MATLAB Tutorial

Chapter 4. Advanced matrix operations
4.1. Sparse matrices

SPARSE MATRICES
To show the efficiency gained by using sparse matrices, we will solve a PDE using finite differences twice. First, we will use the matrix commands that use the full matrix that we have learned so far. Second, we will use new commands that take advantage of the fact that most of the elements are zero to greatly reduce both the memory requirements and the number of floating point operations required to solve the PDE.

clear all; remove all existing variables from memory

num_pts = 100; number of grid points in simulation

CALCULATION WITH FULL MATRIX FORMAT

The following matrix is obtained from using central finite differences to discretize the Laplacian operator in 1-D.
x = 1:num_pts; grid of x-values

Set the matrix from discretizing the PDE with a 1-D grid containing num_pts points with a spacing of 1 between points.

```
Afull=zeros(100,100);
Afull(1,1) = 1;
Afull(num_pts,num_pts) = 1;
for i=2:(num_pts-1) sum over interior points
Afull(i,i) = 2;
Afull(i,i-1) = -1;
Afull(i,i+1) = -1;
end
```

Dirichlet boundary conditions at x=1 and x=num_pts are set.
BC1 = -10; value of f at x(1);
BC2 = 10; value of f at x(num_pts);

For the interior points, we have a source term.
b_RHS = linspace(0,0,num_pts)'; create column vector of zeros
b_RHS(1) = BC1;
b_RHS(num_pts) = BC2;
b_RHS(2:(num_pts-1)) = 0.05; for interior, b_RHS is source term

We now use the standard MATLAB solver to obtain the solution of the PDE at the grid points.

```
f = Afull\b_RHS;
figure; plot(x,f);
title('PDE solution from FD-CDS method (full matrix)');
xlabel('x'); ylabel('f(x)');
```

Let us now take a closer look at Afull. The command spy(A) makes a plot of the matrix A by writing a point wherever an element of A has a non-zero value.

```
figure;
spy(Afull); title('Structure of Afull');
```

The number nz at the bottom is the number of non-zero elements. We see that only a small fraction of the matrix elements are non-zero. Since we numbered the grid points in a regular manner with the neighbors of each grid point stored in adjacent locations, the non-zero elements in this matrix are on the principal diagonal and the two diagonals immediately above and below. Even if we numbered the grid points irregularly, we would still have this small number of non-zero points. It is often the case, as it is here, that the matrices we encounter in the numerical simulation of PDE's are sparse; that is, only a small fraction of their points are non-zero. For this matrix, the total number of elements is
num_elements = num_pts*num_pts;
nzA = nnz(Afull); returns # of non-zero elements in Afull
fraction_filled = nzA/num_elements

This means that Afull is mostly empty space and we are wasting a lot of memory to store values we know are zero.

Remove all variables from memory except Afull.
clear x f b_RHS BC1 BC2 i num_elements nzA fraction_filled;

SPARSE MATRIX

We can convert a matrix to sparse format using the command "sparse".
Asparse = sparse(Afull)

MATLAB stores a sparse matrix as an NZ by 3 array where NZ is the number of non-zero elements. The first column is the row number and the second the column number of the non-zero element. The third column is the actual value of the non-zero element. The total memory usage is far smaller than with the full matrix format.
whos Afull Asparse;
clear Asparse; get rid of sparse matrix

NOW WE WILL SOLVE USING SPARSE MATRIX FORMAT

Next, we set the grid point values
x = 1:num_pts; grid of x-values

Now we declare the matrix A to have sparse matrix structure from the start. First, we calculate the number of non-zero elements (or an upper bound to this number). We see that for each row corresponding to an interior point, we have 3 values, whereas for the first and last row we only have one value. Therefore, the number of non-zero elements is
nzA = 3*(num_pts-2) + 2;

We now use "spalloc(m,n,nz)" that allocates memory for a m by n dimensioned sparse matrix with no more than nz non-zero elements.
A = spalloc(num_pts,num_pts,nzA);

We now set the values of the A matrix.
```
A(1,1) = 1;
A(num_pts,num_pts) = 1;
for i=2:(num_pts-1)
A(i,i) = 2;
A(i,i-1) = -1;
A(i,i+1) = -1;
end
```

Dirichlet boundary conditions at x=1 and x=num_pts are set.
BC1 = -10; value of f at x(1);
BC2 = 10; value of f at x(num_pts);

For the interior points, we have a source term.
b_RHS = linspace(0,0,num_pts)'; create column vector of zeros
b_RHS(1) = BC1;
b_RHS(num_pts) = BC2;
b_RHS(2:(num_pts-1)) = 0.05; for interior, b_RHS is source term

Now, when we call the MATLAB standard solver, it automatically identifies that A is a sparse matrix, and uses solver algorithms that take advantage of this fact.
f = A\b_RHS;

figure; plot(x,f);
title('PDE solution from FD-CDS method (sparse matrix)');
xlabel('x'); ylabel('f(x)');
whos A Afull;

From the lines for A and Afull, we can see that the sparse matrix format requires far less memory that the full matrix format. Also, if N is the number of grid points, we see that the size of the full matrix is N^2; whereas, the size in memory of the sparse matrix is only approximately 3*N. Therefore, as N increases, the sparse matrix format becomes far more efficient than the full matrix format. For complex simulations with thousands of grid points, one cannot hope to solve these problems without taking advantage of sparsity. To see the increase in execution speed that can be obtained by using sparse matrices, examine the following two algorithms for multiplying two matrices.

FULL MATRIX ALGORITHM FOR MATRIX MULTIPLICATION

Bfull = 2*Afull;
Cfull = 0*Afull; declare memory for C=A*B
num_flops = 0;
for i=1:num_pts
for j=1:num_pts
for k=1:num_pts
Cfull(i,j) = Cfull(i,j) + Afull(i,k)*Bfull(k,j);
num_flops = num_flops + 1;
end
end
end
disp(['# FLOPS with full matrix format = ', int2str(num_flops)]);

SPARSE MATRIX ALGORITHM FOR MATRIX MULTIPLICATION

B = 2*A;
nzB = nnz(B); # of non-zero elements of B
nzC_max = round(1.2*(nzA+nzB)); guess how much memory we'll need for C
C = spalloc(num_pts,num_pts,nzC_max);
[iA,jA] = find(A); find (i,j) elements that are non-zero in A
[iB,jB] = find(B); find (i,j) elements that are non-zero in B
num_flops = 0;
for ielA = 1:nzA iterate over A non-zero elements
for ielB = 1:nzB iterate over B non-zero elements
if(iB(ielB)==jA(ielA)) the pair contributes to C
i = iA(ielA);
k = jA(ielA);

```
j = jB(ielB);
C(i,j) = C(i,j) + A(i,k)*B(k,j);
num_flops = num_flops + 1;
end
end
end
disp(['# FLOPS for sparse matrix format = ', int2str(num_flops)]);
```
D = Cfull - C; check to see both algorithms give same result
```
disp(['# of elements where Cfull ~= C : ' int2str(nnz(D))]);
```

Finally, we note that taking the inverse of a sparse matrix
usually destroys much of the sparsity.
```
figure;
subplot(1,2,1); spy(A); title('Structure of A');
subplot(1,2,2); spy(inv(A)); title('Structure of inv(A)');
```

Therefore, if we have the values of A and of C = A*B and want to calculate the matrix B, we do NOT use inv(A)*C. Rather, we use the "left matrix division" operator A\C. This returns a matrix equivalent to inv(A)*C, but uses the MATLAB solver that takes advantage of the sparsity.
```
B2 = A\C;
figure; spy(B2); title('Structure of B2');
```

We see that the error from the elimination method has introduced very small non-zero values into elements off of the central three diagonals. We can remove these by retaining only the elements that are greater than a tolerance value.
```
tol = 1e-10;
Nel = nnz(B2);
[iB2,jB2] = find(B2);
```
return positions of non-zero elements
```
for iel=1:Nel
if(abs(B2(iB2(iel),jB2(iel))) < tol) set to zero
B2(iB2(iel),jB2(iel)) = 0;
end
end
```
B2 = sparse(B2); reduce memory storage
```
figure; spy(B2); title('Structure of "cleaned" B2');
```

Since we do not want to go through intermediate steps where we have to store a matrix with many non-zero elements, we usually do not calculate matrices in this manner. Rather we limit ourselves to solving linear systems of the form A*x = b, where x and b are vectors and A is a sparse matrix whose value we input directly. We therefore avoid the memory problems associated with generating many non-zero elements from round-off errors.

```
clear all
```

4.2. Common matrix operations/eigenvalues

The determinant of a aquare matrix is calculated using "det".
A = rand(4); creates a random 4x4 matrix
det(A) calculate determinant of A

Other common functions of matrices are
rank(A) rank of A
trace(A) trace of A
norm(A) matrix norm of A
cond(A) condition number of A

A_inv=inv(A) calculates inverse of A
A*A_inv

The eigenvalues of a matrix are computed with the command "eig"
eig(A)

If the eigenvectors are also required, the syntax is
[V,D] = eig(A)

Here V is a matrix containing the eigenvectors as column vectors, and D is a diagonal matrix containing the eigenvalues.

```
for i=1:4
eig_val = D(i,i);
eig_vect = V(:,i);
A*eig_vect - eig_val*eig_vect
end
```

The command "eigs(A,k)" calculates the k leading eigenvalues of A; that is, the k eigenvalues with the largest moduli.
eigs(A,1) estimate leading eigenvalue of A

Similarly, the eigenvectors of the leading eigenvalues can also be calculated with eigs.

```
[V2,D2] = eigs(A,1);
eig_vect = V2; eig_val = D2;
A*eig_vect - eig_val*eig_vect
```

With sparse matrices, only the command "eigs" can be used.

clear all

4.3. LU decomposition

The linear system Ax=b can be solved with multiple b vectors using LU decomposition. Here, we perform the decomposition P*A = L*U, where P is a permutation matrix (hence inv(P)=P'), L is a lower triangular matrix, and U is an upper triangular matrix. P is an identity matrix when no pivoting is done during the factorization (which is essentially Gaussian elimination). Once the LU factorization is complete, a problem Ax=b is solved using the following linear algebra steps.

A*x = b
P*A*x = P*b
L*U*x = P*b

This gives the following two linear problems invloving triangular matrices that may be solved by substitution.
L*y = P*b
U*x = y
The MATLAB command for performing an LU factorization is "lu" We use a random, non-singular matrix to demonstrate the algorithm. Non-singularity is ensured by adding a factor of an identity matrix.

```
A = rand(10) + 5*eye(10);
```

Perform LU factorization.

```
[L,U,P] = lu(A);
```

max(P*P'-eye(10)) demonstrates that P is orthogonal matrix
max(P*A - L*U) shows largest result of round-off error

Compare the structures of the matrices involved
```
figure;
subplot(2,2,1); spy(A); title('Structure of A');
subplot(2,2,2); spy(P); title('Structure of P');
subplot(2,2,3); spy(L); title('Structure of L');
subplot(2,2,4); spy(U); title('Structure of U');
```

LU factorization can be called in exactly the same way for sparse matrices; however, in general the factored matrices L and U are not as sparse as is A, so by using LU factorization, some efficiency is lost. This becomes more of a problem the the greater the bandwidth of the matrix, i.e. the farther away from the principal diagonal that non-zero values are found.

Sometimes we only want an approximate factorization B=L*U where B is close enough to A such that C = inv(B)*A is not too much different from an identity matrix, i.e. the ratio between the largest and smallest eigenvalues of C is less than that for A. In this case, B is called a preconditioner, and is used in methods for optimization and solving certain classes of linear systems. When we perform an incomplete LU factorization, we only calculate the elements of L and U that correspond to non-zero elements in A, or with different options, we neglect elements whose absolute values are less than a specified value.

The following code demonstrates the use of incomplete LU factorization.

make B=A, except set certain elements equal to zero.
```
B=A;
```

set some elements far-away from diagonal equal to zero.
```
for i=1:10
B(i+5:10,i) = 0;
B(1:i-5,i) = 0;
end

B=sparse(B);
[Linc,Uinc,Pinc] = luinc(B,'0');

figure;
subplot(2,2,1); spy(B); title('Structure of B');
subplot(2,2,2); spy(Pinc); title('Structure of Pinc');
subplot(2,2,3); spy(Linc); title('Structure of Linc');
subplot(2,2,4); spy(Uinc); title('Structure of Uinc');

D1 = P*A - L*U;
D2 = Pinc*B - Linc*Uinc;
tol = 1e-10; set tolerance for saying element is zero
for i=1:10
for j=1:10
if(D1(i,j)<tol)
D1(i,j) = 0;
end
if(D2(i,j)<tol)
D2(i,j) = 0;
end
end
end
```

figure;
subplot(1,2,1); spy(D1); title('(P*A - L*U)');
subplot(1,2,2); spy(D2); title('(Pinc*B - Linc*Uinc)');

But, look at the eigenvalues of the B and of the approximate factorization.
Bapprox = Pinc'*Linc*Uinc;
eigs(B) eigenvalues of B matrix
C = Bapprox\B; inv(Bapprox)*B (don't use "inv" for sparse matrices)
eigs(C)

clear all

4.4. QR decomposition

The factorization A*P = Q*R, where P is a permutation matrix, Q is a orthogonal matrix, and R is upper triangular is performed by invoking the command "qr".
A = rand(6);
[Q,R,P] = qr(A);

Q*Q' shows Q is orthogonal
A*P - Q*R

figure;
subplot(2,2,1); spy(A); title('Structure of A');
subplot(2,2,2); spy(P); title('Structure of P');
subplot(2,2,3); spy(Q); title('Structure of Q');
subplot(2,2,4); spy(R); title('Structure of R');

If the decomposition A=QR is desired (i.e. with P=1), the
command is :
[Q,R] = qr(A);

figure;
subplot(2,2,1); spy(A); title('Structure of A');
subplot(2,2,2); spy(Q); title('Structure of Q');
subplot(2,2,3); spy(R); title('Structure of R');

A - Q*R

clear all

4.5. Cholesky decomposition

If A is a Hermetian matrix (i.e. A=A') then we know that all eigenvalues are real. If in addition, all the eigenvalues are greater than zero, then x'*A*x > 0 for all vectors x and we say that A is positive-definite. In this case, it is possible to perform a Cholesky decomposition, i.e. A = R'*R, where R is upper triangular. This is equivalent to writing A = L*L', where L is lower triangular.

First, we use the following positive-definite matrix.
Ndim=10;
Afull=zeros(Ndim,Ndim);
for i=1:Ndim sum over interior points
Afull(i,i) = 2;
if(i>1)

```
Afull(i,i-1) = -1;
end
if(i<Ndim)
Afull(i,i+1) = -1;
end
end

Rfull = chol(Afull);
D = Afull - Rfull'*Rfull; eig(D)

figure;
subplot(1,2,1); spy(Afull); title('Structure of Afull');
subplot(1,2,2); spy(Rfull); title('Structure of Rfull');
```

For sparse matrices, we can perform an incomplete Cholesky decomposition that gives an approximate factorization with no loss of sparsity that can be used as a preconditioner. In this particular case, with a highly structured matrix, the incomplete factorization is the same as the complete one.

```
Asparse = sparse(Afull);
Rsparse = cholinc(Asparse,'0');
D2 = Asparse - Rsparse'*Rsparse; eig(D2)

figure;
subplot(1,2,1); spy(Asparse); title('Structure of Asparse');
subplot(1,2,2); spy(Rsparse); title('Structure of Rsparse');

clear all
```

4.6. Singular value decomposition

Eigenvalues and eigenvectors are only defined for square matrices. The generalization of the concept of eigenvalues to non-square matrices is often useful. A singular value decomposition (SVD) of the (m x n) matrix A is defined as A = U*D*V', where D is a (m x n) diagonal matrix containing the singular values, U is a (m x m) matrix containing the right eigenvectors and V' is the adjoint (transpose and conjugate) of the (n x n) matrix of left eigenvectors.

In MATLAB, a singular value decomposition is peformed using "svd"

```
A = [1 2 3 4; 11 12 13 14; 21 22 23 24];
[U,D,V] = svd(A);
D, U, V
```

U*D*V' show that decomposition works

```
clear all
```