Carleton University
Department of Electronics

ELEC 5508

Computer Methods for Analysis and Design of VLSI and Communication

Sparse Matrix Primer for Matlab

1. About Sparse Matrices

By now, you have probably noticed that the matrices that we use when doing circuit simulations consist mainly of zero elements. These types of matrices are known as *sparse matrices*, and ordinary matrices where there are few zero elements are known as *full* or *dense matrices*. By default, Matlab will treat all matrices as dense. However, large improvements in efficiency can be achieved if you design your programs to take advantage of the large number of zeroes in your matrices. To do this, you need to tell Matlab that your matrices are sparse and use special functions that work with and preserve this structure.

2. Sparse Matrix Creation and Conversion

In this section, the commands that are useful for creating sparse matrices and converting between sparse and dense matrices are presented. Where appropriate, the equivalent dense matrix function is given to assist you in converting your dense matrix code to use sparse matrices.

2.1. Creating an Empty Sparse Matrix

Function: `A = sparse(m,n)` Dense Equivalent: `A = zeros(m,n)`

This function creates an empty *m*-by-*n* sparse matrix.

2.2. Creating a Sparse Identity Matrix

Function: `A = speye(m,n)` Dense Equivalent: `A = eye(m,n)`

This function creates an m-by-n identity matrix (i.e. a matrix where the diagonal elements are ones.)

2.3. Creating a Sparse Diagonal Matrix

Function: `A = spdiags(D, 0, m, n)` Dense Equivalent: `A = diag(D)`

This function creates a sparse *m*-by-*n* matrix with the diagonal entries set to the corresponding elements of the column vector *D*. The zero for the second parameter indicates that the elements should be places on the main (central) diagonal. Both *m* and *n* should be set to `length(D)`.

2.4. Conversion between Sparse and Dense Forms

Function: `S = sparse(D)`

This function will convert the dense matrix *D* into its sparse form stored in *S* by stripping out all zero elements.

Function: `D = full(S)`

This function will convert the sparse matrix *S* to its dense form to be stored in *D* by adding in all omitted zero elements.

3. Other Useful Functions

Function: `n = nnz(S)`

This function returns the number of nonzero elements in matrix *S*.

Function: `spy(S)`

Plots a sparsity pattern of the matrix *S*. All non-zero elements will show up with markers, allowing you to quickly see the pattern of elements in the matrix, and how your operations are affecting the level of sparsity.

4. Working with Sparse Matrices

For the most part, sparse matrices can be used wherever dense matrices are used. In particular, addressing elements of a sparse matrix is done the same way as for dense matrices. If *A* is a sparse matrix, `A(2,3)` evaluates to the element at row 2, column 3. If this value is not present, zero is returned. `A(2,3) = 5` will assign the value 5 to row 2, column 3, creating storage automatically for this element if needed.

The main thing to watch out for is that some operations will result in a conversion to a dense matrix. The rule of thumb is that if all your operands are sparse matrices and scalars, the result will be a sparse matrix. If you include one dense matrix, you will be left with a dense matrix as a result.

To illustrate this, here are a few examples:

A = 3x3 dense matrix
B = 3x3 sparse matrix
v = 3x1 dense vector
x = 3x1 sparse vector

`A * A` → 3x3 dense matrix
`A * B` → 3x3 dense matrix
`B * A` → 3x3 dense matrix
`B * B` → 3x3 sparse matrix
`A\v` → 3x1 dense vector
`A\x` → 3x1 dense vector

$B\backslash v$ → 3x1 dense vector
$B\backslash x$ → 3x1 sparse vector

If you are ever in doubt if an operation on a sparse matrix is returning a dense matrix, you can use the **`issparse()`** function on the result to test for sparsity. It will evaluate to 1 if the result is sparse.

5. Solving Sparse Systems of Equations

The main advantages of sparse matrices are seen when solving systems of equations in the standard form $Ax=b$ where A is sparse. While you can directly solve these systems in the same way as dense systems, there are a few considerations that should be made to get the best efficiency and accuracy from the process. The first is that it is important to ensure the matrix is ordered in a certain way. The process to accomplish this is known as matrix pivoting.

5.1. Pivoting for Accuracy

When solving a system of equations numerically, the order of the rows can have an effect on the accuracy of the solution. This is due to the limited accuracy of floating-point numbers in computers, and certain calculations can introduce large errors. This is easiest to show with an example.

Consider this system:

$$\begin{bmatrix} 10^{-4} & 1 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}=\begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

If we take the LU factorization of this, we get:

$$\begin{bmatrix} 1 & 0 \\ 10^{4} & 1 \end{bmatrix}\begin{bmatrix} 10^{-4} & 1 \\ 0 & -9999 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}=\begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

If we solve this, we get $x_1=-1.0001$ and $x_2=2.0001$. Assume, however, that our machine can only store 3 significant figures. This will give us the following LU factorization:

$$\begin{bmatrix} 1 & 0 \\ 10^{4} & 1 \end{bmatrix}\begin{bmatrix} 10^{-4} & 1 \\ 0 & -9990 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}=\begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

And, solving this, we get $x_1=-19.0$ and $x_2=2.00$, which is clearly very inaccurate.

If, instead, we swapped the two rows in the original matrix, and then did the LU factorization:

$$\begin{bmatrix} 1 & 1 \\ 10^{-4} & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}=\begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 10^{-4} & 1 \end{bmatrix}\begin{bmatrix} 1 & 1 \\ 0 & 0.999 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}=\begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

We would get $x_1=-1.00$ and $x_2=2.00$, which is much closer to the correct solution.

This pivoting method is called *partial pivoting*. At each step of the LU factorization process, we find the row with the largest value in the column below the current diagonal and swap it with the current row. This helps reduce the numerical errors caused by rounding.

5.2. Pivoting for Efficiency

When dealing with sparse matrices, there is another concern on top of accuracy. To have the highest efficiency, we have to avoid the fill-in phenomenon that occurs during LU factorization. This happens when we have a situation like the following matrix:

$$\begin{bmatrix} x & x & x & x \\ x & x & 0 & 0 \\ x & 0 & x & 0 \\ x & 0 & 0 & x \end{bmatrix}$$, where x is any non-zero element.

In this case, as the LU process proceeds, every zero element will be replaced with a value, destroying the sparsity of our matrix, and we will be left with dense L and U matrices. Clearly, we would prefer to try to preserve the sparsity of the matrix. Once again, this is done with pivoting. If we reorder this matrix like this:

$$\begin{bmatrix} x & 0 & 0 & x \\ 0 & x & 0 & x \\ 0 & 0 & x & x \\ x & x & x & x \end{bmatrix}$$

The resulting LU factorization will look like this:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & x & x & 1 \end{bmatrix} \begin{bmatrix} x & 0 & 0 & x \\ 0 & x & 0 & x \\ 0 & 0 & x & x \\ 0 & 0 & 0 & x \end{bmatrix}$$

There are many algorithms to determine how to reorder these matrices, and it is an active research topic. The best algorithm is called *minimum fill-in*, but this is an NP-complete algorithm and is expensive to use. There have been many approximations developed, such as *approximate minimum fill-in* and *minimum degree.* Thankfully, some good algorithms based on minimum degree have been included in Matlab.

5.3. Sparse Matrix Pivoting and LU in Matlab

When working with sparse matrices in Matlab, the **`lu`** function is extended to include support for pivoting. Its syntax is as follows:

Function: **`[`***`L`***`,`***`U`***`,`***`P`***`,`***`Q`***`] = lu(`***`A`***`, `***`threshold`***`)`**

This function performs an LU decomposition of the sparse matrix *A*. Depending on the value of *threshold*, the pivoting method will either be partial pivoting (if *threshold* is 1.0), pure minimum degree (if *threshold* is 0.0), or a combination of the two. This gives a tradeoff between accuracy and efficiency. The default value is 0.1, which is usually a good choice.

The first step in this extended LU process is to perform the matrix pivoting as specified by the threshold. Instead of finding $LU = A$ as in the standard LU process, it finds instead $LU = PAQ$. P and Q are special matrices known as permutation matrices which have the function of exchanging the rows and columns of A through the multiplication process.

To use this LU factorization, we have to first transform $Ax=b$ into an equation containing PAQ. First, we premultiply both sides by P:

$$Ax = b \;\rightarrow\; PA\,x = Pb$$

We then define two new variables, y and b_1, defined by:

$$x = Qy$$

$$b_1 = Pb$$

Substituting these in, we get:

$$PAQy = b_1$$

And, since $PAQ=LU$:

$$LUy = b_1$$

Next, solve for y using forward and back substitution:

$$Lz = b_1$$

$$Uy = z$$

Finally, x is found with:

$$x = Qy$$

For sparse matrices, efficient forward and back substitution algorithms are implemented in the \ operator. Therefore an efficient solution of the sparse system $Ax=b$ could be done with the following:

```
[L,U,P,Q] = lu(A, threshold);
z = L\(P*b);
y = U\z;
x = Q*y;
```