OBJECT ORIENTED MATRIX CLASSES FOR SPARSE MATRICES USING C++

BY

GORDON WILLIAM ZEGLINSKI

B.Sc. (M.E.) University of Manitoba 1991

A Thesis

Submitted to the Faculty of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Mechanical and Industrial Engineering

University of Manitoba

Winnipeg, Manitoba, Canada

March 20, 1993

OBJECT ORIENTED MATRIX CLASSES FOR SPARSE MATRICES USING C++

BY

GORDON WILLIAM ZEGLINSKI

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

ABSTRACT

This thesis presents an object oriented matrix hierarchy. The focus of this pioneering work is to improve the flexibility and ease of programming matrix manipulations without sacrificing performance. This hierarchy encompasses both sparse and full matrix operations allowing a greater ease of programming than before. It includes types to hold full matrices, general sparse matrices and banded sparse matrices. The matrix operations covered by this hierarchy include matrix multiplication, scalar multiplication and division, matrix inversion, and LU decomposition, to name just a few. Speed and accuracy tests show that the developed algorithms are as fast and accurate as those used in procedural based languages such as FORTRAN. Thus, this matrix hierarchy allows for easier program development, maintenance and extensibility without a sacrifice in performance and accuracy.

ACKNOWLEDGMENTS

I would like to thank my advisors, Dr. Ray P. S. Han and Dr. P. W. Aitchison, for their supervision of this thesis and their helpful input. The author would also like to convey his appreciation of the financial support provided by the National Science and Engineering Research Council of Canada.

Much thanks also goes to my parents and the many other people who provided support and understanding during the time spent on this thesis. Their support have made it possible for me to complete my studies.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

LITERATURE SURVEY AND PROPOSED WORK

1.1 Introduction

Sparse and full matrices are frequently generated by numerical methods employed in the various engineering disciplines. For example, large sparse matrices are always formed by the well-known finite element method, the widely-used finite difference technique and the popular boundary element method. Due to this wide spread application of numerical analysis, it is highly desirable to develop an approach that can handle these and other types of general matrices readily and efficiently, such as via an object oriented programming (OOP) environment. Traditionally, the field of matrix computation has been dominated by non-OOP languages such as FORTRAN. Some of the largest and well-known packages such as EISPAC, LINPAC, and SPARPAC are in FORTRAN. However, during the past few years, a number of these packages has migrated towards the use of more modern but still non-OOP languages, like the C language.

There are many OOP-languages currently available to the programmer. C++ first appeared in the PC market about 4 years ago. Since its introduction, It has rapidly become the language of choice by many professional programmers because it combines the OOP features with the versatility of C. OOP languages such as Smalltalk have too much overhead to be useful in numerical fields which C++ does not suffer from. However, an OOP-language offers many advantages over a non-OOP language. One of the most

important advantages is increased programmer productivity. This increase is achieved by the reusability and "blocking" of the code which allow for easier program development, maintenance and upgrading. A layman's definition of reusability is that sections of the code can be used repeatedly even after substantial code modifications have been performed. By "blocking" of the code, we mean that all the code and data that affect a particular object are located in the same section of the program.

1.2 Survey of Existing Work

The literature survey outlined here will concentrate on the use of OOP-languages for numerical analysis in engineering. It is shown in [1] and [2] that the finite element method itself is ideally suited to programming in an OOP-language. In general numerical analysis, they are many sources of algorithms and source code for sparse and full matrix computations. Several of the more common packages have already been identified. Code fragments are also available from a number of sources, see for example, *Numerical Recipes* [3], a very well-known collection of programs for engineering usage and an on-line database, Netlib [4]. For full matrix computation, several C++ packages exist and these include packages like M++[1], Math.h++[2] and newmat[3]. The latter is public domain software. However, in the area of sparse matrix calculation, there is no single dominant package yet available. This is due primarily to the different types of sparsity that can occur in a sparse matrix, and this necessitates the use of different types of storage

[1] Copyright by Dyad Software
[2] Copyright by Rogue Wave Software
[3] Copyright by R. B. Davies

schemes. For example, a banded sparse matrix can be stored in the banded form, but also in a skyline form for greater efficiency. A good discussion of sparse matrix storage types and computations can be found in [5] and [6]. Due to the variety of storage schemes that can be employed, it is impossible to come up with a single package that can handle this situation without resorting to OOP. There currently is no object oriented generalized sparse matrix package available for usage and it is the intent of this thesis to fill this void.

1.3 Proposed Research

An OOP-based package for handling general matrices is developed in this work. The benefits of OOP-styles in numerical programming are utilized by the object oriented matrix hierarchy so that it offers greater flexibility and ease of programming than the conventional approach without a sacrifice in performance. The hierarchy includes objects to store full matrices, banded sparse matrices and general sparse matrices. It allows different types of matrices to be intermixed in equations where necessary and provides a well defined platform for further expansion. The algorithms are tested for speed and accuracy using a series of tests designed to simulate typical applications.

1.4 Thesis Layout and Future Work

The layout of this thesis is as follows. To aid the reader unfamiliar with OOP, a short C++-based OOP primer is included in Chapter 2. An integrated sparse and full matrix library is presented in Chapter 3. Numerical testings

are reported in Chapter 4, in order to demonstrate that the advantages of the proposed matrix hierarchy do not come with a significant performance loss. The matrix library is also tested for speed and accuracy. Finally, conclusions are given in Chapter 5.

It is hoped that future engineering programs will shift away from non-OOP languages so that greater flexibility, improved usability and enhanced ease of programming of new applications can be achieved. Also, it would be desirable to extend the matrix library developed here so that it becomes more usable in specialized engineering applications. This extension should include support for block-sparse storage schemes, improved banded matrices, sub-matrix classes, and better eigenvalue and eigenvector support.

CHAPTER 2

AN INTRODUCTION TO OOP USING C++

Before one can understand some of the important features and benefits of an object oriented matrix library, one should be familiar with some of the basic concepts of OOP. This section will present the reader with some basic OOP concepts and the way they are implemented in C++. *Appendix* I provides the definition of additional terms used in this paper, which do not warrant a lengthy explanation here.

To begin with, OOP is based on a completely different philosophy. In OOP, data and the functions that operate on it are closely bound which is different from non-OOP languages (procedural based languages) as illustrated in *Figure* 1. Observe in the figure that a non-object oriented program is simply a collection of data and functions (procedures) whereas, an object oriented program is made up of one or more objects. The term, closely bound as used here, means the data and functions are grouped together as one object so that the relationship between the two is obvious when viewing the code, a highly desirable feature in very large programs. In C++, close binding namely, an object, is achieved by an extension to the structure, as defined in C. Also, being close together facilitates easy control of their access. Thus, debugging an object oriented program is easier than a non-object oriented program.

Encapsulation is a concept that is closely related to the concept of an object. Encapsulation is the wrapping of a process or group of related objects inside

an object or a hierarchy of objects. In this work, the matrix library encapsulates many matrix operations and storage schemes.

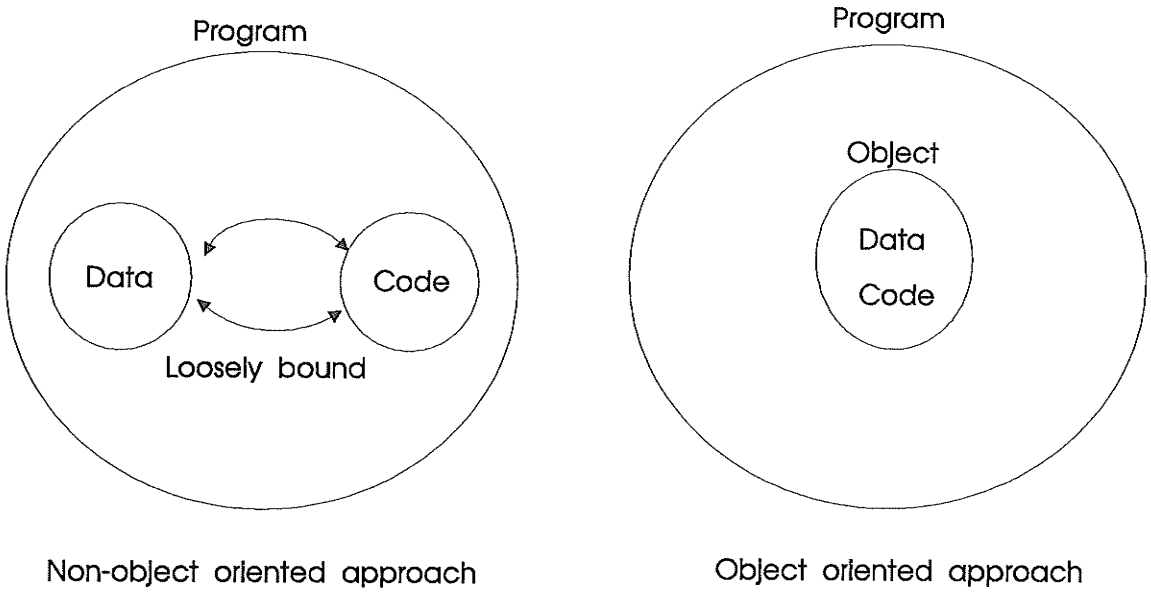

Figure 1. Object Oriented Vs Non-Object Oriented Programming Styles

Data and process abstraction are two important features of an OOP language. They hide the "internals" of the object from the "outside world". This helps to insure that these internals cannot be improperly adjusted by defining a strict interface that prevents other objects from creating errors within the object. It is well-known that C++ does not check to see if an array is accessed beyond the end of its declared length, unlike FORTRAN and PASCAL languages. To overcome this problem, array objects in C++ should be written to prevent the access beyond the end of the array's length. The matrix objects also includes this feature.

Code reusability is largely accomplished by inheritance. In inheritance, the child object retains all the properties of its parent object but the child object is free to redefine any property or add new properties. *Figure* 2 shows an example of single inheritance.

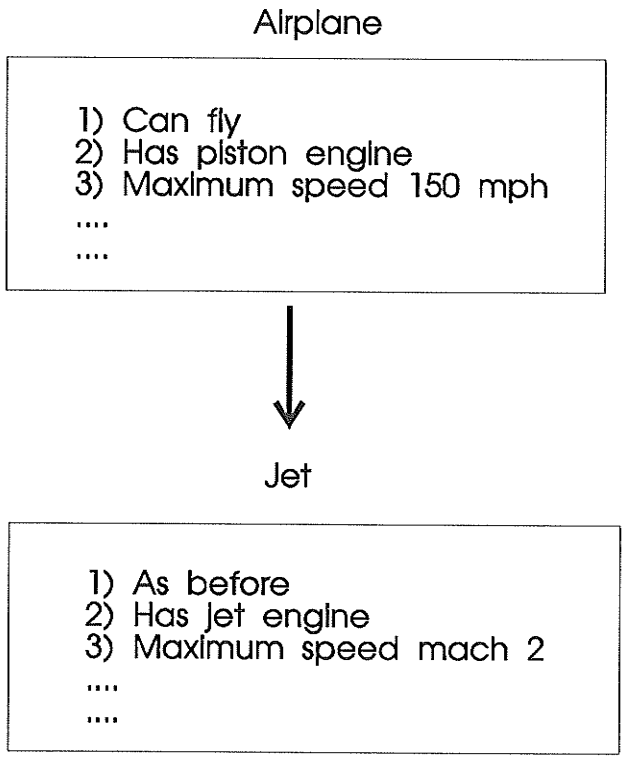

Figure 2. Single Inheritance Example

In this example, the parent class is *Airplane*. *Airplane* has a set of properties that are common to most single engine airplanes. *Jet* is a child of the *Airplane*. Thus, *Jet* inherits property 1 but redefines properties 2 and 3. If

one wanted to extend the hierarchy further, one would simply have to derive a new object from either the *Airplane* or *Jet* object. In this way, any code that makes up the parent object can be reused within the child.

C++ also allows another form of inheritance. This form of inheritance is called multiple inheritance. For instance, the example in *Figure* 2 can be extended to include another object called *Fighter* that has properties common to military fighter planes. Now we can create a *Jet Fighter* by deriving an object from both the *Jet* object and the *Fighter* object.

Polymorphism is one of the more advance concept that the reader should be familiar with. Polymorphism allows a more specific object to be used in place of a more general object. Going back to the previous example, a *Jet* is a more specific definition of an *Airplane* because it is derived from the *Airplane* object. Thus, a *Jet* object could be used wherever the *Airplane* object is used but not vice-versa. Figure 3 illustrates a class hierarchy that allows an *Airplane* object and a *Jet* object to be used interchangeably where *Things that Fly* is called upon.

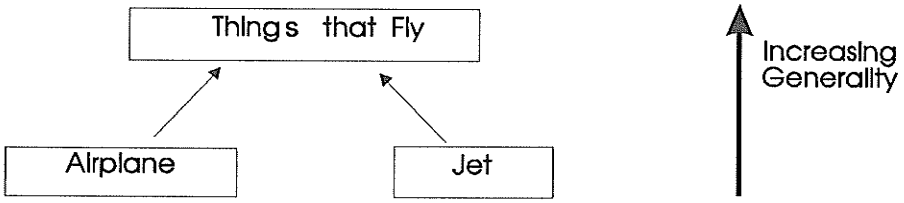

Figure 3. Sample Hierarchy Where the Airplane and Jet are on the Same Level

CHAPTER 3

THE C++ MATRIX LIBRARY

3.1 Overview of the Class Structure

There are several goals that have to be kept in mind when designing the matrix hierarchy. These are portability, efficiency and extensibility. The issue of portability arises due to the differences in the way that C++ compilers handle temporary objects. Temporary objects are frequently used to return the results of matrix operations. Efficiency is important to minimize the amount of time it takes to do a given matrix operation. This is accomplished via optimized routines and by balancing the amount of OOP in the classes' member functions against the speed involved in using a non-OOP approach within the member function. It should be noted that this trade-off is only done inside the member functions and it does not effect the overall object oriented nature of the hierarchy. Because this hierarchy encapsulates both sparse and full matrices, a clear and easy method to allow additional matrix types to be added had to be designed. These three goals will be discussed in greater detail throughout this chapter.

Figure 4 shows the matrix hierarchy in its present state. In keeping with convention, the arrows point from the child class (or derived class) to the parent class. Classes with dashed boxes are abstract base classes. Those with solid boxes are usable matrix objects (matrix types).

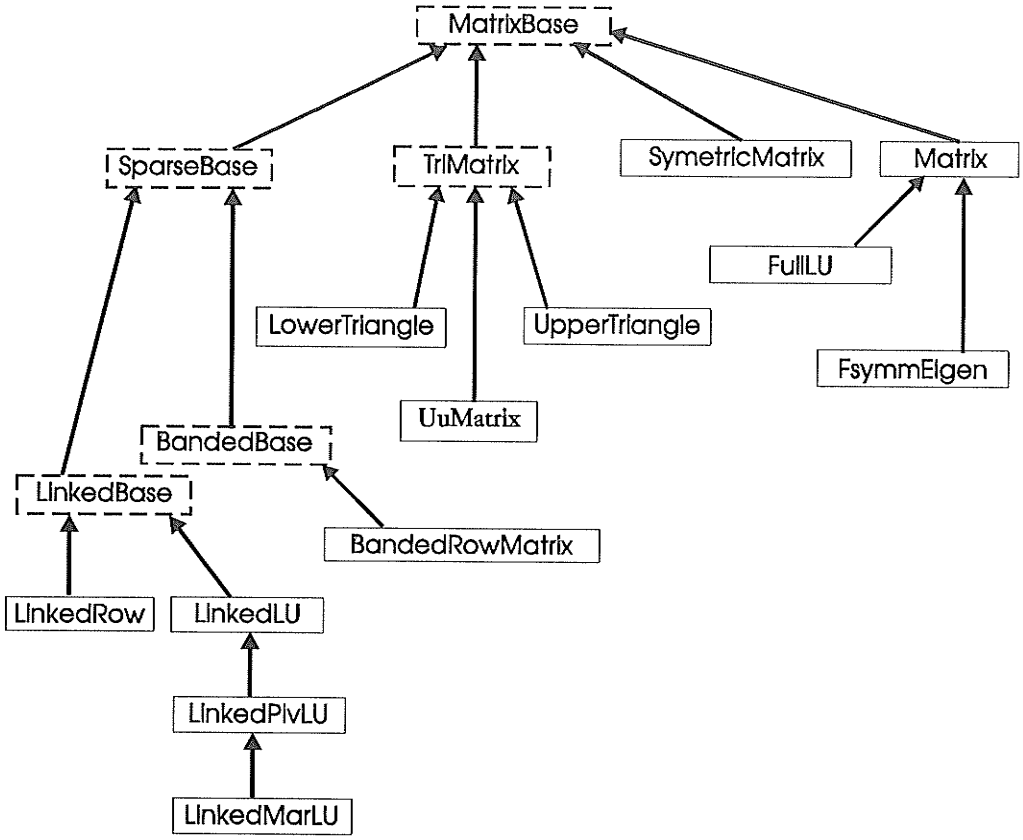

Figure 4. The Matrix Hierarchy

A brief description of the matrix hierarchy is outlined here by discussing the important details of each of the matrix classes. These descriptions are arranged in accordance to their functionality. *Appendix* II provides a pull-out user's guide to this matrix hierarchy. This guide provides more information on how to use and extend this library.

3.1.1 Abstract Classes

The abstract objects are the most important objects to the hierarchy because they provide its order and structure. They specify what properties (i.e. member functions and data) each of their children must define, and the properties that are common to all their children. This section will discuss these objects.

The *MatrixBase* class is the base object for the entire matrix hierarchy. It stores the size of the matrix, the matrix index base, and a tolerance for testing zero. The size of the matrix is unique to each instance but the value of the matrix index base and the tolerance apply to every instance. This class remaps all the common C++ operators (*=, /=, +, -, *, /, *etc.*) into their basic mathematical operations: matrix addition, matrix subtraction, matrix multiplication, scalar division, and scalar multiplication. These basic mathematical operations must be defined by all the objects in the hierarchy. In addition to the functions that provided the matrix operations, this class also provides a set of utility functions for gathering information about a particular matrix instance.

The class, *SparseBase,* provides a set of functions that are unique to sparse matrices. A sparse matrix is a matrix in which most of its terms are zero. A common interface had to be defined so that different storage schemes could interact with each other and still exploit the sparsity in this type of matrix. By exploiting the sparsity, the amount of floating point operations that have

to be performed is reduced. Thus, the speed of the operation can be significantly increased.

The class, *TriMatrix,* provides a common basic storage scheme that is used by all triangular matrices. By employing this class as a base for other triangular matrices, all the code necessary to maintain the memory required to store the triangular matrix can be reused.

The *BandedBase* class provides the basic storage schemes required to hold a variable bandwidth matrix. However, this class uses an interesting concept. In sparse matrices, it is sometimes better to store a matrix by column rather than by row because some algorithms are most efficient when used with a specific storage scheme. For example, sparse matrix multiplication is best done when a row based matrix is multiplied by a column based matrix. However, from a storage perspective, it doesn't matter in what format the data is stored. To address this situation conveniently, the concepts of major and minor are introduced. A major is the index by which the matrix is stored and a minor is the index used to identify the element. The *BandedBase* class stores the matrices by the major. *Figure* 5 illustrates this concept for a row based, ideally-banded matrix.

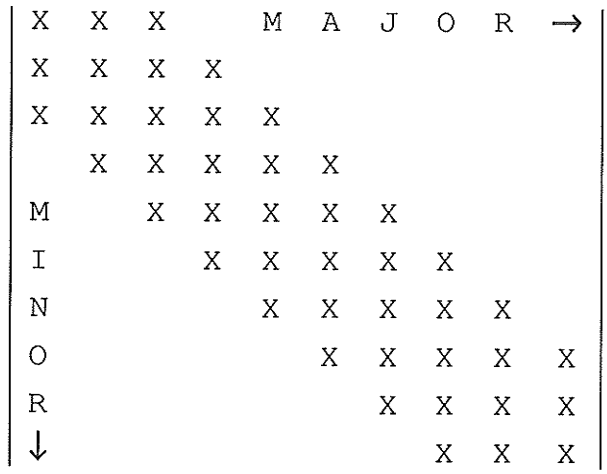

Figure 5. An Ideal Banded Matrix

The class, *LinkedBase,* serves the same function as *BandedBase* except that it is for a generalized sparse matrix. This class provides all the low level list manipulation functions necessary to maintain the storage system. Because this class and its descendants use a linked list storage system, matrix elements can be easily inserted regardless of previous operations performed on the matrix. Because of this flexibility, classes which are derived from this one have much more freedom in the implementation of solvers than those based on the banded scheme. With this type of storage system, pivoting may be done for maximum stability or minimum fill-in. Note the fill-in that occurs when the matrix is decomposed into its LU form is undesirable and should be kept to a minimum, since it increases computational time and decreases accuracy.

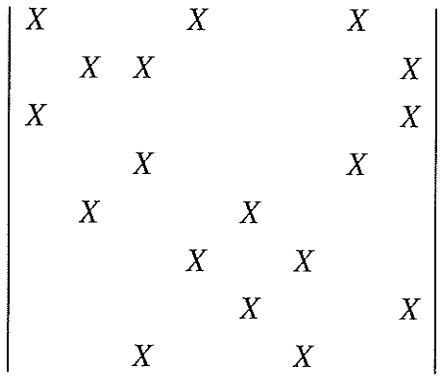

Figure 6. A Typical Sparse Matrix for Linked List Storage

3.1.2 Full Matrices

There are three separate full matrix storage schemes. Each one provides a specialized type of matrix or matrix function. These classes will be discussed in this section.

The class *Matrix* provides the functions necessary to maintain the full matrix storage system. This class uses the Gauss-Jordan based algorithm for solving and inverting matrices. In general, this class is not very efficient for solving systems of linear equations or inverting matrices. However, when it comes to performing full matrix math operations, this class is very efficient. The *Matrix* class also serves as a base for other more specialized classes that provide very efficient specialized matrix functions.

The class *SymetricMatrix* provides the storage system for handling full symmetric matrices. To reduce storage requirements, this class only stores half of the matrix. Thus, it is similar to the triangular matrices. However, it is not a child of the *TriMatrix* class because of differences between the member functions.

3.1.3 Triangular Matrices

The three triangular matrices are all derived from the parent class *TriMatrix*. These different triangular matrix classes simply add a different perspective to the way the triangular matrix storage scheme is interpreted.

The classes *UpperTriangle* and *LowerTriangle* hold upper and lower triangular matrices, respectively. The advantages of these classes are that they can quickly solve systems of equations and have optimized matrix operation routines to take advantage of their unique properties. The class *UuMatrix* provides a U^TU decomposition routine for full symmetric matrices. Unfortunately, this requires that the diagonals of the symmetric matrix be positive. To eliminate this constraint, a U^TDU storage and decomposition system can be implemented.

3.1.4 Full Matrix Decompostions

The class *FullLU* provides the LU decomposition routines for full matrices. This decomposition algorithm is based on the LU decomposition routines developed in LINPAC. These routines are chosen because they are known to

be accurate and efficient. To reduce the effects of ill-conditioning, this class uses partial pivoting in the LU decomposition algorithm. Note that the *FullLU* class performs a specialized task, and as such, it should not be used for general full matrix operations.

The class *FsymmEigen* implements the QR algorithm to find the eigenvalues and eigenvectors of real symmetric matrices. The algorithm used here is a modified version of the algorithms used in EISPAC. Further information about the nature of this algorithm or the LU algorithm can be found in [3]. Like the *FullLU* class, this class performs a specialized task and should not be used for general full matrix operations.

3.1.5 Sparse Matrices

There are currently two sparse matrix storage objects available, *LinkedRow* and *BandedRowMatrix*. *LinkedRow* provides a row oriented linked list matrix structure, while *BandedRowMatrix* provides a row oriented banded matrix structure. Further details about the nature of these classes can be found by looking at the description of their parent classes in Section 3.1.1 *Abstract Classes*.

3.1.6 Sparse Matrix Decomposition

There are three sparse matrix decomposition classes. They are all based on the linked list abstract class but have different variations on the pivoting strategy used. When performing a LU decomposition on a sparse matrix, one

has to consider the amount of fill-in that results. This fill-in greatly effects the computation time and accuracy of the decomposition. Thus, it is very desirable to keep the amount of fill-in to an absolute minimum.

The class *LinkedLU* implements a simple LU decomposition routine and acts as a base for the more advance decomposition classes. This class does not implement pivoting. Thus, its use is highly limited. For illustrative purposes, a sample matrix will be decomposed in *Figure* 7. Step 1 in *Figure* 7 shows the original matrix. Step 2 shows the results of the reductions using the first row. Step 3 shows the results of the reductions using the second row. There is no need in reducing this example further because the LU form has been established after step 3. Note that there is no fill-in in this example due to its structure. In practice, matrices will have very large amounts of fill-in if they are not pivoted to reduce this undesirable phenomenon.

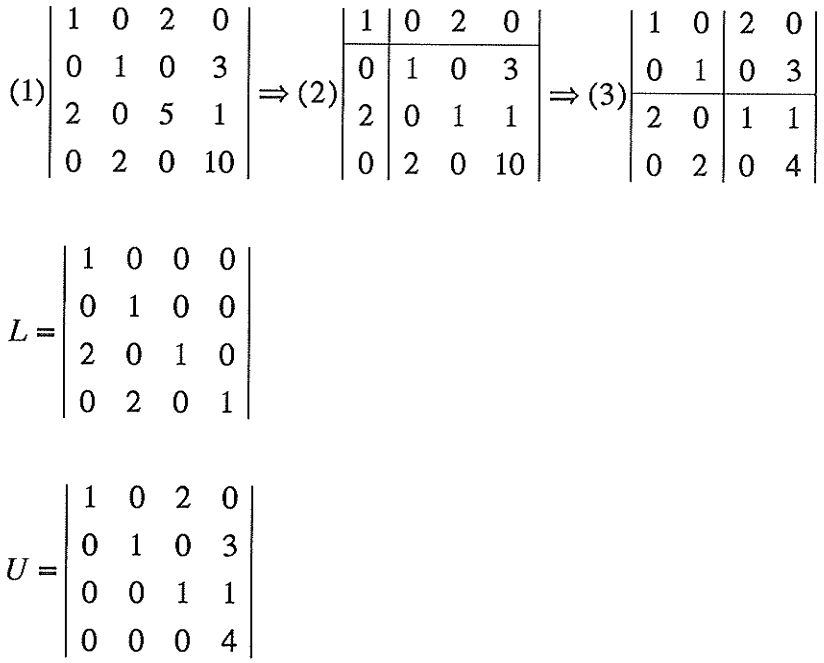

Figure 7. LU Decomposition of a Sparse Matrix

The class, *LinkedPivLU,* implements a full pivoting LU decomposition algorithm. It also provides the additional storage structure necessary to keep track of the rows and columns used in the pivoting operations. Full pivoting is essential in sparse matrices because typically there isn't enough elements to choose from if only partial pivoting is done. Thus, ill-conditioned matrices cannot be accurately solved if only partial pivoting is done. However, pivoting without regards to fill-in typically results in very large amounts of fill-in. Thus, a compromise between fill-in and stability should be used when choosing a pivot element.

The class *LinkedMarLU* implements a widely accepted compromise pivoting strategy. This strategy, know as the Markowitz pivoting strategy, allows one

to customize the amount of trade off between stability and fill-in. The first step, in this strategy, is to find the absolute maximum value of the elements in the matrix, and the number of significant non-zero elements in each row and column. An element is considered significant if it resides in a row and column whose index value is equal to or greater than the number of the current pivoting operation. Then a search is made to find all the elements that have an absolute value greater than or equal to an arbitrary percentage of the maximum value previously obtained. By changing this percentage, the accuracy versus fill-in compromise can be suitably adjusted. Out of these elements that satisfy this criterion, the one with the lowest "area" is chosen. The "area" is defined as

$$\text{area}=(\text{NSER}-1)\cdot(\text{NSEC}-1)$$

in which NSER is the number of significant elements in its row and NSEC is the number of significant elements in its column. Consider the sample matrix shown in *Figure* 8. Assuming that the current pivoting operation is the first, in *Figure* 8, any element in row 1 has 3 significant elements and any element in column 1 has 3 significant elements. Thus, the "area" of the element in position (1,1) is 4.

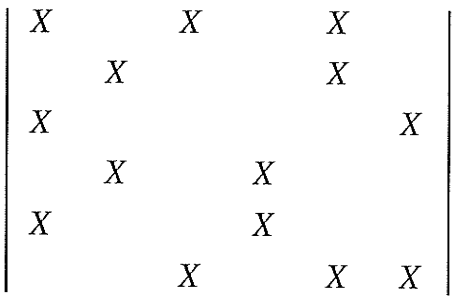

Figure 8. Sample Matrix to Illustrate Markowitz Pivoting

3.2 Matrix Operations

In this section, the matrix operations provided by the hierarchy will be discussed. Two topics of particular interest are, the method by which C++ executes a typical matrix equation and the unique algorithms developed for this hierarchy. *Table* 1 shows the supported matrix operations and their associated function.

Table 1. Matrix Operations and Their Associated Functions

Matrix Operation	C++ Operator[4]	Function
matrix multiplication	* , *=	MMul
scalar multiplication	* , *=	CMul
scalar division	/ , /=	CDiv
matrix addition	+ , +=	MAdd
matrix subtraction	- , -=	MSub
matrix inversion		inv()
solution of linear equations		solve()
transpose of the matrix		trans()

The matrix operations that correspond to C++ operators have been remapped into member functions to minimize the amount of duplicate code. These operations are fairly simple to implement when dealing with full matrices and can be found in any elementary book on matrices or linear algebra [6]. However, the code for the linked list based sparse matrices is not straight forward and warrants an in-depth look. These routines are optimized to make maximum use of the sparsity of the matrix The algorithms used for the linked list based matrix multiplication and addition will be detailed later.

[4] If no entry is given then the matrix operation is implemented by a function rather than by a C++ operator.

Before one can look at the algorithms used, one should be familiar with the manner in which a matrix equation is executed by C++. Consider the following equation:

$$\mathbf{D}=(\mathbf{A}+\mathbf{B})x3x\mathbf{C}.inv()$$

where **A**, **B**, **C**, and **D** are matrices. The exact matrix class does not matter except that it must be compatible with the expected results. For instance, **D** could not be symmetric if the result of the right hand side of the equation is not symmetric. As well, it is fairly unlikely that these would be sparse matrices because in general **C**.inv(), the inverse of **C** would most likely be full. *Figure* 9 shows the execution of this equation.

First, the inverse of **C** is found and stored in a temporary matrix. This temporary matrix is necessary because the functions that perform the matrix operations do not destroy the original matrices. Next, the sum of matrices **A** and **B** is found and stored in another temporary matrix. The temporary matrix is then multiplied by 3. Because the matrix in the scalar multiplication is already temporary, the result of the operation overwrites the original matrix. Next, the two temporaries are multiplied and the result stored in a third temporary matrix. The first two temporaries are deleted and then the contents of the third temporary is copied into matrix **D.** The third temporary is then deleted.

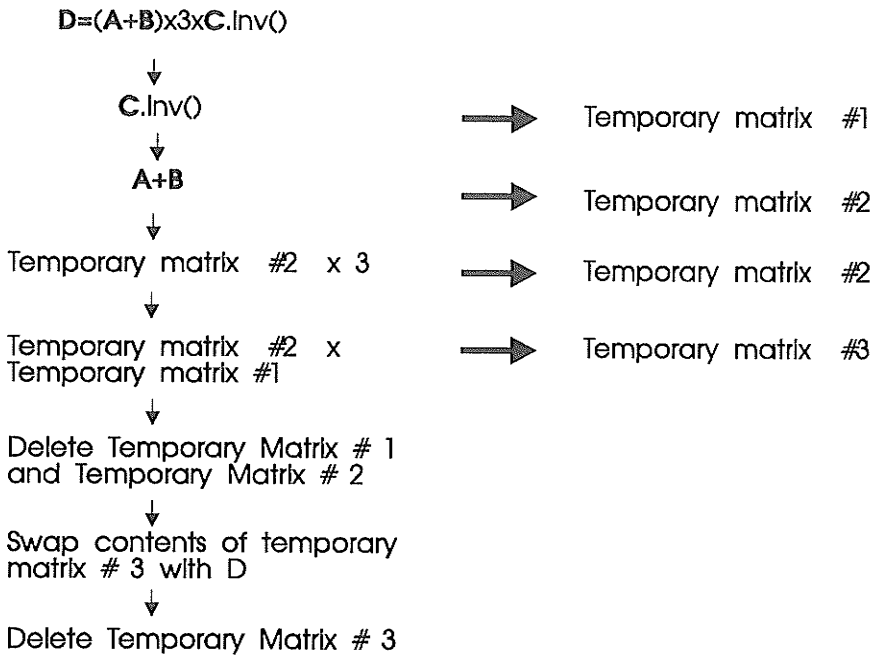

Figure 9. Execution of Sample Equation

Now that the order of execution and need for temporary matrices has been established, the algorithms for the linked list based sparse matrix addition and multiplication routines can be discussed. For example, consider the following two equations:

$$\mathbf{A} + \mathbf{B} \quad \text{and} \quad \mathbf{A} \times \mathbf{B}.$$

For full matrix addition, each element of **A** is added to the corresponding element of **B** and the result is stored in a temporary matrix. However, for sparse matrices, the elements in the two matrices may not coincide. If they

do coincide, the two elements are added and the sum stored in the temporary matrix. If there is no element in **B** that corresponds to the current element in **A**, the value of the current element in **A** is copied into the temporary matrix. For this case fill-in occurs. The opposite is also true, when **B** has an element with no match in **A**. Following is the code used to implement the linked list matrix addition.

```
void LinkedBase::MAdd(MatrixBase & Arg, MatrixBase & Dest){

  if(!CompareSize(Arg)){
      SizeError("Linked Sparse Matrix");
      exit(1);
  }

  Reset();

  if(IsSame(Arg) && IsSame(Dest)){
      MatSize Ma;

      LinkedBase & arg= (LinkedBase &) Arg;
      LinkedBase & dest= (LinkedBase &) Dest;

      arg.Reset();
      dest.Reset();

      if(&arg==this){
          dest=arg*2.0;
          return;
      }

      for(Ma=0;Ma<row;Ma++){
          SetMajor(Ma);
          arg.SetMajor(Ma);
          dest.SetMajor(Ma);
          (*this)++;
          arg++;
          //loop until one has cycled through
          while(arg && (*this) ){
              // continue until arg is greater than current
              while(arg.CurMinor() < CurMinor()){
                  dest.Insert(arg.CurMinor(),arg.CurVal());
                  dest++;
                  arg++;
              }
              // if equal minors store the sum.
              if ( CurMinor() == arg.CurMinor() ){
```

```
                    dest.Insert(CurMinor(),CurVal()+arg.CurVal());
                    arg++;
                    dest++;
                    (*this)++;
                }
                // if Minors are not equal, then this minor must be less
                // than the arguments minor.
                else{
                        dest.Insert(CurMinor(),CurVal());
                        dest++;
                        (*this)++;
                    }
            }
            // One (or both) of the matrices has cycled.
            // finish the rest of the row
            while(arg){
                arg++;
                dest.Insert(arg.CurMinor(),arg.CurVal());
                dest++;
            }
            while( (*this)){
                (*this)++;
                dest.Insert(CurMinor(),CurVal());
                dest++;
            }
        }
    }
}
```

The preceding code is written using the concept of majors and minors as defined previously. This addition routine is fairly easy to understand and is very efficient. The circular linked list is said to have "cycled" when the pointer maintaining the position in the list returns to the beginning of the list. The matrix multiplication routine is somewhat more complex and doesn't really follow the approach that is used for full matrices. For full matrices, each column of **B** is traversed once for each row in **A**. This method was initially tried for the linked list based sparse matrices. However, it was found that it was not as efficient as the following method.

```
void LinkedBase::MMul(LinkedBase & Arg, LinkedBase & Dest){
  MatSize R,C, *Pos;

  Pos=new MatSize[row];
  if((Pos==0) ){
       MemError("Linked Base: MMul");
       exit(3);
  }

  Arg.Reset();
  //Arg.row == the # of columns for a column based list structure
  for(C=Arg.row;C>0;C--){
       for(R=0;R<row;R++)
            *(Pos+R)=*(List+*(Start+R));
       Arg.SetMajor(C-1);
       Arg++;
       while(Arg){
            for(R=0;R<row;R++){
                 while( *(Minor+*(Pos+R)) < Arg.CurMinor())
                      *(Pos+R)=*(List+*(Pos+R));
                 if( (*(Minor+*(Pos+R))) == Arg.CurMinor() ){
                      if(Dest.MinorAtHead(R)==(C-1)){
                           Dest.AddToHead(R,
                      Arg.CurVal()* (*(Storage+*(Pos+R))));
                      }
                      else  // check if entry is zero
                           if (IsZero(Dest.ValAtHead(R)) ){
                                (*(Dest.Minor+*(Dest.List+*(Dest.Start+R))))=C-1;
                                (*(Dest.Storage+*(Dest.List+*(Dest.Start+R))))=0;
                                Dest.AddToHead(R,
                                          Arg.CurVal()* (*(Storage+*(Pos+R))));
                           }
                           else{
                                Dest.InsertAtHead(R,C-1,
                                          Arg.CurVal()* (*(Storage+*(Pos+R))));
                           }
                 }
            }
            Arg++;
       }
  }
  for(R=0;R<row;R++)
       if(IsZero(Dest.ValAtHead(R)))
            Dest.RemoveFromHead(R);

  delete [] Pos;
}
```

This version of the matrix multiplication routine iterates through each column in **B** only once. Because each column is only traversed once, this version of the multiplication routine is approximately 3 times faster than the algorithm that traversed each column once per each row in **A**. The array *Pos* is used to hold the position is each row as the column is traversed. As well, the columns are traversed in reverse order so that when the routine exits, the minor indices of the element list in the destination matrix is in ascending order. Note, this routine assumes that **A** is row based and **B** is column based. *Appendix* III contains the C++ header files for the matrix hierarchy, and the complete source code is contained on the floppy disk attached to the inside back cover.

CHAPTER 4

NUMERICAL RESULTS

4.1 Storage Requirements

In this section, the storage requirements of the matrix classes is given. *Table* 2 defines the terms used in *Table* 3 which shows the storage space required and the storage scheme implemented for the classes in the matrix hierarchy. The storage space required is expressed in terms of a machine independent notation.

Table 2. Definition of Symbols

Symbol	Definition
N	Size of a square matrix
Row	Number of rows
Col	Number of columns
R	Size of an element in bytes
I	Size of the index in bytes
V	Size of a pointer to a function in bytes

Table 3. Matrix Storage Scheme and Space Required

Matrix Class	Storage Scheme	Space Required (in Bytes)
Matrix	array of vectors	R x Row x Col + V x Row
FullLU	array of vectors	R x Row x Col + (I + V) x Row
Symmetric Matrix	array of vectors	R x (N^2 + N) / 2 + V x Row
All Triangular Matrices	array of vectors	R x N^2 + N) / 2 + V x Row
BandedRowMatrix	array of vectors	(Σ Bandwidths) x R + V x Row
LinkedRow	Linked List	(R + I)(# of elements) + I x Row
LinkedLU	Linked List	(R + I)(# of elements) + 2 x I x Row
LinkedPivLU	Linked List	(R + I)(# of elements) + 2 x I x Row + 2 x I x Row
LinkedMarLU	Linked List	(R + I)(# of elements) + 2 x I x Row + 2 x I x Row

The additional storage space required by the matrix classes for implementing a pivoting LU decomposition algorithm, is the space needed to hold the rows and columns used in the pivoting operations. In addition, each matrix instance requires (24xV + 2xI + 1xC) bytes of storage to hold the virtual function tables and other miscellaneous variables. Here, C is the size of a character in bytes, I and V are as before in *Table* 2.

4.2 Operation Speed

In order to clarify the conditions under which the tests are performed, the details of the tools used are described. The test programs are implemented via the GNU C++[5] compiler version 2.2.2 ported to the OS/2 version 2.0[6] operating system. The compiler generates true 32 bit code under a true 32 bit operating system for maximum speed on a 486-PC. The code is compiled using the compiler's optimizations for maximum speed. The test machine is a PC based on the Intel 80486 DX 33 cpu. The timings are taken when only the tested program was running to minimize the effects of multi-tasking. The timings generated by the 32 bit OS/2 executables are compared to the timing generated by 16 bit DOS executables. The 32 bit code is found to be at least twice as fast in every instance. Because DOS is currently such a limited environment, there will be no further mention of the code compiled for DOS.

Several tests are performed to compare the speed of the various functions of the matrix hierarchy. Tests are performed to determine the speed of the matrix multiplication, matrix inversion and equation solving. These tests are divided into two categories. The first category is a set of tests for the full matrix section of the hierarchy, and the second is a different set of tests for the sparse section of the hierarchy. For the sparse equation solving tests, matrices are generated by a program written here to solve heat transfer problems using the finite difference method. This program is given in

[5] TM the Free Software Foundation Inc.
[6] TM International Business Machines Inc.

Appendix IV. This problem, along with the nature of the matrix generated by it are discussed later in this section.

4.2.1 Timing Test for the Full Matrix

The full matrix operations are tested using randomly filled matrices that ranged from 50X50 to 250X250 in size. Each section of the test consisted of 100 iterations to minimize the influence of the multitasking nature of OS/2 on the recorded timings. As well, with the current compiler and tools being used, the accuracy between any two time locations is only to the nearest second. Thus, by increasing the number of tests, and thereby increasing the difference between the two time locations, a more accurate time per operation can be obtained. The source code for this test is located in *Appendix* V. *Table* 4 shows the results of this test.

Table 4. The Timing Results for the Full Matrix Tests

Matrix Operation	Time for the 50X50 Matrix (seconds)	Time for the 100X100 Matrix (seconds)	Time for the 150X150 Matrix (seconds)	Time for the 200X200 Matrix (seconds)	Time for the 250X250 Matrix (seconds)
Multiply	0.58	4.86	16.85	39.86	80.41
LU Decompose	0.10	0.60	2.23	5.25	10.69
Solve	0.004	0.0162	0.0402	0.0710	0.1156
Addition	0.01	0.03	0.07	0.12	0.18

Figure 10 shows the results of the timing tests in a graphical form. Observe in these plots, the time to multiply two NXN matrices or to carry out matrix decomposition, are each proportional to N^3, namely, a cubic fit. In contrast, the time consumed by the addition or solving operations is quadratic. Also, the results from this test are compared to the timings from the conventional C routines. No significant differences are found and therefore, to avoid unnecessary cluttering of results, these comparisons are not plotted in the figures.

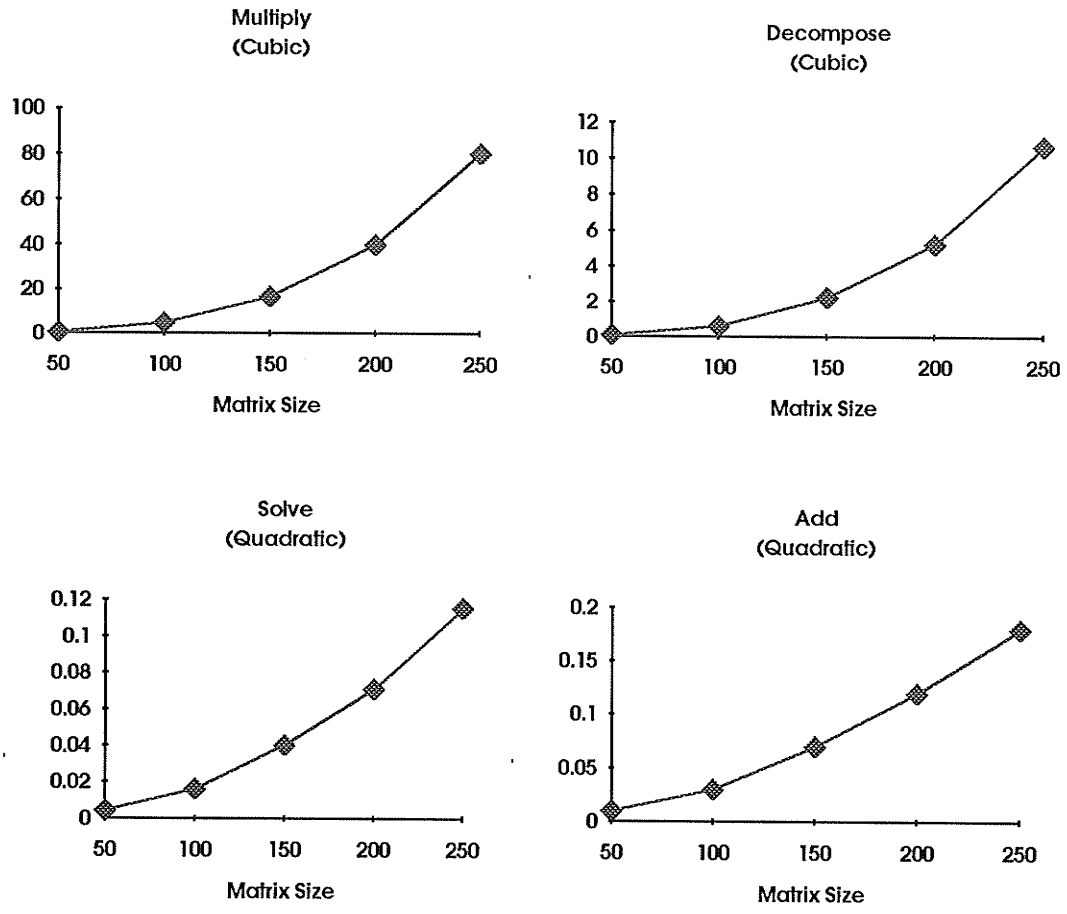

Figure 10. Time Plots for Full Matrix Tests

4.2.2 Timing Test for the Sparse Matrix

Determining the efficiency of sparse matrix algorithms is more difficult than the same task for full matrix algorithms because the matrix size is not a good indicator of the computational effort required to perform the task. What is important is the sparsity of the matrices before and after the matrix operation. As well, inverting a sparse matrix will most often result in a full matrix. Thus, no matrix inversion testing will be performed because the

complex storage scheme used to store sparse matrices makes it very inefficient for use with full matrices. Randomly filled sparse matrices are the worst type of matrix when considering fill-in. Thus, these tests represent the worst case scenario. The solver used is the one coded into the *LinkedMarLU* object. This is the only sparse object that has a solver capable of minimizing the fill-in. Two general sparse matrix tests have been performed. The first simply tests the multiplication operation, the addition operation and the solver operation using general sparse matrices. The source code for this test is given in *Appendix* V. *Table* 5 shows the timing results. The second uses a matrix generated by the finite difference method, the results of which are presented later.

Table 5. Timing Results for Sparse Matrix Operations

Matrix Operation	Time for the 500X500 Matrix (seconds)	Time for the 550X550 Matrix (seconds)	Time for the 600X600 Matrix (seconds)	Time for the 650X650 Matrix (seconds)	Time for the 700X700 Matrix (seconds)
Multiplication	3.88	4.42	5.46	7.32	8.2
Decomposition	14.68	18.28	23.1	28.48	34.26
Solve & Decompose	16.68	21.42	26.52	32.92	40.42
Add	0.05	0.055	0.06	0.07	0.08

The randomly filled matrix test is designed to simulate typical matrices generated by many common engineering applications. That is, even though the size of the matrix is increasing, the number of non-zero elements per row

remains constant. Thus, with this constraint, all the plots in *Figure* 11 show an approximately linear relationship between the size of the matrix and the time it takes to perform the test operations. Note also, the fill-in ratio[7] remained relatively constant for different matrix sizes.

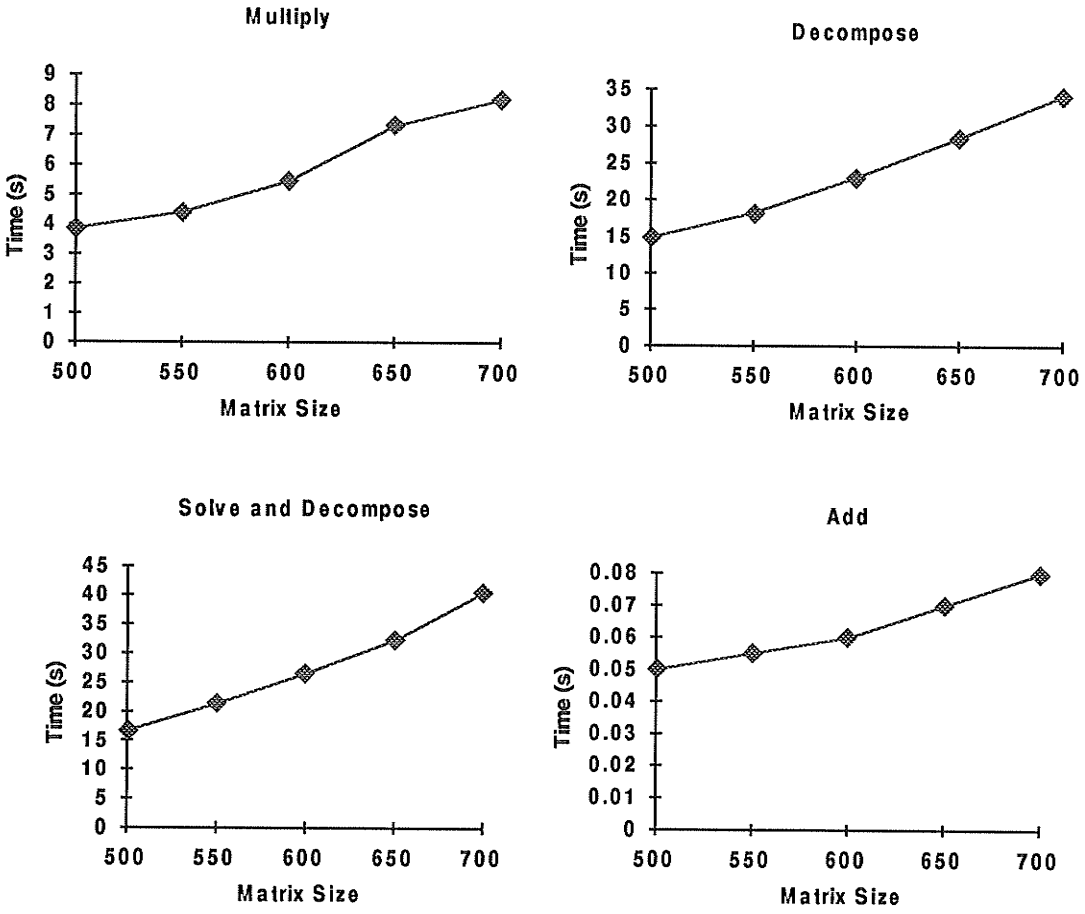

Figure 11. Time Plots for Random-Fill Sparse Matrix Tests

[7] defined as: (# of non-zero elements after decomposition) / (# of non-zero elements after decomposition)

4.2.3 Timing Test for Matrix Generated by the Finite Difference Method

The matrix used in the finite difference test is generated by a program written to solve the heat transfer through a fin. As such, the matrix is strongly diagonal and, for this particular discretization, the matrix has at most five non-zero elements per row. Because the matrix is strongly diagonal, it can be easily solved using the Gauss-Seidel iterative method. A C-based program is written for this purpose. To avoid unnecessary repetition of effort, only the solver part of the program needs to be compared with an OOP-based approach. In this OOP version, the C-based program is rewritten to export the matrices so that they could be used to test the linked list based sparse matrix structure. Note that the matrix generated by the finite difference program cannot be efficiently stored by any method not capable of holding general sparse matrices because the elements in each row are separated by large numbers of zero entries.

The test compares the performance of the non-OOP solver based on the Gauss-Seidel procedure with the OOP solver coded from the Markowitz pivoting technique, in terms of the timing results and fill-in. The issue of accuracy will be discussed in the next section. Since the non-OOP solver cannot pivot, an OOP-based solver with a non-pivoting strategy is tested, in addition to the Markowitz solver. This solver will be referred simply to as the No-Pivot solver. *Table* 6 shows the results of the timing tests. The size of the matrix is given in the first row and the solver type in the first column. The results are given in seconds and are accurate to one second. For instance 25.7 seconds would be given as 25 seconds. This order of accuracy is the result of the limitation of the compiler and ANSI C timing routines used.

Note that later entries of the No-Pivot solver could not be obtained because there is too much fill-in and the matrix could not be stored in memory. *Table* 7 shows the fill-in ratios. Recall, the Gauss-Seidel procedure is an iterative method and hence, does not produce any fill-in.

Table 6. Timing Comparison for the Solver Test

	Size (N)								
Solver	**70**	**216**	**470**	**781**	**1220**	**1696**	**2320**	**2961**	**3770**
Gauss-Seidel	0	2	8	23	58	115	247	430	604
No-Pivot	0	0	4	12	30	-	-	-	-
Markowitz Pivoting	0	1	8	24	65	130	263	429	762

Table 7. Comparison of the Fill-In Ratios

	Size (N)								
Solver	**70**	**216**	**470**	**781**	**1220**	**1696**	**2320**	**2961**	**3770**
No-Pivot	4.75	8.11	11.92	14.67	18.15	20.64	-	-	-
Markowitz Pivoting	2.26	2.91	3.56	4.27	4.97	5.09	5.94	6.19	7.27

Figure 12 shows a plot of the data in *Table* 6. Observe that the time taken by the linked list based solver did not increase in an approximately linear fashion as it did in *Figure* 11 for the general sparse matrices because the fill-in ratio depicted in *Table* 7 did not remain constant. Usually, for this type of

matrix, the merits of the Gauss-Seidel solver far outweighs those of the linked list solver. However, it is important to note that for most part, the time difference between the two OOP and non-OOP solvers is not too great. Thus, for matrices whose solutions are unstable under the Gauss-Seidel method, the linked list based solver can be used without a significant speed deterioration. As well, if several solutions are needed using different right hand side matrices, the LU decomposition in the OOP solvers will be much faster than the Gauss-Seidel solver because the decomposed matrix will only have to be formed once.

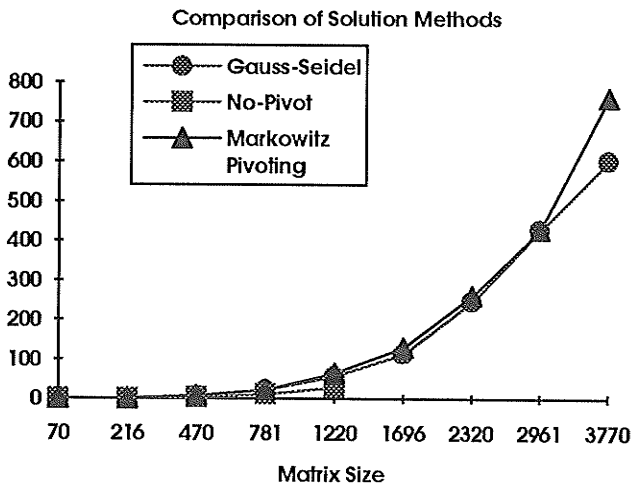

Figure 12. Timing Comparison Between the Gauss-Seidel, No-Pivot and Markowitz Pivoting Solvers

4.3 Accuracy Testing

To complement the previously presented timing tests, accuracy tests for the OOP solvers are carried out. The solver used here are the LU decomposition solver for the full matrix and the Markowitz solver for the sparse matrix. They are coded in the *FullLU* and the *LinkedMarLU* matrix objects respectively. Basically, accuracy in the solving process is assessed here. Two types of accuracy tests are performed. The first is to calculate the inverse of a matrix $\mathbf{A}^{-1}$, and then to determine how accurate this inverse is formed, the $(\mathbf{A}\cdot\mathbf{A}^{-1} - \mathbf{I})$ operation is computed and compared to the null matrix. Note that both the solving and multiplication operations are tested in this process. This test is carried out for the full matrix only. The second test actually involves solving for $\mathbf{x}$ in the expression $\mathbf{A}\cdot\mathbf{x}=\mathbf{b}$, where both **A, b** are known. To gauge the accuracy, the operation $(\mathbf{A}\cdot\mathbf{x} - \mathbf{b})$ is formed and compared with the null vector. This test is carried out for both the full and sparse matrices.

The error in the solution is obtained by calculating the root mean square value as follows:

$$\text{Error} = \frac{\sqrt{\sum (\mathrm{Ax} - \mathrm{b})^2}}{\mathrm{n}}$$

In the same manner, the inversion error is computed by taking the root mean square value of the elements in the matrix $(\mathbf{A}\cdot\mathbf{A}^{-1} - \mathbf{I})$. *Table* 8 shows the error for the $(\mathbf{A}\cdot\mathbf{A}^{-1} - \mathbf{I})$ and the $(\mathbf{A}\cdot\mathbf{x} - \mathbf{b})$ operations, for the full matrix. Observe that these errors are well within the acceptable levels of accuracy.

Table 8. Root Mean Square Error for the Full Matrix Solver

Type of Operations	Order of Magnitude	RMS Errors 50X50 Matrix	100X100 Matrix	150X150 Matrix	200X200 Matrix	250X250 Matrix
$A \cdot A^{-1} - I$	$X10^{-18}$	1.99	5.50	28.4	6.45	5.84
Ax - b	$X10^{-14}$	2.88	1.51	2.30	2.44	13.6

Table 9 shows the results of the accuracy test for the sparse matrix. To more accurately reflect a typical compromise between accuracy and minimization of fill-in, the Markowitz pivoting solver for sparse matrices is given a fairly large choice of pivot elements. This is done by setting the pivot tolerance to a relatively low value of 0.0001. In this test the matrices are more ill-conditioned than in the full tests. Thus, the accuracy in these results is lower than that of the full matrix results because of the increased size of the matrices and because of the chosen pivoting compromise.

Table 9. Root Mean Square Error for the Sparse Matrix Solver

Type of Operations	Order of Magnitude	RMS Errors 500X500 Matrix	550X550 Matrix	600X600 Matrix	650X650 Matrix	700X700 Matrix
Ax - b	$X10^{-9}$	6.00	6.70	383	5.26	0.68

CHAPTER 5

SUMMARY AND CONCLUSIONS

It is shown in this work that the object oriented matrix hierarchy is superior to the non-object oriented way of incorporating matrix functions into applications. The hierarchy developed here uses a blend of unique and well establish algorithms to maximize the performance of the code. It includes functions for many of the common matrix operations that occur in engineering applications. The OOP solvers have been thoroughly tested and found to be just as fast and accurate as the non-OOP solvers.

Using the OOP method, programs can be easily written to take advantage of the best matrix object. The best matrix object is the one that provides for the lowest storage space, fastest solver, or whatever function the application needs. More importantly, when new and better matrix functions are developed, they can be quickly incorporated into the application without changing a single line of code other than the declaration of the matrix instance in question. This kind of flexibility cannot be achieved in non-object oriented programming.

BIBLIOGRAPHY

[1] R. I. Mackie, 'Object Oriented Programming of the Finite Element Method', *International Journal for Numerical Methods in Engineering*, **35**, 425-436, 1992.

[2] G. W. Zeglinski, Ray P. S. Han, and Peter Aitchison, 'C++ Based Object Oriented Matrix Classes for Use in a Finite Element Code', submitted: *International Journal for Numerical Methods in Engineering*, 1993.

[3] W. H. Press, S. A. Teukolsky, B. P. Flannery, and W. T. Vetterling, *Numerical Recipes in C,* Cambridge University Press, Cambridge, 1986.

[4] Netlib, an online data base, netlib@ornl.gov.

[5] S. Pissanetsky, *Sparse Matrix Technology*, Academic Press Inc., London, 1984

[6] Fernando L. Alvarado, William F. Tinney, and Mark K. Enns, 'Sparsity in Large-Scale Network Computation', *Control and Dynamic Systems*, **41**, 207-271, 1991.

[7] Nicholson Keith W., *Elementary Linear Algebra with Applications*, Wadsworth Publisher of Canada Ltd., Toronto, 1986.

APPENDIX I

NOMENCLATURE

abstract object	An object with one or more pure virtual member functions.
class	In C++, a class is the same as a structure from C. However, the class has private access by default. In C++, the terms class and object are often used interchangeably.
fill-in	Fill-in occurs in sparse matrix operations when a perviously zero term has a non-zero term assigned to it.
instance	An instance is roughly equivalent to a variable in a non-object oriented language.
member function	A function that is defined within an object is said to be a meber function of that object.
object	The object roughly equivalent to the type in a non-object oriented language but it is much more powerful.
pure virtual function	A virtual function that exists only in definition. It has no body.
virutal function	A virtual function has the feature of being called independently of the level in the heirarchy at which the object is being accessed. It is used to implement polymorphism.

APPENDIX II

USER'S GUIDE

User's Guide For Matrix Class Library

Introduction

This manual is divided into two sections. The first section describes the class hierarchy so that additional classes may easily be incorporated into it. The second section details the implementation of the functions. In the second section, the emphasis is on how to use the library.

Class Hierarchy

Figure 1 shows the class structure. As by usual standard, the arrows point from the child class to its parent class.

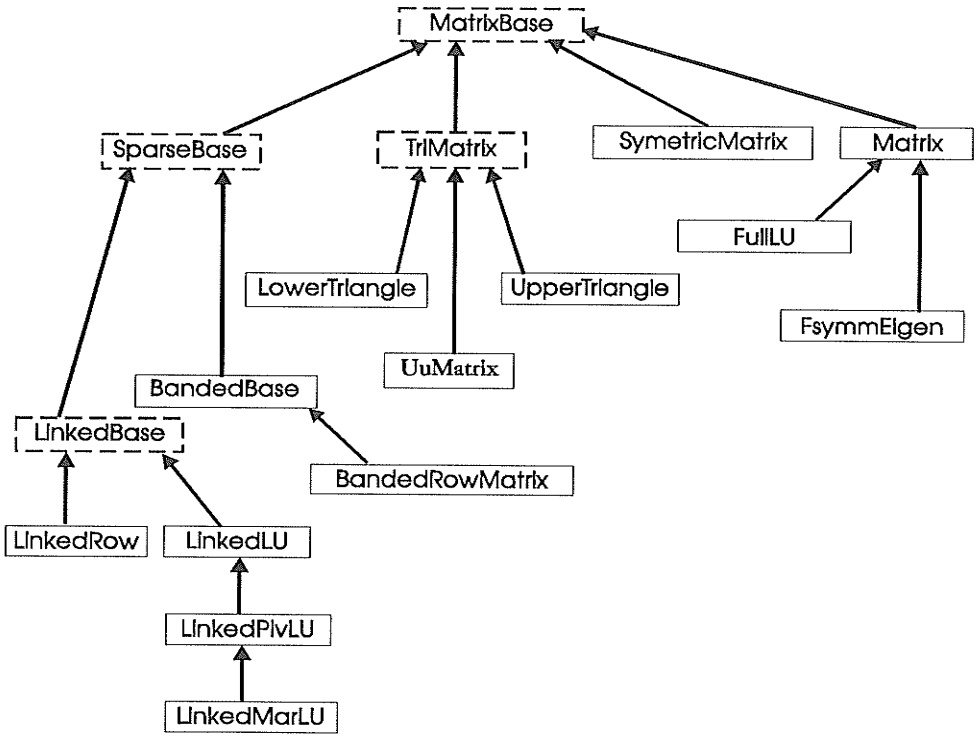

Figure 1. Matrix Class Hierarchy

Abstract classes are shown with a dashed box, while normal classes are shown with a solid box.

MatrixBase

MatrixBase serves as the base class for all the matrix types. It provides the framework for the member function which all derived types must define. It also stores the # of rows, # of columns, and the base. It also provides, functions for logging errors and testing for zero.

Member Functions

Constructor MatrixBase()	Default Constructor. Sets # of rows and columns=0.
Constructor MatrixBase(MatSize R, MatSize C, char IsTemp)	Normal Constructor for in process creation.
Constructor MatrixBase(ifstream & Source)	Constructor for creating Matrices from a file.
Destructor ~MatrixBase()	Virtual destructor.
operator+ (MatrixBase& Arg)	Matrix Addition operator. Branches to MAdd routine.
operator- (MatrixBase& Arg)	Matrix Subtraction operator. Branches to MSub routine.
operator* (MatrixBase& Arg)	Matrix Multiplication operator. Branches to MMul.
operator* (MatPrec X)	Scalar to Matrix Multiplication operator. Branches to CMul.
operator/ (MatPrec X)	Scalar to Matrix Division operator. Branches to Cdiv.
operator+= (MatrixBase& Arg)	Branches to MAdd.
operator-= (MatrixBase& Arg)	Branches to MSub.
operator*= (MatrixBase& Arg)	Branches to MMul.
operator/= (MatPrec X)	Branches to CDiv.
operator*= (MatPrec X)	Branches to CMul
operator= (MatrixBase& Arg)^{PV}	Equality operator. It can assign matrices of all types to its current instance.
IsSymetric()^{V}	Tests if matrix is symmetric.
MemError(char * mat)	Logs memory allocation errors.
SizeError(char * mat)	Logs matrix sizing errors.
IncTError(char * mat)	Logs mixed matrix type errors.
InvError(char *mat)	Logs matrix inversion errors.
AccessError(char * mat)	Logs bounds errors.
MAdd(MatrixBase & Arg, MatrixBase & Dest)^{PV}	Adds the matrix Arg to its instance and stores the results in the matrix Dest.
MSub(MatrixBase & Arg, MatrixBase & Dest)^{PV}	Subtracts the matrix Arg from its instance and stores the results in the matrix Dest.
MMul(MatrixBase & Arg, MatrixBase & Dest)^{PV}	Multiplies the matrix Arg with its instance and stores the results in the matrix Dest.
CMul(MatPrec X, MatrixBase & Dest)^{PV}	Multiplies each element in the instance by the scalar X .
CDiv(MatPrec X, MatrixBase & Dest)^{PV}	Divides each element in the instance by the scalar X .
CreateTempMat(_MatrixOp Operation, MatrixBase & Arg)	Creates a "temporary" matrix to be used as "Dest" for MAdd, MSub,...., depending upon the type of operation.
SetTemp()	Sets the temporary flag.
ReSetTemp()	Resets the temporary flag.

MatrixType()	Returns the type of storage scheme.
GetVal(MatSize R, MatSize C)[PV]	Returns the value of element (R,C). This routine has full error checking to insure that the bounds are not exceeded.
QGetVal(MatSize R, MatSize C)[PV]	Returns the value of element (R,C). Does not do any error checking.
PutVal(MatSize R, MatSize C)[PV]	Sets the value of element (R,C). This routine has full error checking to insure that the bounds are not exceeded.
QPutVal(MatSize R, MatSize C)[PV]	Sets the value of element (R,C). This routine does not do any error checking.
IsSparse()[V]	Returns true if the matrix is derived from SparseBase otherwise, it returns false.
IsTemp()	Returns true if the temporary flag has been set otherwise, it returns false.
IsSquare()	Returns true if the number of rows equals the number of columns otherwise, it returns false.
operator[](MatSize row)	Returns the matrix manipulator with its row indicator set to "row".
operator()(int x,int y)	Returns the matrix manipulator with its row indicator set to "x" and its column indicator set to "y".
GetNumRow()	Returns the number of rows.
GetNumCol()	Returns the number of columns.
CompareSize(MatrixBase & Arg)	Compares the number of rows and columns of its instance to the instance given by Arg and returns true if they are equal.
IsSame(MatrixBase & Arg)	Compares the matrix type of its instance to that of the instance given by Arg and returns true if the are equal.
GetRow(MatSize R)	Returns row "R" of its instance.
PutRow(MatSize R, VectorBase & row)	Inserts the row given by "row" into row "R" of its instance.
GetCol(MatSize C)	Returns the column "C" of its instance.
PutCol(MatSize C, VectorBase & col)	Inserts the column given by "col" into column "C" of its instance.
ident(MatPrec val=0)[PV]	Changes its instance into an "identity matrix" with the value "val" along the diagonal.
SetTol(MatPrec tol)	Sets the tolerance used in computing equality.
SetBase(MatSize B)	Sets the base used for all instances of all matrix types.
IsZero(MatPrec Val)	Tests whether the value "Val" is zero using the tolerance. Returns true if "-tolerance ≤ Val ≤ tolerance" is true.

V - Virtual Function, PV - Pure Virtual Function

Matrix

This class implements the full matrix storage structure. The largest number which "MatSize" can hold, determines the maximum size of the matrix. For the DOS version of this code, this maximum is the 65535 rows by 65535 columns. However, a matrix of this size could not be created because of memory constraints.

Member Functions

Constructor Matrix(MatSize R, MatSize C, char IsTemp=0)	
Constructor Matrix(MatSize R, MatSize C, MatPrec El1, MatPrec El2, ...)	
Constructor Matrix(Matrix & arg)	
Constructor Matrix(ifstream & Source)	
Destructor ~Matrix()	
AddRows(MatSize Rarg1, MatSize Rarg2, MatSize Rdest, MatPrec Mult, MatSize Start, MatSize End)	Private member function which adds the elements in "Rarg1" and "Rarg2" multiplied by "Mult" together and stores the results in "Rdest". The operations are carried out for the elements between the "Start"ing and "End"ing columns of these rows.
AddCols(MatSize Carg1, MatSize Carg2, MatSize Cdest, MatPrec Mult, MatSize Start, MatSize End)	Private member function which does the same operation as AddRows but does it for columns.
MulRows(MatSize Rsource, MatSize Rdest, MatPrec Mult, MatSize Start, MatSize End)	Private member function which multiplies row "Rsource" by "Mult" and stores the results in the row indicated by "Rdest". It does this for the columns between the "Start" and "End", inclusive.
MulCols(MatSize Csource, MatSize Cdest, MatPrec Mult, MatSize Start, MatSize End)	This private member function does for columns, what "MulRows" dose for rows.
AddRows(MatSize Rarg1, MatSize Rarg2, MatSize Rdest, MatPrec Mult=1.0)	This is the public version of "AddRows". It takes its arguments in the user defined base.
AddCols(MatSize Carg1, MatSize Carg2, MatSize Cdest, MatPrec Mult=1.0)	This is the public version of "AddCols". It takes its arguments in the user defined base.
MulRows(MatSize Rsource, MatSize Rdest, MatPrec Mult)	This is the public version of "MulRows". It takes its arguments in the user defined base.
MulCols(MatSize Csource, MatSize Cdest, MatPrec Mult)	This is the public version of "MulCols". It takes its arguments in the user defined base.

TriMatrix

This abstract class, encapsulates the storage scheme used for triangular matrices. Both upper and lower matrices are derived from this class.

Member Functions

Constructor TriMatrix(MatSize R, MatSize C, char IsTemp)	This is the normal constructor.
Constructor TriMatrix(TriMatrix & arg)	The copy constructor.
Destructor ~TriMatrix()	The destructor frees up the storage space.
AddRows(MatSize Rarg1, MatSize Rarg2, MatSize Rdest, MatPrec Mult=1.0)	This function performs the same operation as the function in the Matrix class of the same name. NOTE: there is no private version of this function
AddCols(MatSize Carg1, MatSize Carg2, MatSize Cdest, MatPrec Mult=1.0)	This function performs the same operation as the function in the Matrix class of the same name. NOTE: there is no private version of this function
MulRows(MatSize Rsource, MatSize Rdest, MatPrec Mult)	This function performs the same operation as the function in the Matrix class of the same name. NOTE: there is no private version of this function
MulCols(MatSize Csource, MatSize Cdest, MatPrec Mult)	This function performs the same operation as the function in the Matrix class of the same name. NOTE: there is no private version of this function

LowerTriangle

This class is derived from TriMatrix. It is the implementation of the parent class in its lower triangular form.

Member Functions

Constructor LowerTriangle(MatSize R, MatSize C, char IsTemp=0)	
Destructor ~LowerTriangle()	

UpperTriangle

This class is derived from TriMatrix and implements the upper triangular matrix version of those routines.

Member Functions

Constructor
UpperTriangle(MatSize R, MatSize C, char IsTemp)
Destructor
~UpperTriangle()

UuMatrix

This class implements a UTU matrix decomposition scheme. As such, it can only hold symmetric matrices. Also, the diagonal terms must be positive numbers when using real elements.

Member Functions

Constructor UuMatrix(MatSize R, MatSize C, char IsTemp=0)	Normal constructor.
Constructor UuMatrix(MatrixBase & ARG)	Converts other symmetric matrices to this type of matrix.
Destructor ~UuMatrix()	
UnPack()	Returns a full matrix holding the original, undecomposed matrix.
Pack(MatrixBase & ARG)	Decomposes the matrix "ARG" into its instance. Note ARG must be symmetric.

SymmetricMatrix

This class implements a storage system for symmetric matrices. As such, it holds only half of the elements.

Member Function	
Constructor SymmetricMatrix(MatSize R, MatSize C, char IsTemp=0)	Creates a symmetric matrix of size R X C. If IsTemp is true, the temporary flag is set.
Destructor ~SymmetricMatrix()	
IsSymetric()	Returns the value true.

FullLU

This class performs a LU decomposition for full matrices that cannot be decomposed by *UuMatrix*. This class contains the fastest solver and inversion routines for full matrices. The LU decomposition routine also performs partial pivoting to help maintain stability. This matrix is stored as a matrix of type *Matrix* until the inversion function, solver or the LU decomposition routine is called. At this point, the matrix is decomposed into the LU form. In the LU form, non of the matrix operators effect the stored data.

Member Functions	
Constructor FullLU(MatSize R, MatSize C, char IsTemp=0)	Create a matrix of size R X C.
Constructor FullLU(MatrixBase & Mat)	Copy the contents of any matrix into the instance being created. This routine uses the most general access methods.
Constructor FullLU(Matrix & Mat)	Copy the contents of a matrix of type *Matrix* into the current instance. This routine is more efficient than the previous constructor.
Destructor ~FullLU()	
void LUForm()	Performs a LU decomposition if the matrix is not already in the LU form.
void RowForm()	Reverses the LU decomposition returning the matrix back to an undecomposed state if the matrix is already decomposed.
MatPrec det()	Calculates the determinant of the matrix. If the matrix is not already decomposed, it is decomposed into the LU form.
MatrixBase & inv()	Calculates the inverse of the matrix. If the matrix is not already decomposed, it is decomposed into the LU form.
MatrixBase & trans()	Returns the transpose of the matrix.

Matrix Base & solve(MatrixBase & COL)	Return the solution to **Ax=COL**. Where **A** is the current instance, x is the solution and **COL** is the **RHS** matrix. If more than one set of equations is to be solved, **COL** can hold them in column order.

FSymmEigen

This class implements a QR algorithm to find the eigen values and vectors of symmetric matrices. It is similar to the *FullLU* class in that it also has two forms. The first form is the same as that use by Matrix. The second form occurs after the QR routine has been called. In this form, the previous matrix has been destroyed and replace by the eigen vectors in column form. Thus, the first column of the "decomposed" matrix hold the eigen vector to the first eigen value.

Member Functions

Constructor FSymmEigen(MatSize R, MatSize C)	Creates a matrix of size R X C. This matrix can be initialized by using the function *PutVal* or the *operator []*.
Constructor FSymmEigen(MatSize R, MatSize C, MatSize N, MatPrec El1, MatPrec El2, ...)	Creates a matrix of size R X C and initializes it with the elements *El1*.
Constructor FSymmEigen(MatrixBase & arg)	Copies the matrix **arg** into the instances data structure .
Destructor ~FSymmEigen()	
Matrix & GetEigenValue()	Returns all the eigen value in a column matrix.
Matrix & GetEigenVector()	Returns all the eigen values in a matrix of type *Matrix*.
Matrix & GetEigenVector(MatSize N)	Returns the *Nth* eigen vector in a column matrix

SparseBase

This class lays out the framework to which sparse storage schemes are to conform. It defines several functions which allow efficient inter-sparse type matrix operations.

Member Functions

Constructor
SparseBase(MatSize R, MatSize C, char IsTemp)
Destructor
SparseBase(MatSize R, MatSize C, char IsTemp)
IsSparse()
GetRowRange(MatSize row, MatSize & StartCol, MatSize & EndCol)

LinkBase

This class provides the base functions for maintaining and manipulating the linked list structure for the sparse storage classes based upon linked lists. This class uses the concept of a "major" and "minor". A major is the index by which the data is stored, and a minor is the other index. For example, in a row based list, the row index is the major and the column index is the minor.

Member functions

Constructor
LinkedBase(MatSize R, MatSize C, MatSize deep, char IsTemp=0)
Destructor
~LinkedBase()

Reset()	The following functions, up to MAdd, work as a set. They are all related in that the all work with the current list position stored in the current instance. This function resets all the list position pointers for the current instance.
CurVal()	This function returns the value of the element pointed to by the current list pointer.
CurMinor()	This function returns the value of the minor index pointed to by the current list position pointer.
NextVal()	This function returns the value of the element pointed to by the next list position pointer.
NextMinor()	This function returns the value of the minor index pointed to by the next list position pointer.
SetMajor(MatSize M)	This function resets the current and next list pointers to point to the beginning to the major "M".

operator ++(int)	This operator is used to traverse a major. It checks to see if the major has cycled. If it has, it sets a flag to indicate this.
operator int()	This is the default type conversion operator. It returns the value of the cycled flag. Thus, it can be used to check if the current major list has cycled.
Insert(MatSize Minor, MatPrec Val)	This function inserts an entry between the elements indicated by the current list position and the next list position. It does not check if the minor of the inserted element already exists.
MAdd(MatrixBase & Arg, MatrixBase & Dest)	This function is specialized to add two matrices of identical types based upon this class.
MSub(MatrixBase & Arg, MatrixBase & Dest)	This function is specialized to subtract two matrices of identical types based upon this class.
MMul(MatrixBase & Arg, MatrixBase & Dest)	This function is specialized to multiply two matrices of identical types based upon this class. This function is valid only if the major index represents rows.
CMul(MatPrec X, MatrixBase & Dest)	This function is specialized to multiply a constant and a matrix derived this class.
CDiv(MatPrec X, MatrixBase & Dest)	This function is specialized to multiply a constant and a matrix derived this class.
ReSize(unsigned long Size)	This function expands the memory pool and initializes the free space list.
CompareSame(LinkedBase & arg)	This function compares two matrices of the same type that are derived from this class. It returns true if they are the same.
void Zero()	This function "zeros" the matrix. That is it adjusts the linked list so that there are no elements in them.
float Sparseness()	This function returns the sparseness of the matrix. The sparseness is defined as the number of non-zero elements divided by the maximum number of possible elements.

LinkedRow

This class implements the row based linked list sparse matrix class. Most of its member functions are directly supplied by *LinkedBase*.

Member Function

Constructor

LinkedRow(MatSize R, MatSize C, MatSize
NumPer=0, char IsTemp=0)

Destructor

~LinkedRow()

LinkedLU

This class forms the basis of the LU decomposition classes for sparse matrices. It also adds the additional data and member functions necessary to maintain the LU structure. The primary problem with using this class is that very large amounts of fill-in will occur in most cases. The *LinkedMarLU* class is a much better choice for performing sparse LU decompositions.

Member Functions

Constructor LinkedLU(LinkedRow &Mat, char Form=0)	This constructor is used to create a LU matrix from an existing row based linked list matrix. If the *Form* parameter is non-zero, then the matrix **MAT** will be decomposed by the constructor as well.
Constructor LinkedLU(MatSize R, MatSize C, MatSize deep, char Form=0, char IsTemp=0)	This constructor creates initializes and allocates space for a matrix of size R X C with *deep* elements per row.
Constructor LinkedLU(ifstream & Source)	This constructor creates a matrix using the data is the file stream *Source*. First, *Source* is traversed to find the total number of elements and the size of the matrix. Then during the second pass, the elements are read in and stored.
Destructor ~LinkedLU()	
void LUForm()	This function performs the LU decompositions if the matrix is not already decomposed.
void RowForm()	This function reverses the decomposition function if the matrix is not already in row form.
LinkedRow &GetL()	If the matrix is decomposed, this function returns the lower triangular matrix
LinkedRow &GetU()	If the matrix is decomposed, this function returns the upper triangular matrix

LinkedPivotLU

The *LinkedLU* class provides the basic LU decomposition routine. However, it does not perform any pivoting. Thus, if the matrix is near singular or if there is a zero

diagonal element, the LU decomposition algorithm in *LinkedLU* will fail. This class adds the basic data structures necessary to keep track of the pivot elements used. The LU decomposition routine here uses full pivoting. The criteria for the pivot element is simply the largest element in the matrix. For general sparse matrices, this class is of no value. The sparseness after the decomposition is typically 40 to 50 times higher than it was before. The net result is that the routine becomes too slow and the storage requirements become too great.

Member Functions

Constructor LinkedPivotLU(LinkedRow &Mat, char Form=0)	This constructor is used to create a LU matrix from an existing row based linked list matrix. If the *Form* parameter is non-zero, then the matrix **MAT** will be decomposed by the constructor as well.
Constructor LinkedPivotLU(MatSize R, MatSize C, MatSize deep, char Form=0, char IsTemp=0)	This constructor creates initializes and allocates space for a matrix of size R X C with *deep* elements per row.
Constructor LinkedPivotLU(ifstream & Source)	This constructor creates a matrix using the data is the file stream *Source*. First, *Source* is traversed to find the total number of elements and the size of the matrix. Then during the second pass, the elements are read in and stored.
Destructor ~LinkedPivotLU()	

LinkedMarLU

This is the most useful of the linked list LU routines. It has the least amount of fill in of the three classes. The LU decomposition routine is based upon the MarKowitz pivoting strategy. In this pivoting algorithm, a pivoting tolerance is used to control how many eligible pivot choices are present. The lower the tolerance value, the more possible pivoting elements are possible. Out of the possible elements, the one which has the lowest theoretical fill-in is used.

Member Functions

Constructor LinkedMarLU(LinkedRow &Mat, char Form=0)	This constructor is used to create a LU matrix from an existing row based linked list matrix. If the *Form* parameter is non-zero, then the matrix **MAT** will be decomposed by the constructor as well.

Constructor LinkedMarLU(MatSize R, MatSize C, MatSize deep, char Form=0, char IsTemp=0)	This constructor creates initializes and allocates space for a matrix of size R X C with *deep* elements per row.
Constructor LinkedMarLU(ifstream & Source)	This constructor creates a matrix using the data is the file stream *Source*. First, *Source* is traversed to find the total number of elements and the size of the matrix. Then during the second pass, the elements are read in and stored.
Destructor ~LinkedMarLU()	
static void SetPivotTol(MatPrec X)	This function set the Markowitz pivoting tolerance to X.

BandedBase

This class serves as the base class for banded matrices. This class is similar to the LinkedBase class in that it also uses the concept of a major to allow both the creation of row oriented and column oriented banded matrices.

Member Function	
Constructor BandedBase(MatSize Major, MatSize Minor, MatSize B, char IsTemp=0)	
Destructor ~BandedBase()	
NumMinor(MatSize Major)	This function returns the number of minors in the major band given by "Major".
ExpandBand(MatSize Increase)	This function increases the bandwidth by the amount "Increase".
ShrinkBand(MatSize Decrease)	This function decreases the bandwidth by the amount "Decrease".

BandedRowMatrix

This is the row oriented implementation of banded matrix. Most of its member functions are supplied by *BandedBase*.

Member Functions

Constructor

BandedRowMatrix(MatSize R, MatSize C,
MatSize B, char IsTemp=0)

Destructor

~BandedRowMatrix()

Example Usage

In this section, code example will be given to illustrate how to do many common matrix operations with the hierarchy. Where appropriate, only one example will be given using the class of type *Matrix*. Other classes can be used simply by changing the objects that are being operated on.

```
Matrix A(3,3);                  //declare an instance of object Matrix. size is 3 X 3.
Matrix B(3,3);
ifstream Input("Input.file");   //open the input file. The first file format follows
Matrix C(Input);                // create a matrix using the data in the
                                // file stream Input

A.PutVal(1,1,5.0)               // Assigns the value 5.0 to element (1,1)
A[1][2]=4.0;                    // an alternative method of assigning values
A[1][3]=5.0;                    // to elements.

// perform other initialization of A

B=A*3+A*C/5;

cout << B[1][1]<<B.GetVal(1,2);  //Two methods of retrieving element from
                                 // a matrix object.
                                 // In this case it writes them to the standard
                                 // output stream.
```

The above example illustrates how to perform several basic matrix operations. In this example, matrix **B** is assign the value of $3 \cdot \mathbf{A} + \mathbf{A} \cdot \mathbf{C}/5$. The format of the input stream is :

```
Row Col
Row Col Val
Row Col Val
.....
.....
```

The first line contains the size of the matrix. The following lines contain the elements of the matrix. These line have the elements row and column location and its value.

```
Matrix A(3,3);
Matrix Ainv(3,3);
Matrix B(3,1);
Matrix Sol(3,1);

/* Initialize A
.....
.....
*/

/* Initialize B
.....
.....
*/

FullLU ALU(A);          //Copy matrix A into the LU object
Sol=ALU.solve(B);       //LU decomposes ALU and solves system of
                        //equations
Ainv=ALU.inv();         //Find the inverse of A by using the LU
                        //decomposition method.
```

The above example illustrates how to LU decompose a matrix without destroying the original. If the original matrix is not needed, the matrix **A** could have been dispensed with and all operations involving **A** could have been replaced by operations of **ALU**.

```
ifstream InFileA("Inputfile.A");
ifstream InFileRHS("inputfile.RHS");

LinkedMarLU A(InFile);      // read in the square matrix
Matrix RHS(InFileRHS);      //read in the Right Hand Side matrix

//Create a matrix to hold the solution that is the same size as the
// right hand side matrix.
Matrix Sol(RHS.GetNumRow(), RHS.GetNumCol() );

//Set the Markowitz pivoting tolerance
LinkedMarLU::SetPivotTol(0.0000001);

//solve the set of equations
Sol=A.solve(RHS);
```

The above example shows how to use the LinkedMarLU class. The tolerance used here should be set on an application by application basis. Note, in this example, the original matrix **A** is not saved.

APPENDIX III

C++ HEADER FILES

BANDED.H

```
/*****************************************************************
*                                                                *
* This file is part of MATCLASS,a Matrix Class Library,          *
*  written by:                                                   *
*        Gordon W. Zeglinski                                     *
*                                                                *
*                                                                *
*   (c) 1992, (beta test version)                                *
*                                                                *
*****************************************************************/

// this file contains the sparse matrix base data structure

#if !defined(__BANDED_H)
#define __BANDED_H

#include <sparse.h>


class BandedRowMatrix:public BandedBase{
  static ClassMarker ClassID;
protected:

  virtual void MAdd(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MSub(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MMul(MatrixBase & Arg, MatrixBase & Dest);
  virtual void CMul(MatPrec X, MatrixBase & Dest);
  virtual void CDiv(MatPrec X, MatrixBase & Dest);
  virtual MatrixBase& CreateTempMat(_MatrixOp Opperation,
            MatrixBase & Arg);

  virtual MatPrec BGetVal(MatSize R, MatSize C);
  virtual void BPutVal(MatSize R, MatSize C, MatPrec VAL);


//void AddRows(MatSize Rarg1, MatSize Rarg2, MatSize Rdest, MatPrec Mult);
//void AddCols(MatSize Carg1, MatSize Carg2, MatSize Cdest, MatPrec Mult);

//void MulRows(MatSize Rsource, MatSize Rdest, MatPrec Mult);
//void MulCols(MatSize Csource, MatSize Cdest, MatPrec Mult);

//void SwapRow(MatSize To, MatSize From);
//  void SwapCol(MatSize To, MatSize From);

public:
  BandedRowMatrix(MatSize R, MatSize C, MatSize B, char IsTemp=0);
```

```
  ~BandedRowMatrix();

  virtual MatrixBase& operator= (MatrixBase& Arg);
  virtual const MatType MatrixType();
  static const MatType GetID(){return ClassID;}
  virtual MatPrec GetVal(MatSize R, MatSize C);
  virtual void PutVal(MatSize R, MatSize C, MatPrec VAL);
  virtual void ident(MatPrec val=1.0);

  virtual MatPrec QGetVal(MatSize R, MatSize C);
  virtual void QPutVal(MatSize R, MatSize C, MatPrec VAL);
  virtual void* GetAddress(){return (void *) this;}

  virtual void GetRowRange(MatSize row, MatSize & StartCol, MatSize & EndCol);

  MatrixBase & Solve(MatrixBase & COL);
};

#endif
```

FEIGN.H

```
/*************************************************************************
*                                                                        *
* This file is part of MATCLASS,a Matrix Class Library,                  *
*  written by:                                                           *
*        Gordon W. Zeglinski                                             *
*                                                                        *
*                                                                        *
*   (c) 1992, (beta test version)                                        *
*                                                                        *
*************************************************************************/

// fegien.h
// The header file for the Matrix class


#if !defined( __FEIGEN_H)
#define __FEIGEN_H

#include <matrix.h>

class FSymmEigen: public Matrix{
  static ClassMarker ClassID;

protected:
  enum Forms{Normal, Decomposed, EigenValue};

  Forms Form;
  MatPrec *Diag, *OffDiag;

  void tqli();
  void tred2();

public:
  FSymmEigen(MatSize R, MatSize C);
  FSymmEigen(MatSize R, MatSize C, MatSize N, MatPrec El1, MatPrec El2, ...);
  FSymmEigen(MatrixBase & arg);

  ~FSymmEigen();

  virtual MatPrec QGetVal(MatSize R, MatSize C){
      return (Form==Normal) ? Matrix::QGetVal(R,C) :0;
  }

  virtual void QPutVal(MatSize R, MatSize C, MatPrec VAL){
      if(Form==Normal) Matrix::QPutVal(R,C,VAL);
  }
  virtual MatPrec BGetVal(MatSize R, MatSize C){
      return (Form==Normal) ? Matrix::BGetVal(R,C) :0;
```

```
  }
  virtual void BPutVal(MatSize R, MatSize C, MatPrec VAL){
      if (Form==Normal) Matrix::BPutVal(R,C,VAL);
  }

  Matrix & GetEigenValue();
  Matrix & GetEigenVector();
  Matrix & GetEigenVector(MatSize N);
  void ReduceByHouseHolder();
  void SolveEigenProb();
};

#endif
```

FULLU.H

```
/*****************************************************************
*                                                                *
* This file is part of MATCLASS,a Matrix Class Library,          *
*  written by:                                                   *
*         Gordon W. Zeglinski                                    *
*                                                                *
*                                                                *
*   (c) 1992, (beta test version)                                *
*                                                                *
*****************************************************************/

// FullLu.h
// The header file for the Matrix class


#if !defined( __FULL_LU_H)
#define __FULL_LU_H

#include <matrix.h>


class FullLU:public Matrix{
  static ClassMarker ClassID;
protected:
  char    D:1;
  char    InLU:1;
  MatSize * Indx;

  virtual void MAdd(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MSub(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MMul(MatrixBase & Arg, MatrixBase & Dest);
  virtual void CMul(MatPrec X, MatrixBase & Dest);
  virtual void CDiv(MatPrec X, MatrixBase & Dest);
  virtual MatrixBase& CreateTempMat(_MatrixOp Opperation,
            MatrixBase & Arg);

  void Decompose();
  void lubksb(Matrix & Sol);
  void ludcmp();
public:
  FullLU(MatSize R, MatSize C, char IsTemp=0);
  FullLU(MatrixBase & Mat);
  FullLU(Matrix &Mat);
  ~FullLU();

  virtual MatrixBase& operator= (MatrixBase& Arg);

  virtual const MatType MatrixType();
  static const MatType GetID(){return ClassID;}
  virtual MatPrec GetVal(MatSize R, MatSize C);
```

```
  virtual void PutVal(MatSize R, MatSize C, MatPrec VAL);
  virtual void ident(MatPrec val=1.0);

  virtual MatPrec QGetVal(MatSize R, MatSize C){
      return *(*(Storage+R-Base)+C-Base);
  }
  virtual void QPutVal(MatSize R, MatSize C, MatPrec VAL){
      *(*(Storage+R-Base)+C-Base)=VAL;
  }

  virtual MatPrec BGetVal(MatSize R, MatSize C){
      return *(*(Storage+R)+C);
  }

  virtual void BPutVal(MatSize R, MatSize C, MatPrec VAL){
      *(*(Storage+R)+C)=VAL;
  }


  virtual void* GetAddress(){return (void*)this;}

  MatPrec det();
  MatrixBase & inv();
  MatrixBase & trans();
  MatrixBase & Solve(MatrixBase & COL);

  void LUForm();
  void RowForm();
};

#endif
```

LINKBASE.H

```
/*****************************************************************
*                                                                *
* This file is part of MATCLASS,a Matrix Class Library,          *
*  written by:                                                   *
*         Gordon W. Zeglinski                                    *
*                                                                *
*                                                                *
*   (c) 1992, (beta test version)                                *
*                                                                *
*****************************************************************/

// this file contains the linked base data structures to be used
// in sparse matrices using linked lists

#if !defined(__LINKEDBASE_H)
#define __LINKEDBASE_H


#include "sparse.h"



class SparseVec;
class SparseListIterator;



////////////////////////
// Linked Base
////////////////////////

class LinkedBase:public SparseBase{

protected:
  MatPrec _HUGE_PTR *Storage;
  MatSize _HUGE_PTR *Minor;
  MatSize _HUGE_PTR *List;
  MatSize _HUGE_PTR *Start;
  MatSize Free;
  MatSize Depth;
  unsigned long Size;
  MatSize CurMajor, CurListPos, NextListPos;
  int Valid    :1;
  int          :15;

  void Reset();
  MatPrec CurVal();
  MatSize CurMinor();
```

```
  MatPrec NextVal();
  MatSize NextMinor();

  void SetMajor(MatSize M);
  void operator ++(int);
  void operator ++();

  operator int();

  void Insert(MatSize Minor, MatPrec Val);
  void Remove();

  void JumpTo(MatSize TO);
  MatPrec ValAtHead(MatSize R);
  MatSize MinorAtHead(MatSize R);
  void AddToHead(MatSize R, MatPrec V);
  void InsertAtHead(MatSize R, MatSize C, MatPrec V);
  void RemoveFromHead(MatSize R);

  virtual void MAdd(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MSub(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MMul(MatrixBase & Arg, MatrixBase & Dest);
          void MMul(LinkedBase & Arg, LinkedBase & Dest);
// This next function should really be a static Function !!!
/*
      void MMul(SparseListIterator &LHS, SparseListIterator &RHS,
                  LinkedBase & Dest);
*/
  virtual void CMul(MatPrec X, MatrixBase & Dest);
  virtual void CDiv(MatPrec X, MatrixBase & Dest);

  virtual MatrixBase& CreateTempMat(_MatrixOp Opperation,
              MatrixBase & Arg)=0;

  void ReSize(unsigned long Size);
  int CompareSame(LinkedBase & arg);
  void ListScalarSub(MatPrec Val, MatSize IdxA);

  virtual MatPrec BGetVal(MatSize R, MatSize C);
  virtual void BPutVal(MatSize R, MatSize C, MatPrec VAL);
  void FillDiag(MatPrec Val);

  static MatSize MaxIndex;


public:
  LinkedBase(MatSize R, MatSize C,MatSize deep, char IsTemp=0);
  LinkedBase(LinkedBase & Mat);
  LinkedBase(ifstream & Source);
  ~LinkedBase();

  virtual MatrixBase& operator= (MatrixBase& Arg)=0;
  virtual const MatType MatrixType()=0;
  virtual MatPrec GetVal(MatSize R, MatSize C);
```

```
  virtual void PutVal(MatSize R, MatSize C, MatPrec VAL);
  virtual void ident(MatPrec val=1.0)=0;

  virtual MatPrec QGetVal(MatSize R, MatSize C);
  virtual void QPutVal(MatSize R, MatSize C, MatPrec VAL);
  virtual void* GetAddress()=0;

  int GetDepth();
  void RandomFill();

  virtual void GetRowRange(MatSize row, MatSize & StartCol, MatSize & EndCol)=0;

  void MajorToMinorList();
  void ShowDiff(ostream &Out, LinkedBase& arg);
  void Dump(ostream &Out);
  virtual void Zero();
  float Sparseness();


  friend class SparseListIterator;
  friend class MinorIterator;
  friend class MajorIterator;
};


//////////////////////
// Inline functions for LinkedBase
//////////////////////

inline void LinkedBase::Reset(){
  CurMajor=0;
  CurListPos=*(Start);
  NextListPos=*(List+CurListPos);
  Valid=0;
}

inline MatPrec LinkedBase::CurVal(){
  return *(Storage+CurListPos);
}

inline MatSize LinkedBase::CurMinor(){
  return *(Minor+CurListPos);
}

inline MatPrec LinkedBase::NextVal(){
  return *(Storage+NextListPos);
}

inline MatSize LinkedBase::NextMinor(){
  return *(Minor+NextListPos);
}

inline void LinkedBase::SetMajor(MatSize M){
  CurMajor=M;
```

```
  CurListPos=*(Start+M);
  NextListPos=*(List+CurListPos);
//if (CurMinor() >= NextMinor())
//      Valid=0;
//else
        Valid=1;
}

inline void LinkedBase::operator ++(int){
  CurListPos=NextListPos;
  NextListPos=*(List+CurListPos);

  if (CurMinor() >= NextMinor())
        Valid=0;
}

inline void LinkedBase::operator ++(){
  CurListPos=NextListPos;
  NextListPos=*(List+CurListPos);

  if (CurMinor() >= NextMinor())
        Valid=0;
}


inline LinkedBase::operator int(){
  return Valid;
}

inline void LinkedBase::JumpTo(MatSize TO){
  *(List+CurListPos)=TO;
  NextListPos=TO;
}

inline MatPrec LinkedBase::ValAtHead(MatSize R){
  return (*(Storage+*(List+*(Start+R))));
}

inline MatSize LinkedBase::MinorAtHead(MatSize R){
  return (*(Minor+*(List+*(Start+R))));
}

inline void LinkedBase::AddToHead(MatSize R, MatPrec V){
  (*(Storage+*(List+*(Start+R))))+=V;
}

inline void LinkedBase::InsertAtHead(MatSize R, MatSize C, MatPrec V){
  MatSize TFree,NxLP;
        // check if storage is exhausted
  if (Free==(Size-1) )
        ReSize(Size+( (((Depth*row)*0.2)>1)? (MatSize)((Depth*row)*0.2):1));
  NxLP=*(List+*(Start+R));
  *(Storage+Free)=V;
  *(Minor+Free)=C;
```

```
  *(List+*(Start+R))=Free;
  TFree=(*(List+Free));
  *(List+Free)=NxLP;
  Free=TFree;
}

inline void LinkedBase::RemoveFromHead(MatSize R){
  MatSize TO,NxLP;

  NxLP=*(List+*(Start+R));
  TO=*(List+NxLP);
  *(List+NxLP)=Free;
  Free=NxLP;
  *(List+*(Start+R))=TO;
}

inline int LinkedBase::GetDepth(){
  return Depth;
}



//////////////////////
// Linked Row
//////////////////////

class LinkedRow:public LinkedBase{
  static ClassMarker ClassID;

  friend class LinkedLU;

protected:
  virtual void MMul(MatrixBase & Arg, MatrixBase & Dest);
  virtual MatrixBase& CreateTempMat(_MatrixOp Opperation,
                MatrixBase & Arg);

  virtual MatPrec BGetVal(MatSize R, MatSize C);
  virtual void BPutVal(MatSize R, MatSize C, MatPrec VAL);


public:
  LinkedRow(MatSize R, MatSize C, MatSize NumPer=1, char IsTemp=0);
  ~LinkedRow();

  virtual MatrixBase& operator= (MatrixBase& Arg);
  virtual const MatType MatrixType();
  static const MatType GetID(){return ClassID;}
  virtual MatPrec GetVal(MatSize R, MatSize C);
  virtual void PutVal(MatSize R, MatSize C, MatPrec VAL);
  virtual void ident(MatPrec val=1.0);

  virtual MatPrec QGetVal(MatSize R, MatSize C);
  virtual void QPutVal(MatSize R, MatSize C, MatPrec VAL);
```

```
  virtual void* GetAddress();


  virtual void GetRowRange(MatSize row, MatSize & StartCol, MatSize & EndCol);

  int operator==(MatrixBase & Arg);
};

/* left for future expansion
class LinkedCol:public LinkedBase{
  static ClassMarker ClassID;

public:
  LinkedCol(MatSize R, MatSize C, MatSize NumPer=0, char IsTemp=0);
  ~LinkedCol();

  virtual MatrixBase& operator= (MatrixBase& Arg);
  virtual const MatType MatrixType();
  static const MatType GetID(){return ClassID;}
  virtual MatPrec GetVal(MatSize R, MatSize C);
  virtual void PutVal(MatSize R, MatSize C, MatPrec VAL);
  virtual void ident(MatPrec val=1.0);

  virtual MatPrec QGetVal(MatSize R, MatSize C);
  virtual void QPutVal(MatSize R, MatSize C, MatPrec VAL);
  virtual void* GetAddress();


  virtual void GetRowRange(MatSize row, MatSize & StartCol, MatSize & EndCol);


};
*/


//-------------------------------------------------------------------
// Iterators for Linked List structures
//-------------------------------------------------------------------


class SparseListIterator{
protected:
  MatSize Along, CurListPos;
  LinkedBase *Mat;
  char Valid;
public:
  SparseListIterator(LinkedBase * LB);
  virtual ~SparseListIterator();

  operator int() {return Valid;}

  virtual void Next()=0;

  virtual MatPrec CurVal()=0;
```

```
  virtual MatSize CurIndex()=0;

  virtual MatPrec NextVal()=0;
  virtual MatSize NextIndex()=0;

  virtual void SetAlong(MatSize M)=0;
  virtual void operator ++(int)=0;
  virtual void Insert(MatSize Index, MatPrec Val)=0;
  virtual void Remove()=0;
  virtual void JumpTo(MatSize TO)=0;
  virtual void Reset()=0;
  virtual SparseListIterator* CreateAnother()=0;

  MatSize GetNumRow(){return Mat->GetNumRow();}
  MatSize GetNumCol(){return Mat->GetNumCol();}
};



//////////////////
// Major Iterator
//////////////////

class MajorIterator:public SparseListIterator{

public:
  MajorIterator(LinkedBase *LB);
  MajorIterator(MajorIterator & Arg);
  ~MajorIterator();

  void Next();

  MatPrec CurVal();
  MatSize CurIndex();

  MatPrec NextVal();
  MatSize NextIndex();

  void SetAlong(MatSize M);
  void operator ++(int);

  void Insert(MatSize Index, MatPrec Val);
  void Remove();

  void JumpTo(MatSize TO);

  void Reset(){};
  SparseListIterator* CreateAnother();
};



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

```
// inline Functions
//////////////////////


inline void MajorIterator::Next(){
  SetAlong(Along+1);
}

inline MatPrec MajorIterator::CurVal(){
  return *(Mat->Storage+CurListPos);
}

inline MatSize MajorIterator::CurIndex(){
  return *(Mat->Minor+CurListPos);
}

inline MatPrec MajorIterator::NextVal(){
  return *(Mat->Storage+*(Mat->List+CurListPos) );
}

inline MatSize MajorIterator::NextIndex(){
  return *(Mat->Minor+*(Mat->List+CurListPos));
}

inline void MajorIterator::SetAlong(MatSize M){
  Along=M;
  CurListPos=*(Mat->Start+M);
//NextListPos=*(Mat->List+CurListPos);
//if (CurIndex() >= NextIndex())
//      Valid=0;
//else
        Valid=1;
}

inline void MajorIterator::operator ++(int){
  CurListPos=*(Mat->List+CurListPos);

  if (CurIndex() >= NextIndex())
        Valid=0;
}

inline void MajorIterator::JumpTo(MatSize TO){
  *(Mat->List+CurListPos)=TO;
}

inline SparseListIterator* MajorIterator::CreateAnother(){
  return new MajorIterator(*this);
}



////////////////
// Minor Iterator
////////////////
```

```
class MinorIterator:public SparseListIterator{

  MatSize _HUGE_PTR *List,Count,NextListPos;


public:
  MinorIterator(LinkedBase *LB);
  MinorIterator(MinorIterator &Arg);
  ~MinorIterator();

  void Next();


  MatPrec CurVal();
  MatSize CurIndex();

  MatPrec NextVal();
  MatSize NextIndex();

  void SetAlong(MatSize M);

  void operator ++(int);


  void Insert(MatSize Index, MatPrec Val);
  void Remove();

  void JumpTo(MatSize TO);

  void Reset();

  SparseListIterator* CreateAnother();
};



//////////////////
// Inline functions
//////////////////



inline void MinorIterator::Next(){SetAlong(Along+1);}


inline MatPrec MinorIterator::CurVal(){
  return *(Mat->Storage+ *(List+CurListPos));
}

//MatSize CurIndex(){return *(Mat->Minor+ *(List+CurListPos));}

inline MatSize MinorIterator::CurIndex(){
  return CurListPos-1;
```

```
}

inline MatPrec MinorIterator::NextVal(){
  return *(Mat->Storage+*(List+NextListPos) );
}

//MatSize NextIndex(){return *(Mat->Minor+*(List+NextListPos));}

inline MatSize MinorIterator::NextIndex(){
  return NextListPos-1;
}

inline SparseListIterator* MinorIterator::CreateAnother(){
  return new MinorIterator(*this);
}



#endif
```

LINKLU.H

```
/*********************************************************************
*                                                                    *
* This file is part of MATCLASS,a Matrix Class Library,              *
*  written by:                                                       *
*         Gordon W. Zeglinski                                        *
*                                                                    *
*                                                                    *
*   (c) 1992, (beta test version)                                    *
*                                                                    *
*********************************************************************/

// this file contains the linked LU data structures to be used
// in sparse matrices using linked lists

#if !defined(__LINKEDLU_H)
#define __LINKEDLU_H

#include <linkbase.h>



class LinkedLU: public LinkedBase{
  static ClassMarker ClassID;

protected:
  MatSize _HUGE_PTR *LStart;
  MatSize CurMinorIndex, CurMinorListPos, NextMinorListPos;

  char InLU           :1;
  char ValidMinorList   :1;
  char                :6;


  MatPrec MinorListCurVal(){return *(Storage+CurMinorListPos);}
  MatSize _CurMajor(){return *(Minor+CurMinorListPos);}

  MatPrec MinorListNextVal(){return *(Storage+NextMinorListPos);}
  MatSize _NextMajor(){return *(Minor+NextMinorListPos);}

  void SetMinorList(MatSize M){
      CurMinorIndex=M;
      CurMinorListPos=*(LStart+M);
      NextMinorListPos=*(List+CurMinorListPos);
      ValidMinorList=1;
  }
  // copied from parent do to some kind of inheritance problem
  void operator ++(int){
      CurListPos=NextListPos;
      NextListPos=*(List+CurListPos);
```

```
        if (CurMinor() >= NextMinor())
            Valid=0;

    }

    void operator ++(){
        CurListPos=NextListPos;
        NextListPos=*(List+CurListPos);

        if (CurMinor() >= NextMinor())
            Valid=0;

    }

    void IncL(){
        CurMinorListPos=NextMinorListPos;
        NextMinorListPos=*(List+CurMinorListPos);

        if (_CurMajor() >= MaxIndex)
            ValidMinorList=0;
    }

    int MinorListValid(){return ValidMinorList;}
    void InsertInMinorList(MatSize Major, MatPrec Val);
    void InsertInMinorList(MatSize Here, MatSize Major, MatPrec Val){
        *(Storage+Here)=Val;
        *(Minor+Here)=Major;
        *(List+CurMinorListPos)=Here;
        *(List+Here)=NextMinorListPos;
        NextMinorListPos=Here;
    }

    virtual void MAdd(MatrixBase & Arg, MatrixBase & Dest);
    virtual void MSub(MatrixBase & Arg, MatrixBase & Dest);
    virtual void MMul(MatrixBase & Arg, MatrixBase & Dest);
    virtual void CMul(MatPrec X, MatrixBase & Dest);
    virtual void CDiv(MatPrec X, MatrixBase & Dest);

    virtual MatrixBase& CreateTempMat(_MatrixOp Opperation,
                MatrixBase & Arg);

    virtual MatPrec BGetVal(MatSize R, MatSize C);
    virtual void BPutVal(MatSize R, MatSize C, MatPrec VAL);

    virtual void Decompose();
    void ListScalarSub(MatPrec Val, MatSize IdxA);
    void LColtoRow();

  public:
    LinkedLU(LinkedRow &Mat,char Form=0);
    LinkedLU(MatSize R, MatSize C,MatSize deep,char Form=0, char IsTemp=0);
    LinkedLU(ifstream & Source);
    ~LinkedLU();
```

```
  void LUForm();
  void RowForm();

  virtual MatrixBase& operator= (MatrixBase& Arg);
  virtual const MatType MatrixType();
  static const MatType GetID(){return ClassID;}
  virtual MatPrec GetVal(MatSize R, MatSize C);
  virtual void PutVal(MatSize R, MatSize C, MatPrec VAL);
  virtual void ident(MatPrec val=1.0);
  MatrixBase& solve(MatrixBase &RightSide);
  MatrixBase& inv();

  virtual MatPrec QGetVal(MatSize R, MatSize C);
  virtual void QPutVal(MatSize R, MatSize C, MatPrec VAL);
  virtual void* GetAddress();


  virtual void GetRowRange(MatSize row, MatSize & StartCol, MatSize & EndCol);

  LinkedRow& GetL();
  LinkedRow& GetU();
  void Zero();
  float Sparseness();
};


#endif
```

LINKMARLU.H

```
/*************************************************************************
*                                                                        *
* This file is part of MATCLASS,a Matrix Class Library,                  *
*  written by:                                                           *
*          Gordon W. Zeglinski                                           *
*                                                                        *
*                                                                        *
*   (c) 1992, (beta test version)                                        *
*                                                                        *
*************************************************************************/

// this file contains the linked LU data structures with Markowitz Pivoting
// to be used in sparse matrices using linked lists

#if !defined(__LINKEDMARLU_H)
#define __LINKEDMARLU_H

#include <linpivlu.h>



class LinkedMarLU: public LinkedPivotLU{
  static ClassMarker ClassID;
  static MatPrec PivTol;

protected:
/*
  virtual void MAdd(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MSub(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MMul(MatrixBase & Arg, MatrixBase & Dest);
  virtual void CMul(MatPrec X, MatrixBase & Dest);
  virtual void CDiv(MatPrec X, MatrixBase & Dest);
*/
  virtual MatrixBase& CreateTempMat(_MatrixOp Opperation,
              MatrixBase & Arg);

  void Decompose();
  void ListMax(MatPrec &MaxVal, MatSize RowRet);
  void LUSetup(MatPrec &MaxVal, MatSize *RowCount, MatSize *ColCount,
         MatSize RowRet);
//void ListIntChg(MatSize IdxA, MatSize IdxB);
  void ListScalarSub(MatPrec Val, MatSize IdxA,
      MatSize * RowCount, MatSize * ColCount);

public:
  LinkedMarLU(LinkedRow &Mat,char Form=0);
  LinkedMarLU(MatSize R, MatSize C,MatSize deep,char Form=0, char IsTemp=0);
  LinkedMarLU(ifstream & Source);
  ~LinkedMarLU();
```

```
  virtual MatrixBase& operator= (MatrixBase& Arg);
  virtual const MatType MatrixType();
  static const MatType GetID(){return ClassID;}
  static void SetPivotTol(MatPrec X){PivTol=X;}

//MatrixBase& solve(MatrixBase& RHS);
//MatrixBase& inv();
//void Zero();
  virtual void* GetAddress();

};


#endif
```

LINKPIVLU.H

```
/*************************************************************************
*                                                                        *
* This file is part of MATCLASS,a Matrix Class Library,                  *
*  written by:                                                           *
*          Gordon W. Zeglinski                                           *
*                                                                        *
*                                                                        *
*   (c) 1992, (beta test version)                                        *
*                                                                        *
*************************************************************************/

// this file contains the linked LU data structures to be used
// in sparse matrices using linked lists

#if !defined(__LINKEDPIVLU_H)
#define __LINKEDPIVLU_H

#include <linklu.h>



class LinkedPivotLU: public LinkedLU{
  static ClassMarker ClassID;

protected:
  MatSize _HUGE_PTR *PivRow, _HUGE_PTR *PivCol;
/*
  virtual void MAdd(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MSub(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MMul(MatrixBase & Arg, MatrixBase & Dest);
  virtual void CMul(MatPrec X, MatrixBase & Dest);
  virtual void CDiv(MatPrec X, MatrixBase & Dest);
*/
  virtual MatrixBase& CreateTempMat(_MatrixOp Opperation,
              MatrixBase & Arg);

  void Decompose();
  void ListMax(MatPrec &MaxVal, MatSize &PivRow, MatSize &PivCol,
          MatSize RowRet);
  void ListIntChg(MatSize IdxA, MatSize IdxB);

public:
  LinkedPivotLU(LinkedRow &Mat,char Form=0);
  LinkedPivotLU(MatSize R, MatSize C,MatSize deep,char Form=0, char IsTemp=0);
  LinkedPivotLU(ifstream & Source);
  ~LinkedPivotLU();


  virtual MatrixBase& operator= (MatrixBase& Arg);
  virtual const MatType MatrixType();
```

```
  static const MatType GetID(){return ClassID;}
  MatrixBase& solve(MatrixBase& RHS);
  MatrixBase& inv();
  void Zero();
  virtual void* GetAddress();

  friend class LinkedMarLU;
};


#endif
```

MATBASE.H

```
/*************************************************************************
*                                                                        *
* This file is part of MATCLASS,a Matrix Class Library,                  *
*  written by:                                                           *
*         Gordon W. Zeglinski                                            *
*                                                                        *
*                                                                        *
*   (c) 1992, (beta test version)                                        *
*                                                                        *
*************************************************************************/

// matbase.h
// The Header files for the Matrix Base class


#if !defined( __MATBASE_H)
#define __MATBASE_H


//allow for matrices > 64K in DOS and windows

#if defined(__MSDOS__) || defined (__WINDOWS__)
  #define _HUGE_PTR huge
  #define __USE__FAR
#else
  #define _HUGE_PTR
#endif


//allow debugging code in constructors to be initiated
//#define __DEBUG__


#include <fstream.h>
#include <math.h>

typedef double MatPrec;
typedef unsigned int MatType;
typedef unsigned int MatSize;


class MatrixManip;


//Define data structure for unique ClassIDs-- automatic definition

class ClassMarker{
  static MatType MarkerBase;
  MatType MarkerNumber;
public:
```

```
  ClassMarker(){ MarkerNumber=MarkerBase++;}
  operator MatType () {return MarkerNumber;}
};


enum _MatrixOp{_Add,_Sub,_MulMat,_MulConst,_DivConst,
  _AddToSelf, _SubFromSelf, _MulMatToSelf, _MulConstToSelf,
  _DivConstFromSelf };


#include "vectbase.h"


// turn off unused parameter warning messages
#pragma warn -par




class MatrixBase{
  char Temp:1;              // set to 1 if matrix is temp
  char Reserved:7;

protected:
  MatSize row, col;
  static MatPrec Tolerance;
  static MatSize Base;

  virtual void MAdd(MatrixBase & Arg, MatrixBase & Dest)=0;
  virtual void MSub(MatrixBase & Arg, MatrixBase & Dest)=0;
  virtual void MMul(MatrixBase & Arg, MatrixBase & Dest)=0;
  virtual void CMul(MatPrec X, MatrixBase & Dest)=0;
  virtual void CDiv(MatPrec X, MatrixBase & Dest)=0;
  virtual MatrixBase& CreateTempMat(_MatrixOp Opperation,
              MatrixBase & Arg)=0;


  static ofstream errorfl;

#if defined(__DEBUG__)
  static ofstream DebugLog;
  static void VerifyHeap();
#endif

  static void MemError(char * mat);
  static void SizeError(char * mat);
  static void IncTError(char * mat);
  static void InvError(char * mat);
  static void AccessError(char * mat);


public:
  MatrixBase();
```

```
MatrixBase(MatSize R, MatSize C, char IsTemp=0);
MatrixBase(ifstream & Source);
virtual ~MatrixBase(){};

MatrixBase& operator+ (MatrixBase& Arg);
MatrixBase& operator- (MatrixBase& Arg);
MatrixBase& operator* (MatrixBase& Arg);
MatrixBase& operator* (MatPrec X);
MatrixBase& operator/ (MatPrec X);

virtual MatrixBase& operator= (MatrixBase& Arg)=0;
MatrixBase& operator+= (MatrixBase& Arg);
MatrixBase& operator-= (MatrixBase& Arg);
MatrixBase& operator*= (MatrixBase& Arg);
MatrixBase& operator*= (MatPrec X);
MatrixBase& operator/= (MatPrec X);

virtual const MatType MatrixType()=0;

virtual MatPrec GetVal(MatSize R, MatSize C)=0;
virtual void PutVal(MatSize R, MatSize C, MatPrec VAL)=0;
virtual MatPrec QGetVal(MatSize R, MatSize C)=0;
virtual void QPutVal(MatSize R, MatSize C, MatPrec VAL)=0;


virtual int IsSparse(){return 0;}
virtual int IsTemp(){return Temp;}
virtual int IsSymetric();
        int IsSquare(){return row==col;}

MatrixManip operator[](MatSize row);
MatrixManip operator()(int x,int y);
MatSize GetNumRow(){return row;}
MatSize GetNumCol(){return col;}
int CompareSize(MatrixBase & Arg){
    return (col==Arg.col)&&(row==Arg.row);
}
int IsSame(MatrixBase & Arg){
    return (MatrixType())==(Arg.MatrixType());
}
virtual void* GetAddress()=0;

virtual MatrixBase & GetSubMatrix(MatSize R, MatSize C);
virtual int PutSubMatrix(MatSize R, MatSize C, MatrixBase & arg);

virtual VectorBase & GetRow(MatSize R)
    {return (VectorBase&)*((VectorBase*)000);}
virtual int PutRow(MatSize R, VectorBase & row){return 0;}
virtual VectorBase & GetCol(MatSize C)
    {return (VectorBase&)*((VectorBase*)000);}
virtual int PutCol(MatSize C, VectorBase & col){return 0;}

virtual void ident(MatPrec val=1.0)=0;
static void SetTol(MatPrec tol){Tolerance=tol;}
```

```
  static void SetBase(MatSize B){Base=B;}
  static int IsZero(MatPrec Val){
      return (fabs(Val) <= Tolerance);
  }
  virtual MatPrec BGetVal(MatSize R, MatSize C)=0;
  virtual void BPutVal(MatSize R, MatSize C, MatPrec VAL)=0;


  void SetTemp(){Temp=1;}
  void ReSetTemp(){Temp=0;}
};




class MatrixManip{

public:
  enum _FNC{SubMat,GetCol,GetRow};

  MatrixManip(MatSize R, MatrixBase* Mat);
  MatrixManip(MatSize R, MatSize C, MatrixBase* Mat,_FNC Do=SubMat);
  MatrixManip & operator[] (MatSize C){
      col=C;
      return *this;
  }
  MatrixManip & operator <<(MatrixBase & Arg);
  MatrixManip & operator >>(MatrixBase & Arg);
  operator MatPrec(){
      return theMatrix->GetVal(row,col);
  }
  MatPrec operator =(MatPrec Arg){
      theMatrix->PutVal(row,col,Arg);
      return Arg;
  }

private:
  MatrixBase * theMatrix;
  MatSize row,col;
  _FNC DoWhat;
};


ostream& operator <<(ostream& s, MatrixManip & manip);


#endif
```

MATRIX.H

```
/*********************************************************************
*                                                                    *
*  This file is part of MATCLASS,a Matrix Class Library,             *
*   written by:                                                      *
*          Gordon W. Zeglinski                                       *
*                                                                    *
*                                                                    *
*    (c) 1992, (beta test version)                                   *
*                                                                    *
*********************************************************************/

// matrix.h
// The header file for the Matrix class


#if !defined( __MATRIX_H)
#define __MATRIX_H

#include <matbase.h>



class Matrix: public MatrixBase{
  static ClassMarker ClassID;
protected:
  MatPrec ** Storage;

  MatPrec** GetStorage(){return Storage;}

  virtual void MAdd(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MSub(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MMul(MatrixBase & Arg, MatrixBase & Dest);
  virtual void CMul(MatPrec X, MatrixBase & Dest);
  virtual void CDiv(MatPrec X, MatrixBase & Dest);
  virtual MatrixBase& CreateTempMat(_MatrixOp Opperation,
              MatrixBase & Arg);

  void AddRows(MatSize Rarg1, MatSize Rarg2, MatSize Rdest, MatPrec Mult,
      MatSize Start, MatSize End);
  void AddCols(MatSize Carg1, MatSize Carg2, MatSize Cdest, MatPrec Mult,
      MatSize Start, MatSize End);

  void MulRows(MatSize Rsource, MatSize Rdest, MatPrec Mult,
      MatSize Start, MatSize End);
  void MulCols(MatSize Csource, MatSize Cdest, MatPrec Mult,
      MatSize Start, MatSize End);


public:
  Matrix(MatSize R, MatSize C, char IsTemp=0);
```

```
Matrix(MatSize R, MatSize C, MatPrec El1, MatPrec El2, ...);
Matrix(Matrix & arg);
Matrix(MatrixBase & arg);
Matrix(ifstream & Source);
~Matrix();

virtual MatrixBase& operator= (MatrixBase& Arg);

virtual const MatType MatrixType();
static const MatType GetID(){return ClassID;}
virtual MatPrec GetVal(MatSize R, MatSize C);
virtual void PutVal(MatSize R, MatSize C, MatPrec VAL);
virtual void ident(MatPrec val=1.0);

virtual MatPrec QGetVal(MatSize R, MatSize C){
    return *(*(Storage+R-Base)+C-Base);
}
virtual void QPutVal(MatSize R, MatSize C, MatPrec VAL){
    *(*(Storage+R-Base)+C-Base)=VAL;
}
virtual void* GetAddress(){return (void*)this;}

MatPrec det();
MatrixBase & inv();
MatrixBase & trans();

VectorBase & GetRow(MatSize R);
int PutRow(MatSize R, VectorBase& arg);
VectorBase & GetCol(MatSize C);
int PutCol(MatSize C, VectorBase & arg);

void AddRows(MatSize Rarg1, MatSize Rarg2, MatSize Rdest, MatPrec Mult=1.0){
    AddRows(Rarg1,Rarg2,Rdest,Mult,0,col);
}
void AddCols(MatSize Carg1, MatSize Carg2, MatSize Cdest, MatPrec Mult=1.0){
    AddCols(Carg1,Carg2,Cdest,Mult,0,row);
}
void MulRows(MatSize Rsource, MatSize Rdest, MatPrec Mult){
    MulRows(Rsource,Rdest,Mult,0,col);
}
void MulCols(MatSize Csource, MatSize Cdest, MatPrec Mult){
    MulCols(Csource,Cdest,Mult,0,row);
}

void SwapRows(MatSize from, MatSize to);

MatrixBase & Solve(MatrixBase & COL);

MatPrec Max();
MatPrec Min();
MatPrec MaxInRow(MatSize R);
MatPrec MaxInCol(MatSize C);
MatPrec MinInRow(MatSize R);
MatPrec MinInCol(MatSize C);
```

```
  void rand(MatPrec val);
  int operator!=(Matrix &Arg){return !(*this==Arg);}
  int operator==(Matrix &Arg);

  virtual MatPrec BGetVal(MatSize R, MatSize C){
      return *(*(Storage+R)+C);
  }
  virtual void BPutVal(MatSize R, MatSize C, MatPrec VAL){
      *(*(Storage+R)+C)=VAL;
  }

friend class FSymmEigen;
friend class FullLU;
};


#endif
```

SPARSE.H

```
/*****************************************************************
*                                                                *
* This file is part of MATCLASS,a Matrix Class Library,          *
*  written by:                                                   *
*         Gordon W. Zeglinski                                    *
*                                                                *
*                                                                *
*  (c) 1992, (beta test version)                                 *
*                                                                *
*****************************************************************/

// this file contains the sparse matrix base data structure

#if !defined(__SPARSEMATRIX_H)
#define __SPARSEMATRIX_H

#include <matbase.h>

class SparseBase:public MatrixBase{
protected:
  virtual void MAdd(MatrixBase & Arg, MatrixBase & Dest)=0;
  virtual void MSub(MatrixBase & Arg, MatrixBase & Dest)=0;
  virtual void MMul(MatrixBase & Arg, MatrixBase & Dest)=0;
  virtual void CMul(MatPrec X, MatrixBase & Dest)=0;
  virtual void CDiv(MatPrec X, MatrixBase & Dest)=0;
  virtual MatrixBase& CreateTempMat(_MatrixOp Opperation,
              MatrixBase & Arg)=0;

  virtual MatPrec BGetVal(MatSize R, MatSize C)=0;
  virtual void BPutVal(MatSize R, MatSize C, MatPrec VAL)=0;

public:
  SparseBase(MatSize R, MatSize C, char IsTemp);
  SparseBase(ifstream &Source);
  ~SparseBase();

  virtual MatrixBase& operator= (MatrixBase& Arg)=0;
  virtual const MatType MatrixType()=0;
  virtual MatPrec GetVal(MatSize R, MatSize C)=0;
  virtual void PutVal(MatSize R, MatSize C, MatPrec VAL)=0;
  virtual void ident(MatPrec val=1.0)=0;

  virtual MatPrec QGetVal(MatSize R, MatSize C)=0;
  virtual void QPutVal(MatSize R, MatSize C, MatPrec VAL)=0;
  virtual void* GetAddress()=0;

  virtual int IsSparse(){return 1;}

  virtual void GetRowRange(MatSize row, MatSize & StartCol, MatSize & EndCol)=0;
};
```

```
class BandedBase:public SparseBase{
protected:
  MatSize BandWidth;  // The full bandwidth
  MatPrec **Storage;

  virtual void MAdd(MatrixBase & Arg, MatrixBase & Dest)=0;
  virtual void MSub(MatrixBase & Arg, MatrixBase & Dest)=0;
  virtual void MMul(MatrixBase & Arg, MatrixBase & Dest)=0;
  virtual void CMul(MatPrec X, MatrixBase & Dest)=0;
  virtual void CDiv(MatPrec X, MatrixBase & Dest)=0;
  virtual MatrixBase& CreateTempMat(_MatrixOp Opperation,
            MatrixBase & Arg)=0;

  void MAdd(BandedBase & Arg, BandedBase & Dest);
  void MSub(BandedBase & Arg, BandedBase & Dest);
  void MMul(BandedBase & Arg, BandedBase & Dest);
  void CMul(MatPrec X, BandedBase & Dest);
  void CDiv(MatPrec X, BandedBase & Dest);

  virtual MatPrec BGetVal(MatSize R, MatSize C)=0;
  virtual void BPutVal(MatSize R, MatSize C, MatPrec VAL)=0;


//void AddMajors(MatSize Rarg1, MatSize Rarg2, MatSize Rdest, MatPrec Mult);
//void AddMinors(MatSize Carg1, MatSize Carg2, MatSize Cdest, MatPrec Mult);

//void MulMajors(MatSize Rsource, MatSize Rdest, MatPrec Mult);
//void MulMinors(MatSize Csource, MatSize Cdest, MatPrec Mult);

//void SwapMajor(MatSize To, MatSize From);
//  void SwapMinor(MatSize To, MatSize From);

  MatSize NumMinor(MatSize Major);

  void ExpandBand(MatSize Increase);
  void ShrinkBand(MatSize Decrease);

  void GetMajorRange(MatSize row, MatSize & StartCol, MatSize & EndCol);

public:
  BandedBase(MatSize Major, MatSize Minor, MatSize B, char IsTemp=0);
  ~BandedBase();

  virtual MatrixBase& operator= (MatrixBase& Arg)=0;
  virtual const MatType MatrixType()=0;
  virtual MatPrec GetVal(MatSize R, MatSize C)=0;
  virtual void PutVal(MatSize R, MatSize C, MatPrec VAL)=0;
  virtual void ident(MatPrec val=1.0)=0;

  virtual MatPrec QGetVal(MatSize R, MatSize C)=0;
  virtual void QPutVal(MatSize R, MatSize C, MatPrec VAL)=0;
```

```
//virtual void* GetAddress(){return (void *) this;}

  virtual void GetRowRange(MatSize row, MatSize & StartCol, MatSize & EndCol)=0;

//MatrixBase & Solve(MatrixBase & COL);
};

inline MatSize BandedBase::NumMinor(MatSize Major){
      MatSize NMinor;

  if (Major< ((BandWidth-1)/2) )
      NMinor=((BandWidth-1)/2)+Major;
  else {
      if (Major>=(row-(BandWidth-1)/2))
          NMinor=(BandWidth-1)/2+(row-Major-1);
      else
          NMinor=BandWidth;
      }
return NMinor;
}


#endif
```

TRIMAT.H

```
/*****************************************************************
*                                                                *
* This file is part of MATCLASS,a Matrix Class Library,          *
*  written by:                                                   *
*        Gordon W. Zeglinski                                     *
*                                                                *
*                                                                *
*  (c) 1992, (beta test version)                                 *
*                                                                *
*****************************************************************/

// this file contains the triangular matrix data structure.
// it serves as a base for all triangular types of matrices.

#if !defined(__TRIMATRIX_H)
#define __TRIMATRIX_H

#include <matbase.h>
#include <matrix.h>

class TriMatrix: public MatrixBase{
  MatPrec * Storage;
protected:
  virtual void MAdd(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MSub(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MMul(MatrixBase & Arg, MatrixBase & Dest);
  virtual void CMul(MatPrec X, MatrixBase & Dest);
  virtual void CDiv(MatPrec X, MatrixBase & Dest);
  virtual MatrixBase& CreateTempMat(_MatrixOp Opperation,
              MatrixBase & Arg)=0;

  virtual MatPrec BGetVal(MatSize R, MatSize C);
  virtual void BPutVal(MatSize R, MatSize C, MatPrec VAL);


public:
  TriMatrix(MatSize R, MatSize C,char IsTemp);
  TriMatrix(TriMatrix & arg);
  ~TriMatrix();

  virtual MatrixBase& operator= (MatrixBase& Arg)=0;
  virtual const MatType MatrixType()=0;
  virtual MatPrec GetVal(MatSize R, MatSize C);
  virtual void PutVal(MatSize R, MatSize C, MatPrec VAL);
  virtual void ident(MatPrec val=1.0);

  virtual MatPrec QGetVal(MatSize R, MatSize C);
  virtual void QPutVal(MatSize R, MatSize C, MatPrec VAL);
  virtual void* GetAddress()=0;
```

```
  MatPrec det();
  virtual MatrixBase & inv()=0;
  void inv(TriMatrix & Ident);

  VectorBase & GetRow(MatSize R);
  int PutRow(MatSize R, VectorBase & arg);
  VectorBase & GetCol(MatSize C);
  int PutCol(MatSize C, VectorBase & arg);

  void AddRows(MatSize Rarg1, MatSize Rarg2, MatSize Rdest, MatPrec Mult=1.0);
  void AddCols(MatSize Carg1, MatSize Carg2, MatSize Cdest, MatPrec Mult=1.0);

  void MulRows(MatSize Rsource, MatSize Rdest, MatPrec Mult);
  void MulCols(MatSize Csource, MatSize Cdest, MatPrec Mult);

  MatrixBase & Solve(MatrixBase & COL);

};


class LowerTriangle:public TriMatrix{
  static ClassMarker ClassID;

protected:
  virtual MatrixBase& CreateTempMat(_MatrixOp Opperation,
              MatrixBase & Arg);

  virtual MatPrec BGetVal(MatSize R, MatSize C){
      if(R<C)
          return 0;
      return TriMatrix::BGetVal(C,R);
  }
  virtual void BPutVal(MatSize R, MatSize C, MatPrec VAL){
      if(R>C)
          TriMatrix::BPutVal(C,R,VAL);
  }

public:
  LowerTriangle(MatSize R, MatSize C,char IsTemp=0);
  ~LowerTriangle();

  virtual MatrixBase& operator= (MatrixBase& Arg);
  virtual const MatType MatrixType();
  static const MatType GetID(){return ClassID;}
  virtual MatPrec GetVal(MatSize R, MatSize C);
  virtual void PutVal(MatSize R, MatSize C, MatPrec VAL);

  virtual MatPrec QGetVal(MatSize R, MatSize C){
      if(R<C)
          return 0;
      return TriMatrix::QGetVal(C,R);
  }
  virtual void QPutVal(MatSize R, MatSize C, MatPrec VAL){
      if(R>C)
```

```
            TriMatrix::QPutVal(C,R,VAL);
    }
    virtual void* GetAddress(){return (void*)this;}

    MatrixBase & Solve(MatrixBase & COL);
    virtual MatrixBase & inv();
};

class UpperTriangle:public TriMatrix{
    static ClassMarker ClassID;

protected:
    virtual MatrixBase& CreateTempMat(_MatrixOp Opperation,
                MatrixBase & Arg);

    virtual MatPrec BGetVal(MatSize R, MatSize C){
        if(C<R)
            return 0;
        return TriMatrix::BGetVal(R,C);
    }
    virtual void BPutVal(MatSize R, MatSize C, MatPrec VAL){
        if(C>R)
            TriMatrix::BPutVal(R,C,VAL);
    }


public:
    UpperTriangle(MatSize R, MatSize C, char IsTemp);
    ~UpperTriangle();

    virtual MatrixBase& operator= (MatrixBase& Arg);
    virtual const MatType MatrixType();
    static const MatType GetID(){return ClassID;}
    virtual MatPrec GetVal(MatSize R, MatSize C);
    virtual void PutVal(MatSize R, MatSize C, MatPrec VAL);

    virtual MatPrec QGetVal(MatSize R, MatSize C){
        if(C<R)
            return 0;
        return TriMatrix::QGetVal(R,C);
    }
    virtual void QPutVal(MatSize R, MatSize C, MatPrec VAL){
        if(C>R)
            TriMatrix::QPutVal(R,C,VAL);
    }
    virtual void* GetAddress(){return (void*)this;}

    MatrixBase & Solve(MatrixBase & COL);
    virtual MatrixBase & inv();
};


#endif
```

UUMAT.H

```
/*****************************************************************
*                                                                *
* This file is part of MATCLASS,a Matrix Class Library,          *
*  written by:                                                   *
*        Gordon W. Zeglinski                                     *
*                                                                *
*                                                                *
*  (c) 1992, (beta test version)                                 *
*                                                                *
*****************************************************************/


// this file holds the data structure for uTu, decomposed matrices


#if !defined(__UUMATRIX_H)
#define __UUMATRIX_H

#include <trimat.h>
#include <matrix.h>

class UuMatrix: public TriMatrix{
  static ClassMarker ClassID;

  virtual void MAdd(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MSub(MatrixBase & Arg, MatrixBase & Dest);
  virtual void MMul(MatrixBase & Arg, MatrixBase & Dest);
  virtual void CMul(MatPrec X, MatrixBase & Dest);
  virtual void CDiv(MatPrec X, MatrixBase & Dest);
  virtual MatrixBase& CreateTempMat(_MatrixOp Opperation,
              MatrixBase & Arg);

  virtual MatPrec BGetVal(MatSize R, MatSize C);
  virtual void BPutVal(MatSize R, MatSize C, MatPrec VAL);

public:
  UuMatrix(MatSize R, MatSize C,char IsTemp=0);
  UuMatrix(MatrixBase & ARG);
  ~UuMatrix();

  virtual MatrixBase& operator= (MatrixBase& Arg);
  virtual const MatType MatrixType();
  static const MatType GetID(){return ClassID;}
  virtual MatPrec GetVal(MatSize R, MatSize C);
  virtual void PutVal(MatSize R, MatSize C, MatPrec VAL);
  virtual void ident(MatPrec val=1.0);

  virtual MatPrec QGetVal(MatSize R, MatSize C);
  virtual void QPutVal(MatSize R, MatSize C, MatPrec VAL);
```

```
  Matrix & UnPack();
  void Pack(MatrixBase & ARG);
  virtual void* GetAddress(){return (void*)this;}

  MatPrec det();
  MatrixBase & inv();

  MatrixBase & Solve(MatrixBase & COL);
};

#endif
```

APPENDIX IV

CODE FOR TEST MATRIX GENERATOR

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <math.h>
#include <fstream.h>


#define huge
#define farmalloc malloc
#define farfree free
#define cprintf printf

int checkmap(int i,int j, int *map, int &count);
int fndpos(int *ICOL, int row, int col);
void solvemat(int size, double huge *a, double *x, double *b, int *ICOL);
float XX(float x, float y, float z);
float YY(float x, float y, float z);

  float AS ,AP;


void main(void){
  time_t T1, T2;

  float sizex, sizey, dx, dy;
  int numx, numy;
  float *xx, *xx1;
  float *yy, *yy1;
  float *temp, *Qj, *H, *K, tinfnt;
  float kval,hval;
  int k, i, j, num_unknowns;
  unsigned char *BC, *NODE_TYPE;
  float mltx, mlty, heat_in, heat_out;
  int *map, cnt, tmp, row, col, tmp2;
  int *ICOL;
  double huge *A;
  double *B;
  double *X;
  float x,y,z;
  float xxmax, yymax, xxmin, yymin;
  float axisx,axisy,axisz;
  float xxt[2],yyt[2],xxx[2],yyx[2],xxy[2],yyy[2],xxz[2],yyz[2];
  char *format;



//AS=3.1415927/180.0*60.0;
//AP=3.1415927/180.0*20.0;
```

```
  AS=(-4.537843);
  AP=3.05432;


  numx=7;
  numy=3;
//numx=19;
//numy=7;
//numx=25;
//numy=9;
  cprintf("Enter the number of nodes in the X-direction\n\rin the Y-direction
(seperatted by a space)\n\r");
  scanf("%i %i", &numx,&numy);
  cprintf("\n\rEnter the value of k H\n\r");
  scanf("%f %f",&kval,&hval);

  sizex=0.06;
  sizey=0.02;
  dx=sizex / (numx-1);
  dy=sizey / (float) (numy-1);

  temp= new float [(numx+2)*(numy+2)] ; //(0);
  K= new float [numx*numy] ; //(0.5);
  Qj= new float [numx*numy] ; //(0);
  BC= new unsigned char [(numx+2)*(numy+2)] ; //(0);
  H= new float [numx*numy] ; //(0);
  NODE_TYPE= new unsigned char [numx*numy] ; //(0);

  for(i=0;i<numx*numy;i++){
      *(K+i)=0.5;
      *(Qj+i)=0;
      *(H+i)=0;
      *(NODE_TYPE+i)=0;
  }

  for(i=0;i<(numx+2)*(numy+2);i++){
      *(temp+i)=0;
      *(BC+i)=0;
  }

  if ((temp==NULL)||(K==NULL)||(Qj==NULL)||
      (BC==NULL)||(H==NULL)){
      printf("error in mememory allocation");
      exit(0);
  }

  for(j=0;j<numy;j++)
      for(i=0;i<numx;i++){
          Qj[i+j*numx]=0.0;
          K[i+j*numx]=kval;
          H[i+j*numx]=0.0;
          NODE_TYPE[i+j*numx]=4;
      }
```

```
  memset(BC,0, sizeof(char)*((numx+2)*(numy+2)));
  memset(temp,0, sizeof(float)*((numx+2)*(numy+2)));

// set the boundry conditions, node types and known temperatures ...

// left boundry conditions
  for(j=0;j<numy+1;j++)
      BC[j*(numx+2)]=2;          // convection
  for(j=0;j<numy+1;j++)
      temp[j*(numx+2)]=400;      //temperature of air
  for(j=0;j<numy;j++)
      H[j*numx]=500;                    // H value for left face
  for(j=1;j<numy-1;j++)
      NODE_TYPE[j*numx]=1;  //main wall nodes
  NODE_TYPE[0]=6;                       //upper corner
  NODE_TYPE[(numy-1)*numx]=7;               //lower corner

// bottom boundry
  for(i=0;i<=numx+1;i++)
      BC[i+(numy+1)*(numx+2)]=3; // insulated
  for(i=1;i<numx-1;i++)
      NODE_TYPE[i+(numy-1)*numx]=2;
  NODE_TYPE[numx-1+(numy-1)*numx]=9;
  H[numx-1+(numy-1)*numx]=hval;

// top boundry
  k=0.01/dx;
  for(i=0;i<k+2;i++){
      BC[i]=3;              //top point insulated
  }
  for(j=0;j<0.01/dy;j++)  //set air temp
      for(i=k+2;i<=numx+1;i++){
          BC[i+j*(numx+2)]=1;
          temp[i+j*(numx+2)]=20;
      }
  j=0.01/dy;
  for(i=k+2;i<numx+1;i++){
      BC[i+j*(numx+2)]=2;  //convection
      H[i-1+j*numx]=hval;
      NODE_TYPE[i-1+j*numx]=0;
  }
  NODE_TYPE[k+j*numx]=5;
  H[k+j*numx]=hval;
  NODE_TYPE[numx-1+j*numx]=8;
  H[numx-1+j*numx]=hval;

// right boundry
  k=0.01/dy;
  i=0.01/dx;
  for(j=1;j<k;j++){
      temp[i+2+j*(numx+2)]=20;
      BC[i+2+j*(numx+2)]=2;   // convection
      NODE_TYPE[i+j*numx]=3;
      H[i+j*numx]=hval;
```

```
}
NODE_TYPE[k]=8;
H[k]=hval;
for(j=k+1;j<=numy;j++){
    temp[numx+1+j*(numx+2)]=20;
    BC[numx+1+j*(numx+2)]=2;   // convection
}
for(j=k+2;j<numy;j++){
    NODE_TYPE[numx-1+(j-1)*numx]=3;
    H[numx-1+(j-1)*numx]=hval;
}


k=0;
for (j=1;j<=numy;j++)
    for(i=1;i<=numx;i++)
        if (BC[i+j*(numx+2)]==0)
            k+=1;
num_unknowns=k;
A=(double huge*) farmalloc((long) ((long) num_unknowns)*5*sizeof(double));
ICOL=new int [num_unknowns*5];
B=new double[num_unknowns];
X=new double[num_unknowns];
map= new int [num_unknowns*2] ; // (0);
if ((map==NULL)||(A==NULL)||(B==NULL)||(X==NULL)||(ICOL==NULL))
    exit(1);

for(i=0;i<num_unknowns*2;i++)
    *(map+i)=0;


memset( map,0,(num_unknowns*sizeof(int)));
memset( B,0,(num_unknowns*sizeof(double)));
for(i=0;i<(num_unknowns*5);i++){
    *(A+i)=0.0;
    *(ICOL+i)=1;
}



for(i=0;i<num_unknowns;i++)
    *(X+i)=200.0;

cnt=0;
for(j=1;j<=numy;j++)
    for(i=1;i<=numx;i++){

        if(BC[(i)+(j)*(numx+2)]==0){
            if (NODE_TYPE[(i-1)+(j-1)*numx] != 5){
                switch(NODE_TYPE[(i-1)+(j-1)*numx]){
                    case 0:
                        mlty=0.5;
                        mltx=1.0;
                        break;
```

```
                    case 1:
                        mltx=0.5;
                        mlty=1.0;
                        break;
                    case 2:
                        mlty=0.5;
                        mltx=1.0;
                        break;
                    case 3:
                        mltx=0.5;
                        mlty=1.0;
                        break;
                    case 4:
                        mlty=mltx=1.0;
                        break;
                    case 6:
                        mlty=mltx=0.5;
                        break;
                    case 7:
                        mlty=mltx=0.5;
                        break;
                    case 8:
                        mlty=mltx=0.5;
                        break;
                    case 9:
                        mlty=mltx=0.5;
                }
                row=checkmap(i,j,map,cnt);
// check in horizontal direction
                tmp=i-1;
                switch (BC[(tmp)+j*(numx+2)]){
                    case 2:  //convection
                        B[row]-=H[(i-1)+(j-1)*numx]*
                                temp[(tmp)+(j)*(numx+2)]*dy*mlty;
                        A[0 + row*5]-=H[(i-1)+(j-1)*numx]*dy*mlty;
                        break;

                    case 0: //unknown temp
                        tmp2=checkmap(tmp,j,map,cnt);
                        A[fndpos(ICOL,row,tmp2) + 5*row]+=K[(i-1)+(j-1)*
                                numx]*dy/dx*mlty;
                        A[0 + 5*row]-=K[(i-1)+(j-1)*numx]*dy/dx*mlty;
                        break;

                    case 1: //known temp
                        B[row]-=K[(i-1)+(j-1)*numx]*
                                temp[(tmp)+(j)*(numx+2)]*dy/dx*mlty;
                        A[0 + row*5]-=K[(i-1)+(j-1)*numx]*dy/dx*mlty;
                }

                tmp=i+1;
                switch (BC[(tmp)+j*(numx+2)]){
                    case 2:  //convection
                        B[row]-=H[(i-1)+(j-1)*numx]*
```

```
                        temp[(tmp)+(j)*(numx+2)]*dy*mlty;
                    A[0+row*5]-=H[(i-1)+(j-1)*numx]*dy*mlty;
                    break;

                case 0: //unknown temp
                    tmp2=checkmap(tmp,j,map,cnt);
                    A[fndpos(ICOL,row,tmp2) + 5*row]+=K[(i-1)+(j-1)*

 numx]*dy/dx*mlty;
                    A[0 + row*5]-=K[(i-1)+(j-1)*numx]*dy/dx*mlty;
                    break;

                case 1: //known temp
                    B[row]-=K[(i-1)+(j-1)*numx]*
                        temp[(tmp)+(j)*(numx+2)]*dy/dx*mlty;
                    A[0 + row*5]-=K[(i-1)+(j-1)*numx]*dy/dx*mlty;
            }

// check in vertical direction

            tmp=j-1;
            switch (BC[(i)+tmp*(numx+2)]){
                case 2:  //convection
                    B[row]-=H[(i-1)+(j-1)*numx]*
                            temp[(i)+(tmp)*(numx+2)]*dx*mltx;
                    A[0 + row*5]-=H[(i-1)+(j-1)*numx]*dx*mltx;
                    break;

                case 0: //unknown temp
                    tmp2=checkmap(i,tmp,map,cnt);
                    A[fndpos(ICOL,row,tmp2) + 5*row]+=K[(i-1)+(j-1)*
                        numx]*dx/dy*mltx;
                    A[0 + row*5]-=K[(i-1)+(j-1)*numx]*dx/dy*mltx;
                    break;

                case 1: //known temp
                    B[row]-=K[(i-1)+(j-1)*numx]*
                            temp[(i)+(tmp)*(numx+2)]*dx/dy*mltx;
                    A[0 + row*5]-=K[(i-1)+(j-1)*numx]*dx/dy*mltx;
            }

            tmp=j+1;
            switch (BC[(i)+tmp*(numx+2)]){
                case 2:  //convection
                    B[row]-=H[(i-1)+(j-1)*numx]*
                            temp[(i)+(tmp)*(numx+2)]*dx*mltx;
                    A[0 + row*5]-=H[(i-1)+(j-1)*numx]*dx*mltx;
                    break;

                case 0: //unknown temp
                    tmp2=checkmap(i,tmp,map,cnt);
                    A[fndpos(ICOL,row,tmp2) + 5*row]+=K[(i-1)+(j-1)*

 numx]*dx/dy*mltx;
```

```
                    A[0 + row*5]-=K[(i-1)+(j-1)*numx]*dx/dy*mltx;
                    break;

                case 1: //known temp
                    B[row]-=K[(i-1)+(j-1)*numx]*
                                temp[(i)+(tmp)*(numx+2)]*dx/dy*mltx;
                    A[0 + row*5]-=K[(i-1)+(j-1)*numx]*dx/dy*mltx;
            }
        }
        else{
            row=checkmap(i,j,map,cnt);
// check in horizontal direction
            tmp=i-1; //left
            switch (BC[(tmp)+j*(numx+2)]){
                case 2:  //convection
                    B[row]-=H[(i-1)+(j-1)*numx]*
                            temp[(tmp)+(j)*(numx+2)]*dy;
                    A[0 + row*5]-=H[(i-1)+(j-1)*numx]*dy;
                    break;

                case 0: //unknown temp
                    tmp2=checkmap(tmp,j,map,cnt);
                    A[fndpos(ICOL,row,tmp2) + 5*row]+=K[(i-1)+(j-1)*
                            numx]*dy/dx;
                    A[0 + row*5]-=K[(i-1)+(j-1)*numx]*dy/dx;
                    break;

                case 1: //known temp
                    B[row]-=K[(i-1)+(j-1)*numx]*
                                temp[(tmp)+(j)*(numx+2)]*dy/dx;
                    A[0 + row*5]-=K[(i-1)+(j-1)*numx]*dy/dx;
            }

            tmp=i+1; //right
            switch (BC[(tmp)+j*(numx+2)]){
                case 2:  //convection
                    B[row]-=H[(i-1)+(j-1)*numx]*
                                temp[(tmp)+(j)*(numx+2)]*dy*0.5;
                    A[0 + row*5]-=H[(i-1)+(j-1)*numx]*dy*0.5;
                    break;

                case 0: //unknown temp
                    tmp2=checkmap(tmp,j,map,cnt);
                    A[fndpos(ICOL,row,tmp2) + 5*row]+=K[(i-1)+(j-1)*
                            numx]*dy/dx*0.5;
                    A[0 + row*5]-=K[(i-1)+(j-1)*numx]*dy/dx*0.5;
                    break;

                case 1: //known temp
                    B[row]-=K[(i-1)+(j-1)*numx]*
                                temp[(tmp)+(j)*(numx+2)]*dy/dx*0.5;
                    A[0 + row*5]-=K[(i-1)+(j-1)*numx]*dy/dx*0.5;
            }
```

```
// check in vertical direction

                tmp=j-1; //up
                switch (BC[(i)+tmp*(numx+2)]){
                    case 2:  //convection
                        B[row]-=H[(i-1)+(j-1)*numx]*
                                        temp[(i)+(tmp)*(numx+2)]*dx*0.5;
                        A[0 + row*5]-=H[(i-1)+(j-1)*numx]*dx*0.5;
                        break;

                    case 0: //unknown temp
                        tmp2=checkmap(i,tmp,map,cnt);
                        A[fndpos(ICOL,row,tmp2) + 5*row]+=K[(i-1)+(j-1)*
                                numx]*dx/dy*0.5;
                        A[0 + row*5]-=K[(i-1)+(j-1)*numx]*dx/dy*0.5;
                        break;

                    case 1: //known temp
                        B[row]-=K[(i-1)+(j-1)*numx]*
                                        temp[(i)+(tmp)*(numx+2)]*dx/dy*0.5;
                        A[0 + row*5]-=K[(i-1)+(j-1)*numx]*dx/dy*0.5;
                }

                tmp=j+1; //down
                switch (BC[(i)+tmp*(numx+2)]){
                    case 2:  //convection
                        B[row]-=H[(i-1)+(j-1)*numx]*
                                        temp[(i)+(tmp)*(numx+2)]*dx;
                        A[0 + row*5]-=H[(i-1)+(j-1)*numx]*dx;
                        break;

                    case 0: //unknown temp
                        tmp2=checkmap(i,tmp,map,cnt);
                        A[fndpos(ICOL,row,tmp2) + 5*row]+=K[(i-1)+(j-1)*
                                numx]*dx/dy;
                        A[0 + row*5]-=K[(i-1)+(j-1)*numx]*dx/dy;
                        break;

                    case 1: //known temp
                        B[row]-=K[(i-1)+(j-1)*numx]*
                                        temp[(i)+(tmp)*(numx+2)]*dx/dy*mltx;
                        A[0 + row*5]-=K[(i-1)+(j-1)*numx]*dx/dy;
                }
            // now take care of wierd corner
            tmp=i+1;
            tmp2=j-1;
            B[row]-=H[(i-1)+(j-1)*numx]*
                        temp[(tmp)+(tmp2)*(numx+2)]*dx*0.5;
            A[0 + row*5]-=H[(i-1)+(j-1)*numx]*dx*0.5;
            B[row]-=H[(i-1)+(j-1)*numx]*
                        temp[(tmp)+(tmp2)*(numx+2)]*dy*0.5;
            A[0 + row*5]-=H[(i-1)+(j-1)*numx]*dy*0.5;
            }
        }
```

```
        }

/*
  printf("\n  MATRIX A  & B\n\r");
  for(j=0;j<num_unknowns;j++){
        for(i=0;i<num_unknowns;i++)
            printf("%4.1lf ",A[i+j*num_unknowns] );
        printf("= %6.1lf \n\r",B[j]);
//      getch();
  }
*/

//Dump Matrix to file
  ofstream MatrixOut("NxN.out");
  ofstream RHS("Rhs.out");

  MatrixOut<<num_unknowns<<" "<<num_unknowns<<endl;
  for(i=0;i<num_unknowns;i++){
        MatrixOut<<i+1<<" "<<i+1<<" "<<*(A+5*i)<<endl;
        for(j=1;j<*(ICOL+i*5);j++)
            MatrixOut<<i+1<<" "<<*(ICOL+j+i*5)+1<<" "<<*(A+j+5*i)<<endl;
  }
  RHS<<num_unknowns<<" "<<1<<endl;
  for(i=0;i<num_unknowns;i++)
        RHS<<i+1<<" "<<1<<" "<<*(B+i)<<endl;

//exit(1);

  time(&T1);
  solvemat(num_unknowns, A, X, B, ICOL);
  time(&T2);
  cout<< difftime(T2,T1)<<" seconds"<<endl;
  for(k=0;k<num_unknowns;k++)
        temp[map[k*2]+(numx+2)*map[1+k*2]]=X[k];

  delete X;
  delete B;
  farfree(A);
  delete ICOL;

/*
// text output
  printf("\n\r    Temperature Distribution\n\n\r   dx= %5.5f   dy=%5.5f\n\n\r",
            dx,dy);
  for(j=1;j<=numy;j++){
        for(i=1;i<=numx;i++)
            printf("%4.1f ",temp[i+j*(numx+2)] );
        printf("\n\r");
  }
//getch();
*/

// print temp at corners and at mid-points
  printf("\n\rNumber of Unknows  %10i", num_unknowns);
```

```
  printf("\n\rX co-ordinate  Y co-ordinate   temperature\n\r");
  format=    "  %-10.3f      %-10.3f          %-10.4f\n\r";
  printf(format,0.0,0.0,temp[1+1*(numx+2)]);
  printf(format,1.0,0.0,temp[1+(int (0.01/dx)) +1*(numx+2)]);
  printf(format,0.0,1.0,temp[1 +(int (0.01/dy)+1)*(numx+2)]);
  printf(format,1.0,1.0,temp[(1+(int (0.01/dx)))
          +(int(0.01/dy)+1)*(numx+2)]);
  printf(format,6.0,1.0,temp[(1+(int (0.06/dx)))
          +(int (0.01/dy)+1)*(numx+2)]);
  printf(format,0.0,2.0,temp[1 +(int (0.02/dy+1))*(numx+2)]);
  printf(format,6.0,2.0,temp[(1+(int (0.06/dx)))
          +(int (0.02/dy)+1)*(numx+2)]);

// calculate heat in and heat out
  heat_in=heat_out=0.0;
  k=0.01/dy;
  i=0.01/dx;
  for(j=0;j<numy;j++){
      if( (j==0)||(j==(numy-1)) )
          heat_in+=500.0*dy*(400.0-temp[1+(j+1)*(numx+2)] )*0.5;
      else
          heat_in+=500.0*dy*(400.0-temp[1+(j+1)*(numx+2)] );
      if (j<k) {
          if(j==0)
              heat_out+=hval*dy*(temp[(i+1)+(j+1)*(numx+2)]-20.0)*0.5;
          else
              heat_out+=hval*dy*(temp[(i+1)+(j+1)*(numx+2)]-20.0);
      }
      if (j==k){
          heat_out+=hval*(dy+dx)/2.0*(temp[(i+1)+(j+1)*(numx+2)]-20.0);
          heat_out+=hval*(dy+dx)/2.0*(temp[(numx)+(j+1)*(numx+2)]-20.0);
      }
      if (j>k){
          if( (j==(k+1)) || (j==(numy-1)) )
              heat_out+=hval*dy*(temp[(numx)+(j+1)*(numx+2)]-20.0)*0.5;
          else
              heat_out+=hval*dy*(temp[(numx)+(j+1)*(numx+2)]-20.0);
      }
  }
  for(j=i+1;j<numx-1;j++)
      heat_out+=hval*dx*(temp[(j+1)+(k+1)*(numx+2)]-20);

  printf("\n\rHeat in     Heat out      Difference\n\r");
  printf("%-10.2f  %-10.2f     %-10.2f",heat_in,heat_out,
          (heat_in-heat_out) );
//getch();

}

int checkmap(int i,int j, int *map, int &count){
  int k;

  for(k=0;k<count;k++)
```

```
        if ((map[0+k*2]==i)&&(map[1+k*2]==j))
            break;
    if (k==count){
        map[k*2]=i;
        map[1+k*2]=j;
        count+=1;
    }
return k;
}

void solvemat(int n, double huge *a, double *x, double *b, int *ICOL){
    int i,j,j1;
    unsigned long maxiter, iter;
    double dfmin;
    double rhs, jend, df, alpha;

    dfmin=0.001;
    maxiter=((long) 200) * ((long) n);
    alpha=1.6;

    iter=0;
    df=10;
    while( (iter<maxiter) && (df>dfmin)){
        df=0.0;
        for(i=0;i<n;i++){
            rhs=(*(b+i)) - (*(a+i*5)) * (*(x+i));
            jend=*(ICOL+i*5);
            for(j=1;j<jend;j++){
                j1=*(ICOL+j+i*5);
                rhs-=(*(a+j+i*5)) * (*(x+j1));
            }
            df+=rhs<=0?(-1.0*rhs):rhs;
            (*(x+i))+=alpha*rhs/(*(a+i*5));
        }
        iter+=1;
    }
    printf("\n\r %5lu   Iterations\n\rConvergence =%10.3e\n\r",iter,df);
}

float XX(float x, float y, float z){
    return y*cos(AS)-x*sin(AS)*cos(AP)-z*sin(AS)*sin(AP);
}

float YY(float x, float y, float z){
    return x*sin(AP)-z*cos(AP);
}

int fndpos(int *ICOL,int row, int col){
    int i;

    for(i=1;i<ICOL[0+row*5];i++)
        if(col==ICOL[i+row*5])
            break;
```

```
  if(i==ICOL[0+row*5]){
       ICOL[i+row*5]=col;
       ICOL[0+row*5]+=1;
  }

  return i;
}
```

APPENDIX V

CODE FOR ACCURACY AND SPEED TESTS

FULL MATRIX TEST

```
/*************************************************************************
*                                                                        *
* This file is part of MATCLASS,a Matrix Class Library,                  *
*  written by:                                                           *
*         Gordon W. Zeglinski                                            *
*                                                                        *
*                                                                        *
*   (c) 1992, (beta test version)                                        *
*                                                                        *
*************************************************************************/

// FullBench.cpp
// this file implements a test of the various matrix classes

#include <time.h>
#include <matrix.h>
#include <fulllu.h>
#include <stdlib.h>

#if defined(__MSDOS__) || defined (__WINDOWS__)
  #include <iostreams.h>
#else
  #include <iostream.h>
#endif

#include <iomanip.h>



#if defined(__MSDOS__) || defined (__WINDOWS__)
  #define _OUT setiosflags(ios::showpoint||ios::scientific)<<setw(15)
#else
  #define _OUT " "
#endif



int main(){
  time_t t1,t2;
  int i,j;

  const int NumAddTest=100;
  const int NumLUTest=100;

  float TotalTime, SolAccuracy=0.0, InvAccuracy=0.0;
  MatSize DIM;
```

```
cout<<"Enter Matrix Size ";
cin>>DIM;


Matrix Org(DIM,DIM);
Matrix Inv(DIM,DIM);
Matrix Col(DIM,1);
Matrix Sol(DIM,1);
Matrix RHS(DIM,1);
FullLU LUOrg(DIM,DIM);
Matrix Temp(DIM,DIM);



// time to decompose a matrix

time(&t1);
for(i=0;i<NumLUTest;i++){
     cout << i <<" ";
     Org.rand(100);
     LUOrg=Org;
     LUOrg.LUForm();
}
time(&t2);
cout<<endl;
TotalTime=difftime(t2,t1);
cout <<"Approx. time to LU Decompose "<<DIM<<" X "<<DIM<<" matrix "<<
          (TotalTime/NumLUTest)<<" sec."<<endl;

//time to solve system of equations

time(&t1);
for(i=0;i<NumLUTest;i++){
     cout<< i << " ";
     Org.rand(100);
     RHS.rand(200);
     LUOrg=Org;
     Sol=LUOrg.Solve(RHS);
     Inv=LUOrg.inv();
}
time(&t2);
cout<<endl;
TotalTime=difftime(t2,t1);
cout <<"Approx. time to solve "<<DIM<<" X "<<DIM<<" matrix "<<
          (TotalTime/(NumLUTest+NumLUTest*DIM))<<" sec."<<endl;



// time to add two matrices

time(&t1);
for(i=0;i<NumAddTest;i++){
     cout<<i<< " ";
```

```
        Temp=Org+Inv;
    }
    time(&t2);
    cout<<endl;
    TotalTime=difftime(t2,t1);
    cout<<"Approx. time to add two "<<DIM<<" X "<<DIM<<
            " matices together "<< (TotalTime/NumLUTest)<<" sec."<<endl;



    // time to multiply matrices

    time(&t1);
    for(i=0;i<NumLUTest;i++){
        cout<<i<< " ";
        Temp=Org*Inv;
    }
    time(&t2);
    cout<<endl;
    TotalTime=difftime(t2,t1);
    cout<<"Approx. time to multiply two "<<DIM<<" X "<<DIM<<
            " matices together "<< (TotalTime/NumLUTest)<<" sec."<<endl;


    // accuracy test

    Col=Org*Sol;
    for(i=1;i<=DIM;i++){
        InvAccuracy+= pow(Temp.GetVal(i,i)-1.0, 2);
        SolAccuracy+= pow(Col.GetVal(i,1)-RHS.GetVal(i,1) ,2);
    }
    for(i=1;i<=DIM;i++)
        for(j=1;j<=DIM;j++)
            if (i!=j)
                InvAccuracy+=pow(Temp.GetVal(i,j) ,2);

    InvAccuracy=sqrt(InvAccuracy)/DIM/DIM;
    SolAccuracy=sqrt(SolAccuracy)/DIM;

    cout << "Total Inversion Error " << InvAccuracy<<endl;
    cout << "Total Solver Error "<< SolAccuracy<<endl;

    return 0;
}
```

SPARSE CODE

```
/*************************************************************************
*                                                                        *
* This file is part of MATCLASS,a Matrix Class Library,                  *
*  written by:                                                           *
*         Gordon W. Zeglinski                                            *
*                                                                        *
*                                                                        *
*  (c) 1992, (beta test version)                                         *
*                                                                        *
*************************************************************************/

// this file contains A test of the sparse LU routines on matrix from files.



#include <fulllu.h>
#include <stdlib.h>
#include <linmarlu.h>
#include <time.h>
#include <fstream.h>


#if defined(__MSDOS__) || defined (__WINDOWS__)
  #include <iostreams.h>
#else
  #include <iostream.h>
#endif

#include <iomanip.h>



int main(){
  time_t t1, t2;
  int i,j;


  const int NumAddTest=200;
  const int NumLUTest=50;


  MatSize DIM;
  const MatSize FILL=3;

  cout<<"Enter Matrix Size ";
  cin>>DIM;

  LinkedMarLU::SetPivotTol(0.0001);
  MatrixBase::SetTol(0.00000000000001);
```

```
  float SparBef,SparAft;
  float TotalTime, SolAccuracy;

  LinkedRow Org(DIM,DIM,FILL);
  LinkedRow Mul(DIM,DIM,FILL);
  LinkedRow Temp(DIM,DIM,FILL);
  LinkedMarLU OrgLU(DIM,DIM,FILL);
  Matrix Col(DIM,1);
  Matrix RHS(DIM,1);
  LinkedRow Sol(DIM,1,2);


//Test Addition Time

  Org.Rand(100);
  Mul.Rand(100);
  time(&t1);
  for(i=0;i<NumAddTest;i++){
//      cout<<i<<" ";
      Temp=Org+Mul;
  }
  time(&t2);
  TotalTime=difftime(t2,t1);
  cout<<endl;
  cout<<"Approx. time to Add "<<DIM<<" X "<<DIM<<
      " matrix "<<(TotalTime/NumAddTest)<<" seconds"<<endl;



//Test Decomposition Time

  time(&t1);
  for(i=0;i<NumLUTest;i++){
      cout<<i<<" ";
      OrgLU.Rand(100);
//      cout<<OrgLU.Sparseness()<<endl;
      OrgLU.LUForm();
  }
  time(&t2);
  TotalTime=difftime(t2,t1);
  cout<<endl;
  cout<<OrgLU.Sparseness();
  cout<<"Approx. time to decompose "<<DIM<<" X "<<DIM<<
      " matrix "<<(TotalTime/NumLUTest)<<" seconds"<<endl;



//Test Multiplication Time

  Org.Rand(100);
  Mul.Rand(100);
  time(&t1);
  for(i=0;i<NumLUTest;i++){
      cout<<i<<" ";
```

```
        Temp=Org*Mul;
    }
    time(&t2);
    TotalTime=difftime(t2,t1);
    cout<<endl;
    cout<<"Approx. time to Multiply "<<DIM<<" X "<<DIM<<
        " matrix "<<(TotalTime/NumLUTest)<<" seconds"<<endl;



//Test Solution Time

    time(&t1);
    for(i=0;i<NumLUTest;i++){
        cout<<i<<" ";
        Org.Rand(100);
        for(j=1;j<=DIM;j++) {
            MatPrec Val;
            const MatPrec Scale=10.0;
            Val=((float)::rand())/RAND_MAX*Scale;
            Org.PutVal(j,j,Val);
        }
        RHS.rand(100);
        OrgLU=Org;
//      cout<<OrgLU.Sparseness()<<endl;
        Sol=OrgLU.solve(RHS);
    }
    time(&t2);
    TotalTime=difftime(t2,t1);
    cout<<endl;
    cout<<OrgLU.Sparseness();
    cout<<"Approx. time to solve "<<DIM<<" X "<<DIM<<
        " matrix "<<(TotalTime/NumLUTest)<<" seconds"<<endl;



//Determine solver Accuracy.
    Col=Org*Sol;
    for(i=1;i<=DIM;i++){
        SolAccuracy+= pow(Col.GetVal(i,1)-RHS.GetVal(i,1),2 );
    }

    SolAccuracy=sqrt(SolAccuracy)/DIM;
    cout << "Total Solver Error "<< SolAccuracy<<endl;


return 0;
}
```