

Webnucleo Technical Report: Sparse Matrix Storage Schemes

David Adams

April 7, 2008

The following will provide information regarding sparse matrices, sparse matrix storage, and the particular formats thereof that the Sparse Matrix Tool itself applies. Explicitly, these formats are: Compressed Sparse Row Matrix, Coordinate Matrix, and Yale Sparse Matrix. Each will be described in detail in the sections of this report dedicated to them. The latest release of the Tool can be found at:

http://www.webnucleo.org/home/online_tools/sparse_matrix/

1 Overview of Sparse Matrices

A sparse matrix is one which a nontrivial fraction of the elements in the matrix are zero. Such a matrix is encountered frequently when dealing with real-world systems, with diagonalized matrices (see eq. 1 below) or in linear models. In nuclear astrophysics, for example, the Jacobian matrix for the reactions among nuclides in a nuclear network often has a substantial number of zero elements because each nuclear species is typically coupled only with a small fraction of the other species.

$$\begin{pmatrix} 3 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 9 \end{pmatrix} \quad (1)$$

Even with the large memory capacity of modern computers, attempting to fully model any moderately complex physical system can be quite taxing on a computer's resources. That is why when dealing with computations involving large (millions of elements) matrices with many zero elements, it is advantageous to only store the non-zero elements and to assume any element not stored in memory is zero. This is the basis of sparse matrix storage schemes, which achieve memory efficiency through elegant data handling.

2 Sparse Matrix Storage Schemes

There are many different storage mechanisms for sparse matrices, though, as previously mentioned, the Sparse Matrix Tool only currently implements three of the most prominent ones. These three vary in how they store identical data, and so will be described separately. To compare them, the storage of the same matrix

$$\begin{pmatrix} 0 & 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 5 & 0 & 0 \\ 6 & 0 & 7 & 0 & 8 \end{pmatrix} \quad (2)$$

will be demonstrated for each.

2.1 Coordinate Matrix

The Coordinate Matrix is the simplest of the three storage schemes. It is composed of three arrays:

```
int row[];   \\Size = number of non-zero elements
int column[]; \\Size = number of non-zero elements
double value[]; \\Size = number of non-zero elements
```

which, as indicated, contain the row, column, and value, respectively, for each non-zero element in the array. The elements with a value of zero are not stored. Referring to example matrix (2), the arrays would contain the following data:

index	1	2	3	4	5	6	7	8
row	1	1	2	3	4	5	5	5
column	2	4	2	5	3	1	3	5
value	1	2	3	4	5	6	7	8

where index is the index in the arrays where the corresponding value is located.

2.2 Compressed Sparse Row Matrix

The Compressed Sparse Row (or CSR) storage schemes also uses three arrays:

```
int row_pointer[]; \\Size = number of rows + 1
int column[]; \\Size = number of non-zero elements
double value[]; \\Size = number of non-zero elements
```

where column[] and value[] store the same data as the Coordinate Matrix storage scheme, and row_pointer[] stores the index of the first non-zero element in each row. To understand what this means, consider again the example matrix (2):

index	1	2	3	4	5	6	7	8
row_pointer	1	3	4	5	6	9		
column	2	4	2	5	3	1	3	5
value	1	2	3	4	5	6	7	8

The first value in `row_pointer[]` is the row in which the first non-zero element appears. Subtracting this value from the next value you gives you the number of elements in that row, and so on with the successive values in the array. The last value is the number of non-zero elements in the matrix + 1.

2.3 Yale Sparse Matrix

The Yale Sparse Matrix (or YSM) storage scheme is both the most convoluted and generally the most memory efficient of the three storage schemes. It needs only two arrays:

```
int ija[];  \Size = number of rows + number of off-diagonal elements + 1
double sa[]; \Size = number of rows + number of off-diagonal elements + 1
```

For an NxN matrix, `ija[]` and `sa[]` work in the following ways:

- The first N values in `sa[]` are the diagonal elements.
- The first N values in `ija[]` are the indices in `sa[]` of the first off-diagonal element in that row.
- N + 1 in `ija[]` is the size of the arrays + 1. N + 1 in `sa[]` is not used.
- The values starting at N + 2 in `sa[]` are the values of the off-diagonal elements.
- The values starting at N + 2 in `ija[]` are the columns for the off-diagonal elements.

For the example matrix (2), this means:

index	1	2	3	4	5	6	7	8	9	10	11	12
ija	7	8	9	10	11	13	2	4	5	3	1	3
sa	0	3	0	0	8	X	1	2	4	5	6	7

Note that YSM stores all diagonal elements, even if they are zero. Only non-zero off-diagonal elements are stored.

3 Storage Scheme Selection

Each of the three schemes provided carry with them their own strengths and weaknesses. The Coordinate Matrix storage scheme has the benefit of being the most straightforward of the three, but suffers from having the least efficient

storage. Compressed Sparse Row is slightly more complicated than the Coordinate Matrix and, in general, slightly less efficient than the Yale Sparse Matrix. However, if there are many zero elements on the diagonals of the matrix, it may be more efficient. The Yale Sparse Matrix is certainly the most complex of the three, but generally the most memory efficient, and especially when most of the non-zero elements are on the diagonals.

As stated earlier, the storage schemes provided by the Sparse Matrix Tool represent only a portion of the total number of sparse matrix storage schemes in existence. While there are currently no plans to add any more of them to the Tool, if a new scheme is desired, users should request it on the Tool Request a Feature page.