9

Sparse Matrices

MATLAB supports *sparse matrices*, matrices that contain a small proportion of nonzero elements. This characteristic provides advantages in both matrix storage space and computation time.

This chapter explains how to create sparse matrices in MATLAB, and how to use them in both specialized and general mathematical operations.

The sparse matrix functions are located in the `sparfun` directory in the MATLAB `toolbox` directory.

Category	Function	Description
Elementary sparse matrices	`speye`	Sparse identity matrix.
	`sprand`	Sparse uniformly distributed random matrix.
	`sprandn`	Sparse normally distributed random matrix.
	`sprandsym`	Sparse random symmetric matrix.
	`spdiags`	Sparse matrix formed from diagonals.
Full to sparse conversion	`sparse`	Create sparse matrix.
	`full`	Convert sparse matrix to full matrix.
	`find`	Find indices of nonzero elements.
	`spconvert`	Import from sparse matrix external format.
Working with sparse matrices	`nnz`	Number of nonzero matrix elements.
	`nonzeros`	Nonzero matrix elements.
	`nzmax`	Amount of storage allocated for nonzero matrix elements.
	`spones`	Replace nonzero sparse matrix elements with ones.
	`spalloc`	Allocate space for sparse matrix.
	`issparse`	True for sparse matrix.

Category	Function	Description
	spfun	Apply function to nonzero matrix elements.
	spy	Visualize sparsity pattern.
	gplot	Plot graph, as in “graph theory.”
Reordering algorithms	colmmd	Column minimum degree permutation.
	symmmd	Symmetric minimum degree permutation.
	symrcm	Symmetric reverse Cuthill-McKee permutation.
	colperm	Column permutation.
	randperm	Random permutation.
	dmperm	Dulmage-Mendelsohn permutation.
Linear algebra	eigs	A few eigenvalues.
	svds	A few singular values.
	luinc	Incomplete LU factorization.
	cholinc	Incomplete Cholesky factorization.
	normest	Estimate the matrix 2-norm.
	condest	1-norm condition number estimate.
	sprank	Structural rank.
Linear equations (iterative methods)	pcg	Preconditioned Conjugate Gradients Method.
	bicg	BiConjugate Gradients Method.
	bicgstab	BiConjugate Gradients Stabilized Method.
	cgs	Conjugate Gradients Squared Method.
	gmres	Generalized Minimum Residual Method.

Category	Function	Description
	qmr	Quasi-Minimal Residual Method.
Miscellaneous	symbfact	Symbolic factorization analysis.
	spparms	Set parameters for sparse matrix routines.
	spaugment	Form least squares augmented system.

Introduction

Sparse matrices are a special class of matrices that contain a significant number of zero-valued elements. This property allows MATLAB to:

- Store only the nonzero elements of the matrix, together with their indices.
- Reduce computation time by eliminating operations on zero elements.

Sparse Matrix Storage

For full matrices, MATLAB stores internally every matrix element. Zero-valued elements require the same amount of storage space as any other matrix element. For sparse matrices, however, MATLAB stores only the nonzero elements and their indices. For large matrices with a high percentage of zero-valued elements, this scheme significantly reduces the amount of memory required for data storage.

MATLAB uses three arrays internally to store sparse matrices with real elements. Consider an *m*-by-*n* sparse matrix with `nnz` nonzero entries:

- The first array contains all the nonzero elements of the array in floating-point format. The length of this array is equal to `nnz`.
- The second array contains the corresponding integer row indices for the nonzero elements. This array also has length equal to `nnz`.
- The third array contains integer pointers to the start of each column. This array has length equal to `n`.

This matrix requires storage for `nnz` floating-point numbers and `nnz+n` integers. At 8 bytes per floating-point number and 4 bytes per integer, the total number of bytes required to store a sparse matrix is

```
8*nnz + 4*(nnz+n)
```

Sparse matrices with complex elements are also possible. In this case, MATLAB uses a fourth array with `nnz` elements to store the imaginary parts of the nonzero elements. An element is considered nonzero if either its real or imaginary part is nonzero.

Creating Sparse Matrices

MATLAB never creates sparse matrices automatically. Instead, you must determine if a matrix contains a large enough percentage of zeros to benefit from sparse techniques.

The *density* of a matrix is the number of non-zero elements divided by the total number of matrix elements. Matrices with very low density are often good candidates for use of the sparse format.

Converting Full to Sparse

You can convert a full matrix to sparse storage using the `sparse` function with a single argument.

```
S = sparse(A)
```

For example

```
A = [ 0   0   0   5
      0   2   0   0
      1   3   0   0
      0   0   4   0];

S = sparse(A)
```

produces

```
   S =

      (3,1)         1
      (2,2)         2
      (3,2)         3
      (4,3)         4
      (1,4)         5
```

The printed output lists the nonzero elements of `S`, together with their row and column indices. The elements are sorted by columns, reflecting the internal data structure.

You can convert a sparse matrix to full storage using the `full` function, provided the matrix order is not too large. For example `A = full(S)` reverses the example conversion.

Converting a full matrix to sparse storage is not the most frequent way of generating sparse matrices. If the order of a matrix is small enough that full storage is possible, then conversion to sparse storage rarely offers significant savings.

Creating Sparse Matrices Directly

You can create a sparse matrix from a list of nonzero elements using the `sparse` function with five arguments.

```
S = sparse(i,j,s,m,n)
```

`i` and `j` are vectors of row and column indices, respectively, for the nonzero elements of the matrix. `s` is a vector of nonzero values whose indices are specified by the corresponding `(i,j)` pairs. `m` is the row dimension for the resulting matrix, and `n` is the column dimension.

The matrix `S` of the previous example can be generated directly with

```
S = sparse([3 2 3 4 1],[1 2 2 3 4],[1 2 3 4 5],4,4)

S =

   (3,1)        1
   (2,2)        2
   (3,2)        3
   (4,3)        4
   (1,4)        5
```

The `sparse` command has a number of alternate forms. The example above uses a form that sets the maximum number of nonzero elements in the matrix to `length(s)`. If desired, you can append a sixth argument that specifies a larger maximum, allowing you to add nonzero elements later without changing storage requirements.

Example: The Second Difference Operator

The matrix representation of the second difference operator is a good example of a sparse matrix. It is a tridiagonal matrix with –2s on the diagonal and 1s

on the super- and subdiagonal. There are many ways to generate it – here's one possibility.

```
D = sparse(1:n,1:n,-2*ones(1,n),n,n);
E = sparse(2:n,1:n-1,ones(1,n-1),n,n);
S = E+D+E'
```

For `n = 5`, MATLAB responds with

```
S =

   (1,1)       -2
   (2,1)        1
   (1,2)        1
   (2,2)       -2
   (3,2)        1
   (2,3)        1
   (3,3)       -2
   (4,3)        1
   (3,4)        1
   (4,4)       -2
   (5,4)        1
   (4,5)        1
   (5,5)       -2
```

Now `F = full(S)` displays the corresponding full matrix.

```
F = full(S)

F =

    -2     1     0     0     0
     1    -2     1     0     0
     0     1    -2     1     0
     0     0     1    -2     1
     0     0     0     1    -2
```

Creating Sparse Matrices from Their Diagonal Elements

Creating sparse matrices based on their diagonal elements is a common operation, so the function `spdiags` handles this task. Its syntax is

```
S = spdiags(B,d,m,n)
```

To create an output matrix S of size *m*-by-*n* with elements on p diagonals:

- B is a matrix of size `min(`*m*`,`*n*`)`-by-*p*. The columns of B are the values to populate the diagonals of S.
- d is a vector of length p whose integer elements specify which diagonals of S to populate.

That is, the elements in column j of B fill the diagonal specified by element j of d. As an example, consider the matrix B and the vector d.

```
B =

    41    11     0
    52    22     0
    63    33    13
    74    44    24

d =

    -3
     0
     2
```

Use these matrices to create a 7-by-4 sparse matrix A.

```
A = spdiags(B,d,7,4)

A =

   (1,1)        11
   (4,1)        41
   (2,2)        22
   (5,2)        52
   (1,3)        13
   (3,3)        33
   (6,3)        63
   (2,4)        24
   (4,4)        44
   (7,4)        74
```

In its full form, `A` looks like this.

```
full(A)

ans =

    11     0    13     0
     0    22     0    24
     0     0    33     0
    41     0     0    44
     0    52     0     0
     0     0    63     0
     0     0     0    74
```

`spdiags` can also extract diagonal elements from a sparse matrix, or replace matrix diagonal elements with new values. Type `help spdiags` for details.

Importing Sparse Matrices from Outside MATLAB

You can import sparse matrices from computations outside MATLAB. Use the `spconvert` function in conjunction with the `load` command to import text files containing lists of indices and nonzero elements. For example, consider a three-column text file `T.dat` whose first column is a list of row indices, second column is a list of column indices, and third column is a list of nonzero values. These statements load `T.dat` into MATLAB and convert it into a sparse matrix `S`:

```
load T.dat
S = spconvert(T)
```

The `save` and `load` commands can also process sparse matrices stored as binary data in MAT-files. Finally, a Fortran utility routine `hbo2mat` is available to convert a file containing a sparse matrix in the Harwell-Boeing format into a MAT-file that `load` can process. The Harwell-Boeing data is available through anonymous ftp or the World Wide Web from `ftp.mathworks.com` in the directory `pub/mathworks/toolbox/matlab/sparfun`.

Viewing Sparse Matrices

MATLAB provides a number of functions that let you get quantitative or graphical information about sparse matrices.

General Storage Information

The `whos` command provides high-level information about matrix storage, including size and storage class. For example, this `whos` listing shows information about sparse and full versions of the same matrix.

```
whos
  Name          Size         Bytes  Class

  M_full      1100x1100     9680000  double array
  M_sparse    1100x1100        4404  sparse array

Grand total is 1210000 elements using 9684404 bytes
```

Notice that the number of bytes used is much less in the sparse case, because zero-valued elements are not stored. In this case, the density of the sparse matrix is 4404/9680000, or approximately .00045%.

Information About Nonzero Elements

There are several commands that provide high-level information about the nonzero elements of a sparse matrix:

- `nnz` returns the number of nonzero elements in a sparse matrix.
- `nonzeros` returns a column vector containing all the nonzero elements of a sparse matrix.
- `nzmax` returns the amount of storage space allocated for the nonzero entries of a sparse matrix.

To try some of these, load the supplied sparse matrix `west0479`, one of the Harwell-Boeing collection.

```
load west0479
whos
  Name           Size         Bytes  Class

  west0479     479x479        24576  sparse array
```

This matrix models an eight-stage chemical distillation column.

Try these commands.

```
nnz(west0479)

ans =

        1887

format short e
west0479

west0479 =

  (25,1)       1.0000e+00
  (31,1)      -3.7648e-02
  (87,1)      -3.4424e-01
  (26,2)       1.0000e+00
  (31,2)      -2.4523e-02
  (88,2)      -3.7371e-01
  (27,3)       1.0000e+00
  (31,3)      -3.6613e-02
  (89,3)      -8.3694e-01
  (28,4)       1.3000e+02
    .
    .
    .

nonzeros(west0479);
```

```
ans =

    1.0000e+00
   -3.7648e-02
   -3.4424e-01
    1.0000e+00
   -2.4523e-02
   -3.7371e-01
    1.0000e+00
   -3.6613e-02
   -8.3694e-01
    1.3000e+02
       .
       .
       .
```

Note Use **Ctrl-C** to stop the `nonzeros` listing at any time.

Note that initially `nnz` has the same value as `nzmax` by default. That is, the number of nonzero elements is equivalent to the number of storage locations allocated for nonzeros. However, MATLAB does not dynamically release memory if you zero out additional array elements. Changing the value of some matrix elements to zero changes the value of `nnz`, but not that of `nzmax`.

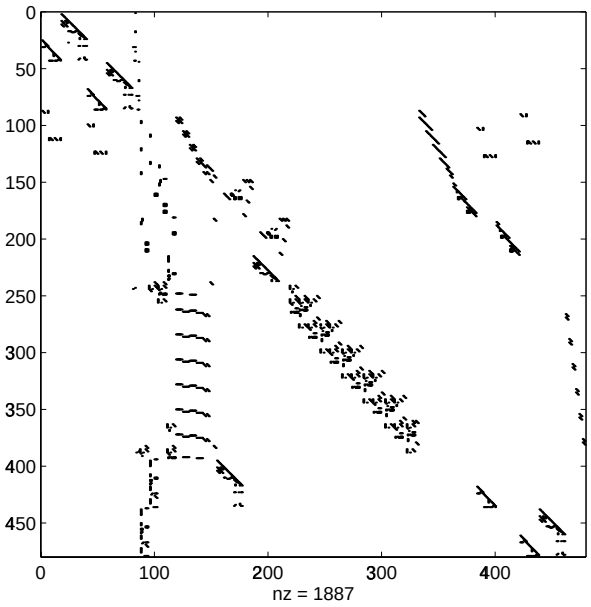
You can add as many nonzero elements to the matrix as desired, however; you are not constrained by the original value of `nzmax`.

Viewing Sparse Matrices Graphically

It is often useful to use a graphical format to view the distribution of the nonzero elements within a sparse matrix. MATLAB's `spy` function produces a template view of the sparsity structure, where each point on the graph represents the location of a nonzero array element.

For example,

```
spy(west0479)
```



The find Function and Sparse Matrices

For any matrix, full or sparse, the `find` function returns the indices and values of nonzero elements. Its syntax is:

```
[i,j,s] = find(S)
```

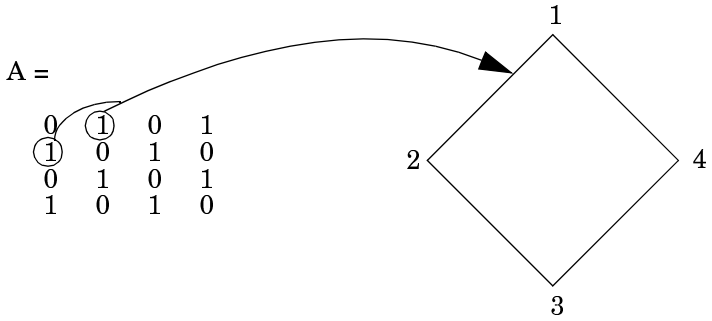
`find` returns the row indices of nonzero values in vector `i`, the column indices in vector `j`, and the nonzero values themselves in the vector `s`. The example below uses `find` to locate the indices and values of the nonzeros in a sparse matrix. The `sparse` function uses the `find` output, together with the size of the matrix, to recreate the matrix.

```
[i,j,s] = find(S)
[m,n] = size(S)
S = sparse(i,j,s,m,n)
```

Example: Adjacency Matrices and Graphs

The formal mathematical definition of a *graph* is a set of points, or nodes, with specified connections between them. An economic model, for example, is a graph with different industries as the nodes and direct economic ties as the connections. The computer software industry is connected to the computer hardware industry, which, in turn, is connected to the semiconductor industry, and so on.

This definition of a graph lends itself to matrix representation. The *adjacency matrix* of an *undirected* graph is a matrix whose `(i,j)`-th and `(j,i)`-th entries are 1 if node `i` is connected to node `j`, and 0 otherwise. For example, the adjacency matrix for a diamond-shaped graph looks like



Since most graphs have relatively few connections per node, most adjacency matrices are sparse. The actual locations of the nonzero elements depend on how the nodes are numbered. A change in the numbering leads to permutation of the rows and columns of the adjacency matrix, which can have a significant effect on both the time and storage requirements for sparse matrix computations.

Graphing Using Adjacency Matrices

MATLAB's `gplot` function creates a graph based on an adjacency matrix and a related array of coordinates. To try `gplot`, create the adjacency matrix shown above by entering

```
A = [0 1 0 1; 1 0 1 0; 0 1 0 1; 1 0 1 0];
```

The columns of `gplot`'s coordinate array contain the Cartesian coordinates for the corresponding node. For the diamond example, create the array by entering

```
xy = [1 3; 2 1; 3 3; 2 5];
```

This places the first node at location `(1,3)`, the second at location `(2,1)`, the third at location `(3,3)`, and the fourth at location `(2,5)`. To view the resulting graph, enter

```
gplot(A,xy)
```

The Bucky Ball

One interesting construction for graph analysis is the *Bucky ball*. This is composed of 60 points distributed on the surface of a sphere in such a way that the distance from any point to its nearest neighbors is the same for all the points. Each point has exactly three neighbors. The Bucky ball models four different physical objects:

- The geodesic dome popularized by Buckminster Fuller
- The C_{60} molecule, a form of pure carbon with 60 atoms in a nearly spherical configuration
- In geometry, the truncated icosahedron
- In sports, the seams in a soccer ball

The Bucky ball adjacency matrix is a 60-by-60 symmetric matrix `B`. `B` has three nonzero elements in each row and column, for a total of 180 nonzero values. This matrix has important applications related to the physical objects listed earlier. For example, the eigenvalues of `B` are involved in studying the chemical properties of C_{60}.

To obtain the Bucky ball adjacency matrix, enter

```
B = bucky;
```

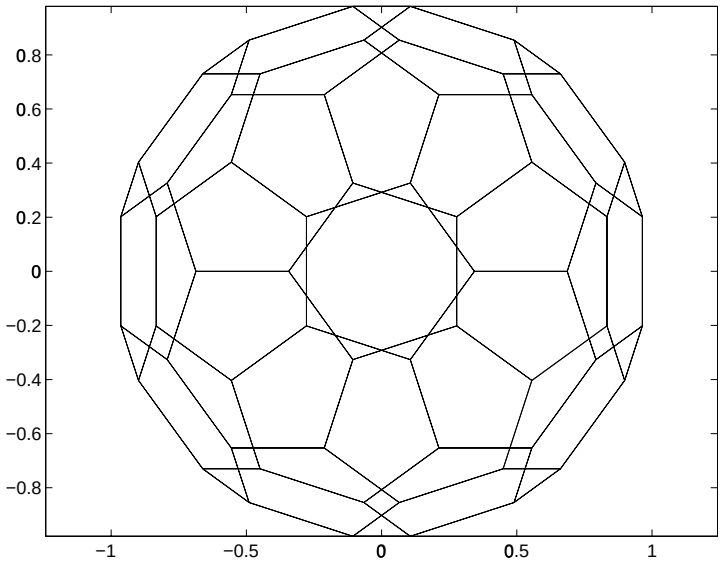
At order 60, and with a density of 5%, this matrix does not require sparse techniques, but it does provide an interesting example.

You can also obtain the coordinates of the Bucky ball graph using

```
[B,v] = bucky;
```

This statement generates `v`, a list of *xyz*-coordinates of the 60 points in 3-space equidistributed on the unit sphere. The function `gplot` uses these points to plot the Bucky ball graph.

```
gplot(B,v)
axis equal
```

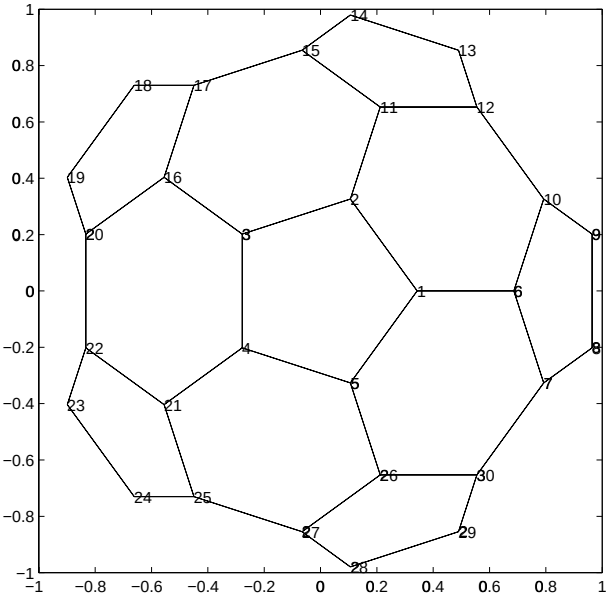


It is not obvious how to number the nodes in the Bucky ball so that the resulting adjacency matrix reflects the spherical and combinatorial symmetries of the graph. The numbering used by `bucky.m` is based on the pentagons inherent in the ball's structure.

The vertices of one pentagon are numbered 1 through 5, the vertices of an adjacent pentagon are numbered 6 through 10, and so on. The picture on the following page shows the numbering of half of the nodes (one hemisphere); the numbering of the other hemisphere is obtained by a reflection about the

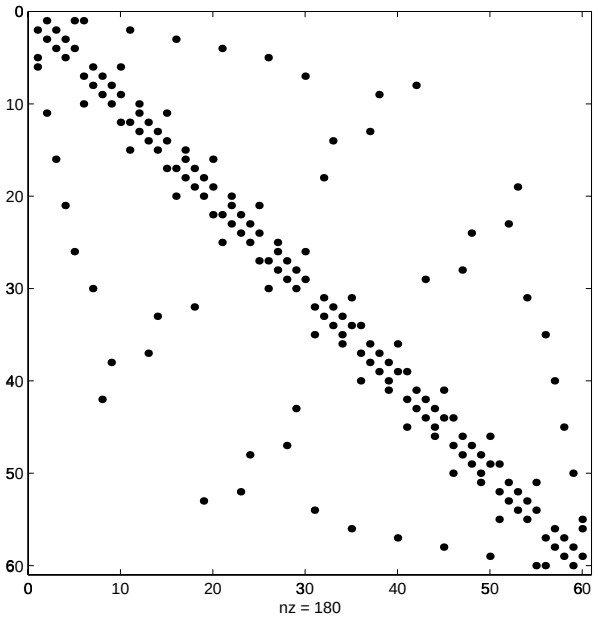
equator. Use `gplot` to produce a graph showing half the nodes. You can add the node numbers using a `for` loop.

```
k = 1:30;
gplot(B(k,k),v);
axis square
for j = 1:30, text(v(j,1),v(j,2), int2str(j)); end
```



To view a template of the nonzero locations in the Bucky ball's adjacency matrix, use the `spy` function:

```
spy(B)
```

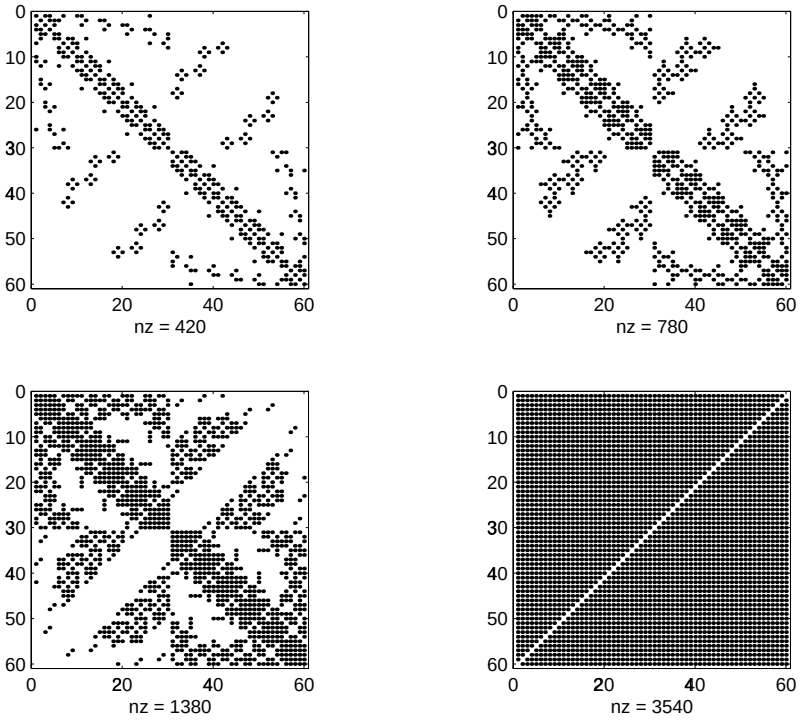


The node numbering that this model uses generates a spy plot with twelve groups of five elements, corresponding to the twelve pentagons in the structure. Each node is connected to two other nodes within its pentagon and one node in some other pentagon. Since the nodes within each pentagon have consecutive numbers, most of the elements in the first super- and sub-diagonals of `B` are nonzero. In addition, the symmetry of the numbering about the equator is apparent in the symmetry of the spy plot about the antidiagonal.

Graphs and Characteristics of Sparse Matrices

Spy plots of the matrix powers of `B` illustrate two important concepts related to sparse matrix operations, fill-in and distance. `spy` plots help illustrate these concepts.

```
spy(B^2)
spy(B^3)
spy(B^4)
spy(B^8)
```

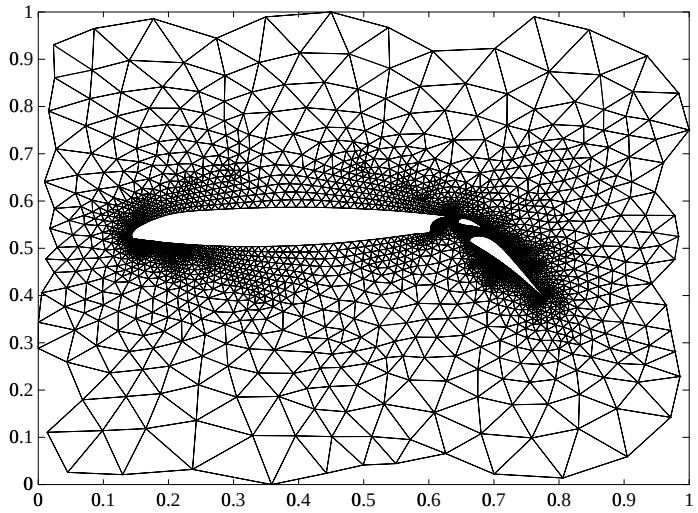


Fill-in is generated by operations like matrix multiplication. The product of two or more matrices usually has more nonzero entries than the individual terms, and so requires more storage. As `p` increases, `B^p` fills in and `spy(B^p)` gets more dense.

The *distance* between two nodes in a graph is the number of steps on the graph necessary to get from one node to the other. The spy plot of the p-th power of `B` shows the nodes that are a distance `p` apart. As `p` increases, it is possible to get to more and more nodes in `p` steps. For the Bucky ball, `B^8` is almost completely full. Only the antidiagonal is zero, indicating that it is possible to get from any node to any other node, except the one directly opposite it on the sphere, in eight steps.

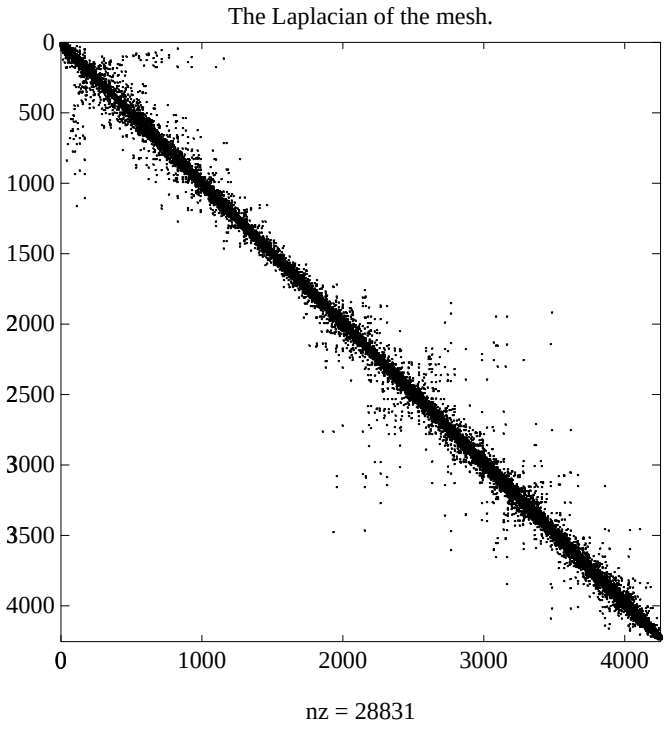
An Airflow Model

A calculation performed at NASA's Research Institute for Applications of Computer Science involves modeling the flow over an airplane wing with two trailing flaps.



In a two-dimensional model, a triangular grid surrounds a cross section of the wing and flaps. The partial differential equations are nonlinear and involve several unknowns, including hydrodynamic pressure and two components of velocity. Each step of the nonlinear iteration requires the solution of a sparse linear system of equations. Since both the connectivity and the geometric location of the grid points are known, the `gplot` function can produce the graph shown above.

In this example, there are 4253 grid points, each of which is connected to between 3 and 9 others, for a total of 28831 nonzeros in the matrix, and a density equal to 0.0016. This spy plot shows that the node numbering yields a definite band structure.



Sparse Matrix Operations

Most of MATLAB's standard mathematical functions work on sparse matrices just as they do on full matrices. In addition, MATLAB provides a number of functions that perform operations specific to sparse matrices. This section discusses:

- Computational Considerations
- Standard Mathematical Operations
- Permutation and Reordering
- Factorization
- Simultaneous Linear Equations
- Eigenvalues and Singular Values

Computational Considerations

The computational complexity of sparse operations is proportional to `nnz`, the number of nonzero elements in the matrix. Computational complexity also depends linearly on the row size `m` and column size `n` of the matrix, but is independent of the product `m*n`, the total number of zero and nonzero elements.

The complexity of fairly complicated operations, such as the solution of sparse linear equations, involves factors like ordering and fill-in, which are discussed in the previous section. In general, however, the computer time required for a sparse matrix operation is proportional to the number of arithmetic operations on nonzero quantities. This is the "time is proportional to flops" rule.

Standard Mathematical Operations

Sparse matrices propagate through computations according to these rules:

- Functions that accept a matrix and return a scalar or vector always produce output in full storage format. For example, the `size` function always returns a full vector, whether its input is full or sparse.
- Functions that accept scalars or vectors and return matrices, such as `zeros`, `ones`, `rand`, and `eye`, always return full results. This is necessary to avoid introducing sparsity unexpectedly. The sparse analog of `zeros(m,n)` is simply `sparse(m,n)`. The sparse analogs of `rand` and `eye` are `sprand` and `speye`, respectively. There is no sparse analog for the function `ones`.

- Unary functions that accept a matrix and return a matrix or vector preserve the storage class of the operand. If `S` is a sparse matrix, then `chol(S)` is also a sparse matrix, and `diag(S)` is a sparse vector. Columnwise functions such as `max` and `sum` also return sparse vectors, even though these vectors may be entirely nonzero. Important exceptions to this rule are the `sparse` and `full` functions.
- Binary operators yield sparse results if both operands are sparse, and full results if both are full. For mixed operands, the result is full unless the operation preserves sparsity. If `S` is sparse and `F` is full, then `S+F`, `S*F`, and `F\S` are full, while `S.*F` and `S&F` are sparse. In some cases, the result might be sparse even though the matrix has few zero elements.
- Matrix concatenation using either the `cat` function or square brackets produces sparse results for mixed operands.
- Submatrix indexing on the right side of an assignment preserves the storage format of the operand. `T = S(i,j)` produces a sparse result if `S` is sparse whether `i` and `j` are scalars or vectors. Submatrix indexing on the left, as in `T(i,j) = S`, does not change the storage format of the matrix on the left.

Permutation and Reordering

A permutation of the rows and columns of a sparse matrix `S` can be represented in two ways:

- A permutation matrix `P` acts on the rows of `S` as `P*S` or on the columns as `S*P'`.
- A permutation vector `p`, which is a full vector containing a permutation of `1:n`, acts on the rows of `S` as `S(p,:)`, or on the columns as `S(:,p)`.

For example, the statements

```
p = [1 3 4 2 5]
I = eye(5,5);
P = I(p,:);
e = ones(4,1);
S = diag(11:11:55) + diag(e,1) + diag(e,-1)
```

produce

```
p =

     1     3     4     2     5

P =

     1     0     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     1     0     0     0
     0     0     0     0     1

S =

    11     1     0     0     0
     1    22     1     0     0
     0     1    33     1     0
     0     0     1    44     1
     0     0     0     1    55
```

You can now try some permutations using the permutation vector `p` and the permutation matrix `P`. For example, the statements `S(p,:)` and `P*S` produce

```
ans =

    11     1     0     0     0
     0     1    33     1     0
     0     0     1    44     1
     1    22     1     0     0
     0     0     0     1    55
```

Similarly, `S(:,p)` and `S*P'` produce

```
ans =

    11     0     0     1     0
     1     1     0    22     0
     0    33     1     1     0
     0     1    44     0     1
     0     0     1     0    55
```

If P is a sparse matrix, then both representations use storage proportional to n and you can apply either to S in time proportional to nnz(S). The vector representation is slightly more compact and efficient, so the various sparse matrix permutation routines all return full row vectors with the exception of the pivoting permutation in LU (triangular) factorization, which returns a matrix compatible with earlier versions of MATLAB.

To convert between the two representations, let I = speye(n) be an identity matrix of the appropriate size. Then,

```
P = I(p,:)
P' = I(:,p).
p = (1:n)*P'
p = (P*(1:n)')'
```

The inverse of P is simply R = P'. You can compute the inverse of p with r(p) = 1:n.

```
r(p) = 1:5

r =

     1     4     2     3     5
```

Reordering for Sparsity

Reordering the columns of a matrix can often make its Cholesky, LU, or QR factors sparser. The simplest such reordering is to sort the columns by nonzero count. This is sometimes a good reordering for matrices with very irregular structures, especially if there is great variation in the nonzero counts of rows or columns.

The function p = colperm(S) computes this column-count permutation. The colperm M-file has only a single line.

```
[ignore,p] = sort(full(sum(spones(S))));
```

This line performs these steps:

1. The inner call to spones creates a sparse matrix with ones at the location of every nonzero element in S.
2. The sum function sums down the columns of the matrix, producing a vector that contains the count of nonzeros in each column.

3 `full` converts this vector to full storage format.

4 `sort` sorts the values in ascending order. The second output argument from `sort` is the permutation that sorts this vector.

Reordering to Reduce Bandwidth

The reverse Cuthill-McKee ordering is intended to reduce the profile or bandwidth of the matrix. It is not guaranteed to find the smallest possible bandwidth, but it usually does. The function `symrcm(A)` actually operates on the nonzero structure of the symmetric matrix `A + A'`, but the result is also useful for asymmetric matrices. This ordering is useful for matrices that come from one-dimensional problems or problems that are in some sense "long and thin."

Minimum Degree Ordering

The degree of a node in a graph is the number of connections to that node, which is the same as the number of nonzero elements in the corresponding row of the adjacency matrix. The minimum degree algorithm generates an ordering based on how these degrees are altered during Gaussian elimination. It is a complicated and powerful algorithm that usually leads to sparser factors than most other orderings, including column count and reverse Cuthill-McKee. MATLAB has two versions, `symmd` for symmetric matrices and `colmmd` for nonsymmetric matrices. You can change various parameters associated with details of the algorithm using the `spparms` function.

For more details on the algorithm and MATLAB's version of it, see Gilbert, John R., Cleve Moler, and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM J. Matrix Anal. Appl.*, Vol. 13, No. 1. January 1992: pp. 333-356.

Factorization

This section discusses four important factorization techniques for sparse matrices:

- LU, or triangular, factorization
- Cholesky factorization
- QR, or orthogonal factorization
- Incomplete factorizations

LU Factorization

If S is a sparse matrix, the statement below returns three sparse matrices L, U, and P such that P*S = L*U.

```
[L,U,P] = lu(S)
```

lu obtains the factors by Gaussian elimination with partial pivoting. The permutation matrix P has only n nonzero elements. As with dense matrices, the statement [L,U] = lu(S) returns a permuted unit lower triangular matrix and an upper triangular matrix whose product is S. By itself, lu(S) returns L and U in a single matrix without the pivot information.

The sparse LU factorization does not pivot for sparsity, but it does pivot for numerical stability. In fact, both the sparse factorization (line 1) and the full factorization (line 2) below produce the same L and U, even though the time and storage requirements might differ greatly:

```
[L,U] = lu(S)                 % sparse factorization

[L,U] = sparse(lu(full(S)))   % full factorization
```

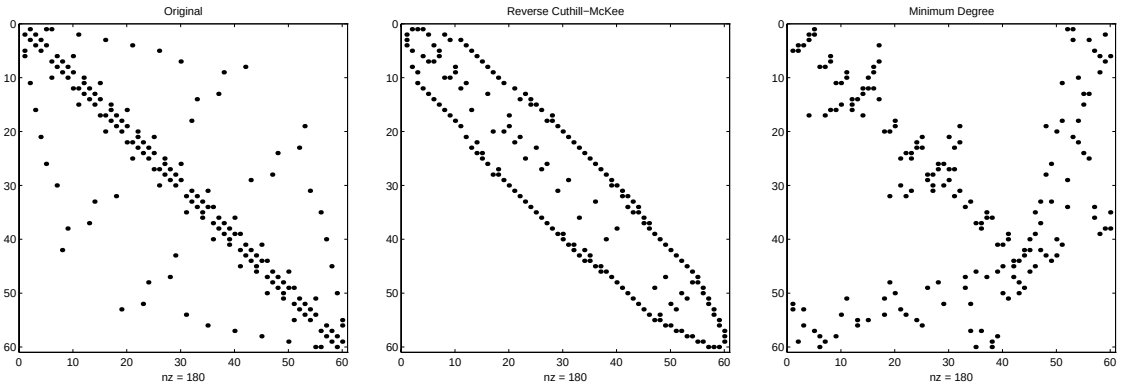
MATLAB automatically allocates the memory necessary to hold the sparse L and U factors during the factorization. MATLAB does not use any symbolic LU prefactorization to determine the memory requirements and set up the data structures in advance.

Reordering and factorization. If you obtain a good column permutation p that reduces fill-in, perhaps from symrcm or colmmd, then computing lu(S(:,p)) will take less time and storage than computing lu(S). Two permutations are the symmetric reverse Cuthill-McKee ordering and the symmetric minimum degree ordering.

```
r = symrcm(B);
m = symmmd(B);
```

The three spy plots produced by the lines below show the three adjacency matrices of the Bucky Ball graph with these three different numberings. The local, pentagon-based structure of the original numbering is not evident in the other three.

```
spy(B)
spy(B(r,r))
spy(B(m,m))
```



The reverse Cuthill-McGee ordering, `r`, reduces the bandwidth and concentrates all the nonzero elements near the diagonal. The minimum degree ordering, `m`, produces a fractal-like structure with large blocks of zeros.

To see the fill-in generated in the LU factorization of the Bucky ball, use `speye(n,n)`, the sparse identity matrix, to insert -3s on the diagonal of `B`.

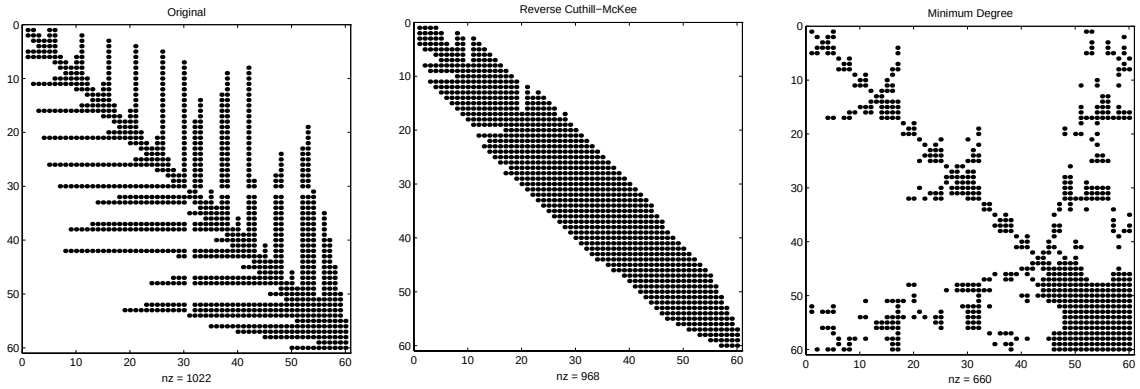
```
B = B - 3*speye(n,n)
```

Since each row sum is now zero, this new `B` is actually singular, but it is still instructive to compute its LU factorization. When called with only one output argument, `lu` returns the two triangular factors, `L` and `U`, in a single sparse matrix. The number of nonzeros in that matrix is a measure of the time and storage required to solve linear systems involving `B`. Here are the nonzero counts for the three permutations being considered.

Original	`lu(B)`	1022
Reverse Cuthill-McKee	`lu(B(r,r))`	968
Minimum degree	`lu(B(m,m))`	660

Even though this is a small example, the results are typical. The original numbering scheme leads to the most fill-in. The fill-in for the reverse Cuthill-McKee ordering is concentrated within the band, but it is almost as extensive as the first two orderings. For the minimum degree ordering, the relatively large blocks of zeros are preserved during the elimination and the

amount of fill-in is significantly less than that generated by the other orderings. The spy plots below reflect the characteristics of each reordering.



Cholesky Factorization

If `S` is a symmetric (or Hermitian), positive definite, sparse matrix, the statement below returns a sparse, upper triangular matrix `R` so that `R'*R = S`.

```
R = chol(S)
```

`chol` does not automatically pivot for sparsity, but you can compute minimum degree and profile limiting permutations for use with `chol(S(p,p))`.

Since the Cholesky algorithm does not use pivoting for sparsity and does not require pivoting for numerical stability, it is possible to do a quick calculation of the amount of memory required and allocate all the memory at the start of the factorization.

QR Factorization

MATLAB will compute the complete QR factorization of a sparse matrix `S` with

```
[Q,R] = qr(S)
```

but this is usually impractical. The orthogonal matrix `Q` often fails to have a high proportion of zero elements. A more practical alternative, sometimes known as “the Q-less QR factorization,” is available.

With one sparse input argument and one output argument

```
R = qr(S)
```

returns just the upper triangular portion of the QR factorization. The matrix `R` provides a Cholesky factorization for the matrix associated with the normal equations,

```
R'*R = S'*S
```

However, the loss of numerical information inherent in the computation of `S'*S` is avoided.

With two input arguments having the same number of rows, and two output arguments, the statement

```
[C,R] = qr(S,B)
```

applies the orthogonal transformations to `B`, producing `C = Q'*B` without computing `Q`.

The Q-less QR factorization allows the solution of sparse least squares problems

$$\text{minimize} \|Ax - b\|$$

with two steps

```
[c,R] = qr(A,b)
x = R\c
```

If `A` is sparse, but not square, MATLAB uses these steps for the linear equation solving backslash operator

```
x = A\b
```

Or, you can do the factorization yourself and examine `R` for rank deficiency.

It is also possible to solve a sequence of least squares linear systems with different right-hand sides, `b`, that are not necessarily known when `R = qr(A)` is computed. The approach solves the "semi-normal equations"

```
R'*R*x = A'*b
```

with

```
x = R\(R'\(A'*b))
```

and then employs one step of iterative refinement to reduce roundoff error

```
r = b - A*x
e = R\(R'\(A'*r))
x = x + e
```

Incomplete Factorizations

The `luinc` and `cholinc` functions provide approximate, *incomplete* factorizations, which are useful as preconditioners for sparse iterative methods.

The `luinc` function produces two different kinds of incomplete LU factorizations, one involving a drop tolerance and one involving fill-in level. If `A` is a sparse matrix, and `tol` is a small tolerance, then

```
[L,U] = luinc(A,tol)
```

computes an approximate LU factorization where all elements less than `tol` times the norm of the relevant column are set to zero. Alternatively,

```
[L,U] = luinc(A,'0')
```

computes an approximate LU factorization where the sparsity pattern of `L+U` is a permutation of the sparsity pattern of `A`.

For example,

```
load west0479
A = west0479;
nnz(A)
nnz(lu(A))
nnz(luinc(A,1e-6))
nnz(luinc(A,'0'))
```

shows that `A` has 1887 nonzeros, its complete LU factorization has 16777 nonzeros, its incomplete LU factorization with a drop tolerance of 1e–6 has 10311 nonzeros, and its `lu('0')` factorization has 1886 nonzeros.

The `luinc` function has a few other options. See the online help for details.

The `cholinc` function provides drop tolerance and level 0 fill-in Cholesky factorizations of symmetric, positive definite sparse matrices. See the online help for more information.

Simultaneous Linear Equations

Systems of simultaneous linear equations can be solved by two different classes of methods:

- Direct methods. These are usually variants of Gaussian elimination and are often expressed as matrix factorizations such as LU or Cholesky factorization. The algorithms involve access to the individual matrix elements.
- Iterative methods. Only an approximate solution is produced after a finite number of steps. The coefficient matrix is involved only indirectly, through a matrix-vector product or as the result of an abstract linear operator.

Direct Methods

Direct methods are usually faster and more generally applicable, if there is enough storage available to carry them out. Iterative methods are usually applicable to restricted cases of equations and depend upon properties like diagonal dominance or the existence of an underlying differential operator. Direct methods are implemented in the core of MATLAB and are made as efficient as possible for general classes of matrices. Iterative methods are usually implemented in MATLAB M-files and may make use of the direct solution of subproblems or preconditioners.

The usual way to access direct methods in MATLAB is not through the `lu` or `chol` functions, but rather with the matrix division operators `/` and `\`. If `A` is square, the result of `X = A\B` is the solution to the linear system `A*X = B`. If `A` is not square, then a least squares solution is computed.

If `A` is a square, full, or sparse matrix, then `A\B` has the same storage class as `B`. Its computation involves a choice among several algorithms:

- If `A` is triangular, perform a triangular solve for each column of `B`.
- If `A` is a permutation of a triangular matrix, permute it and perform a sparse triangular solve for each column of `B`.
- If A is symmetric or Hermitian and has positive real diagonal elements, find a symmetric minimum degree order `p` and attempt to compute the Cholesky factorization of `A(p,p)`. If successful, finish with two sparse triangular solves for each column of `B`.
- Otherwise (if `A` is not triangular, or is not Hermitian with positive diagonal, or if Cholesky factorization fails), find a column minimum degree order `p`.

Compute the LU factorization with partial pivoting of `A(:,p)`, and perform two triangular solves for each column of `B`.

For a square matrix, MATLAB tries these possibilities in order of increasing cost. The tests for triangularity and symmetry are relatively fast and, if successful, allow for faster computation and more efficient memory usage than the general purpose method.

For example, consider the sequence below.

```
[L,U] = lu(A);
y = L\b;
x = U\y;
```

In this case, MATLAB uses triangular solves for both matrix divisions, since `L` is a permutation of a triangular matrix and `U` is triangular.

Use the function `spparms` to turn off the minimum degree preordering if a better preorder is known for a particular matrix.

Iterative Methods

Six functions are available that implement iterative methods for sparse systems of simultaneous linear systems.

Function	Description
`bicg`	Biconjugate gradient.
`bicgstab`	Biconjugate gradient stabilized.
`cgs`	Conjugate gradient squared.
`gmres`	Generalized minimum residual.
`pcg`	Preconditioned conjugate gradient.
`qmr`	Quasiminimal residual.

All six methods are designed to solve $Ax = b$. The preconditioned conjugate gradient method, `pcg`, is restricted to symmetric, positive definite matrix A. The other five can handle nonsymmetric, square matrices.

All six methods can make use of left and right preconditioners. The linear system

$$Ax = b$$

is replaced by the equivalent system

$$M_1^{-1}AM_2^{-1}M_2x = M_1^{-1}b$$

The preconditioners M_1 and M_2 are chosen to accelerate convergence of the iterative method. In many cases, the preconditioners occur naturally in the mathematical model. A partial differential equation with variable coefficients may be approximated by one with constant coefficients, for example. Incomplete matrix factorizations may be used in the absence of natural preconditioners.

The five-point finite difference approximation to Laplace's equation on a square, two-dimensional domain provides an example. The following statements use the preconditioned conjugate gradient method with an incomplete Cholesky factorization as a preconditioner.

```
A = delsq(numgrid('S',50));
b = ones(size(A,1),1);
tol = 1.e-3;
maxit = 10;
R = cholinc(A,tol);
[x,flag,err,iter,res] = pcg(A,b,tol,maxit,R',R);
```

Only four iterations are required to achieve the prescribed accuracy.

Background information on these iterative methods and incomplete factorizations is available in:

Saad, Yousef. *Iterative Methods for Sparse Linear Equations*. PWS Publishing Company: 1996.

Barrett, Richard et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics: 1994.

Eigenvalues and Singular Values

Two functions are available which compute a few specified eigenvalues or singular values.

Function	Description
eigs	Few eigenvalues.
svds	Few singular values.

These functions are most frequently used with sparse matrices, but they can be used with full matrices or even with linear operators defined by M-files.

The statement

```
[V,lambda] = eigs(A,k,sigma)
```

finds the k eigenvalues and corresponding eigenvectors of the matrix A which are nearest the "shift" sigma. If sigma is omitted, the eigenvalues largest in magnitude are found. If sigma is zero, the eigenvalues smallest in magnitude are found. A second matrix, B, may be included for the generalized eigenvalue problem

$$Av = \lambda Bv$$

The statement

```
[U,S,V] = svds(A,k)
```

finds the k largest singular values of A and

```
[U,S,V] = svds(A,k,0)
```

finds the k smallest singular values.

For example, the statements

```
L = numgrid('L',65);
A = delsq(L);
```

set up the five-point Laplacian difference operator on a 65-by-65 grid in an L-shaped, two-dimensional domain. The statements

```
size(A)
nnz(A)
```

show that `A` is a matrix of order 2945 with 14,473 nonzero elements.

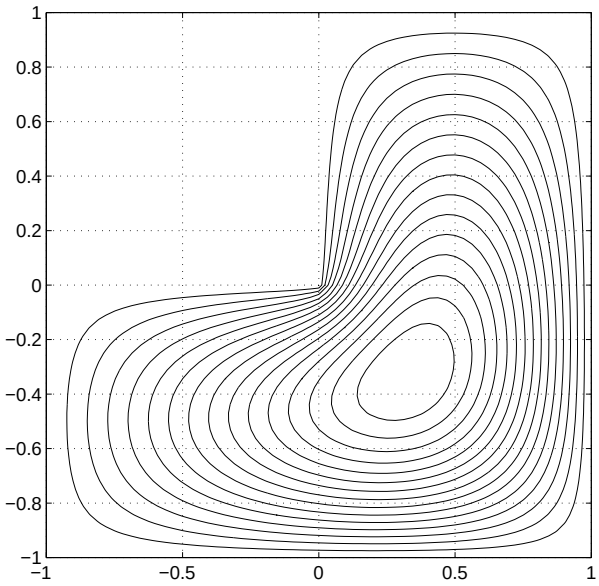
The statement

```
[v,d] = eigs(A,1,0);
```

computes the smallest eigenvalue and eigenvector. Finally,

```
L(L>0) = full(v(L(L>0)));
x = -1:1/32:1;
contour(x,x,L,15)
axis square
```

distributes the components of the eigenvector over the appropriate grid points and produces a contour plot of the result.



The numerical techniques used in `eigs` and `svds` are described in a paper by D. C. Sorensen, *Implicitly Restarted Arnoldi/Lanczos Methods for Large Scale*

Eigenvalue Calculations. A copy of the paper is available through the MATLAB Help Desk.