SPARSE LU DECOMPOSITION AND GA-BASED MATRIX REORDERING

A.N. YZELMAN

Suppose A is an $n \times n$ sparse matrix. An *upper triangular matrix* U has for all $i > j$ ($i, j \in [0, n-1]$) that $u_{ij} = 0$. Similarly, a *lower triangular matrix* L has for all $i < j$ that $l_{ij} = 0$ (note that diagonal entries may be nonzero). LU-decomposition attempts to construct such matrices L, U so that their product equals A:

$$A = LU.$$

Solving linear systems using LU decompositions (or Cholesky-factorisations in the case that A is symmetric positive definite[1]) fall under the *direct* methods for solving linear systems; a (mathematically) precise solution to $Ax = b$ is calculated.

Assignment A. (1 pt.)
Sketch a method to solve x from $L\tilde{x} = b$ in an efficient way, with L a lower triangular matrix as defined above. Using this, motivate why decomposing A into L and U hence is useful for solving $Ax = b$.

Iterative methods solve $Ax = b$ approximately, with (usually) increasing accuracy as more iterations are performed. Generally, an initial solution x_0 is guessed (or taken equal to $\vec{0}$), and the residual $r_0 = b - Ax_0$ of this initial approximation is used to find a better approximation x_1. This is applied iteratively (r_1 is used to find x_2, and so on), until the norm of the last residual r_k is deemed small enough (and the final approximation x_k hence approximates the true solution x well enough).

Example. A (relatively) easy to understand iterative method is, for example, Generalised Conjugate Residuals (GCR). In this method, update vectors u_i are sought such that $r_{i+1} = b - Ax_{i+1} = b - A(x_i + u_i)$ is reduced (according to some norm). GCR chooses the u_i to be of the following form: $u_i = \alpha_i r_i$; hence $r_{i+1} = r_i - \alpha_i A r_i = r_i - \alpha_i c_i$, with $c_i = Ar_i$ the so-called correction vector. This means that actually, $r_{i+1} = r_0 - \alpha_0 c_0 - \cdots - \alpha_i c_i$.

Now comes in the main idea. We may choose the α_i such that $r_0 - \alpha_i c_i$ is orthogonal to r_0; in other words, we subtract (from r_0) the part of the c_i upon which r_0 was projected. This causes a strict reduction of the length (or rather, norm) of each subsequent r_i, and the residuals will approach norm zero. This works, as long as the c_i are not completely orthogonal to r_0; in such case this method would break down.

Assignment B. (1 pt.)
Consider a case in which you would have a matrix A, and right-hand side vectors $b_0, \ldots, b_{k-1}$, $k \in \mathbb{N}$. For small dimensions of A and small k, would you rather use a direct or iterative method to solve the systems $Ax_i = b_i$, and why? Answer the same for (very) large A with small k, and large A and large k.

1. The algorithm

The LU algorithm is as follows. First, the top row $u^{(0)}$ from $A = A^{(0)}$ is copied to the top (empty) row of U. Then the leftmost column $l^{(0)}$ from $A^{(0)}$ is copied to the leftmost (empty) column of L, and its entries are divided by a_{00}. This causes the diagonal entries of L to all equal 1. Note that $(LU)_{0j} = u_j^{(0)} l_0^{(0)} = u_j^{(0)} 1 = a_{0j}$ due to this normalisation and the fact that L, U are lower or upper triangular, respectively; the first row of (LU) is already correct. Also because of this normalisation, $(LU)_{i0} = u_0^{(0)} l_i^{(0)} = a_{00} \frac{a_{i0}}{a_{00}} = a_{i0}$ and the first column of (LU) is correct. Define $A^{(1)}$ to be the submatrix of $A^{(0)}$ with both the first row and first column deleted, thus a matrix of dimensions $(n-1) \times (n-1)$. The idea is to apply the algorithm recursively on

Date: September 2009.

[1] A matrix A is positive definite when for all nonzero $z \in \mathbb{R}^n$, $zAz^T > 0$.

this smaller matrix. However, if we would do this, then when calculating $(LU)_{ij}$ the current recursion step would add the factor $u_j^{(0)} l_i^{(0)}$ (that is, $(LU)_{ij} = l_i u_j + \dots$); the second row and column would no longer be correct. To offset this, we subtract this value $u_j^{(0)} l_i^{(0)}$ from each entry $a_{ij}^{(0)}$ found in $A^{(1)}$. Now the algorithm recurses correctly, and terminates at the 1×1 matrix $A^{(n-1)}$.

Example 1.1. *Let us decompose*

$$A = A^{(0)} = \begin{pmatrix} 1 & 0 & 3 \\ 1 & 2 & 0 \\ 7 & 0 & 5 \end{pmatrix}.$$

The first step $k = 0$ constructs the following L, U:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 1 & & 0 \\ 7 & & \end{pmatrix} \qquad U = \begin{pmatrix} 1 & 0 & 3 \\ 0 & & \\ 0 & 0 & \end{pmatrix}.$$

The submatrix used for the next iteration will be

$$A^{(1)} = \begin{pmatrix} 2 & \mathbf{-3} \\ 0 & 5\mathbf{-21} \end{pmatrix}.$$

At $k = 1$, the L, U decomposition matrices are updated:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 7 & 0 & \end{pmatrix} \qquad U = \begin{pmatrix} 1 & 0 & 3 \\ 0 & 2 & \mathbf{-3} \\ 0 & 0 & \end{pmatrix},$$

and

$$A^{(2)} = -16$$

so that $l^{(2)} = u^{(2)} = -16$ and our decomposition is final:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 7 & 0 & 1 \end{pmatrix} \qquad U = \begin{pmatrix} 1 & 0 & 3 \\ 0 & 2 & \mathbf{-3} \\ 0 & 0 & -16 \end{pmatrix};$$

Assignment C. (1 pt.)

Let L, U and L', U' both be LU-decompositions of A (that is, $A = LU = L'U'$). Suppose the diagonal of both L and L' consists of entries equal to 1. Prove that the LU decomposition is unique under this assumption, that is: $L = L'$ and $U = U'$.

Assignment D. (1 pt.)

It is important to be able to rely on an algorithm to find proper solutions; that is, not yield arbitrary answers when no answer is possible, or, on the other hand, not yield any answer while an answer exists. The following questions focus on the first item:

(1) *Is it possible that a singular*[2] *matrix A has an LU decomposition? Give a proof or a small example in either case.*
(2) *Can the* algorithm *execute successfully if the input matrix is singular? Explain why.*
(3) *Find a non-singular (invertible) matrix B that does* not *have an LU decomposition. Would the algorithm execute successfully for this matrix B you found?*

In the sparse case[3], LU decomposition may result in (relatively) dense L, U factors. This can be caused when the algorithm subtracts $u_j^{(k)} l_i^{(k)}$ where appropriate; for example, such subtraction may occur for some

[2] A matrix A is singular if and only if A does *not* have an inverse matrix A^{-1} (such that $AA^{-1} = I$).
[3] that is, cases where a typically large matrix contains a relatively large amount of zeroes.

i, j while a_{ij} previously was zero. Such effect is called *fill-in*. Fill-in occurs when at some point, both $l_i^{(k)}$ and $u_j^{(k)}$ are nonzero while $a_{ij}^{(k)}$ is zero; in boolean logic, defining 0 to be false and anything nonzero true, fill-in occurs at the kth recursive step at position i, j when:

$$l_i^{(k)} \wedge u_j^{(k)} \wedge !a_{ij}^{(k)}$$

is true. The boldface nonzero in Example 1.1 is the fill in caused by the LU algorithm. Note that fill-in can recursively again cause fill-in at other locations in the other matrices $A^{(>k)}$, so that, as one can imagine, exceedingly many nonzeroes will appear in L and U.

2. Row- and column permutations

Instead of applying the LU decomposition algorithm on the matrix A, we could also apply this on row permutations thereof (PA), column permutations (AQ), or both (PAQ); with P, Q permutation matrices of the form $P = (e_{p_0}|e_{p_1}|\dots|e_{p_{n-1}})$ with p_i a permutation of $(0, 1, \dots, n-1)$, and similarly for Q.

In general, row permutations are necessary to ensure a nonzero is placed in the top-left of the $A^{(k)}$, especially so in our sparse case; this was not an issue in the previous example since all entries on the diagonal of A were, conveniently, nonzero. Usually, a sparse LU algorithm will do these necessary row-permutations (also called *pivoting*); perhaps even on the fly, only looking for and swapping rows when a top-left zero is encountered.

Assignment E. (0.5 pt.)

Is a nonzero diagonal enough to ensure proper execution of the LU decomposition algorithm? Consider the following matrix:

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}.$$

Note that the diagonal of A indeed is nonzero. Is the matrix invertible? Apply the LU algorithm on A. Does it terminate successfully? If not, can you find a row or column permutation (or both) so that the algorithm runs successfully on PA or AQ (or PAQ)?

Assignment F. (1 pt.)

Assume the following lemma is true.

> **Lemma 2.1.** *For any non-singular matrix A, there exists a permutation matrix P such that for the row-permuted matrix PA the LU decomposition algorithm outlined above executes successfully.*

Prove that all linear systems $Ax = b$ with A non-singular can be solved using row permutations and the algorithm outlined above. Note you can use the above lemma and refer back to earlier questions (even if you could not prove them).

Does what you prove here contradict with what you proved in Assignment D.3?

Column reordering can be used to reduce fill-in. Let us change the order of the columns of A in the previous example to $(1, 2, 0)$. We then obtain:

$$A = \begin{pmatrix} 0 & 3 & 1 \\ 2 & 0 & 1 \\ 0 & 5 & 7 \end{pmatrix}.$$

Now, to ensure proper execution of the decomposition algorithm, a suitable row-permutation has to be found. Changing the row order to, for example, $(1, 0, 2)$ will fix this ($(1, 2, 0)$ is also possible):

$$A = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 3 & 1 \\ 0 & 5 & 7 \end{pmatrix}.$$

In this case, by the simple fact that $A^{(1)}$ is fully dense, no extra fill-in can occur (although the number of nonzeroes of course equals the amount of those in the original matrix). Column reordering thus completely eliminated fill-in here.

Assignment G. (0.5 pt.)
Do you think that a method for finding a column permutation to reduce fill-in, should be dependent on the method used to find a row permutation? Briefly motivate your answer.

3. Sparse LU and GA

How to find column permutations for large sparse matrices such that fill-in is reduced? Here, instead of more classical graph-based algorithms, we will consider the use of genetic algorithms for finding such a permutation.

Genetic algorithms iterate on *populations* of candidate solutions, also called *chromosomes*. These chromosomes can mutate or yield chromosomes (offspring) through cross-fertilisation. In this case, chromosomes can be represented as an array of integers, containing a permutation of $(0, 1, \ldots, n-1)$; this is very similar to chromosomes typically used when solving the Travelling Salesman problem (using GAs).

Besides how to implement mutation or cross-fertilisation, a big issue is how to define the cost function $f : X \to \mathbb{R}$ to-be maximised, where X is the space of all possible permutations of $(0, 1, \ldots, n-1)$. Assumed is that the matrix to be decomposed is readily available. Known from the first section is when fill-in occurs at some stage k; this rule can be applied recursively to calculate exactly the number of fill-ins for a given permutation.

This is costly, however, to the point one wonders if the total time spent calculating costs could not have been better spent actually solving $LUx = b$, and take possible extra fill-in for granted. An upper bound on the number of fill-in may be more appropriate, although this is not useful if this bound is not strict enough; then no difference between 'ok' solutions and good solutions may be seen through the cost function (in the worst case there is no difference between good and bad solutions). For the same reasons, using a lower bound on fill-in may be a bad idea as well; the cost function may make solutions look pretty good while, in fact, they are not.

3.1. **Sparsity structure.** Since pivoting is necessary, one could think only column reordering is possible to reduce fill-in. This is not true, however, if two-sided reordering infers certain *sparsity structures*. In general, the sparsity structure of A is that of a single sparse block. Other sparsity structures include, for example, block diagonal:

$$A = \begin{pmatrix} A_0 & 0 & 0 \\ 0 & A_1 & 0 \\ 0 & 0 & A_2 \end{pmatrix},$$

bordered block diagonal (BBD):

$$A = \begin{pmatrix} A_{00} & 0 & A_{02} \\ 0 & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix},$$

and separated block diagonal (SBD):

$$A = \begin{pmatrix} A_{00} & A_{01} & 0 \\ A_{10} & A_{11} & A_{12} \\ 0 & A_{21} & A_{22} \end{pmatrix};$$

where the A_{ij} are, not necessarily square, sparse submatrices of A. The zeroes 0 in this case refer to submatrices of appropriate size consisting of only zero entries.

If a sparse matrix has enough (hidden) structure, row and column permutations can be used to bring it in BBD or SBD form. Block diagonal form is quite uncommon, since it would mean the linear system can be decoupled ($A_{\{0,1,2\}}$ are independent). Assuming pivoting is not necessary (all diagonal elements were

nonzero), then it is easy to see that *no fill-in can occur in the* 0*-matrices*; the $l^{(k)}, u^{(k)}$ cannot have nonzero entries outside the A_{ij} submatrices, and hence no fill-in can occur there.

Assignment H. (1 pt.)

Assume the LU decomposition algorithm would run successfully on PAQ (without pivoting). Which sparsity structure would benefit reduction of fill-in more; BBD or SBD? Motivate your answer.

Given this, we now look at the case where pivoting *is* needed. To bring nonzeroes on the diagonal, further row permutations are made; this may cause rows from submatrices to be swapped with those from other submatrices (either one of the zero submatrices or one of the other sparse blocks). Does this affect the reason why there was no fill-in in the zero blocks in the case without pivoting? The answer is sometimes, for example with BBD if rows from the middle matrix parts are permuted to the upper parts, thereby spoiling the nonzeroes of the first few $u^{(i)}$. Swapping from middle to lower parts is not a problem, however; so if the pivoting algorithm can live with such restrictions, BBD would reduce fill-in even when pivoting. Similar considerations would hold for the SBD form.

Summarising, row- (P) and column-permutations (Q) of A, can be used in LU decomposition, if the matrix PAQ has some favourable sparsity structure (e.g., SBD or BBD) such that LU without pivoting would not cause fill-in in some blocks. Then, with some restrictions on further row permutations $\tilde{P}$ due to pivoting, no extra fill-in is induced. A full (pivoting) LU algorithm run on $\tilde{P}PAQ$ (taking into account the restrictions on $\tilde{P}$) reduces fill-in in the same way as a non-pivoting LU on PAQ would.

Assignment I. (3 pt.)

Suppose you would want to use a GA to reorder a sparse matrix A to reduce fill-in during LU decomposition. Think of a cost function for use in this GA to find such a reordering. Write tersely on how the cost function works and why you expect it to work well. Also, sketch methods to apply mutation or crossover to candidate solutions (chromosomes) of this problem. No testing (programming) is required for this assignment.

Hint: *One possible approach would be to consider band matrices. A banded matrix with bandwidth b has zero entries a_{ij} if $|i-j| > b$. Again by looking at the $l^{(k)}, u^{(k)}$ for each LU step k, no fill-in can occur at those entries i, j 'outside' of the band. For the same reasons as with BBD, this also holds after row-permutations due to pivoting.*