

Notes for Ma221 Lecture 14, Apr 19, 2022

Krylov subspace methods for  $A^*x=b$  and  $A^*x = \lambda x$

Next we consider Krylov Subspace Methods, of which there are many, depending on the matrix structure. For spd matrices like our model problem, Conjugate Gradients (CG) is the most widely used, and the one we will discuss in most detail. We will also summarize the other methods with a "decision tree" that chooses the right algorithm depending on properties of your matrix (like symmetry, positive definiteness, etc), see Fig 6.8 in the text.

Unlike Jacobi, GS and SOR, a Krylov subspace method for  $Ax=b$  or  $A^*x = \lambda x$  need only assume that you have a "black-box" subroutine for multiplying  $A$  times a vector by  $x$  (or  $A^T$  times  $x$ , for some methods). So you cannot ask for the diagonal part, or subdiagonal part, etc. that you would need for methods like Jacobi.

This is particularly useful in two situations:

(1) If you are writing a library for solving  $A^*x=b$ , the user can simply input a pointer to a function that computes  $A^*z$  for any  $z$ , and your algorithm calls it without having to know any details about how  $A$  is represented.

(2)  $A$  may not actually be represented as a matrix with explicit entries, but as the result of evaluating some complicated function.

For example, suppose you have a system, like a car engine or airplane wing, governed by a complicated set of PDEs. You'd like to optimize some set  $y$  of  $n$  outputs of the PDE as a function of some set  $x$  of  $n$  inputs,  $y=f(x)$ , say the pressure on the wing as a function of its shape. For some optimization methods, this means that you need to solve linear systems with the  $n \times n$  coefficient matrix  $A = \text{del } f$ , the Jacobian of  $f$ .

$A$  is not written down, it is defined implicitly as the derivative of the outputs of the simulation with respect to its inputs, and varies depending on the inputs. The easiest way to access  $A$  is to note that  $A^*z = \text{del } f * z \sim (f(x+h*z) - f(x))/h$  when  $h$  is small enough, so multiplication by  $A$  requires running the simulation twice to get  $f(x+h*z)$  and  $f(x)$  (care must be taken in choosing  $h$  small enough but not too small, because otherwise the result is dominated by error).

There is no such simple way to get  $A^T * z$ , unfortunately.

There is actually software that takes an arbitrary collection of C or Fortran subroutines computing an arbitrary function  $y = f(x)$ , and produces new C or Fortran subroutines that compute  $\text{del } f(x)$ , by differentiating each line of code automatically; google "ADIFOR" or see [wiki.mcs.anl.gov/autodiff/](http://wiki.mcs.anl.gov/autodiff/)

Another more specialized and widely used example, for training neural nets, is TensorFlow.

On the other hand, if we assume that the matrix  $A$  is available explicitly, then it is possible to reorganize many Krylov subspace methods to significantly reduce their communication costs, which are dominated by moving the data needed to represent the sparse matrix  $A$ . For example, when  $A$  is too large to fit in cache (a typical situation), each step of a conventional sequential method does a multiplication  $A*x$ , which requires reading the matrix  $A$  from slow memory to cache. Thus the communication cost is proportional to the number of steps, call it  $k$ . However the reorganized Krylov subspace methods can take  $k$  steps and only read  $A$  from slow memory once, a potential speedup of a factor of  $k$ . See the link to the 2014 Acta Numerica survey article on the class webpage for more details. In practice  $k$  is limited by both the sparsity structure of  $A$  and issues of numerical stability; numerical stability is also discussed in more detail in Chap 7 of the text.

We begin by discussing how to extract information about  $A$  from a subroutine that does  $A*z$ . Given a starting vector  $y_1$  (say  $y_1 = b$  for solving  $A*x=b$ ), we can compute  $y_2 = A*y_1, \dots, y_{i+1} = A*y_i, \dots, y_n = A*y_{n-1} = A^{n-1}*y_1$ . Letting  $K = [y_1, y_2, \dots, y_n]$  we get

$$A*K = [A*y_1, \dots, A*y_n] = [y_2, y_3, \dots, y_n, A^n*y_1]$$

Assuming for a moment that  $K$  is nonsingular, write  $c = -K^{-1}*A^n*y_1$  to get

$$A*K = K*[e_2, e_3, \dots, e_n, -c] = K*C$$

where

$$C = K^{-1}*A*K = \begin{bmatrix} 0 & 0 & \dots & 0 & -c_1 \\ 1 & 0 & \dots & 0 & -c_2 \\ 0 & 1 & \dots & 0 & -c_3 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 & -c_n \end{bmatrix}$$

is upper Hessenberg, and a companion matrix, i.e. its characteristic polynomial is simply  $p(x) = x^n + \sum_{i=1}^n c_i*x^{i-1}$ . So just by matrix-vector multiplication we have reduced  $A$  to a simple form, and could imagine using  $C$  to solve  $A*x=b$  or find eigenvalues.

But this would be a bad idea for two reasons:

- (1)  $K$  is likely to be dense (if  $y_1$  is dense), and solving with  $K$  is probably harder than solving with  $A$
- (2)  $K$  is likely to be very ill-conditioned, since it is basically running the power method, so the vectors  $y_i$  are converging to the eigenvector of the largest eigenvalue in absolute value.

Krylov subspace methods address these two drawbacks, implicitly or explicitly, as follows:

We will not compute  $K$  but rather an orthogonal  $Q$  such that the leading  $k$  columns of  $K$  and of  $Q$  span the same space, called a Krylov subspace:

$$\text{span}\{y_1, y_2, \dots, y_k\} = \text{span}\{y_1, A^*y_1, \dots, A^{(k-1)}y_1\} \\ = \text{span}\{K\}_k(A, y_1)$$

The relationship between  $K$  and  $Q$  is simply the QR decomposition:  $K = Q^*R$ . Furthermore, we won't compute all  $n$  columns of  $K$  and  $Q$ , but just the first  $k \ll n$  columns, and settle for the "best" approximate solution  $x$  of  $A^*x=b$  or  $A^*x = \lambda x$  that we can find in the Krylov subspace they span (the definition of "best" depends on the algorithm).

(We pause the recorded lecture here.)

To proceed, substitute  $K = Q^*R$  into  $C = K^{-1}A^*K$  to get

$$C = R^{-1}Q^T A^* Q R$$

or

$$(*) \quad R^* C \text{inv}(R) = Q^T A^* Q = H$$

since  $R$  and so  $R^{-1}$  are triangular, and  $C$  is upper Hessenberg, then  $H$  is also upper Hessenberg (Question 6.11 in the text).

If  $A$  is symmetric, then so is  $H$ , and so  $H$  is tridiagonal.

To see how to compute the columns of  $Q$  and  $H$  one at a time, write

$Q = [q_1, q_2, \dots, q_n]$ , rewrite  $(*)$  as

$$A^*Q = Q^*H$$

and equate the  $j$ -th column on both sides to get

$$A^*q_j = \sum_{i=1}^{j+1} q_i^* H(i,j)$$

Since the  $q_j$  are orthonormal, multiply both sides of the last equality by  $q_m^T$  to get

$$q_m^T A^* q_j = \sum_{i=1}^{j+1} H(i,j) q_m^T q_i = H(m,j) \text{ for } 1 \leq m \leq j$$

and so

$$H(j+1,j) q_{j+1} = A^*q_j - \sum_{i=1}^j q_i^* H(i,j)$$

This yields the following algorithm:

Arnoldi algorithm for (partial) reduction of  $A$  to Hessenberg form:

$$q_1 = y_1 / \|y_1\|_2$$

for  $j = 1$  to  $k$

$$z = A^*q_j$$

for  $i = 1$  to  $j$  ... run Modified Gram-Schmidt (MGS) on  $z$

$$H(i,j) = q_i^T z$$

$$z = z - H(i,j)q_i$$

end for

$$H(j+1,j) = \|z\|_2$$

$$q_{j+1} = z / H(j+1,j)$$

end for

The  $q_j$  vectors are called Arnoldi vectors, and the cost is  $k$  multiplications by  $A$ , plus  $O(k^2 \cdot n)$  flops for MGS. If we stop the algorithm here, at  $k < n$ , what have we learned about  $A$ ? Write  $Q = [Q_k, Q_u]$  where  $Q_k = [q_1, \dots, q_k]$  is computed as well as the first column  $q_{k+1}$  of  $Q_u$ ; the other columns of  $Q_u$  are unknown. Then

$$H = Q^T A Q = [Q_k, Q_u]^T A [Q_k, Q_u] \\ = [Q_k^T A Q_k \quad Q_k^T A Q_u] = [H_k \quad H_{uk}] \\ [Q_u^T A Q_k \quad Q_u^T A Q_u] \quad [H_{ku} \quad H_u]$$

Since  $H$  is upper Hessenberg, so are  $H_k$  and  $H_u$ , and  $H_{ku}$  has a single nonzero entry in its upper right corner, namely  $H(k+1, k)$ . So  $H_{uk}$  and  $H_u$  are unknown, and  $H_k$  and  $H_{ku}$  are known.

When  $A$  is symmetric, so that  $H$  is symmetric and tridiagonal, we may write

$$H = T = \begin{bmatrix} \alpha_1 & \beta_1 & & 0 & & \\ & \beta_1 & \alpha_2 & \beta_2 & & \\ & & \beta_2 & \alpha_3 & \beta_3 & \\ & & & \dots & & \\ & & & & \dots & 0 & \beta_{n-1} & \alpha_n \end{bmatrix}$$

Equating column  $j$  on both sides of  $A^*Q = Q^*T$  yields

$$A^*q_j = \beta_{j-1} q_{j-1} + \alpha_j q_j + \beta_j q_{j+1}$$

Multiplying both sides by  $q_j^T$  yields

$$q_j^T A^* q_j = \alpha_j$$

leading to the following simpler version of the Arnoldi algorithm, called Lanczos:

Lanczos algorithm for (partial) reduction of  $A=A^T$  to tridiagonal form

$$q_1 = y_1 / \|y_1\|_2, \beta_0 = 0, q_0 = 0$$

for  $j = 1$  to  $k$

$$z = A^*q_j$$

$$\alpha_j = q_j^T z$$

$$z = z - \alpha_j q_j - \beta_{j-1} q_{j-1}$$

$$\beta_j = \|z\|_2$$

$$q_{j+1} = z / \beta_j$$

end for

Here is some notation to describe the Arnoldi and Lanczos algorithms.

As above, the space spanned by  $\{q_1, \dots, q_k\}$  is called a Krylov subspace:

$$\{\mathcal{K}\}_k(A, y_1) = \text{span}\{q_1, \dots, q_k\}$$

or  $\{\mathcal{K}\}_k$  for short.

We call  $H_k$  ( $T_k$  for Lanczos) the projection of  $A$  onto  $\{\mathcal{K}\}_k$ .

Our goal is to solve  $A^*x=b$  or  $A^*x = \lambda x$  by somehow finding the "best" solution in the Krylov subspace.

For the eigenvalue problem, the answer is simple: use the eigenvalues of  $H_k$  (or  $T_k$ ) as approximate eigenvalues of  $A$ . To see what the error is, suppose  $H_k y = \lambda y$  and write

$$H \begin{bmatrix} y \\ 0 \end{bmatrix} = \begin{bmatrix} H_k & H_{uk} \\ 0 & H_{ku} \ H_u \end{bmatrix} \begin{bmatrix} y \\ 0 \end{bmatrix} = \begin{bmatrix} H_k y \\ H_{ku} y \end{bmatrix} = \begin{bmatrix} \lambda y \\ H_{(k+1,k)} y_k e_1 \end{bmatrix}$$

and since  $AQ = QH$

$$A \begin{bmatrix} Q y \\ 0 \end{bmatrix} = \lambda \begin{bmatrix} Q y \\ 0 \end{bmatrix} + q_{(k+1)} H_{(k+1,k)} y_k$$

so if the product of  $H_{(k+1,k)} y_k$  is small, the eigenvalue/vector pair  $(\lambda, y) = (\lambda, Q[y; 0])$  has a small residual

$$\|A y - \lambda y\|_2 = |H_{(k+1,k)} y_k|$$

When  $A$  is symmetric, we know from Chapter 5, Thm 5.5, that this is in fact a bound on the error in  $\lambda$ .

To see how the eigenvalues of  $T_k$  approximate the eigenvalues of  $A$  as  $k$  grows, we consider the symmetric case (Lanczos), and plot the eigenvalue of  $T_k$  versus  $k$  (Figure 7.2: LanczosConvergence9.eps and LanczosConvergence29.eps). Chapter 7 discusses finding eigenvalues and eigenvectors in more detail. The monotonic convergence of the largest and smallest eigenvalues shown in these pictures is due to the Cauchy Interlace Thm (homework Q5.4, part 1).

This presentation has ignored floating point errors so far. In practice, it is common for the vectors  $q_j$  to lose orthogonality as the number of iterations grows, leading to more complicated behavior, and interesting algorithmic changes. See Chap 7 for more details.