nag®

CSC / NAG Autumn School on

Core Algorithms in High-Performance Scientific Computing

Maths IV

Dwight Barkley

Linear Equations II

Maths IV: Iterative Methods for Linear Systems

4.1 Overview of Iterative Methods

The most significant advances in the field of numerical linear algebra in the past several years have come in the area of iterative methods. These have resulted in changes to the way large-scale linear equations and eigenvalue problems are solved. This lecture focuses on linear systems, but much of the general discussion in this overview applies equally to the eigenvalue problem.

The essence of iterative methods is to find the solution to $Ax = b$ by starting from an initial guess for x, call this $x^{(0)}$, and iterating some procedure that produces a sequence of improved approximations

$$x^{(0)}, x^{(1)}, x^{(2)}, \ldots, x^{(k)}, \cdots \rightarrow x$$

The common way to quantify the accuracy of the approximate solution $x^{(k)}$ is in terms of the **residual**

$$r^{(k)} \equiv b - A\,x^{(k)}$$

The residual is a vector measuring the failure of an approximate solution $x^{(k)}$ to solve the linear system. **It is not a direct measure the accuracy of** $x^{(k)}$. It should not be confused with error in the solution: $x^{(k)} - x$.

The norm of the residual, $||r^{(k)}||$, which is sometimes itself referred to as the residual in practice, provides a readily computable scalar measure of how well the system is satisfied by approximation $x^{(k)}$.

Thus, we have the three sequences of interest

$$x^{(k)} \rightarrow x \qquad r^{(k)} \rightarrow 0 \qquad ||r^{(k)}|| \rightarrow 0$$

One should not necessarily interpret the arrows here as an indication of $k \rightarrow \infty$. In fact, many methods are guaranteed to converge (in exact arithmetic), for $k = n$. In practice, even this many iterates is far beyond anything one would be interested in.

Iterative vs Direct

The essential distinction between iterative and direct methods is as follows.

With a direct method one supplies input, e.g. A and b say for solving linear equations, and at the end of the $O(n^3)$ operations the solution x is produced. The residual corresponding to the returned solution will

be $O(\epsilon_{machine})$. Prior to termination, nothing is known about the answer. The number of operations can be determined ahead of time and is independent of the particular numerical values in A and b. While we are here considering linear equations, direct solution of the eigenvalue problem also requires $O(n^3)$ work and the same plot applies in that case.

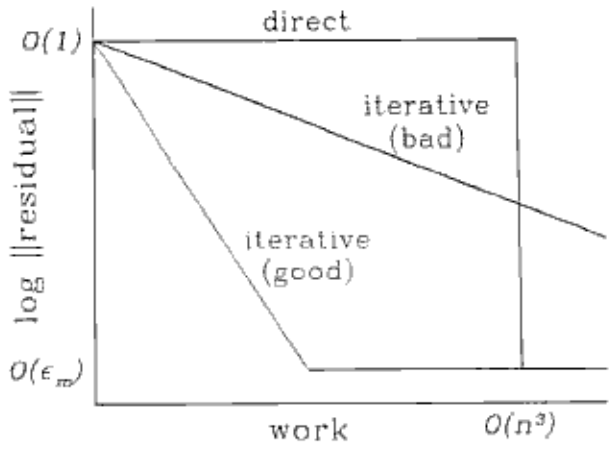

Figure 4.1: Illustration comparing direct and iterative methods. Residual is plotted as function of work. For an iterative method this axis is equivalent to the number of iterations.

By contrast iterative methods reduce the residual through the sequence of iterations such that approximate solutions are available at any point in the iteration process. Importantly, the number of iterations depends very much on the particular matrix A and this means that the work needed to arrive at an acceptable solution depends on A and generally cannot be known in advance.

A is a black box

All modern iterative methods for linear systems, as well as for eigenvalues, are based on repeated "matrix-vector multiplication" by the "matrix" A. We put "matrix" and "matrix-vector multiplication" in quotes here for a reason that is at the heart of the matter, as we explain.

First it is important to understand that iterative methods are appropriate for problems in which given a vector x, the result Ax can be obtained in fewer than the $O(n^2)$ flops needed for a full (dense) matrix-vector multiplication. The standard example of such a case is a **sparse matrix** in which a large proportion of the entries are zero. Assuming one can exploit the sparsity, the computational work of computing Ax can be reduced, in many cases significantly, from $O(n^2)$. The lower bound for a non-singular matrix is n flops.

However, it would be a mistake to view the reduction from $O(n^2)$ operations as simply the result of working with a matrix A with many zeros. We already have seen that if A is highly structured, then one can obtain Ax at lower than $O(n^2)$ cost, even though if one were to write down the matrix A none of the entries would be zero. A simple example is provided by the Fourier transform. **In fact, in numerous applications, the object that we think of as A does not even exist except as a series of calculations inside one or more subroutines.** This is one of the beauties of iterative methods, and one of the ways in which they excel over direct methods – one need never actually construct the linear system whose solutions are sought.

You should think of A as a black box, or as a subroutine, that takes input x, acts linearly on it, and produces output which we call Ax. We say "A acts on x". Some methods also require the "adjoint action": A^*x.

The cost of iterative methods

The cost of computing a solution iteratively is

$$C(n) = (\text{work per iteration}) \times (\text{number of iterations})$$

The cost of each iteration is dominated by the cost of one, or at most a few, actions of A, plus the cost of any preconditioners. (Preconditioners are discussed below and may be significant factor in the cost per iteration.) Other costs incurred each iteration are generally small in comparison.

The number of iterations required to find a solution to desired accuracy generally cannot be known in advance. Theoretic bounds, such as at most n iterations, are always too large to be of use. The ideal case, which does occur in practice with well conditioned problems, is that the number of iterations is somewhere in the range $O(10)$ to $O(100)$, independently of the size n. Often you will not be so lucky. It is worth noting that in the worse case, it takes $O(n)$ iterations to converge and with each iteration taking $O(n^2)$ work, given a cost of $O(n^3)$, which is that of a direct method.

Finally, while CPU time is normally the most important aspect of costing a method, memory requirements should not be completely overlooked. As a rule, the memory requirements of iterative methods are not large in comparison with direct methods because a lot of matrix elements do not need storage. In the worst case one needs to store one new n-vector each iteration (GMRES is such an example). For some applications this can mean significant memory requirements.

Preconditioning

A preconditioner is a matrix M, or more accurately another black box, which is a lot like A, but which is easier to invert than A. If we want to solve

$$Ax = b$$

we first formally pre-multiply both side of the equation on the left by M^{-1} to obtain

$$M^{-1}Ax = M^{-1}b$$

M is useful as a preconditioner if the new system is cheaper to solve iteratively than the original system, taking into account the added work. As you are used to by now, writing M^{-1} in the above does not mean we form a matrix M^{-1} and multiply by it. For example, let $c = M^{-1}b$ be the right-hand-side of the preconditioned system. In practice c is probably found by solving $Mc = b$.

Extreme cases are often used to illustrate the point. Let $M = I$. Then M^{-1} is trivial to compute and use, but it gives zero improvement. Let $M = A$, then M^{-1} is as difficult to compute as the original problem, but once applied, the iterative method can be solved in one iteration. In practice one uses M between these extremes.

A vast amount of research, and practical trial and error, has been devoted to preconditioners for iterative solutions. **Preconditioning can be crucial**. This cannot be emphasized enough.

Discussion

Here we note some practical issues arising with iterative methods and make some further comparisons with direct methods. Caveat. The following is based on my experiences with solving large linear algebra problems, some of the statement may not be precisely true. "Mileage may vary" as they say.

- **Cost estimation**

  While the cost-per-iteration for a particular problem can generally be known in advance, (because the black box which computes Ax is understood), the number of iteration often cannot be known and may vary considerably with parameters of a particular problem.

  What happens when you submit a job with a fixed CPU limit?

- **Stopping tests**

  Iterative methods require stopping test(s). While we shall not consider these in the methods presented later in the lecture, in practice stopping tests must be implemented. The ideal scenario is that the modulus of the residual decreases monotonically and one stops when $||r^{(k)}||$ falls below some desired tolerance. In practice things are not always so simple. The residual may behave as in Fig. 4.2.

- **Possibility of Failure**

  In many cases iterative schemes are not guaranteed to converge, even in infinite precision. Even if convergence is guaranteed in infinite precision, in practice for a particular problem the convergence may be too slow or not obtainable. The residual may reach a minimum and then increase again as in Fig. 4.2

  What does do in the case of failure ? Does one have a backup plan ?

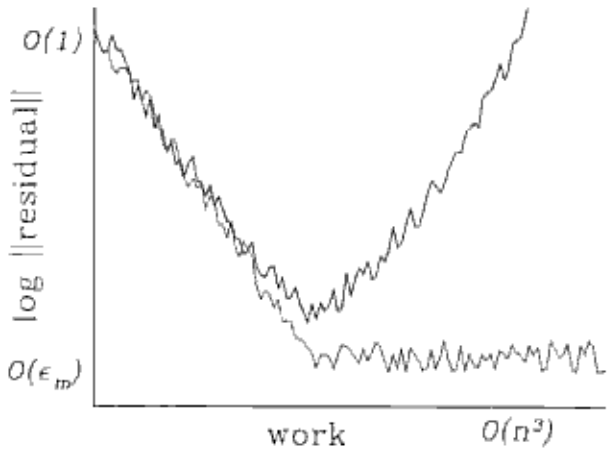

Figure 4.2: Various scenarios for the residual in a real iterative calculation.

- **Software**

  While packages exists for iterative methods, there is no universally accepted package/interface such as BLAS and LAPACK, in part because the field is still advancing. One advantage of using packaged software is that there can be a lot of trial and error in optimizing the performance for a particular problem. It is nice to be able to easily test different methods.

  There is no one iterative method that works optimally for all problems.

4.2 Classic Methods

We briefly consider three classic iterative methods. These are not methods based on the action of A. They are all what are called **linear methods** since the relationship between $x^{(k+1)}$ and $x^{(k)}$ is linear. Consider a splitting the matrix A into a sum of matrices:

$$A = L + D + U$$

where D, L, and U are, respectively, the diagonal of A, the strictly lower triangular part of A, and the strictly upper triangular part of A. This splitting is a *sum* and the L and U here are not the L and U matrices from an LU decomposition. With this splitting, our linear system becomes

$$Lx + Dx + Ux = b$$

The iterative schemes are obtained by replacing some occurrences of x in this expression with $x^{(k-1)}$ and some with $x^{(k)}$, thereby obtaining a relationship between $x^{(k+1)}$ and $x^{(k)}$. If the resulting scheme converges, $x^{(k)} \to x$, then x is the solution to the linear system, since upon convergence $x^{(k-1)} = x^{(k)} = x$ and so $Ax = b$ is satisfied.

Jacobi

The Jacobi method is obtained by using:

$$Lx^{(k)} + Dx^{(k-1)} + Ux^{(k)} = b$$

Assuming all elements of D are non-zero this gives:

$$x^{(k+1)} = D^{-1}\left(b - (L+U)x^{(k)}\right)$$

Requiring only D, the diagonal part of A, to be explicitly inverted and this is trivial: D^{-1} is the diagonal matrix with entries $1/d_{jj}$.

Convergence is dictated by the eigenvalues of the iteration matrix: $-D^{-1}(L+U)$. Converges is typically slow and this method is primarily useful when A is diagonally dominant, meaning that $|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|$.

Gauss-Seidel

The Gauss-Seidel method is given by the following iteration scheme:

$$Lx^{(k+1)} + Dx^{(k+1)} + Ux^{(k)} = b$$

or

$$x^{(k+1)} = (L+D)^{-1}\left(b - Ux^{(k)}\right) \tag{4.1}$$

Gauss-Seidel converges faster than the Jacobi method, but typically not by much. Intuitively one would expect Gauss-Seidel to be an improvement over Jacobi method because both L and D appear on the left-hand-side of (4.1).

The matrix inversion $(L+D)^{-1}$ appearing in Gauss-Seidel iteration could be accomplished by a forward substitution since $(L+D)$ is lower triangular. However, the implementing is actually easier. Write (4.1) as

$$x^{(k+1)} = D^{-1}\left(b - Lx^{(k+1)} - Ux^{(k)}\right) \tag{4.2}$$

Even though $x^{(k+1)}$ appears on the right-hand-side of this equation, if one evaluates the components $x_i^{(k+1)}$ in order $i = 1, 2, \ldots, n$ (from top to bottom), one finds that each time a component $x_i^{(k+1)}$ is needed on the right-hand-side, it has already been obtained. Moreover once $x_i^{(k-1)}$ is computed, $x_i^{(k)}$ is no longer needed. Difference between the Gauss-Seidel and Jacobi methods is that Gauss-Seidel iterations are done *in place* with updated values used immediately in each iteration. Again no actual matrix inversions are needed except for the trivial diagonal matrix D. Note, however, Gauss-Seidel updates cannot be performed in parallel, while Jacobi updates can be.

Successive Over-Relaxation (SOR)

The ideal behind SOR is to improve the convergence of Gauss-Seidel by over compensating a little bit. Specifically, an SOR iteration is given by

$$x^{(k+1)} = \omega x_{GS}^{(k+1)} + (1-\omega)x^{(k)}$$

where $x_{GS}^{(k+1)}$ is what Gauss-Seidel iteration would have given for $x^{(k+1)}$ and ω **is called the over-relaxation parameter**. Gauss-Seidel corresponds to $\omega = 1$.

In terms of the matrices, SOR can be written

$$x^{(k+1)} = (\omega L + D)^{-1}\left(\omega b - \omega U x^{(k)} + (1-\omega)Dx^{(k)}\right)$$

The useful range of ω is $0 < \omega < 2$. There is an optimal ω, which depends on the particular matrix A, for which over-relaxation substantially reduced the number of iterations needed to reach a certain residual. This optimal ω is usually found heuristically or through numerical experiments.

These classic methods each give rise to a preconditioner:

$$\begin{aligned} M_J &= D \\ M_{GS} &= L + D \\ M_{SOR} &= \frac{1}{\omega}(\omega L + D) \end{aligned}$$

and these may be their greatest use.

4.3 Conjugate Gradient

There is both a general and a specific aspect to the following discussion. The general aspect is that the concepts introduced (minimization, line searches, gradients, and conjugate vectors), are very general and applicable broadly to optimization problems. The specific aspect is that we shall focus on the cleanest case of conjugate gradient iteration in which one solves a linear systems where A is a symmetric or Hermitian positive-definite matrix.

Throughout this section and the next we will use the following notation, which is also common in optimization. We denote the solution to the linear system by x_* so that

$$Ax_* = b.$$

This frees up x to be general vector and not specifically the solution to the linear system. For simplicity we shall treat A as real, but there is nothing in what follows that does not apply equally to the complex case.

Minimization problem

A simple idea for solving a linear system system of equations is to convert the problem to a minimization problem by defining the real-valued function

$$f(x) = \frac{1}{2}||Ax - b||^2 \tag{4.3}$$

The function has a minimum at the desired solution x_* since $f(x) > f(x_*) = 0$ for $x \neq x_*$. In optimization, x_* is called a **minimizer** and f the **objective function**.

The problem with using equation (4.3) is that the square of the matrix has been introduced, and as a result, condition number of the problem has been squared: from $\kappa(A)$ to $\kappa^2(A)$. For a general matrix A, there is nothing to be done about this if one wants to generate an objective function from A and b. However, for the case where A is a symmetric or Hermitian positive-definite matrix, this squaring is not necessary. The following function has a unique minimum at the desired solution x^*, but involves only the first power of A

$$f(x) = \frac{1}{2}\langle x, Ax\rangle - \langle x, b\rangle \tag{4.4}$$

Minimizing this function will provide the desired solution x_*. Any iterative method for minimization, when applied to this function, becomes an iterative method for solving the linear equations.

Steepest descents

Steepest descents is one of the oldest methods in optimization. While it should not be used in general, it is instructive to consider. The idea is simple. At any given point x, the negative gradient of f, $-\nabla f(x)$, points in the steepest downhill direction from x. Hence, given the current approximation to the minimum, $x^{(k)}$, find the next, better, approximation $x^{(k-1)}$ by moving in this direction until f reaches a minimum along this direction.

Let us write this more generally as

$$x^{(k-1)} = x^{(k)} + \alpha^{(k)} d^{(k)}$$

where $d^{(k)}$ is a vector - **the search direction** and $\alpha^{(k)}$ is a scalar - **the step size** (how far we move in search direction). Ideally $\alpha^{(k)}$ is such that $x^{(k+1)}$ minimizes f along the line defined by $x^{(k)}$ and $d^{(k)}$. Finding the minimum along this line is called **a line search**.

In the case where f is given by equation (4.4) and search directions are steepest descents, everything can be worked exactly out to give:

$$\begin{aligned} d^{(k)} &= -\nabla f(x^{(k)}) = -(Ax^{(k)} - b) = r^{(k)} \\ \alpha^{(k)} &= \frac{\langle d^{(k)}, d^{(k)}\rangle}{\langle d^{(k)}, Ad^{(k)}\rangle} \end{aligned}$$

(Details can be filled in by the reader). Note that the search direction is just the current residual vector.

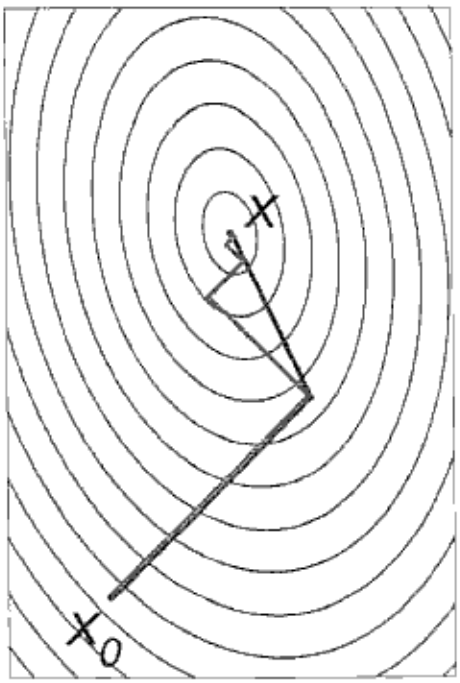

Figure 4.3: Illustration of steepest descents and conjugate gradient iteration for $n = 2$.

Thus we have the following:
Algorithm (Steepest Descents)

Choose $x^{(0)}$
For $k = 0, 1, 2, \cdots$
$d^{(k)} = b - Ax^{(k)}$ (compute search direction = residual)
$\alpha^{(k)} = \frac{\|d^{(k)}\|^2}{\langle d^{(k)}, Ad^{(k)} \rangle}$ (compute step size)
$x^{(k+1)} = x^{(k)} + \alpha^{(k)} d^{(k)}$ (update approximation)
Stopping tests
End

While steepest descents works, it suffers from slow convergence even for relatively simple problems. Many more iterations are required than should be necessary. For $n = 2$, only 2 iterations should be needed to find the solution, but steepest descents can take many many. This is illustrated in figure 4.3.

This is the problem that conjugate gradients solves.

Conjugate Gradient

Conjugate gradient iteration can be viewed as a simple improvement on steepest descents in which search directions are modified to be conjugate. To explain this we need to define conjugate vectors and show why using conjugate vectors fixes the problem with steepest descents. The final algorithm itself is quite simple.

Given symmetric positive definite matrix A, non-zero vectors p and q are conjugate (or A-conjugate) if

$$\langle p, Aq \rangle = 0$$

Sometimes you will see this written $\langle p, q \rangle_A = 0$.

Suppose for the moment that we have n mutually conjugate vectors p_i with

$$\langle p_i, Ap_j \rangle = 0 \quad \text{if } i \neq j$$

Since A is positive definite $\langle p_i, Ap_i \rangle \neq 0$. It is easy to show that these p_i are linearly independent and hence form a basis for $\mathbb{R}^n$ or $\mathbb{C}^n$. The desired minimum x_* can be written in as a linear combination of these vectors:

$$x_* = \sum_{j=1}^{n} \alpha_j p_j \tag{4.5}$$

First write the linear system using the α_j's and p_j's as follows:

$$\begin{aligned} Ax_* &= b \\ A\sum_{j=1}^{n} \alpha_j p_j &= b \\ \sum_{j=1}^{n} \alpha_j Ap_j &= b \end{aligned}$$

Because the p_i are mutually conjugate it is easy to obtain the coefficients α_i. Taking the inner product of both sides of this equation with p_i gives

$$\begin{aligned} \langle p_i, \sum_{j=1}^{n} \alpha_j Ap_j \rangle &= \langle p_i, b \rangle \\ \sum_{j=1}^{n} \alpha_j \langle p_i, Ap_j \rangle &= \langle p_i, b \rangle \\ \alpha_i \langle p_i, Ap_i \rangle &= \langle p_i, b \rangle \end{aligned}$$

Giving

$$\alpha_i = \frac{\langle p_i, b \rangle}{\langle p_i, Ap_i \rangle}$$

Thus, given the p_i we could find the minimum x_* by the following procedure. Start with $x^{(0)} = 0$. Perform line minimization along p_1 (that is move distance α_1). Then perform line along p_2, p_3 etc. After n iterations the minimum will be found exactly as $x_* = \sum_{j=1}^{n} \alpha_j p_j$. None of the back-and-forth behavior of steepest descents. Of course in practice the p_i are not known in advance but must be found as part of the iteration process. Also, one expects that the minimum can be found to within a desired accuracy in many fewer than n iterations. The n is simply an upper bound.

Before considering the actual algorithm let us see how the conjugate directions prevent the back-and-forth of steepest descents. One can view (4.5) as a change of basis from the standard basis in $\mathbb{R}^n$ to the basis given by the p_i. In the standard basis point x_* has coordinates $(x_1, x_2, \cdots, x_n)$. In the basis given by the conjugate vectors, the coordinates of this point are $(\alpha_1, \alpha_2, \cdots, \alpha_n)$. A small calculation shows that in the α-coordinates the contours of f are hyperspheres. Thus conjugate gradient iteration can be viewed as steepest descents in a coordinate system which the contours of f are hyperspheres. In these coordinates, the steepest descent direction points to the minimum a single steepest descents move, starting from anywhere, will find the minimum x_*.

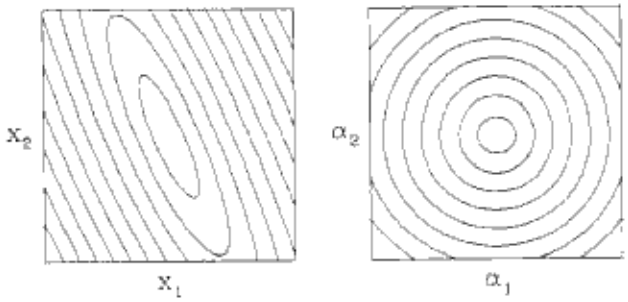

Figure 4.4: Contours of the objective function f in the original coordinate system and in the coordinate system defined by conjugate vectors.

We have not yet presented a method, since we started this discussion by assuming we had n mutually conjugate vectors. The discussion shows, both algebraically, and geometrically, why conjugate vectors are the right vectors to use in such a minimization problem.

We now turn to conjugate gradient proper. In essence, what we are going to do is modify our steepest decent method so that each time we generate a new search direction it is conjugate to all preceding search direction. Thus, as we generate new approximations $x^{(k)}$ to x_*, we at the same time generate a basis of conjugate vectors.

At step k we compute the current residual $r^{(k)}$, which is also the steepest descent direction from the current approximation $x^{(k)}$. Rather than using search direction $d^{(k)} = r^{(k)}$ we use

$$d^{(k)} = r^{(k)} + \beta^{(k)} d^{(k-1)} \qquad \text{where} \qquad \beta^{(k)} = -\frac{\langle r^{(k)}, Ad^{(k-1)} \rangle}{\langle d^{(k-1)}, Ad^{(k-1)} \rangle}$$

This choice of $\beta^{(k)}$ makes $d^{(k)}$ and $d^{(k-1)}$ conjugate. The only non-trivial part left is that **if one starts the whole process off correctly, then making $d^{(k)}$ conjugate to $d^{(k-1)}$ actually makes $d^{(k)}$ conjugate to all previous search directions.** This is not hard to prove, but will take too long here. Rather we just give the final algorithm in the form in which it is most commonly stated.

Algorithm (Conjugate Gradient)

$x^{(0)} = 0$, $r^{(0)} = b$, $d^{(0)} = r^{(0)}$
For $k = 0, 1, 2, \cdots$
$\alpha^{(k)} = \frac{\langle r^{(k-1)}, r^{(k-1)} \rangle}{\langle d^{(k-1)}, Ad^{(k-1)} \rangle}$ (compute step size)
$x^{(k)} = x^{(k-1)} + \alpha^{(k)} d^{(k-1)}$ (update approximation)
$r^{(k)} = r^{(k-1)} - \alpha^{(k)} Ad^{(k-1)}$ (compute residual)
$d^{(k)} = r^{(k)} + \beta^{(k)} d^{(k-1)}$, where $\beta^{(k)} = \frac{\langle r^{(k)}, r^{(k)} \rangle}{\langle r^{(k-1)}, r^{(k-1)} \rangle}$ (compute search direction)
Stopping tests
End

4.4 Krylov Methods - GMRES

We now consider the case where A is general matrix (a general linear black box). The fundamental tools for solving both linear system $Ax = b$, and the eigenvalue problem for A in the next lecture, is the Krylov sequence and associated Krylov subspace.

Starting from the vector b generate a sequence of k vectors by $k-1$ actions of A

$$\{b, Ab, A^2b, \cdots, A^{k-1}b\}$$

This is a **Krylov sequence**. The subspace of $\mathbb{R}^n$ spanned by these k vectors is called as a **Krylov subspace** $\mathcal{K}$:

$$\mathcal{K} = span\{b, Ab, A^2b, \cdots, A^{k-1}b\}$$

When needed, we shall indicate the dimension explicitly as $\mathcal{K}_k$.

We also need the the subspace of $\mathbb{R}^n$ obtained by acting with A on all points in $\mathcal{K}$. We denote $A\mathcal{K}$ and have

$$A\mathcal{K} = span\{Ab, A^2b, \cdots, A^{k-1}b, A^kb\}$$

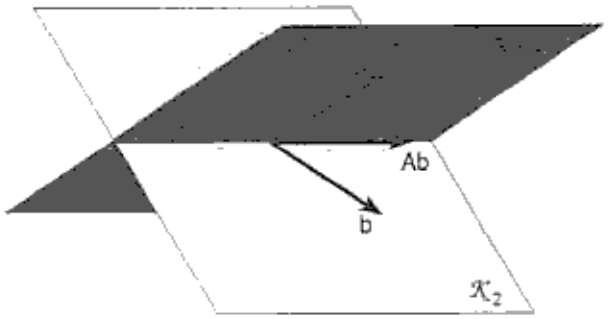

Figure 4.5: Illustration of Krylov subspaces.

GMRES, which stands for **generalized minimal residual** is extremely simple conceptually. We are going to take our k^{th} approximation of x_* to be the point in the Krylov subspace $\mathcal{K}_k$ that minimizes the residual norm of the linear system:

$$x^{(k)} = x \in \mathcal{K}_k \quad \text{such that } ||Ax - b|| \text{ is minimized}$$

The norm is necessarily the 2-norm here.

From allowed x in the k-dimensional Krylov subspace, this is the best approximation, in terms of residual, than one can make to the solution. The residual $r^{(k)}$ is orthogonal to $A\mathcal{K}_k$; if it weren't it could be made smaller.

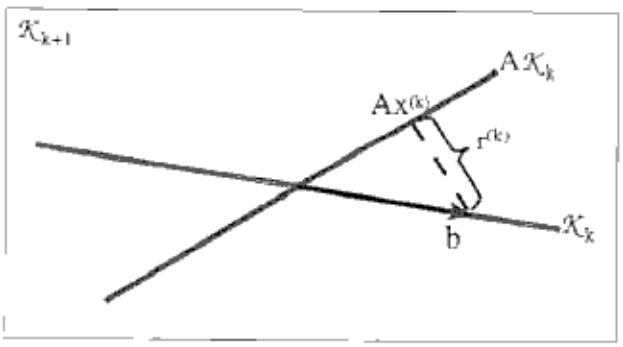

Figure 4.6: Illustration of Krylov subspaces.

The essence of the actual computation is easily described in a few words.

Generate orthogonal basis vectors for the Krylov subspaces. Using the orthogonal bases, exactly project the minimization of the residual norm to a least squares problem of size $(k+1) \times k$. Since k will not be

large, typically $O(100)$ say, this least squares problem can easily be solved by direct methods (discussed in the last lecture).

Intuitively we are able to *exactly* project the minimization problem involving an $n \times n$ matrix onto a small problem because of the way the Krylov spaces are embedded in one another. Essentially by definition $\mathcal{K}_k \subset \mathcal{K}_{k+1}$, but also $A\mathcal{K}_k \subset \mathcal{K}_{k+1}$.

$$\mathcal{K}_{k+1} = \underbrace{span\{b, \overbrace{Ab, A^2b, \cdots, A^{k-1}b, A^kb}^{A\mathcal{K}_k}\}}_{\mathcal{K}_k}$$

Hence the whole problem sits exactly inside a $k+1$ dimensional subspace of $\mathbb{R}^n$. All we need are orthogonal matrices to project onto these subspaces. In practice they are generated by Arnoldi's method discussed in the next lecture. Let us now just suppose that we have an orthogonal set of basis vectors for each $\mathcal{K}_k$:

$$q_1, q_2, q_3, \cdots, q_k \qquad \text{with} \qquad \langle q_i, q_j \rangle = \delta_{ij}$$

These are vectors each of length n. Arrange these as columns of a matrix Q_k:

$$Q_k = \left(q_1 \;\middle|\; q_2 \;\middle|\; \cdots \;\middle|\; q_k \right)$$

Similarly for $\mathcal{K}_{k+1}$ we have Q_{k+1} where the first k columns of Q_{k+1} are the same as Q_k.

$$Q_{k+1} = \left(q_1 \;\middle|\; q_2 \;\middle|\; \cdots \;\middle|\; q_k \;\middle|\; q_{k+1} \right)$$

It is important to understand that $k \ll n$, and hence these matrices are extremely tall and skinny.

$$\left[\begin{array}{c} \\ Q_k \\ \\ \end{array} \right]$$

Given the basis vectors, we can represent any vector in $\mathcal{K}_k$ with just k-coordinates: $(w_1, w_2, w_2, \ldots, w_k)$. Then $x \in \mathcal{K}_k$ is given by

$$x = w_1 q_1 + w_2 q_2 + \cdots w_k q_k$$

Denoting these k coordinates by a vector w we have:

$$x = Q_k w \qquad Q_k^* x = w$$

Again, the relative dimensions are

$$\left[\begin{array}{c} \\ x \\ \\ \end{array} \right] = \left[\begin{array}{c} \\ Q_k \\ \\ \end{array} \right] \left[w \right] \qquad \left[\qquad Q_k^* \qquad \right] \left[\begin{array}{c} \\ x \\ \\ \end{array} \right] = \left[w \right]$$

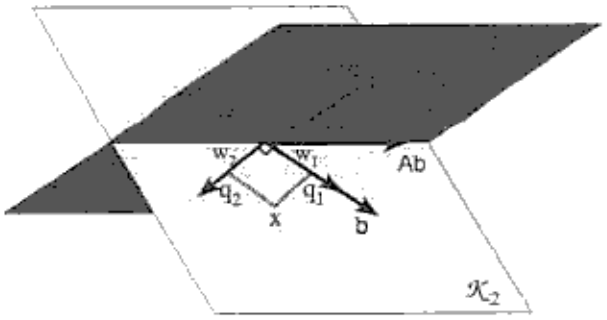

Figure 4.7: Orthogonal basis vectors in a Krylov subspaces.

We can now use Q_k and Q_{k-1} to express the minimization of the residual norm completely in term of the small system. Replacing x by $Q_k w$ gives

$$||Ax - b|| = ||AQ_k w - b||$$

Now, both Ax and b lie in $\mathcal{K}_{k+1}$. Hence if we left multiply by Q_{k-1}^* we will not change the norm of $Ax - b$ so

$$||AQ_k w - b|| = ||Q_{k+1}^*(AQ_k w - b)|| = ||Q_{k+1}^* AQ_k w - Q_{k+1}^* b||$$

$Q_{k-1}^* AQ_k$ is a $(k+1) \times k$ matrix denoted $\tilde{H}_k$. In pictures:

$$\begin{bmatrix} & Q_{k+1}^* & \end{bmatrix} \begin{bmatrix} & A & \end{bmatrix} \begin{bmatrix} Q_k \end{bmatrix} = \begin{bmatrix} \tilde{H}_k \end{bmatrix}$$

Finally, we note that b is parallel to q_1 so in small coordinates b is simply $(||b||, 0, 0, \ldots, 0)$. or $Q_{k-1}^* b = ||b||e_1$, where e_1 is the unit vector in $\mathbb{R}^k$.

Hence finally the quantity we need to minimize is

$$\|\tilde{H}_k w - ||b||e_1\|$$

over all $w \in \mathbb{R}^k$. Once the w is found which minimizes this expression, we form our k^{th} approximation to the solution $x^{(k)} = Q_k w$.

Note, the matrix $\tilde{H}_k$ is upper Hessenberg and in practice it is generated along side the the vectors $q_1, q_2, \ldots$ by Arnoldi iteration.

Other Methods

There is a whole host of iterative methods for linear systems that can be expressed in terms of Krylov sequences. The most important ones:

- CGN - Conjugate Gradient applied to the Normal Equations. For A not symmetric positive definite, form an equivalent problem which is symmetric positive definite

$$A^* A x = A^* b$$

  This gives precisely the minimization problem stated at the outset of conjugate gradient discussion.

- BCG - Biconjugate Gradient. For general matrices but requires A^* as well as A to generate two (biorthogonal) sequences.

- CGS - Conjugate Gradient Squared. For general matrices, improves converges rate of biconjugate gradient and does not require the adjoint A^*. Is considered erratic.

- Bi-CGSTab - stabilized CGS (van der Vorst, 1992). For general matrices, and does not require the adjoint A^*. Improves converges properties of CGS. Probably the method of choice in most cases.