n,i+1:n)=B(i+1:n,i+1:n)-L(i+1:n,i)*L(i+1:n,i)';
  B(i, i:n)=0;
  B(i:n, i)=0;
end
spy(L,32)
```

B.5: Red-black ordering algorithm

```
function p=redblack(A)
%REDBLACK:  Computes the permutation vector implementing the red-black
%ordering algorithm
```

```
n=size(A,1);
temp = 1:n;
count = 0;
odd = 2;
flag =1;
for m = 1:n
    if flag
       p(m)=temp(m+count);
       count = count+1;
       if p(m)==n|p(m)+2>n
          flag = 0;
       end
    else
       p(m)=temp(odd);
       odd = odd+2;
    end
end
```

B.5: Red-black algorithm: comparison code

```
%Comparison code: red-black algorithm
T=3:2:233;
for m = 1:length(T)
    N = T(m);
    A=sparse(toeplitz([2,-1,zeros(1,N-2)]));
    I=speye(N,N);
    B=kron(A,I)+kron(I,A);
    clear N I A;
    tic
    p=redblack(B);
    time = toc;
    C=B(p,p);
    tic
    [L,U]= lu(B);
    time2 = toc;
    tic
    [L2,U2]= lu(C);
    time3 = toc;
    NoN1(m)=nnz(L);
    NoN2(m)=nnz(L2);
    TiMe(m)=time;
    TiMeLUor(m)=time2;
    TiMeLUrb(m)=time3;
    clear L L2 U U2 time time2 time3 p B C m;
end

figure;
plot(T.^2,NoN1,T.^2,NoN2)
title('Number of nonzeros in L as a function of matrix size')
legend('L<=Original K2D','L<=Reordered K2D');
xlabel('Size: N^2')
ylabel('Number of nonzeros in L')

figure;
plot(T.^2,TiMe)
title('Computational time for red-black ordering algorithm')
xlabel('Size: N^2')
ylabel('Time (sec)')

figure;
plot(T.^2,TiMeLUor,T.^2,TiMeLUrb)
title('Computational time of the LU decomposition algorithm')
legend('L<=Original K2D','L<=Reordered K2D');
xlabel('Size: N^2')
ylabel('Time (sec)')
```

B.6: Nested dissection ordering algorithm: comparison code

```
%Comparison code: red-black algorithm
T=3:4:100;
for m = 1:length(T)
    N = T(m);
    A=sparse(toeplitz([2,-1,zeros(1,N-2)]));
    I=speye(N,N);
    B=kron(A,I)+kron(I,A);
    clear N I A;
    tic
    p = gsnd(B);
    time = toc;
    tic
    d = specnd(B);
```

```
    time4 = toc;
    C=B(p,p);
    D = B(d,d);
    tic
    [L,U]= lu(B);
    time2 = toc;
    tic
    [L2,U2]= lu(C);
    time3 = toc;
    tic
    [L3,U3]= lu(D);
    time6 = toc;
    NoN1(m)=nnz(L);
    NoN2(m)=nnz(L2);
    NoN3(m)=nnz(L3);
    TiMe(m)=time;
    TiMe2(m)=time4;
    TiMeLUor(m)=time2;
    TiMeLUnd1(m)=time3;
    TiMeLUnd2(m)=time6;
    clear L L2 U U2 time time2 time3 p B C m time4 time6 D d;
end

figure;
plot(T.^2,NoN1,T.^2,NoN2,T.^2,NoN3)
title('Number of nonzeros in L as a function of matrix size')
legend('L<=Original K2D','L<=Reordered by "gsnd"','L<=Reordered by "specnd"');
xlabel('Size: N^2')
ylabel('Number of nonzeros in L')

figure;
plot(T.^2,TiMe,T.^2,TiMe2)
title('Computational times for nested dissection algorithms: "gsnd" and "specnd"')
legend('"gsnd"','"specnd"')
xlabel('Size: N^2')
ylabel('Time (sec)')

figure;
plot(T.^2,TiMeLUor,T.^2,TiMeLUnd1,T.^2,TiMeLUnd2)
title('Computational time of the LU decomposition algorithm')
legend('L<=Original K2D','L<=Reordered by "gsnd"','L<=Reordered by "specnd"');
xlabel('Size: N^2')
ylabel('Time (sec)')
```