Notes for Ma221 Lecture 17, Dec 5, 2024

Preconditioning to accelerate iterative methods

Our last topic in the course is preconditioning, i.e. changing A in a cheap way to make it better conditioned, to accelerate convergence. The simplest idea is to solve M^(-1)*A*x = M^(-1)*b, where M^(-1) is cheap to multiply by, and M^(-1)*A is better conditioned than A. Given M, this is straightforward for algorithms like GMRES, but not CG, since M^(-1)*A will not generally be spd. But if M is spd, with eigendecomposition M = Q*Lambda*Q^T, then we could define

M^(1/2) = Q*Lambda^(1/2)*Q^T,

and imagine applying CG to the equivalent spd system:

M^(-1/2)*A*M^(-1/2) * M^(1/2)*x = M^(-1/2)*b.

Since M^(-1)*A and M^(-1/2)*A*M^(-1/2) are similar, if one is well-conditioned, both are. It turns out that we can apply CG implicitly to this system without needing M^(1/2) or M^(-1/2):

Preconditioned CG:

```
k = 0 ; x(0) = 0, r(0) = b, p(1) = M^(-1)*b, y(0) = M^(-1)*r(0)
repeat
    k = k+1
    z = A*p(k)
    nu(k) = (y(k-1)^T*r(k-1)) / (p_k^T*z)
    x(k) = x(k-1) + nu(k)*p(k)
    r(k) = r(k-1) - nu(k)*z
    y(k) = M^(-1)*r(k)
    mu(k+1) = (y(k)^T*r(k))/(y(k-1)^T*r(k-1))
    p(k+1) = y(k) + mu(k+1)*p(k)
 until || r_k ||_2 small enough
```

Thm 6.9 in the text shows how the above algorithm is implicitly running CG on M^(-1/2)*A*M^(-1/2). (See posted errata for page 317 of the textbook.)

Recall that our goal is to choose M so that (1) it is cheap to multiply a vector by M^(-1) and (2) M^(-1)A is (much) better conditioned than A. Obviously choosing M=I satisfies goal (1) but not goal (2), and M=A is the opposite. So there is a wide array of preconditioners M that have been proposed, the best depending very strongly the structure of A. We give some examples below, from simple to more complicated.

(1) If A has widely varying diagonal entries, M = diag(a_11,a_22,...,a_nn) works. This is also called Jacobi preconditioning. Note that M^(-1)*A = I - R_J, so if the spectral radius rho(R_J) is small, i.e. Jacobi converges quickly, then M^(-1)*A is well-conditioned.

(2) More generally, one can take M = diag(A_11,A_22,...,A_kk), where each A_ii is
a diagonal block of A of some size. Choosing larger blocks makes multiplying by
M^(-1) more expensive, but M^(-1)*A better conditioned and so convergence is faster.
This is called block Jacobi preconditioning. For example, the A_ii might be chosen
corresponding to different physical regions of a simulation in which different
methods may be used to solve each A_ii (a similar idea is used in domain
decomposition below). Note that by replacing A by P*A*P^T where
P is a permutation, the blocks A_ii can correspond to any subset of rows and columns.

(3) "Incomplete Cholesky" means computing an approximate factorization of a spd
A ~ L*L^T = M, where for example, one might limit L to a particular sparsity pattern
to keep it cheap, such as A's original nonzeros, and then multiplying by
M^(-1) = (L*L^T)^(-1) = L^(-T) * L^(-1) by doing two (cheap) triangular solves.
The same idea for general A is called "incomplete LU", or ILU.

(4) One or more multigrid V cycles can be used as a preconditioner, given a
suitable A (or A_ii in (2)).

(We pause the recorded lecture here.)

(5) Finally, we mention domain decomposition, discussed more extensively in
section 6.10 of the text. To motivate, imagine we have a sparse matrix that,
like 2D Poisson, has a sparsity structure whose graph is a 2D n x n mesh.
However, suppose the matrix entries corresponding to left half of the mesh are
quite different from the right half, perhaps because they correspond to
different parts of a physical domain being modeled (eg finite element models
of steel and concrete). And the mesh points corresponding to the boundary between
these two domains, representing the interactions between steel and concrete,
are different again. Suppose we number the mesh points (matrix rows and columns)
in the following order: left half (steel), right half (concrete), and interface,
yielding the matrix

$$A = \begin{bmatrix} A_{ss} & 0 & A_{si} \\ 0 & A_{cc} & A_{ci} \\ A_{is} & A_{ic} & A_{ii} \end{bmatrix}$$

The subscript s refers to mesh points (rows and columns) in the steel,
c in the concrete, and i in the interface. Thus A_ss and A_cc are square
of dimension n*(n-1)/2 (assuming n is odd) and A_ii is square of dimension n,
so much smaller. The zero blocks arise because there is no direct connection
between steel and concrete, only via the interface.
Thus one may factor

$$A = \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ A_{is}A_{ss}^{-1} & A_{ic}A_{cc}^{-1} & I \end{bmatrix} \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & S \end{bmatrix} \begin{bmatrix} A_{ss} & 0 & A_{si} \\ 0 & A_{cc} & A_{ci} \\ 0 & 0 & I \end{bmatrix}$$

where S = A_ii - A_is*A_ss^(-1)*A_si - A_ic*A_cc^(-1)*A_ci is the Schur complement. Thus

```
A^(-1) = [ A_ss^(-1)       0          -A_ss^(-1)*A_si ]      [ I  0       0    ]
         [    0          A_cc^(-1)   -A_cc^(-1)*A_ci ]  *  [ 0  I       0    ]
         [    0             0                  I        ]     [ 0  0   S^(-1) ]

       [ I                          0              0  ]
   *   [ 0                          I              0  ]
       [ -A_is*A_ss^(-1)    -A_ic*A_cc^(-1)        I  ]
```

Given this factorization, it is natural to use any available preconditioners for A_ss and A_cc to approximate multiplication by the submatrices like (-A_is*A_ss^(-1))*x = -A_is*(A_ss^(-1)*x). S would be expensive to compute, and factorize, even though it is much smaller than A_ss or A_cc. On the other hand, multiplying (approximately) by S can also take advantage of the ability to multiply (approximately) by A_ss^(-1) and A_cc^(-1) using their preconditioners. And if we can multiply (approximately) by S, we can multiply by S^(-1) by using one of the other iterative methods already discussed, such as CG. This often works well because S can be much better conditioned than A (condition number growing like O(N) instead of O(N^2) for Poisson).

The above idea generalizes naturally when there are more than 2 domains (like s and c above).

The above idea is called "nonoverlapping" domain decomposition, because the s, c and i domains are disjoint. It is also possible to have "overlapping" domain decomposition, which can lead to faster convergence in some cases. For example, consider the 2D Poisson equation on a domain which is not a rectangle, but two partially overlapping rectangles, say forming an L-shaped domain. Given the availability of fast solvers for rectangular domains, how can we best combine them to solve the problem on overlapping domains? We could obviously use nonoverlapping domain decomposition, treating the L-shaped domains as two rectangles sharing an interface, but it turns out one can converge faster by making one rectangle larger, so it overlaps the other. Intuitively, this increases how fast information can move from one domain to the other, analogous to the motivation for multigrid, addressing the problem of data moving to just one neighboring mesh point per iteration illustrated in Fig 6.9 in the text. To express this mathematically, suppose we number the mesh points just inside one of the rectangles first, then the mesh point inside both, and then the mesh points just inside the second rectangle, yielding the matrix

A = [A_11 A_12 0]
    [A_21 A_22 A_23]
    [0   A_32 A_33]

In other words, R_1 = [A_11, A12; A_21, A22] is the entire first rectangle, and R_2 = [A_22, A23; A_32, A_33] is the entire second rectangle.

This suggests using the following preconditioner, sometimes also called "additive Schwartz" or "overlapping block Jacobi":

M^(-1) = [R_1^(-1) 0] + [0 0]
         [0   0]   [0 R_2^(-1)]

In words, this means solving (or approximately solving) the two rectangles separately. Just as standard Jacobi can be improved by using the most recent updates (Gauss-Seidel), the same idea can be used here, first updating rectangle 1, and then rectangle 2 (also called "multiplicative Schwartz"). And combining this technique with a multigrid-like idea of using a coarser grid approximation is also possible, see sec 6.10 for more details.