CS 542G: Solving Sparse Linear Systems

Robert Bridson

November 26, 2008

1 Direct Methods

We have already derived several methods for solving a linear system, say $Ax = b$, or the related least-squares problem with A rectangular. These assumed the matrix A is **dense**, i.e. all entries are stored explicitly regardless of whether they are zero or not. We can certainly apply these methods to sparse matrices stored densely, but they're not very scalable: dense Cholesky, LU, QR etc. are all $O(n^3)$ algorithms in their usual form (as in LAPACK). To get accurate approximate solutions to PDEs in 3D, we already are routinely facing values of n in the tens of millions today, and this is sure to increase in the future—not only will the dense routines be far too slow for this value of n, even the $O(n^2)$ memory cost will be far too big.

The eigen decomposition and the SVD are likely to be fully dense even when A is sparse. However, a quick check of the results of Cholesky, LU or QR applied to sparse matrices suggest the resulting factors often have some degree of sparsity themselves. For example, if you think about the first step of right-looking LU (i.e. row reduction), the rest of the sparse matrix is updated with a rank-one product of the first column and row—which therefore is sparse itself—and so the updated matrix will probably be sparse as well. We can therefore translate the dense Cholesky, LU, and QR (and related algorithms) into sparse algorithms, where we are careful only to ever operate on and store nonzero values. These methods for solving sparse linear systems and least-squares problems are termed **direct methods**, since they directly get the solution in a fixed number of steps (unlike iterative methods).

Writing a basic sparse version of, say, Cholesky isn't too hard at all: if you go the right-looking route, for example, you would want to store the nonzero entries in the matrix in some sort of dynamic data structure—maybe some sort of balanced tree for each column—and apart from loops going over just the nonzeroes instead of all entries, the code would look remarkably similar. Writing code to solve a linear system with a sparse triangular factor is similarly pretty simple (and could also be used in the context of

a left-looking or up-looking variant of Cholesky). These are usually far, far better than using fully dense methods to solve sparse linear systems. However, we can do significantly better with more sophisticated approaches.

There are two big issues for an advanced sparse direct method:

- ensuring the sparsity of the factors, and
- optimizing performance.

We'll take a brief look at these, focusing on Cholesky factorization as the most straightforward case, though much of this work generalizes to LU and QR (in the latter case, particularly since R is the Cholesky factor of A^TA, if you recall). The huge benefit of SPD matrices and Cholesky factorization is the guaranteed stability: partial pivoting is unnecessary, and thus we can exactly model the "structure" of the factorization without needing to know the actual numerical entries.

1.1 The Graph Model

The **sparsity pattern** (or "nonzero structure") of a matrix is simply a notion of where the nonzero entries are, or more precisely which entries are explicitly stored in the data structure[1]. A very useful model of the sparsity pattern is to construct a graph, where the vertices correspond to rows of the matrix and the edges correspond to the off-diagonal nonzero entries in the matrix. For a symmetric-structure matrix, the graph is undirected.

One step of right-looking Cholesky, or row-reduction, involves subtracting multiples of the first row from all other rows with a nonzero in the first column. This set of rows corresponds exactly to the neighbours of the first row in the graph. By subtracting off a multiple of the first row from any of these, discounting fluke cancellations, in graph terms we will be adding edges to all of the first row's neighbours. That is, we can think of one step of Cholesky as **eliminating** (deleting) the first vertex from the graph after connecting up all of its neighbours (adding an edge between any two if there isn't already an edge)—note the neighbours will then induce a complete subgraph.

[1] This more precise definition allows us to include cases where an unexpected cancellation causes a zero which we nonetheless store as if it was a general nonzero entry.

1.2 Ordering for Sparsity

Each step of Cholesky potentially could add new edges to the graph that didn't exist before—in other words, create new nonzeros in the L factor that weren't present in the original A. These new entries are called **fill**. If there is a lot of fill, the L factor will take a lot more memory to store (perhaps not even fit in main memory anymore) and the Cholesky factorization is going to run slowly. The first issue confronting direct methods is to try to keep fill as low as possible.

Since A is given to us—we can't change the initial graph—it may seem at first like there's no opportunity to affect the amount of fill we get. However, we do have one option: **ordering**. If we permute the rows of the matrix, and symmetrically the columns with the same permutation, we don't really change the linear system—it's trivial to similarly permute the right-hand-side and solution to make the equivalence. Such a symmetric permutation also obviously preserves the SPD property of the matrix, so Cholesky factorization still works. However, the Cholesky factorization of the reordered matrix may end up producing a completely different L than the original.

In graph terms, this amounts to relabeling the vertices of the graph—deciding on a different order of elimination. This can have a profound impact on fill. The most extreme example is the "star" graph, where a central vertex is connected to the $n-1$ other vertices and no other edges exist. If the central vertex is eliminated first (corresponding to a sparse matrix that's diagonal apart from a dense first row and column), immediately all the remaining $n-1$ vertices are connected to each other (the matrix fills in completely, becoming fully dense. However, if any of the other vertices—which each only have a single neighbours—are eliminated first, no fill occurs. Repeating the argument, if we order the central vertex last (a sparse matrix that's diagonal apart from a dense last row and column) we end up with zero fill. In this example, just by reversing the ordering we can go from $O(n^2)$ fill to zero fill.

This example generalizes to some extent. For example, if the graph is a tree (or forest), a zero fill ordering can be constructed as follows:

- Pick an arbitrary vertex as the root.
- Make any traversal of the tree starting at the root, i.e. only visiting a non-root vertex after a neighbour has been previously visited. Examples include depth-first and breadth-first search.
- Order the vertices in the reverse order of traversal, so the root is last.

One particular case we've already seen is the tridiagonal matrix that popped up in solving the 1D version of the Poisson problem with finite differences: its graph is a single path, a trivial tree.

Unfortunately, this is a rare case. Usually some amount of fill in unavoidable. Even more unfortunately, determining a minimum fill ordering is probably NP-hard (the unsymmetric version is definitely NP-hard, and many other related sparse matrix/graph problems are NP-hard). We instead need to rely on heuristic algorithms that apparently perform well in practice—well enough to make direct methods useful. There are two particular approaches which have arisen as the best.

The first is to take a purely greedy strategy: order the vertices one by one, at each step picking the vertex which will cause the least fill. Even simpler, we can just pick a vertex with minimum degree (number of neighbours) as a way to bound possible fill. This is the **Minimum Degree** algorithm. There have been a number of breakthroughs in making this algorithm more sophisticated, so that modern variants not only give somewhat higher quality orderings than the simple greedy approach, but also run in linear space and nearly linear time (even if the fill and Cholesky factorization time are much more than linear). The standard software today is AMD by Tim Davis et al., though further incremental improvements have been made by some researchers. (Incidentally, Tim Davis at the University of Florida not only has written some of the best direct method software available—and has released it as open-source—but also maintains a list of available direct method software and more.)

The second heuristic approach to ordering is based on **graph partitioning**. This is based on finding a good "vertex separator" can be found, i.e. a small set of vertices whose removal from the graph partitions it into roughly equal sized disconnected components. Finding optimal partitionings of graphs is NP-hard as well, but very good heuristic methods have been developed—with the Metis package of Karypis et al. being probably the most popular open-source software today. If the vertex separator is ordered last, with all the components ordered before it, then no fill edges can be created between the components. If the components are recursively ordered the same way, we arrive at the **nested dissection** algorithm. It has been shown that for reasonable meshes (as we might use for solving the Poisson problem with finite differences or finite elements) nested dissection can produce orderings that only produce some constant factor more than the minimum possible fill: $O(n \log n)$ fill for 2D problems, and $O(n^{5/3})$ fill for 3D.

Hybrids of minimum degree and nested dissection (e.g. switching to minimum degree in the recursion for small-enough components) currently are the best known algorithms for fill reduction. For many problems, including 2D PDEs, the low amount of fill they produce often makes direct methods the best choice for a linear solver. In other cases, especially for 3D PDEs on large meshes, the amount of fill may be too large for direct solvers to be feasible.

1.3 High Performance Direct Solvers

For Cholesky, where again partial pivoting isn't needed and the computation can be modeled exactly without knowing the numerical values of the nonzeroes in the matrix, significant gains in performance can be realized from the graph model. In particular, the exact nonzero structure of the factor L can be predicted in advance of the actual numerical factorization—in far less time and using less memory. This "symbolic factorization" step can be done after ordering, and obviate the need for dynamic data structures. This already makes for a huge gain in performance, since not only are overheads related to maintaining dynamic data structures eliminated, the factor can also be stored in a single contiguous block of memory with attendant cache benefits.

Even greater acceleration is possible for some problems. Since each step of the factorization induces a complete subgraph (on the neighbours of the eliminated vertex), it's quite natural for dense submatrices to arise during the course of factorization. The numerical operations on these submatrices can then be carried out as a whole with level 3 BLAS and LAPACK routines, which may come close to the peak floating point performance of the processor; regular sparse operations usually can only achieve a tenth of peak performance, due to cache misses, branch misprediction, and pipeline bubbles.

There are many excellent modern direct solver packages out there, many of which are open-source. See `http://www.cise.ufl.edu/research/sparse/codes/` for Tim Davis's list—I'd highlight UMFPACK and PARDISO in particular, and also recommend taking a look at my own code for Cholesky (and generalizations to some symmetric indefinite matrices) called KKTDirect, which is in the public domain but not yet on Davis's list.

2 Iterative Methods

In cases where direct solvers use too much memory or are too slow, a different approach is called for: **iterative methods**. Here we start with an initial guess at the solution and steadily refine it, hopefully converging to the correct answer quickly, and hopefully with control over the amount of memory needed.

We have of course already seen iterative methods in the context of more general optimization problems. One of the first things to concern ourselves with is knowing when to stop. Thankfully, for a linear system, that's fairly easy to determine. For the system $Ax = b$, with a particular guess x_i at the solution, we can define the **residual** as:

$$r_i = b - Ax_i$$

This is the same as our definition of residual for least squares problems earlier. A well-posed linear system

is solved if and only if the residual is zero, and it's not too difficult to work out a bound on the relative error of the solution based on the condition number of A: our original analysis of the well-posedness of a linear system pretty much did this. Thus we usually stop an iterative linear solver once the norm[2] of the residual is below some user-supplied tolerance.

One of the nice aspects of iterative methods is that if we have a good initial guess (perhaps the previous solution from a sequence of related linear systems) then that should speed up the method. However, it's often more convenient mathematically to assume an all zero initial guess. In fact, if the nonzero initial guess is x_0, we can instead think of solving $A(x_0 + \Delta x) = b$ for the correction Δx; this linear system is

$$\begin{aligned} A\Delta x &= b - Ax_0 \\ &= r_0 \end{aligned}$$

and it starts naturally with a guess of $\Delta x_0 = 0$.

Now let's turn to example iterative methods. We'll divide them loosely into two categories:

- **stationary** methods, where the i'th guess depends just linearly on the initial right-hand-side, and
- **non-stationary** methods, where more interesting formulas can be used.

2.1 Jacobi

Probably the simplest stationary method is Jacobi. It isn't really considered a practically useful method anymore, but related generalizations and combinations of Jacobi with other methods are significant so it's well worth knowing—and it serves as a good starting place for the next method, which is much more useful.

The idea behind Jacobi is to look at an individual equation in $Ax = b$, say the i'th row:

$$a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n = b_i$$

Assuming the diagonal a_{ii} entry is the largest in this row of the matrix, the left-hand-side of this equation responds most sensitively to adjustments in x_i. Therefore, if we have an "old" guess for the x values, in some sense the most efficient way to update it to exactly satisfy the i'th equation is to modify x_i:

$$x_i^{\text{new}} = \frac{b_i - a_{i1}x_1^{\text{old}} - \ldots - a_{in}x_n^{\text{old}}}{a_{ii}}$$

[2]Usually the 2-norm is most convenient, but if the method has really gotten close to convergence it shouldn't really matter which of the standard norms is used.

Here the ... do not include the x_i term of course. If we simultaneously update x_1 from the first equation all the way to x_n from the last equation, we have the Jacobi step.

Jacobi has many nice features: it needs no extra storage for a factorization, each step runs basically as fast as a sparse matrix multiplication with A, and every entry of x can be updated in parallel. It's fairly simple to prove it does indeed converge for diagonally dominant matrices. However, it usually fails to converge for more general linear systems, and even when it does work its convergence rate can be very slow for systems with large condition numbers.

2.2 Gauss-Seidel

One very small change to Jacobi can significantly improve its power. When updating x_i, instead of using just old values for the other entries in x—from the previous iteration—we can instead use the newest values available. That is, we update the values one at a time, in order from x_1 to x_n, always using the latest known values. At equation i we have:

$$x_i^{\text{new}} = \frac{b_i - a_{i1}x_1^{\text{new}} - \ldots - a_{i,i-1}x_{i-1}^{\text{new}} - a_{i,i+1}x_{i+1}^{\text{old}} - \ldots - a_{in}x_n^{\text{old}}}{a_{ii}}$$

This is the **Gauss-Seidel** method.

Gauss-Seidel is guaranteed to work for diagonally dominant matrices, just like Jacobi, but is often faster to converge. More importantly, it is guaranteed to converge for any arbitrary SPD matrix A, though this isn't obvious at first blush. Despite other methods having better asymptotic guarantees, it's often the best bet—and certainly the simplest—when time constraints mean only a few iterations can be taken but a crude approximation to the solution is acceptable.

Successive Over-Relaxation (SOR) is a common extension of Gauss-Seidel, with the idea that if going from x_i^{old} to x_i^{new} gets you closer to the solution, going a little bit further in the same direction may do even better. A **relaxation parameter**[3] $\omega > 1$ is introduced, similar to a step-length in line search. The SOR update for equation i is:

$$x_i^{\text{new}} = x_i^{\text{old}} + \omega \left(\frac{b_i - a_{i1}x_1^{\text{new}} - \ldots - a_{i,i-1}x_{i-1}^{\text{new}} - a_{i,i+1}x_{i+1}^{\text{old}} - \ldots - a_{in}x_n^{\text{old}}}{a_{ii}} - x_i^{\text{old}} \right)$$

With the right choice of ω, this can converge an order of magnitude faster than regular Gauss-Seidel. Unfortunately, the optimal ω is strongly dependent on A and difficult to determine cheaply—and is often

[3]The term "relaxation" is often associated with older iterative methods; I suspect the term is physically motivated from PDE problems in elasticity where solving the linear system corresponds to finding a minimal stress, i.e. optimally relaxed, configuration of an elastic object.

quite sensitive in that a slightly larger or smaller ω may give much slower convergence, and if too large can even cause SOR not to converge at all. If the type of problem is known in advance, a good value may be determined—for example, $\omega = 1.3$ is often reasonable for the Poisson problem—but for "black box" software which is just given an arbitrary matrix this may be out of the question.

Gauss-Seidel and SOR are significantly more powerful than Jacobi, but do suffer one disadvantage in comparison: they aren't as obviously parallelizable. Updating x_i would appear to require variables x_1 to x_{i-1} to be updated already, necessarily sequentializing the code. However, if A is sufficiently sparse, it may be symmetrically permuted (as we discussed for direct methods) to allow for some degree of parallelism. The update of x_i actually only has to wait for those x_j where $a_{ij} \neq 0$, and an appropriate reordering can exploit this to allow parallelism.

2.3 Steepest Descent

While there several other useful stationary methods[4] let us now turn to nonstationary iterative solvers. The first example is one we have seen already for general optimization: **Steepest Descent**.

When A is SPD, solving the linear system $Ax = b$ is equivalent to minimization the following quadratic:

$$\min_x \frac{1}{2}x^T Ax - x^T b$$

Differentiating this objective with respect to x and setting the gradient to zero immediately gives the linear system $Ax - b = 0$. Any of the optimization algorithms we looked at before are thus applicable to solving an SPD linear system, short of Newton (which relies on the ability to solve the linear system itself).[5] For steepest descent, we in fact already worked out the optimal step length for line search when we took a stab at analyzing its convergence properties.

Let's look at steepest descent applied to SPD systems in more detail. First we need the search direction, the negative of the gradient. The gradient of the quadratic form is $Ax - b$, therefore the search direction is $b - Ax$, i.e. the residual r. With step length α, we have a new guess $x^{\text{new}} = x + \alpha r$ which plugged in to the objective gives:

$$\begin{aligned} &\frac{1}{2}(x + \alpha r)^T A(x + \alpha r) - (x + \alpha r)^T b \\ = &\frac{1}{2}x^T Ax + \alpha r^T Ax + \frac{1}{2}\alpha^2 r^T Ar - x^T b - \alpha r^T b \end{aligned}$$

[4] If interested in learning more, "Chebyshev acceleration", "domain decomposition", and "multigrid" are all powerful classes of stationary methods you might look up.

[5] For example, Cyclic Coordinate Descent ends up being equivalent to Gauss-Seidel.

The locally optimal α which minimizes this results from differentiating and setting to zero:

$$\begin{aligned} r^T Ax + \alpha r^T Ar - r^T b &= 0 \\ \Rightarrow \qquad \alpha &= \frac{r^T b - r^T Ax}{r^T Ar} \\ &= \frac{r^T (b - Ax)}{r^T Ar} \\ &= \frac{r^T r}{r^T Ar} \end{aligned}$$

Note calculating this α requires (as the numerator) the 2-norm of r, which is convenient for testing for convergence as well. Once we have it, x is updated as $x^{\text{new}} = x + \alpha r$, and the new residual can be calculated as:

$$r^{\text{new}} = b - Ax^{\text{new}}$$

We can actually save a little computation here, as this is equivalent to

$$\begin{aligned} r^{\text{new}} &= b - A(x + \alpha r) \\ &= (b - Ax) - \alpha Ar \\ &= r - \alpha Ar \end{aligned}$$

and we have to compute Ar in finding α anyhow.

Putting this together into a reasonably efficient form, we get the following algorithm:

- Set $x_0 = 0$ and $r_0 = b$ (or correct for a nonzero initial guess)
- For $i = 0, 1, \ldots$
  - Compute $\rho = r_i^T r_i$; if less than user-specified ϵ stop (converged)
  - Multiply $s = Ar$
  - Set $\alpha = \rho / (r^T s)$, the optimal step length
  - Update $x_{i+1} = x_i + \alpha r_i$
  - Update $r_{i+1} = r_i - \alpha s$

I've included distinct subscripts for the guesses at x and r, but typically the update would overwrite the existing values since the older values need not be kept around.

In general nonlinear optimization, steepest descent is a useful method to be able to fall back on (or otherwise incorporate into fancier algorithms) since it has strong guarantees about converging for many classes of problems—even if it can be slow. However, in the SPD linear system case, steepest descent is asymptotically no faster than Gauss-Seidel and can be slower than SOR—and in real terms is almost always somewhat slower than Gauss-Seidel. This makes it nearly irrelevant for solving linear systems.

However, it does form the basis for a much more powerful algorithm that has become essentially the standard iterative method for SPD problems, and has variations for more general linear systems.

2.4 Conjugate Gradient

In steepest descent, the first guess is $x_0 = 0$ with residual $r_0 = b$. Every subsequent guess x is a linear combination of the previous guess and the previous residual; the new residual is a linear combination of the previous residual and A times the previous residual. It's not hard to prove, then, that

$$x_i \in \text{span}(b, Ab, \ldots, A^{i-1}b)$$

This linear subspace of dimension i is called a **Krylov subspace**, formed from powers of the matrix multiplying a given vector b.

One fundamental reason who steepest descent is slow is that it's too greedy: it finds a new guess by optimizing over just a line, a one-dimensional space, parallel to the current residual. We can do far, far better by instead seeking a minimum over the entire Krylov space seen so far. This is one interpretation of the **Conjugate Gradient** (CG) algorithm.

The first comment to make about this is that we've seen this idea many times before. What it amounts to is taking a large problem (of dimension n), finding a much smaller subspace we think might be useful (of dimension i), and solving the problem restricted to that subspace with the hope that the result should be a good approximation to the full problem. We did exactly the same in Rayleigh-Ritz for approximating eigenvalues and eigenpairs and in the Finite Element Method for approximating solutions of PDEs.

The second comment is that this might appear expensive: at the i'th iteration we are finding the solution of an i-dimensional quadratic minimization, i.e. an $i \times i$ linear system, which could take as much as $O(i^3)$ time. If it takes 1000 iterations to converge, that would be enormously expensive in total. The near-miracle of CG is that a basis for the Krylov space can be constructed which allows us to go from the i'th guess to the next with a constant number of operations: one multiply with A, a few dot-products, and a few vector updates. CG is hardly more expensive than steepest descent in its final form. We don't have time in this course to derive it, but I'll give you the usual version for reference:

- Start with $x_0 = 0$, $r_0 = b$, $p_1 = r_0$.
- Compute $\rho_0 = r_0^T r_0$ and if zero already, return.
- For $i = 1, 2, \ldots$
  - Multiply $q = Ap_i$.

- Compute $\alpha = \rho_{i-1}/(p_i^T q)$.
- Update $x_i = x_{i-1} + \alpha p_i$ and $r_i = r_{i-1} - \alpha q$.
- Compute $\rho_i = r_i^T r_i$, and if small enough return (converged).
- Compute $\beta = \rho_i/\rho_{i-1}$.
- Update $p_{i+1} = r_i + \beta p_i$.

As with steepest descent, the updates to x, r, and the auxiliary vector p are usually done in-place, and past ρ values are discarded when no longer needed. Also, for robustness there's generally some maximum iteration count at which point, if still not converged, we return with an error flag.

It should also be pointed out that, just like steepest descent, A only appears in the algorithm in the form of a matrix-vector product, $q = Ap$. This gives the method a lot of flexibility that methods like Gauss-Seidel (which depend on knowing the entries in A) don't necessarily have. For example, A might not be explicitly stored as a sparse matrix, but only arise as a black-box function call which gives the effect of multiplying a vector by A: A might be most naturally given as a product of factors; in FEM where A is a global stiffness matrix assembled from local stiffness matrices, we can skip the assembly step and directly compute Ap element by element; or the application of A might be approximated with algorithms like Barnes-Hut or the Fast Multipole Method.

CG can be shown to converge, in the worst case, in the square root of the number of iterations that steepest descent takes: it depends on the square root of the condition number of A. This is as fast as SOR can converge in general, but it happens automatically without the tricky business of finding an optimal relaxation parameter. In fact, CG does even better. For an $n \times n$ matrix, it must get the exact solution by the n'th iteration, since at that point it's optimizing over the entire n-dimensional space—at least in exact arithmetic. In floating-point arithmetic it might never get there exactly, but this is a strong hint that CG has to speed up as it progresses. Another way of thinking about it is that the Krylov spaces over which CG runs contain increasingly good approximations of the biggest and smallest eigenvectors (due to the Power Method), and so CG in essence solves the largest and smallest components early reducing the effective condition number of what's left...

Back when it was invented in the early 1950s, CG was considered more of a direct method since in exact arithmetic it should get the exact answer after n steps. It was observed at the time that a good approximation could be found much earlier on, but since direct methods were a decent constant factor faster for dense matrices, CG was shelved for many years. Its full potential as an iterative method was only realized decades later. Further work, mostly finished by the 1990s, has extended the ideas of CG to Krylov subspace methods for other types of matrices: symmetric indefinite problems (with algorithms named MINRES, SYMMLQ, SQMR, and one or two others), least-squares problems (e.g. LSQR), and

general unsymmetric problems (GMRES, BiCGStab, and a few others). These are now almost always the standard iterative approach, particularly thanks to one more advance.

2.5 Preconditioning

While CG is an order of magnitude faster than steepest descent, Gauss-Seidel, etc. and is guaranteed to work, for large or ill-conditioned problems it might still be very slow. Ideally we would be able to change the matrix to have a smaller condition number, which would make CG faster—and in fact, we can do exactly that!

Notice that the linear system $Ax = b$ is exactly equivalent to $MAx = Mb$, for any invertible matrix or linear operator M. Sweeping aside the issue of whether MA is symmetric or not for now, if the condition number of MA is much smaller than that of A, CG applied to this modified system should run much faster. We call M a **preconditioner**.

With some work, it's possible to derive the Preconditioned Conjugate Gradient (PCG) algorithm, which works for any symmetric positive definite matrix A and any symmetric positive definite preconditioner M. For your reference (we have no time to derive it either), here it is:

- Start with $x_0 = 0$, $r_0 = b$, $z = Mr_0$, $p_1 = z_0$.
- Compute $\rho_0 = r_0^T z$ and if zero already, return.
- For $i = 1, 2, \ldots$
  - Multiply $q = Ap_i$.
  - Compute $\alpha = \rho_{i-1}/(p_i^T q)$.
  - Update $x_i = x_{i-1} + \alpha p_i$ and $r_i = r_{i-1} - \alpha q$.
  - If $\|r_i\|$ is small enough return (converged).
  - Precondition $z_i = Mr_i$.
  - Compute $\rho_i = r_i^T z_i$.
  - Compute $\beta = \rho_i/\rho_{i-1}$.
  - Update $p_{i+1} = z_i + \beta p_i$.

We can actually reuse the same storage for q and z, so this is still nice and trim.

Ideally M should approximate the action of A^{-1}, since $\kappa(AA^{-1}) = \kappa(I) = 1$. But of course we also want something that's efficient to evaluate; the art of preconditioning is finding a good trade-off. This is the main area of effort in research for iterative linear solvers today, and is also an example where stationary methods we saw before—even Jacobi—can find a practical use, as linear operators approximating A^{-1} that can be embedded in a robust Krylov subspace solver.