STEP-BY-STEP DIAGONAL DOMINANT PRECONDITION ALGORITHM FOR UNSYMMETRICAL SLE

[prepared by Duc T. NGUYEN, on November 5-2011]

STEP 1: Input/generate the square NxN unsymmetrical, non-singular matrix [A] and rhs vector {b}. Using MATLAB to find the condition # of [A]. Initialize the iteration counter k = 0. Copy matrix [M_k] = [A]. Note: We want to solve [A] * {x} = {b}, using iterative solver, such as GMRES, or Bi-Conjugate_STAB, or etc…, record MATLAB final answer, and the # iterations converged to a user's specified tolerance, say error tolerance = 10 ^ (-7). Also, use MATLAB command spy(A) to plot the non-zero patterns of matrix [A]. Also, record the solution time !

STEP 2: main iteration loop starts here, with k = k + 1

(a) Identify those rows of [A] which are "NOT" diagonal dominant.

(b) For those rows in (a), find the LOCATION of the largest absolute value of the "off-diagonal" term, and denote the corresponding location as (row p, column q), or simply denoted as (p, q).

(c) Solving 2 non-linear equations (discussed in the lecture) to obtain & store 2 angles theta(k) and phi(k), by using the following MATLAB command:
[theta, phi] = solve [' input 1-st nonlinear equation ', ' input the 2-nd nonlinear equation ']

(d) Compute/update matrix [M_k] = [P_k] transpose * [M_k-1] * [Q_k], where Matrices [P_k] transpose and [Q_k] are "ORTHONORMAL matrices", and they look almost like the Identity matrix, with 4 special values at locations (p,p), (p,q), (q,p), (q,q) which can be computed based on the 2 angles theta(k), and phi(k).

Matrix [M_k] is different with matrix [M_k-1] only in rows #p, q; and in columns #p, q. The remaining rows & columns of these 2 matrices were the same !

Also, for a typical non-zero term in rows #p, q; and in columns #p, q of the matrix [M_k], such as M_k(row ii, column jj), we need to check the following criteria:

Sqrt [M_k(ii ,jj) ^ 2 / { M_K(ii ,ii) * M_k(jj, jj) }] < 10 ^ (-s) where s = 0.1 is recommended

If the above criteria is satisfied, it means the non-zero OFF-DIAGONAL term M_k(ii, jj) is relatively small as compared to its diagonal terms [see K.J. Bathe's textbook (Jacobi iteration for eigen-solution) reference, and therefore this non-zero OFF-DIAGONAL term can be neglected !

The above strategy will offer the following advantages:

- It will make the updated matrix [M_k] more sparse, and hence it will be RELATIVELY CHEAPER (as compared to the matrix [A*], to be discussed later) to solve the system

[M] * {unknown vector w} = [A*] * {known vector v}

* It will make the process to transform the original matrix [A] into the "diagonal matrix" [A*] to converge quicker (since we have dropped/neglected a non-zero off-diagonal term).

- It will also save the computer memory required to store matrix [M]

Note: Matrix [M] needs be EXPLICITLY computed, since we have to provide this matrix [M] as a preconditioned matrix for MATLAB/gmres !

STEP 3: Test to see if the process has already converge into a diagonal matrix [A*] = [A_k] ? Where matrix [A*] = matrix [A_k] = [P_k] transpose * [A_k-1] * [Q_k].

Assuming the process will converge in the k-th iteration, then matrix [A*] can be computed as:

[A*] = [P_k] transpose * .. * [P_2] transpose * [P_1] transpose * [A] * [Q_1] * [Q_2] *. [Q-k]

Notes: However, there is no need to compute matrix [A*] EXPLICITLY (??), because in any iterative solver (such as GMRES), we only need to compute the product of [A*] * {a known vector} !?. On the other hand, we may have to compute/update the matrix [A*] EXPLICITLY, in order to check for EACH row of [A*] if already converged to DIAGONAL DOMINANT !??

Also, we should compute/update the matrix [M], right after computing/updating matrix [A*], since the updating matrix [M] is essentially the same as the updating [A*] with some non-zero off-diagonal terms NEGLECTED !!

If the process CONVERGE (or matrix [A*] CONVERGE) to a diagonal dominant matrix, then GO TO STEP 4,
Else
RETURN to STEP 2
endif

STEP 4: Solve the system [A*] * {intermediate unknown vector y} = {b*} by using MATLAB iterative solver, such as GMRES, and with the provided preconditioned matrix [M]

Where {b*} = [P_k] transpose …. [P_2] transpose * [P_1] transpose * {b}

Make sure to report the final answer to the same error tolerance = 10 ^ (-7), and the # iterations converged ! Also, record the solution time !

STEP 5: Compute the original unknown vector {x}, by solving the following equation:

{x} = [Q_1] * [Q_2] * …. * [Q_k] * {y}

For computational efficiency, the above rhs operations should be done from right to left. Furthermore, it should be realized that the operation [Q-k] * {known vector y} will result into an updated/new vector {y} which is different from its previous operation in only 2 locations (corresponding to locations in rows p, and q) !!

Also, record the solution time !

STEP 6: Define/compute matrix [P] = [M] inverse * [A*]

STEP 7: Use MATLAB built-in functions to compute and print the condition # of [P], [A*], [M]

Also, use MATLAB command spy (M) to plot the non-zero sparse pattern of [M] !!