

Modified from *SPARSKIT: a basic tool kit for sparse matrix computations*
by Youcef Saad (June 6, 1994)

1 Introduction

Research on sparse matrix techniques has become increasingly complex, and this trend is likely to accentuate if only because of the growing need to design efficient sparse matrix algorithms for modern supercomputers. In applications, one must often translate the matrix from some initial data structure in which it is generated, into a different desired data structure. One way around this difficulty is to restrict the number of schemes that can be used and set some standards. However, this is not enough because often the data structures are chosen for their efficiency and convenience, and it is not reasonable to ask practitioners to abandon their favorite storage schemes. What is needed is a large set of programs to translate one data structure into another. In the same vein, subroutines that generate test matrices would be extremely valuable since they would allow users to have access to a large number of matrices without the burden of actually passing large sets of data.

A useful collection of sparse matrices known as the Harwell/Boeing collection has been widely used in recent years for testing and comparison purposes...

2 Data structures for sparse matrices and the conversion routines

One of the difficulties in sparse matrix computations is the variety of types of matrices that are encountered in practical applications. The purpose of each of these schemes is to gain efficiency both in terms of memory utilization and arithmetic operations. As a result many different ways of storing sparse matrices have been devised to take advantage of the structure of the matrices or the specificity of the problem from which they arise. For example if it is known that a matrix consists of a few diagonals one may simply store these diagonals as vectors and the offsets of each diagonal with respect to the main diagonal. If the matrix is not regularly structured, then one of the most common storage schemes in use today is ... the Compressed Sparse Row (CSR) scheme. In this scheme all the nonzero entries are stored row by row in a one-dimensional real array A together with an array $JCNA$ containing their column indices and a pointer array $IPTR$ which contains the addresses in A and $JCNA$ of the beginning of each row. The order of the elements within each row does not matter. Also of importance because of its simplicity is the coordinate storage scheme in which the nonzero entries of A are stored in any order together with their row and column indices. Many of the other existing schemes are specialized to some extent.

2.1 Storage Formats

Currently, the conversion routines of SPARSKIT can handle thirteen different storage formats. These include some of the most commonly used schemes but they are by no means exhaustive. We found it particularly useful to have all these storage modes when trying to extract a matrix from someone else's application code in order, for example, to analyze it with the tools described in the next sections or, more commonly, to try a given solution method which requires a different data structure than the one originally used in the application. Often the matrix is stored in one of these modes or a variant that is very close to it...

For convenience we have decided to label by a three character name each format used. We start by listing the formats and then describe them in detail in separate subsections (except for the dense format which needs no detailed description).

DNS Dense format

BND Linpack Banded format

CSR Compressed Sparse Row format

CSC Compressed Sparse Column format

COO Coordinate format

ELL Ellpack-Itpack generalized diagonal format

DIA Diagonal format

BSR Block Sparse Row format

MSR Modified Compressed Sparse Row format

SSK Symmetric Skyline format

NSK Nonsymmetric Skyline format

LNK Linked list storage format

JAD The Jagged Diagonal format

SSS The Symmetric Sparse Skyline format

USS The Unsymmetric Sparse Skyline format

VBR Variable Block Row format

In the following sections we denote by A the matrix under consideration and by N its row dimension and NNZ the number of its nonzero elements.

2.1.1 Compressed Sparse Row and related formats (CSR, CSC and MSR)

The Compressed Sparse Row format is the basic format used in SPARSKIT. Its data structure consists of three arrays.

- A real array A containing the real values a_{ij} is stored row by row, from row 1 to N as a vector AV . The length of AV is NNZ .
- An integer array $JCNA$ containing the column indices of the elements a_{ij} as stored in the array A . The length of $JCNA$ is NNZ .
- An integer array $IPTR$ containing the pointers to the beginning of each row in the arrays A and $JCNA$. Thus the content of $IPTR(i)$ is the position in arrays A and $JCNA$ where the i -th row starts. The length of $IPTR$ is $N + 1$ with $IPTR(N + 1)$ containing the number $IPTR(1) + NNZ$, i.e., the address in A and $JCNA$ of the beginning of a fictitious row $N + 1$.

The order of the nonzero elements within the same row are not important. A variation to this scheme is to sort the elements in each row in such a way that their column positions are in increasing order. When this sorting is enforced, it is often possible to make substantial savings in the number of operations of some well-known algorithms. The Compressed Sparse Column format is identical with the Compressed Sparse Row format except that the columns of A are stored instead of the rows. In other words the Compressed Sparse Column format is simply the Compressed Sparse Row format for the matrix A^T .

The Modified Sparse Row (MSR) format is a rather common variation of the Compressed Sparse Row format which consists of keeping the main diagonal of A separately. The corresponding data structure consists of a real array A and an integer array $JCNA$. The first N positions in A contain the diagonal elements of the matrix, in order. The position $N + 1$ of the array A is not used. Starting from position $N + 2$, the nonzero elements of A , excluding its diagonal elements, are stored row-wise. Corresponding to each element $A(k)$ the integer $JCNA(k)$ is the column index of the element $A(k)$ in the matrix A . The $N + 1$ first positions of $JCNA$ contain the pointer to the beginning of each row in A and $JCNA$. The advantage of this storage mode is that many matrices have a full main diagonal, i.e., $a_{ii} \neq 0, i = 1, \dots, N$, and this diagonal is best represented by an array of length N . This storage mode is particularly useful for triangular matrices with non-unit diagonals. Often the diagonal is then stored in inverted form (i.e. $1/a_{ii}$ is stored in place of a_{ii}) because triangular systems are often solved repeatedly with the same matrix many times, as is the case for example in preconditioned Conjugate Gradient methods. The column oriented analogue of the MSR format, called MSC format, is also used in some of the other modules, but no transformation to/from it to the CSC format is necessary: for example to pass from CSC to MSC one can use the routine to pass from the CSR to the MSR formats, since the data structures are identical. The above three storage modes are used in many well-known packages.

