CS 6210: Matrix Computations

Preconditioning

David Bindel

2025-11-24

The why and how of preconditioning

Two weeks ago, we discussed the idea of approximation from a subspace, an gave a framework for error analysis involving the quality of approximate solutions available in the subspace and the stability of the approximation subproblem (summarized by a *quasi-optimality constant*). In more prosaic language, to get good results from a subspace method, we need good subspaces and good ways to extract approximations from subspaces.

Last week, we gave a brief discussion of two of the most popular Krylov subspace methods (CG and GMRES). In exact arithmetic, both of these methods converge to the exact solution in n steps, where n is the dimension of the problem. But we don't have the luxury of exact arithmetic; and even if we did, we generally want a good answer in far fewer than n steps. Sadly, for ill-conditioned problems, we may be unhappy with the number of steps needed to get to an acceptable answer – or we may simply be unable to achieve an acceptable answer at all in floating point arithmetic!

The role of *preconditioning* is to transform the problems we want to solve into equivalent problems for which Krylov methods converge quickly (or at least more quickly). Rather than solving $Ax = b$, we might solve the equivalent system

$$M_l^{-1} A M_r^{-1} (M_r x) = M_l^{-1} b;$$

where M_l and M_r are left and right preconditioning matrices. We note that

$$\mathcal{K}_k(M_l^{-1} A M_r^{-1}, M_l^{-1} b) = M_l^{-1} \mathcal{K}_k(AM^{-1}, b)$$

where $M = M_l M_r$; that is, we can see M_l and M_r as a splitting of a matrix M. In particular, if M is an spd matrix with $M = R^T R$ (or a similar symmetric factorization), we can often write algorithms that use the split preconditioner with $M_l = R^T$ and $M_r = R$ *implicitly*, only ever working explicitly with solves with the full M matrix – which can be viewed in the spd case as

defining an inner product. We commented on how to do this for the case of CG in last week's lecture notes.

I have commented that one can use classical stationary iterations like Jacobi and Gauss-Seidel as preconditioners, and in the homework you have the opportunity to explore an application-specific preconditioner based on fast solvers for Poisson problem discretizations on regular grids. But this is not exactly comprehensive advice, because I don't know how to give comprehensive advice. What makes a good preconditioner depends a lot on the fine details of the problem structure, and the design of preconditioners is still an active research area. Our goal with this lecture is not to give a detailed map of the preconditioning landscape, but to give some idea of the types of strategies that are employed and the type of intuitions that go into them. In the interest of keeping this to one lecture, I will focus primarily on the preconditioned CG iteration and our 2D Poisson model problem (and near neighbors).

Some guidelines

Chebyshev bounds

A few lectures ago, we noted that for positive definite problems, we expect to reduce the optimal residual norm by about $2/\sqrt{\kappa(A)-1}$ per step when $\kappa(A)$ is not too small. This bound was based on the approximation problem

$$\min_{p \in \mathcal{P}_{k-1}: p(0)=1} \max_{z \in [\alpha_{\min}, \alpha_{\max}]} |p(z)|,$$

where $[\alpha_{\min}, \alpha_{\max}]$ is a bounded interval containing the spectrum of A. The solution to this polynomial approximation problem involves scaled and translated Chebyshev polynomials, and is beautiful in its own right. But the bound really involves a relaxation of the problem

$$\min_{p \in \mathcal{P}_{k-1}: p(0)=1} \max_{\lambda \in \Lambda(A)} |p(\lambda)|;$$

and while the relaxed problem always provides an upper bound, that upper bound does not need to be particularly tight unless the eigenvalues are fairly uniformly spread out through an interval.

Eigenvalue counting

If $A = \eta I$, then $K_1(A, b)$ contains the solution to the problem $Ax = b$. What if A is not an identity, but has only a few distinct eigenvalues?

Consider an invertible diagonalizable matrix $A = V \Lambda V^{-1}$ with exactly k distinct eigenvalues. Then the k-dimensional Krylov subspace

$$\mathcal{K}_k(A, b) = \{p(A)b : p \in \mathcal{P}_{k-1}\} = \{V p(\Lambda) V^{-1} b\}$$

is an invariant subspace of A that contains the *exact* solution $x = A^{-1}b$, associated with a polynomial p derived by the interpolation conditions $p(\lambda_i) = \lambda_i^{-1}$ for $i = 1, \ldots, k$.

Eigenvalue clustering

The two points above suggest a strategy for thinking about the impact of a good preconditioner. We would ideally like $M^{-1}A$ to be well-conditioned, as that will let us apply Chebyshev-type reasoning. But it is OK if the $M^{-1}A$ is "almost" well-conditioned, in the sense that all but a few eigenvalues lie in a interval whose radius is small compared to the distance to one. In this case, we can think of multiplying a Chebyshev polynomial on the interval with a polynomial with zeros at each of the outlier eigenvalues, and this still establishes fast convergence. More generally, if $M^{-1}A$ has eigenvalues that cluster – and none of those clusters is too close to zero – then we expect to see fairly rapid convergence.

How do we find a matrix M that admits fast solves and so that $M^{-1}A$ has clustered eigenvalues? In general, this is pretty hard! So while the eigenvalue clustering picture is nice for thinking abstractly about the convergence of Krylov subspace methods, it is far from giving a practical recipe for constructing preconditioners.

Convergence studies and cost modeling

We usually monitor the convergence of iterative solvers with a semi-logarithmic plot of the (log) residual norm vs iteration count. For stationary iterations, we usually expect that after an initial transient period, the residual will decay by a factor of $\alpha < 1$ at each step, yielding a straight line with slope $\log \alpha$. For Krylov iterations, the convergence curve is usually much more complicated. Indeed, for methods like CG, the convergence curve does not need to decrease monotonically (assuming that we are measuring the ordinary two norm of the residual).

We monitor the residual norm instead of the error norm because we usually don't have an easy way to estimate the latter, with the exception of test problems where we know the solution in advance. However, if M^{-1} is a good approximation to A^{-1}, then the preconditioned residual norm $\|M^{-1}(b - Ax^{(k)})\|$ may provide a better proxy for the error norm than the ordinary residual norm does.

Convergence plots give us a sense of the number of *iterations* to reach a particular error tolerance for some problem. But what we care about is usually the *time* to solution rather than the number of iterations. This time consists of:

- Problem setup time (forming data structures for A and b)
- Preconditioner setup time (e.g. for approximate factorization or grid construction)
- Per iteration costs of
  - Multiply by A

- Solve with M
- Various vector operations in the solve

The preconditioners that most effectively reduce the number of iterations to converge often also cost a significant amount in setup time and time per iteration.

The cost modeling picture becomes more subtle still when we take into account the impact of parallel implementations. Preconditioners that are "better" in terms of more rapidly accelerating convergence may also involve a lot of communication between processors, and hence may scale poorly as a function of the number of parallel processing resources.

Classes of preconditioners

There are a lot of preconditioning ideas in the world, but not an infinite number. Most preconditioners involve a combination of a few ideas, which we will attempt to (partly) catalog here.

Point relaxations

Jacobi and Gauss-Seidel are sometimes called point relaxation methods: at each step in a sweep, we use one equation to update one variable. Other point relaxations include successive over-relaxation (SOR), which is like Gauss-Seidel, but we can take a longer step; symmetric Gauss-Seidel, implemented by applying a Gauss-Seidel sweep in a given order followed by a sweep in the reverse order (this corresponds to a symmetric M if A is symmetric); and symmetric successive over-relaxation (SSOR). These methods are all easy to program, with no setup costs and a relatively cheap cost for a preconditioner solve step.

Block relaxation and domain decomposition

Block Jacobi and block Gauss-Seidel iterations are exactly what the name suggests; rather than an update of the form

$$a_{ii} x_i^{\text{new}} = b_i - \sum_{j \neq i} a_{ij} x_j,$$

we solve

$$A_{ii} x_i^{\text{new}} = b_i = \sum_{j \neq i} A_{ij} x_j.$$

The block solve can be done exactly (e.g. with a factorization method) or it may be approximated with an inexact solve or via a fixed number of steps of a stationary iteration.

The *additive Schwarz* and *multiplicative Schwarz* updates are generalizations of block Jacobi and block Gauss-Seidel. Rather than considering a partitioning of the equations and unknowns

into blocks, the Schwarz methods consider *overlapping* subdomains. In the case of multiplicative Schwarz (generalizing Gauss-Seidel), it is clear how to stage the updates. In the case of additive Schwarz, we need some way to combine the contributions in the overlap region. In ordinary additive Schwarz, the combination usually happens by addition. In *restrictive* additive Schwarz, updates are computed by looking at a full subdomain (including an overlap region), but then each block only updates a subset of the variables that it "owns," with "ownership" corresponding to a partitioning of the variables into disjoint subsets.

The multiplicative Schwarz iteration began life not as a numerical method, but as a proof technique for the existence and uniqueness of solutions to PDEs on domains that were described by non-disjoint unions of domains with simple shapes.

Incomplete factorizations

Incomplete factorization methods try to approximate a Cholesky or LU factorization while restricting the fill. The incomplete LU factorization and incomplete Cholesky factorization with fill level k (written $ILU(k)$ and $ICC(k)$) allow fill associated with length k paths between nodes in the original graph; $ILU(0)$ (and $ICC(0)$) thus produce approximate factors with the same nonzero structure as the original matrix. Formally, given a sparsity pattern S, the incomplete factorization is writen as

$$\text{for each} k, i, j > k: \quad a_{ij} \leftarrow \begin{cases} a_{ij} - a_{ik} a_{kk}^{-1} a_{kj}, & \text{if } (i,j) \in S \\ a_{ij}, & \text{otherwise.} \end{cases}$$

Special care needs to be taken to ensure that an incomplete Cholesky factorization does not encounter a negative pivot (or to fix the factorization if such a pivot does appear).

In *thresholded* incomplete factorizations, the sparsity pattern is not chosen a priori, but is instead chosen based on dropping all entries of the factor that are below a certain size threshold.

Approximate inverses

A sparse approximate inverse is exactly what it sounds like: we compute M with a given sparsity pattern in order to try to achieve $MA \approx I$. For example, one can compute M with a given sparsity pattern to minimize $\|I - AM\|_F^2$ by solving a sequence of linear least squares problems

$$m_{I,j} = A_{:,I}^{\dagger} e_j$$

where I indicates the nonzero pattern of column j of the approximate inverse. One can choose the sparsity pattern in several ways (including adaptively). In general, special care must be taken to ensure symmetry and positive definiteness in cases when those properties matter.

It is also possible to compute the approximate inverse in factored form. This is particularly useful when A is positive definite, as a factored approximate inverse can be made structurally spd.

Polynomial preconditioning

Polynomial preconditioning uses a fixed polynomial approximation $q(A) \approx A^{-1}$. Though polynomial preconditioning does not appear to obviously improve the richness of the space explored relative to number of multiplies by A, it has the advantage that it may improve the available parallelism and decrease the amount of synchronization needed from operations like dot products. Polynomial preconditioners can also help when using solvers like restarted GMRES, since there are more matrix-vector operations (and maybe more progress) between restart cycles.

Coarse grid corrections

Suppose a Krylov iteration on a matrix A tends to converge slowly because of some eigenvalues close to zero. Let U be a basis for a space that contains good approximations to the eigenvectors associated with the smallest eigenvalues; then we can approximate solves with A by a Bubnov-Galerkin approximation on the basis U:

$$A^{-1} \approx U(U^T AU)^{-1} U^T.$$

This approximation to the inverse does not work as a preconditioner for conjugate gradients (for example), because it is not positive definite. However, we can modify the approximation so that it is an identity on the orthogonal complement to U:

$$\begin{aligned} M &= U(U^T AU)^{-1} U^T + (I - U(U^T U)^{-1} U^T) \\ &= I + U\left[(U^T AU)^{-1} - (U^T U)^{-1}\right] U^T \end{aligned}$$

When A comes from a discretization of a partial differential equation using finite differences or finite elements, we can often choose sparse U matrices so that $U^T AU$ corresponds to a discretization on a coarser mesh. In this case, we call the action of M a *coarse grid correction*. As in other cases, the solves with $U^T AU$ and $U^T U$ do not necessarily need to be done exactly to get a good preconditioner.

Coarse grid corrections can be used on their own, or as the building block for *multigrid* methods.

Operator splitting and ADI

In the case of the discrete Laplace equation

$$TU + UT = -h^2 F,$$

we can approximate the solution by two steps, mapping initial data U^0 to a new guess U^2:

$$\begin{aligned} TU^1 + U^0 T &= -h^2 F \\ TU^1 + U^2 T &= -h^2 F \end{aligned}$$

This is an example of the *alternating direction implicit* or ADI method. The more general pattern is that when $A = A_1 + A_2$ and both A_1 and A_2 admit fast inverses, we can use the approximation

$$x^1 = (A_1)^{-1}(b - Ax^0)$$
$$x^2 = (A_2)^{-1}(b - Ax^1)$$

This is closely related to the notion of *operator splitting methods* for time-steppers for certain classes of PDEs.

Fast solvers

We have already discussed how fast solvers (based on Fourier transforms or separation of variables) may be available for certain very special types of problems. There are many cases where the special case is not exactly the problem that we want to solve, but it is close enough to make a good preconditioner (as illustrated in HW 9).

Composing and combining preconditioners

Several of the preconditioning ideas we described above involve additional (smaller) linear solves as part of the preconditioner solve. That could be an approximate solve on a sub-block (as in Jacobi, Gauss-Seidel, or Schwarz methods); on a subspace (as in coarse grid corrections); or on a piece of the matrix (as in operator splitting).

Approximate block updates

In parallel applications, it is common to assign each processor to a part of a domain (or a subset of variables) to be updated independently using a Jacobi or additive Schwarz strategy. This leaves each processor to independently solve a still-large subproblem. Even if it does not make sense to use approximate factorizations or similar methods for the full system solve in this setting, it might make sense to solve these single-processor subproblems with such an approach, possibly combined with some other iterations.

A key point when using approximate solvers for submatrices in a preconditioner solve is that we need to use the *same* approximate solve strategy at each step. If the first step of an outer CG iteration involved a subdomain solve that did three steps of symmetric Gauss-Seidel, the next CG iteration should also do three steps of symmetric Gauss-Seidel on the same subdomain. The key point here is that our analysis of Krylov subspaces does not require that $M^{-1}A$ be an exact identity, but it does require that $M^{-1}A$ should not change from one step to the next.

Multi-level preconditioners

The idea of using an approximate solve on a subproblem is particularly useful when combined with the notion of coarse grid corrections; these two ingredients together underlie *multigrid methods*, which are the most scalable preconditioners available for many elliptic PDE problems. The basic idea is to use a simple iteration like Gauss-Seidel or (damped Jacobi) as a "smoother" that effectively reduces high-frequency error (associated with eigenvalues far from zero) together with a coarse-grid correction to reduce the high-frequency error. The coarse-grid solve in the correction step can itself be solved by a smoothing-and-coarsening pass. Recursively working with coarser and coarser methods until a direct solve is possible and then going back up the tree with another pass of smoothing gives a "multigrid V cycle."

Re-using preconditioners

Linear systems are not always solved in isolation. In nonlinear solvers and time-stepping methods, we often need to solve many linear systems with very similar (but not identical) matrices. Such settings are a target of opportunity when using preconditioners that require an expensive setup phase: we can set up the preconditioner once and then continue using it for new linear systems that arise, recomputing the preconditioner only when we notice convergence starting to slow.

Approximate preconditioning and flexible methods

In our description of subdomain solves above, we emphasized that we need to use a fixed preconditioner (or combination of preconditioners) in order for the theory of Krylov subspaces to hold. But what if we formed a subspace involving a different M at each step? Such a subspace is no longer a Krylov subspace, but that does not necessarily make it a bad space for approximation. Since it is no longer a Krylov subspace, though, we can no longer rely on the identies that hold for a Krylov subspace, and we need to adapt our methods accordingly. The FGMRES (flexible GMRES) algorithm of Saad is precisely designed for such settings, and has seen renewed interest recently in the context of iterative-refinement style algorithms with mixed precisions between factorizations and residual computations.