Notes for CS542G (Congugate Gradient for Linear Systems)

Robert Bridson

November 22, 2007

We now turn to non-stationary methods, i.e. methods for solving $Ax = b$ iteratively that depend nonlinearly on b. We again assume A is symmetric positive definite, so solving the linear system is equivalent to minimizing $\frac{1}{2}x^T Ax - x^T b$. The two methods we look at here are Steepest Descent and Conjugate Gradients. For a more detailed elementary exposition you might also want to read Jonathan Shewchuk's *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*, freely available on the web.

1 Steepest Descent

Steepest Descent is one of the most basic of optimization algorithms, which we saw before in the general nonlinear context, and can easily be used here. At any given x, the gradient of the objective $\frac{1}{2}x^T Ax - x^T b$ is in fact:

$$\begin{aligned}\frac{\partial}{\partial x}\left(\frac{1}{2}x^T Ax - x^T b\right) &= Ax - b \\ &= -r\end{aligned}$$

That is, the steepest descent direction (the negative of the gradient) at guess x_k is just the k'th residual: $r_k = b - Ax_k$.

If you recall, the other important ingredient in Steepest Descent (for general nonlinear optimization) was line search: selecting a step size α that made sure the next iterate is an improvement. In this case the next iterate will be:

$$x_{k+1} = x_k + \alpha r_k$$

We can in fact solve for the locally optimal α that minimizes the objective: just take the derivative with respect to α and set it to zero:

$$\begin{aligned}\frac{\partial}{\partial \alpha}\left(\frac{1}{2}x_{k+1}^T Ax_{k+1} - x_{k+1}^T b\right) &= 0 \\ \frac{\partial}{\partial \alpha}\left(\frac{1}{2}(x_k + \alpha r_k)^T A(x_k + \alpha r_k) - (x_k + \alpha r_k)^T b\right) &= 0 \\ \frac{\partial}{\partial \alpha}\left(\frac{1}{2}r_k^T Ar_k\alpha^2 + r_k^T Ax_k\alpha + \frac{1}{2}x_k^T Ax_k - r_k^T b\alpha - x_k^T b\right) &= 0 \\ r_k^T Ar_k\alpha + r_k^T Ax_k - r_k^T b &= 0 \\ r_k^T Ar_k\alpha + r_k^T(Ax_k - b) &= 0 \\ r_k^T Ar_k\alpha &= r_k^T(b - Ax_k) \\ r_k^T Ar_k\alpha &= r_k^T r_k \\ \alpha &= \frac{r_k^T r_k}{r_k^T Ar_k}\end{aligned}$$

Note that since A is positive definite, the denominator $r_k^T A r_k$ is positive—unless $r_k = 0$ in which case we've already solved the system and there's no point in taking another iteration.

Also note that since $x_{k+1} = x_k + \alpha r_k$, then we can update the residual rather than compute it from scratch:

$$\begin{aligned} r_{k+1} &= b - Ax_{k+1} \\ &= b - A(x_k + \alpha r_k) \\ &= b - Ax_k - \alpha A r_k \\ &= r_k - \alpha A r_k \end{aligned}$$

To compute the denominator of α we already will have computed the matrix-vector product Ar_k, so we can reuse it here to speed up the algorithm.

This gives us the Steepest Descent algorithm:

- Start with $x_0 = 0$, $r_0 = b$.
- For $k = 0, 1, 2, \ldots$
  - Compute $\rho = r_k^T r_k$, which is, in fact $\|r_k\|_2^2$ — if small enough, stop (converged)
  - Multiply $q = Ar_k$
  - Compute $\alpha = \rho/(r_k^T q)$
  - Update $x_{k+1} = x_k + \alpha r_k$ and $r_{k+1} = r_k - \alpha q$

We require just one extra vector (q), and in each iteration perform just one matrix-vector multiply along with two dot-products and two vector updates.

1.1 Convergence of Steepest Descent

So now the question becomes: how good is Steepest Descent? We looked at this before when we analyzed its performance in the context of general nonlinear optimization. Let's go through it again. Before jumping in, let's define the **energy norm** or A**-norm** of the error $e = A^{-1}b - x$:

$$\|e\|_A = \sqrt{e^T A e}$$

Since A is positive definite, this is a well-defined norm. Since we're dealing with a finite dimensional space, it's of course equivalent to any other norm (in the sense it can only be different from another norm of e by at most some constant factor). Another way of looking at the energy norm is in terms of the residual:

$$\begin{aligned} \|e\|_A^2 &= e^T A e \\ &= (A^{-1}b - x)^T A (A^{-1}b - x) \\ &= (b - Ax)^T A^{-1} (b - A^{-1}x) \\ &= r^T A^{-1} r \end{aligned}$$

A different expansion gives another useful relation:

$$\begin{aligned} \|e\|_A^2 &= e^T A e \\ &= (A^{-1}b - x)^T A (A^{-1}b - x) \\ &= b^T A^{-1} b - 2x^T b + x^T A x \\ &= 2\left[\frac{1}{2} x^T A x - x^T b\right] + b^T A^{-1} b \end{aligned}$$

From this it's obvious that minimizing the objective function we saw before is the same as minimizing the energy norm of the error. We should expect, then, that Steepest Descent will improve the energy norm of the error with each step; the question remains how fast?

Expand out the new energy norm using the first relationship above:

$$\begin{aligned}
\|e_{k+1}\|_A^2 &= r_{k+1}^T A^{-1} r_{k+1} \\
&= (r_k - \alpha A r_k)^T A^{-1} (r_k - \alpha A r_k) \\
&= r_k^T A^{-1} r_k - 2\alpha r_k^T r_k + \alpha^2 r_k^T A r_k \\
&= \|e_k\|_A^2 - 2\alpha r_k^T r_k + \alpha^2 r_k^T A r_k
\end{aligned}$$

Now substitute in $\alpha = r_k^T r_k / r_k^T A r_k$:

$$\begin{aligned}
\|e_{k+1}\|_A^2 &= \|e_k\|_A^2 - 2\frac{(r_k^T r_k)^2}{r_k^T A r_k} + \frac{(r_k^T r_k)}{r_k^T A r_k} \\
&= \|e_k\|_A^2 - \frac{(r_k^T r_k)^2}{r_k^T A r_k}
\end{aligned}$$

This handily guarantees that until we converge the energy norm of error will strictly decrease—the term we're subtracting off of course must be positive. Further extracting out the factor by which the energy norm decreases, using $\|e_k\|_A^2 = r_k^T A^{-1} r_k$ again, gives:

$$\|e_{k+1}\|_A^2 = \left(1 - \frac{(r_k^T r_k)^2}{(r_k^T A r_k)(r_k^T A^{-1} r_k)}\right) \|e_k\|_A^2$$

We now want to bound this factor. The denominator in the fractional part is positive, so it's the same as:

$$(r_k^T A r_k)(r_k^T A^{-1} r_k) = \|r_k^T A r_k\|_2 \|r_k^T A^{-1} r_k\|_2$$

These norms are just absolute values of scalars which are already guaranteed to be positive, so we haven't actually changed anything. Now we simply use our definitions of matrix norms to bound this:

$$\begin{aligned}
(r_k^T A r_k)(r_k^T A^{-1} r_k) &\leq \|r_k\|_2 \|A\|_2 \|r_k\|_2 \|r_k\|_2 \|A^{-1}\|_2 \|r_k\|_2 \\
&= \|r_k\|_2^4 \|A\|_2 \|A^{-1}\|_2 \\
&= (r_k^T r_k)^2 \kappa(A)
\end{aligned}$$

where $\kappa(A)$ is the condition number of A (for the 2-norm), which is simply the ratio of the largest eigenvalue to the smallest eigenvalue. This gives us a bound on the error, finally:

$$\begin{aligned}
\|e_{k+1}\|_A^2 &\leq \left(1 - \frac{(r_k^T r_k)^2}{(r_k^T r_k)^2 \kappa(A)}\right) \|e_k\|_A^2 \\
&= \left(1 - \frac{1}{\kappa(A)}\right) \|e_k\|_A^2
\end{aligned}$$

I should say, this is only an upper bound. In some cases Steepest Descent converges much faster, e.g. if b is parallel to a single eigenvector then it converges in one iteration (it should be easy to work out why!). However in practice it appears to be a fairly representative upper bound.

Finally, expanding this out we see the the error in the k'th guess is bounded by:

$$\begin{aligned}
\|e_k\|_A &\leq \left(1 - \frac{1}{\kappa(A)}\right)^{k/2} \|e_0\|_A \\
&\approx e^{-\frac{k}{2}\kappa(A)} \|e_0\|_A
\end{aligned}$$

Therefore we can expect to have to take $O(\kappa(A))$ iterations to reduce the relative error to below a given threshhold. If you recall from the previous notes on stationary methods, for the Poisson problem the condition number was $O(n^2)$, where n is the number of grid points along one side of the mesh—so we are looking at $O(n^2)$ iterations for Steepest Descent to converge. This isn't any better than Gauss-Seidel.

2 Conjugate Gradient

One perspective on the poor performance of Steepest Descent is that it's too greedy: it performs a locally optimal line search along the direction of steepest descent, but globally this is not the fastest way to get to the minimum. For ill-conditioned problems, this ends up with Steepest Descent zig-zagging a lot mostly in the direction of the largest eigenvalue's eigenvector, but not making much progress in the other eigenvector directions.

We've already seen a few ways to fix this: Newton's method[1] picked a different direction which got to the minimum faster, and Barzilai-Borwein from assignment 2 used the steepest descent direction but a smarter step-length. We'll now look at another way to improve it, leading to the Conjugate Gradient (CG) algorithm which has become the de facto standard for iteratively solving SPD linear systems.

Where Steepest Descent is slow because it keeps optimizing along almost the same direction, picking the best x_{k+1} from the one-dimensional set $x_k + \text{span}\{r_k\}$, we'll speed it up by instead picking the best x_{k+1} from the entire subspace we've seen so far. That is, we will choose x_{k+1} as the best guess from $\text{span}\{r_0, r_1, \ldots, r_k\}$. We're now globally optimizing each iteration, so we can't help but do better!

To make this a bit easier, we'll actually introduce another set of vectors $p_1, p_2, \ldots$ called **search directions**, which span the same space as the residuals. This doesn't change the choice of x_{k+1}, but will give us some freedom to find the best x_{k+1} more efficiently. The first search direction will be the initial residual, $p_1 = r_0$ (which is of course just b), and the rest we'll compute as we go.

Define an $n \times k$ matrix P_k whose columns are the first k search directions:

$$P_k = (p_1 | p_2 | \cdots | p_k)$$

When we say we want x_k to the best guess from the span of $\{p_1, p_2, \ldots, p_k\}$ we are simply saying

$$x_k = P_k a_k$$

for some k-dimensional vector a_k, which contains the coefficients of the linear combination:

$$x_k = a_k^{(1)} p_1 + a_k^{(2)} p_2 + \ldots + a_k^{(k)} p_k$$

This linear combination should be the one that minimizes the objective $\frac{1}{2}x^T Ax - x^T b$, or equivalently the energy-norm of the error. That is:

$$a_k = \arg\min_a \frac{1}{2}(P_k a)^T A(P_k a) - (P_k a)^T b$$

Taking the gradient with respect to a and setting it to zero gives us:

$$P_k^T A P_k a_k = P_k^T b$$

Incidentally, this should look very familiar to you! We've taken the original linear system $Ax = b$ and "projected" it into a smaller subspace spanned by P_k; this is the same operation in play for Rayleigh-Ritz to approximate eigenvalues/eigenvectors and for Galerkin Finite Elements to approximate the solution of a PDE.

[1] Of course we can't directly use Newton here, since it requires solving a linear system with the Hessian, which in this case is exactly the linear system we're trying to solve in the first place!

Right now this looks a little dubious: at the k'th iteration we will apparently have to compute a dense $k \times k$ matrix $P_k^T A P_k$, then solve it with Cholesky, at significant cost if k is large. Therefore we'll cleverly choose the search directions so that $P_k^T A P_k$ is just a diagonal matrix. That is, we want $p_i^T A p_j = 0$ for $i \neq j$, a condition sometimes called being A**-orthogonal** or A**-conjugate**. This is in fact the same thing as being orthogonal if you redefine the inner-product to include a multiplication with A (and, not entirely surprisingly, the norm you get by using this modified inner-product is just the energy norm!).

We want p_k to have a component in the direction of the latest residual, r_{k-1}, but we can also include components from previous search directions to make it A-orthogonal. This should ring a bell: we have a set of vectors which we want to make A-orthogonal, which we'll do one by one... Gram-Schmidt! We can use Gram-Schmidt to A-orthogonalize the vectors, just using the A-inner-product instead of the usual dot-product. That is, the new search direction should be:

$$p_k = r_{k-1} - \sum_{j=1}^{k-1} \frac{r_{k-1}^T A p_j}{p_j^T A p_j} p_j$$

Note that we're not normalizing (changing the length) of the search directions, just making them A-orthogonal. You should have no problem verifying that this formula does indeed make $p_k^T A p_j = 0$ for all $j < k$, which in turn implies that all the search directions will be A-orthogonal.

So now $P_k^T A P_k$ is diagonal. Moreover, the first $k-1$ rows and columns will be the previous diagonal matrix $P_{k-1}^T A P_{k-1}$, and the first $k-1$ entries of $P_k^T b$ will be the previous vector $P_{k-1}^T b$. Then the first $k-1$ entries of a_k will be identical to a_{k-1}, meaning we don't have to solve for them again. Putting this together, we get that the new guess will be

$$\begin{aligned} x_k &= P_k a_k \\ &= P_{k-1} a_{k-1} + \frac{p_k^T b}{p_k^T A p_k} p_k \\ &= x_{k-1} + \alpha_k p_k \end{aligned}$$

where the "step length" α_k is now:

$$\alpha_k = \frac{p_k^T b}{p_k^T A p_k}$$

Just like in Steepest Descent, we can also work out the update to the residual:

$$\begin{aligned} r_k &= b - A x_k \\ &= b - A(x_{k-1} + \alpha_k p_k) \\ &= b - A x_{k-1} - \alpha_k A p_k \\ &= r_{k-1} - \alpha_k A p_k \end{aligned}$$

We're in good shape now: the only really expensive part left to tackle is the Gram-Schmidt step, which involves all the previous $k-1$ search directions.

Or does it? If you implement what we've done so far in MATLAB, you'll discover a remarkable property: the terms in the Gram-Schmidt formula for $j = 1, 2, \ldots, k-2$ will all be, up to round-off error, equal to zero. This is no lucky coincidence!

The equation $P_k^T A P_k a_k = P_k^T b$ which gave us the optimal guess from the search directions can be written as:

$$\begin{aligned} P_k^T A x_k &= P_k^T b \\ 0 &= P_k^T b - P_k^T A x_k \\ 0 &= P_k^T (b - A x_k) \\ 0 &= P_k^T r_k \end{aligned}$$

In other words, it is completely equivalent to look for the guess which gives a residual orthogonal to all the search directions so far.[2] And since the search directions span the same subspace as the residuals, this implies that **the residuals are all orthogonal**! In other words, $r_i^T r_j = 0$ for $i \neq j$. This actually gives us some insight as to why Conjugate Gradient will be so much faster than Steepest Descent: at each step it optimizes along a new direction, and can't zig-zag back and forth. It also will help us understand the mystery of the zero terms in Gram-Schmidt. Rearrange the residual update formula:

$$\begin{aligned} r_j &= r_{j-1} - \alpha_j A p_j \\ \Rightarrow \qquad A p_j &= \frac{r_{j-1} - r_j}{\alpha_j} \end{aligned}$$

Now plug this into the Gram-Schmidt formula:

$$\begin{aligned} p_k &= r_{k-1} - \sum_{j=1}^{k-1} \frac{r_{k-1}^T A p_j}{p_j^T A p_j} p_j \\ &= r_{k-1} - \sum_{j=1}^{k-1} \frac{r_{k-1}^T (r_{j-1} - r_j)}{\alpha_j p_j^T A p_j} p_j \end{aligned}$$

Since the residuals are all orthogonal, the terms are zero except when $j = k - 1$, giving:

$$p_k = r_{k-1} + \frac{r_{k-1}^T r_{k-1}}{\alpha_{k-1} p_{k-1}^T A p_{k-1}} p_{k-1}$$

Of course, this only applies for $k > 1$: remember $p_1 = r_0$. Substituting in the formula we have for $\alpha_{k-1} = p_{k-1}^T b / p_{k-1}^T A p_{k-1}$ simplifies this to

$$\begin{aligned} p_k &= r_{k-1} + \frac{r_{k-1}^T r_{k-1}}{p_{k-1}^T b} p_{k-1} \\ &= r_{k-1} + \beta_{k-1} p_{k-1} \end{aligned}$$

where $\beta_{k-1} = r_{k-1}^T r_{k-1} / p_{k-1}^T b$. This means not only is finding the next direction cheap, we don't even need to store previous directions: we can update them in place. It's nearly magical: we're finding a globally optimal solution from purely local calculations.

There's one niggling issue left, which we didn't cover in lectures. If you look at the formula for β, the denominator isn't obviously nonzero—what if p_{k-1} is orthogonal, or nearly orthogonal, to b? Let's take a closer look. Remember that b is actually the first residual r_0, and if we use the search direction update formula, we get (for $k > 1$)

$$\begin{aligned} p_k^T b &= p_k^T r_0 \\ &= (r_{k-1} + \beta_{k-1} p_{k-1})^T r_0 \end{aligned}$$

Since the residuals are orthogonal, this is just:

$$p_k^T b = \beta_{k-1} p_{k-1}^T r_0 = \beta_{k-1} p_{k-1}^T b$$

But now plug in our definition of $\beta_{k-1} = r_{k-1}^T r_{k-1} / p_{k-1}^T b$ to get

$$p_k^T b = r_{k-1}^T r_{k-1}$$

[2] Again, note the connection to Galerkin FEM: there too we find a linear combination of basis functions, where the "residual" (PDE) is orthogonal to all the basis functions.

after which we breathe a sigh of relief because this is definitely nonzero unless we've converged. From this we can get improved formulas for α and β:

$$\begin{aligned} \alpha_k &= \frac{r_{k-1}^T r_{k-1}}{p_k^T A p_k} \\ \beta_{k-1} &= \frac{r_{k-1}^T r_{k-1}}{r_{k-2}^T r_{k-2}} \end{aligned}$$

We get to reuse a lot of computations now.

Putting it all together, the Conjugate Gradient algorithm is as follows:

- Start with $x_0 = 0$, $r_0 = b$, $p_1 = r_0$.
- Compute $\rho_0 = r_0^T r_0$, which is, in fact $\|r_0\|_2^2$ — if zero already, return.
- For $k = 1, 2, \ldots$
  - Multiply $q = Ap_k$.
  - Compute $\alpha = \rho_{k-1}/(p_k^T q)$.
  - Update $x_k = x_{k-1} + \alpha p_k$ and $r_k = r_{k-1} - \alpha q$.
  - Compute $\rho_k = r_k^T r_k$, and if small enough return (converged).
  - Compute $\beta = \rho_k / \rho_{k-1}$.
  - Update $p_{k+1} = r_k + \beta p_k$.

Normally, the updates to x, r, and p are 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. For example: 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.

2.1 Convergence of Conjugate Gradient

Our first observation is that clearly CG will always outperform Steepest Descent: CG is optimizing over a larger set of directions. A somewhat arduous analysis can show that whereas Steepest Descent needs $O(\kappa(A))$ iterations in the worst case, CG only needs $O(\sqrt{\kappa(A)})$ iterations, an order of magnitude better. For the Poisson problem, that means $O(n)$ iterations instead of $O(n^2)$ for a grid of side length n. However, it can be even better.

In exact arithmetic, CG isn't necessarily an iterative solver: after n steps, it will return the best guess in the entire space, which must of course be the exact solution. In fact, CG was originally proposed as an alternative to Gaussian Elimination—but was initially discarded when it became clear that it took a constant factor more operations. In reality, with the presence of round-off perturbing the calculation, CG isn't exact, but will often get the solution to high precision long before the n'th iteration.

Viewed one way, CG enriches the space it works on at each step with one matrix-vector multiply. It's not hard to prove that it chooses the k'th guess x_k from the space

$$K^k(A, b) = \text{span}\{b, Ab, \ldots A^{k-1}b\}$$

which is called a **Krylov** subspace. (CG is one example of a **Krylov subspace method**, as is Steepest Descent, and several other methods not covered in this course that have been adapted to more general linear systems where A may be indefinite or unsymmetric.) Remembering the Power Method for finding eigenvectors, we can expect these Krylov subspaces to get steadily "richer" in being able to resolve the eigenvectors associated with the largest magnitude eigenvalues. In fact, since exactly the same space is generated from a shifted $A - \mu I$, choosing $\mu = \lambda_{\text{min}}$ or $\mu = \lambda_{\text{max}}$ shows that the Krylov space starts to resolve the eigenvectors corresponding to both minimum and maximum eigenvalues. Therefore, we expect CG to converge on those components of the solution faster than the components of the interior eigenvalues.

As it turns out, CG actually converges superlinearly: the speed at which it converges increases as you iterate more. This can be understood in terms of the outer eigenvalue components converging early on, so that the problem ends up on a smaller space, where the condition number of A has effectively been reduced. Instead of λ_1/λ_n, once the components corresponding to λ_1 and λ_n have been solved the effective condition number ends up as λ_2/λ_{n-1}. This also relates to CG giving the exact answer in n iterations using exact arithmetic—in fact, if there are only k distinct eigenvalues with components present in the residual, CG will converge in k iterations.

Also as a quick note—while for reasons of efficiency (reusing a matrix-vector multiply) we chose to use the update formula for the residual, you might be tempted to think that this runs the danger of accumulating rounding errors, and therefore it might be better to recompute the residual as $r_k = b - Ax_k$. It is true that rounding errors do accumulate, so that the updated residual steadily drifts from the true residual. However, remarkably enough, this gives **better** convergence than if you recompute the residual! The underlying reason ends up being that the update formula is better at keeping the residuals orthogonal (as they should be in exact arithmetic), and this ends up being more important for fast convergence.

Finally it should be made clear that while CG is picking guesses that are optimal in the energy norm, this isn't a norm we can actually measure without knowing the exact solution already. In practice people determine when to stop based on the norm of the residual, however the relative 2-norm of the residual $\|r\|_2/\|b\|_2$ can be different from the relative energy norm of the error by a factor as big as $\sqrt{\kappa(A)}$. In practice it's typical to see that in some iterations the 2-norm of the residual increases a lot, before eventually dropping again.

3 Preconditioned Conjugate Gradient

While CG is an order of magnitude faster than Steepest Descent, Gauss-Seidel, etc. and is guaranteed to work (in the energy norm at least), 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 $AMy = b$, where $x = My$, for any matrix or linear operator M. Sweeping aside the issue of whether AM is symmetric or not for now, if the condition number of AM 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. It is essentially equivalent to running regular Conjugate Gradient on the modified symmetric positive definite system $M^{1/2}AM^{1/2}y = M^{1/2}b$, and setting $x = M^{1/2}y$, but managed to avoid ever explicitly needing $M^{1/2}$ or for that matter any mention of y. Rather than use the residuals to construct search directions, it uses the preconditioned residuals $z = Mr$:

- Start with $x_0 = 0$, $r_0 = b$, $z = Mr_0$, $p_1 = z_0$.
- Compute $\rho_0 = r_0^T z$ — if zero already, return.
- For $k = 1, 2, \ldots$
  - Multiply $q = Ap_k$.
  - Compute $\alpha = \rho_{k-1}/(p_k^T q)$.
  - Update $x_k = x_{k-1} + \alpha p_k$ and $r_k = r_{k-1} - \alpha q$.

- If $\|r_k\|$ is small enough return (converged).
- Precondition $z_k = Mr_k$.
- Compute $\rho_k = r_k^T z_k$.
- Compute $\beta = \rho_k / \rho_{k-1}$.
- Update $p_{k+1} = z_k + \beta p_k$.

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. The simplest preconditioner in use is just to take the reciprocals of the diagonal part of A, i.e. $M = D^{-1}$ where the matrix $D = \text{diag}(A)$ is the same one used in Jacobi iteration. Similarly Gauss-Seidel and SOR can be adapted to be preconditioners, though they first need to be "symmetrized", e.g. $M = (U + D)^{-1}(L + D)^{-1}$. Of course in this case we don't actually compute and store this as a matrix, but rather use triangular solves to apply it to the residual. Block versions of these algorithms are particularly effective.

Once the basic Krylov solvers like PCG were established, most of the research in iterative solvers turned to finding effective preconditioners. One particularly popular class are the Incomplete LU (or Incomplete Cholesky) factorizations. Here we start computing the LU or Cholesky factorization of A, but throw out nonzeroes along the way to make sure that the factors stay sparse (and thus cheap). For example, nonzeros outside of the sparsity pattern of A might be discarded, or nonzeroes smaller than some threshhold magnitude, or all but the m largest in each row or column. Ultimately this produces an inexact factorization, but often it is close enough to give an excellent preconditioner.