

## CMSC661, 3/17/2005 Thursday: Scribed notes for lecture, via Sam Lamphier

Purpose: the intent of this document is to capture material which was conveyed in class verbally and on the board that is not in the lecture notes covered on 3/17,

- pages 5-end of “Solution of Sparse Linear Systems, Part 1: Direct Methods” and
- pages 1-5 of “Solution of Sparse Linear Systems, Part 2: Iterative Methods”.

These notes are meant to be read side-by-side with the lecture notes, as we will refer to sections in the lecture notes without copying them here. **NOTE: this is only an approximate [tape was garbled at a few spots, questions edited out, items paraphrased, etc.] transcription** of the taped lecture of Dr. O'Leary.

[START with intro] We're talking about sparse matrices. We're solving sparse systems of linear equations,  $Ax=b$ . We also talking about the eigenvalue problem,  $A\phi = \lambda\phi$ . We have to be ensure that if the matrix had a small number of nz before that we have a small number of nz **[[[unclear: after we factor.]]]**

When  $A$  is symmetric positive definite we might do a Cholesky factorization, where  $LL'x = b$ , then we do forward and backward substitution. We solve  $Ly = b$  (then do forward substitution), then we solve  $L'x = y$  (then do backwards substitution). It's very important that if  $A$  has a small number of nz that we retain a small number of nz in  $L$ . Otherwise we have a full matrix to store and the operation is  $O(n^2)$ , whereas if the  $L$  matrix is sparse, the cost is only proportional to the number of nz. **[[[something unclear]]]**

One of the things we can do is use a permutation matrix  $P$  so that when we apply PAP' when factored retains a small number of nz. **[[[unclear... We have a permutation of A, and we also have a permuted x vector.]]]** The aim is to determine the matrix  $P$ . In general, reordering is complexity NP. So we'll use a heuristic which often works, especially on practical problems. We'll talk about 3 heuristics here; there are much more in the literature.

[Part 1 notes, start p5]

[Strategy 1: Cuthill-McKee] The first one is Cuthill-McKee. Choose your “favorite node”, i.e., one with least degree, thus in a corner. Choose a corner, say the lower left, call it node 1. Label all nodes a distance one from node 1. Next, order them in increasing degree. There are two nodes, both have degree 3, so pick arbitrarily. Then label and order (by smallest degree) all that are distance 2 from the first node. Continue with neighbors (i.e., distance 3 from original starting node); repeat until done. The first 6 orders are shown:

4 \* \*  
2 6 \*  
1 3 5

[Reverse Cuthill-McKee] It doesn't make much difference for symmetric graphs but is sometimes better for others. Reverse says, get your order 1..25 (for example), then reverse the order of the indices you got – the heuristic is a little better than Cuthill-McKee.

If we have a symmetric matrix A (in order 1, 2, 3):

```

  1 2 3
1 2 3 5
2 3 4 7
3 5 7 6

```

and we're given a reordering of 3, 1, 2, then the result PAP' is also symmetric:

```

  3 1 2
3 6 5 7
1 5 2 3
2 7 3 4

```

[Discussion of graphs on p6] Here's an example (spy(S)) of Cuthill-McKee on a matrix with the 5-point operator on a 5x5 grid. Cholesky factorization gives 129 nz. Cuthill-McKee doesn't form a band matrix, but it's profile is good. Finally, chol() after Cuthill-McKee gives nz=115, better than nz=129 without Cuthill-McKee.

[Strategy 2: Minimum degree, p7]

Start out ordering by choosing node that has smallest degree and label node 1 (so again we choose the lower left). However, unlike Cuthill-McKee, the next step is not to choose a neighbor, but to choose the node with the smallest degree (breaking ties arbitrarily); noting that when we mark a node, we remove it's edges. After removing the first node, then we have the original 3 corners and the two neighbors of node 1 that all have degree 2, so pick one and label it node 2, breaking it's edges. Continue until all nodes processed. Shown below are the first 6 orders:

```

2 * * 3
* * * *
5 * * *
1 6 * 4

```

Cuthill-McKee marches across to adjacent nodes, but minimum degree can jump all over the place. You process all nodes with degree 2, then proceed to process all nodes with degree 3, etc.

Minimum degree has the disadvantage that you have to cut the edges for each labeled node, whereas that doesn't occur with Cuthill-McKee.

The ordering determines the construction of the matrix. If you get the ordering first, then you construct the matrix using the ordering you determined. If you get the matrix first, then you permute the matrix with the ordering.

So a sample minimum degree ordering might give:

```

      1 2 3 4 5 6
1  x x x
2  x x   x   x
3  x   x   x x
4
...

```

[referring to  $S(r,r)$  after minimum degree ordering] As you can see, it's not banded, and doesn't even look like it has a good profile on it. But the number of nz is better than the 129. You have to look at the fill-in to determine if it's a good or bad result. The profile shows worst case fill-in, which we don't get here.

[Strategy 3: Nested Dissection, p7] We'll come back to this in iterative methods as well. The idea is to dissect the graph in a nested way. Divide the nodes of the graph in two so that each graph has approximately the same number of nodes and the separator is small. So a graph that's long and skinny is easy to find a separator for – choose a group of nodes in the middle which if we removed, would leave the two sides being completely disjoint. Then apply recursively until the size of the pieces is small enough. Then you order all within all the pieces, then you order the separators.

A long skinny graph:

```

* * * * *
* * * * *

```

might get split like

```

* * * * *
* * * * *

```

Splitting again:

```

* * * * *
* * * * *

```

then numbering the separator nodes last

```

1  9  3 13  5 11  7
2 10  4 14  6 12  8

```

for a final ordering.

For a square graph, applying nested dissection on:

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

could have a split like so

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

then each side is split again

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

giving a plus sign.

The final reordering would be

```
6 7 25 14 15
5 8 24 13 16
17 18 23 19 20
2 3 22 10 11
1 4 21 9 12
```

[S(r,r) after nested dissection ordering, p8] It does something kind of interesting. The four pieces A, B, C, and D are the four pieces up in the left corner of this matrix. The first 20 rows and columns are interesting because we just have a block diagonal matrix – there’s no coupling between those blocks because we pulled out all the coupling nodes and shoved them down to the bottom of the matrix. The bottom of the matrix gets kind of messy because of all the separators and the connections the separators have to the blocks. When you do the decomposition, you get  $nz=115$ . Nested dissection is not the best way to order nodes for sparsity. But what it does for us is that later on, it gives us a real good way to get something called “preconditioning”. So we’ll hold onto this idea. Later, we’ll pull out these main diagonal blocks and use that matrix which has a nice factorization by itself to precondition the rest of the system, throwing away all the trash in the corner (lower and rhs).

[run the demo program spar50.m (see comments in program)] Note the significant improvement for the 50x50 grid for some methods. When we get up to more realistic sizes like 500x500 we can see the effects even more.

So if you have a sparse matrix and you want to solve a linear system, don't just feed it to backslash in Matlab. First, try to reorder, using one of their built-in routines. All of the routines we've discussed are built-in (except nested dissection which was generated manually), so try them out, especially if the problem is something you have to solve over and over again.

[Summary, p8] Key for direct methods is to be careful how you order your rows and columns. You want to reorder so you reduce the fill-in during factorization to save storage, and perhaps more importantly, to save time.

We gave examples of examples of 3 strategies: Cuthill-McKee, minimum degree, and nested dissection. They work on symmetric matrices.

For nonsymmetric and symmetric indefinite, we can do the same thing, but as we do the reordering, we have watch out for numerical stability. The nice thing about SPD is that now matter how you scramble the matrix, the Cholesky factorization is guaranteed to exist and be stable. In general matrices, it's not so easy.

If you had a 0 in the pivoting position after reordering, the matrix algorithm would fail.

If instead you had epsilon, the factorization would exist, but could wash out data, causing the algorithm to go unstable when  $1/\epsilon$  was added to second row.

Have to worry about not just sparsity, but also accuracy. In order to preserve accuracy, when you do the interchange that you don't try to pivot with an element that is too small.

[Recent strategies], about 14 years old, called spectral partitioning. Determine the reordering by looking at the eigenvectors of the graph. They look at the Laplacian of the graph, a matrix related to the graph. You find the elements of the eigenvector corresponding to the second eigenvalue and the ones that are positive you put in one group, the ones that are negative you put in another group, splitting the matrix into two pieces.

[Sparse direct methods for eigenproblems, p9] All these reordering strategies helped us solve linear systems, they can also help us solve eigenvalue problems. The QR algorithm is the standard way to find eigenvalues and eigenvectors; the workhorse for every matrix that we can afford to factor. Still do the reordering so that when you do the factorization involved in the QR you don't get a lot of zeros filling in. The strategy is to focus on bandwidth, not so much profile; reorder the matrix  $A$  so that it has small bandwidth. When we apply  $PAP^T$ , we get the eigenvalues of  $A$  because it's a similarity transformation (we apply a matrix  $P$  in front and  $P^{-1}$  behind); therefore the eigenvalues do not change. The eigenvectors are just a scrambled form of the eigenvectors of  $A$ .

[Part 2: Iterative Methods, p1] Direct factoring methods work pretty well on 2D finite element problems. By the time we get up to 3D, it's very difficult to order a matrix so that it has a smaller band or smaller profile. 3D usually use iterative, and almost certainly for 4D and higher.

Krylov subspace methods are the workhorse methods.

Direct methods: to roundoff, we get the correct answer.

Indirect methods: completely different strategy; rather than getting the exact solution, we'll get an increasingly good approximation to the solution. (Note: we're solving general linear systems; not restricted to symmetric or positive definite).

[Stationary Iterative Methods, p1]

Gauss used to solve a least squares problem in geodesy, to solve for coordinates. Very popular before computers; unfortunately still somewhat popular, but slow.

[p2] M is the "easy" part of the matrix, i.e., the main diagonal; N is the "leftover".

[The Jacobi iteration, p2]  $b_i - \dots$  should give  $r_i = 0$  if we have our true solution. Modify red part,  $x_i$  in order to make  $r_i = 0$ . That's the Jacobi strategy. Look at the  $i$ th equation and change  $x_i$  to satisfy the  $i$ th equation. That might do lousy things to the other equations, but it makes the  $i$ th equation have a zero residual at this moment in time. So  $x_i$  is  $b_i - \text{sum of blue terms} - \text{sum of purple terms}$  and then divide by the main diag  $a_{ii}$ . To work, we have to make the main diagonals nonzero. For a given  $k$ , run from 1 to  $N$  for  $i$  to get the next estimate  $x_i^{(k+1)}$ . If that isn't good enough, do again until we converge (not guaranteed to converge, but if it does, it is correct). First guess is customarily 0.

Note: if  $A$  is sparse, we should only touch the terms that are nz. So for instance, if we're working with the five point operator, we would only have 4 terms in a row.

In Matlab, using  $A(i,:)$  will pull out just the nz.

[If we partition... p2]  $L$  is just the elements below the diagonal,  $D$  is the diagonal,  $U$  is the elements above the diagonal (not a Cholesky factorization, just notation).

$a_{ii} x_i^{(k+1)}$  stacked is  $D x^{(k+1)}$

$b_i$  stacked is  $b$

$-\text{sum}(j=1:i-1, a_{ij} * x_j^k)$  stacked is  $-L x^k$  (red elements in the matrix)

$-\text{sum}(j=i+1:n, a_{ij} * x_j^k)$  stacked is  $-U x^k$  (black elements in the matrix)

[Gauss-Seidel, p2] We really want to just keep one matrix around, especially if the matrix is large (say size 1 million). So use the new guess that we've computed, which means change the equation from  $Lx^k$  to  $Lx^{(k+1)}$ .

Might work a little better. This is still easy system to solve (we're essentially doing a forward substitution as we go down the matrix).

[SOR, p3] Use GS to get  $x^{(k+1)}$ , but multiply by  $\omega$  = maybe 1.6, and average with old  $x^k$ . Use the result as the new  $x^{(k+1)}$ . Converges faster.  $\omega > 1$  is over-relaxation;  $\omega < 1$  is under-relaxation.

SOR works better on many (not all) problems; developed by trial and error.

[Convergence, p5]  $M^{-1}b = c$

Substituting the true solution for  $x^k$  gives  $x^{(k+1)}$  which is also the true solution; for all the stationary iterative methods. Stationary means  $G$  doesn't change.

For the  $\text{error}^{(k+1)}$  to be smaller than  $\text{error}^k$ , then we need  $G$  to be small.

Let  $G$  have eigenvectors  $v_1 \dots v_n$  and eigenvalues  $L_1 \dots L_n$ , assume eigval are linearly independent.

Express error

$e^{(0)}$  as  $\sum_{i=1:n} \alpha_i v_i$

Now,  $G$  times that vector is:

$\sum(\dots, \alpha_i G v_i)$

which equals

$\sum(\dots, \alpha_i L_i v_i)$

Continuing to multiply by  $G$   $k$  times, then we get

$e^k = G^k e^{(0)} = \sum(\dots, \alpha_i L_i^k v_i)$

So if we want the error to get smaller and smaller, we have to have the eigenvalues small, i.e., less than 1 (close to 0).

In general, the eigenvalues are complex. So if we restrict the eigenvalues to lying within the unit circle, then the methods will converge. If diagonally dominant, the methods will converge. The methods have linear convergence.

[Krylov subspace methods, p5]

$$x^{(0)} = 0$$

$$x^{(1)} = c$$

$$x^{(2)} = Gc + c$$

$$x^{(3)} = G^2c + 2Gc + c$$

... work was in accessing the matrix A (in essence multiplying by matrix G) in SIM.

If we allow arbitrary coefficients ( $a_i$ ):

$$\dots a_3 * G^2c + a_2 * Gc + a_1 * c$$

for the terms, we can do much, much, much better -- called Krylov methods. That's why SIM aren't used any more, Krylov is so much more efficient.