2022-11-01

1 Algorithms

There are several flavors of symmetric eigenvalue solvers for which there is no equivalent (stable) nonsymmetric solver. We discuss four algorithmic ideas: the workhorse QR algorithm that we have already seen, the Jacobi iteration, divide-and-conquer, and bisection with inverse iteration.

The details of these algorithms are quite technical (particularly the divide-and-conquer method and bisection with inverse iteration). But even if you are not planning to focus in numerical linear algebra, you should know a little about the shape of these algorithms. Why? Three reasons:

- Intellectual fun!
- To make an informed choice of algorithms.
- To re-use building blocks.

2 Symmetric QR

2.1 The eigenvalues of symmetric A

Like the nonsymmetric QR iteration, the symmetric QR iteration involves an initial reduction, but to a *tridiagonal* form. This is really the same as the Hessenberg reduction step; but a symmetric Hessenberg matrix is tridiagonal, and we can use that. Similarly, when we perform bulge-chasing, the intermediate matrices remain symmetric, and so we never have a matrix that is more than a few elements away from tridiagonal. For the symmetric case, there is a little difference in how we choose shifts: Wilkinson shifts are fine since there are only real eigenvalues – no need for the Francis double-shift strategy. But otherwise, the main difference is that each step of the tridiagonal QR iteration maps between representations with $O(n)$ parameters in $O(n)$ time. Each eigenvalue converges in roughly a constant number of iterations, so the cost to compute all eigenvalues of a tridiagonal is $O(n^2)$. Compared to the $O(n^3)$ cost of reducing to tridiagonal form in the first place, the cost of solving for the eigenvalues of the tridiagonal is thus quit modest.

If we want all the eigenvalues of a sparse matrix, and only want the eigenvalues, the algorithm is basically the fastest option. But if we want eigenvectors as well, then the QR iteration is more expensive, costing an additional $O(n^3)$; other methods run faster.

2.2 QR iteration for singular values

Now consider the case of computing the singular values of a matrix A. We could compute the singular values directly from the eigenvalues of the Gram matrix $A^T A$ (or AA^T, or the Golub-Kahan matrix). But the backward error associated with tridiagonal reduction is proportional to the norm of the matrix $A^T A$ (or the square norm of A) and this can look quite big compared to the square of the smallest singular values. So rather than work with $A^T A$ *explicitly*, we prefer to manipulate A in order to run the same algorithm *implicitly*.

The first step of the QR iteration for the singular value problem is thus *bidiagonalization*; that is, we compute

$$A = \hat{U} B \hat{V}^T$$

where B is an upper bidiagonal matrix. Note that

$$A^T A = \hat{V} B^T B \hat{V}^T = \hat{V} T \hat{V}^T,$$

i.e. B is the Cholesky factor of the tridiagonal matrix A that we would obtain by tridiagonalization of the Gram matrix $A^T A$. But we can compute B directly by alternately applying transformations to A from the left and the right.

After the bidiagonal reduction, we want to do implicit QR steps. As a shift, we use the square root of the trailing corner element of the tridiagonal $B^T B$; in terms of B, this is just the norm of the last column:

$$\sigma = \sqrt{b_{n-1,n}^2 + b_{n,n}^2}.$$

With this shift in hand, we could apply the first step of shifted QR and complete the process implicitly via bulge chasing in the same way we did for the nonsymmetric case. In practice, there is an alternate algorithm (the *dqds* method) that enjoys extra stability benefits, allowing us to compute the singular values of B to high relative accuracy.[1]

[1] Of course, we usually lose high relative accuracy of the small singular values through

3 Jacobi iteration

A *Jacobi rotation* is a symmetric transformation that diagonalizes a 2 × 2 (sub)matrix:

$$J^T A J = \Lambda$$

In terms of scalars, this means solving

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} \alpha & \beta \\ \beta & \gamma \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix},$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$. A numerically stable method for carrying out this computation is described in Golub and Van Loan[2]; we will leave the details aside for the purposes of these notes.

Given an algorithm for computing Jacobi rotations, the idea of the Jacobi iteration is to incrementally apply Jacobi rotations to each 2 × 2 principle minors of A in turn. Each time we apply a Jacobi iteration to rows and columns k and l, we reduce the sum of squares off the main diagonal by a_{kl}^2. The iteration converges quadratically, and typically we can stop after 5–10 sweeps. Each sweep costs $O(n^3)$, and has cost comparable to the cost of a tridiagonalization step before running QR iteration. Hence, Jacobi iteration is rather slow. On the other hand, it tends to compute the small eigenvalues of A much more accurate than competing methods that start with a tridiagonalization step.

3.1 Divide and conquer

The symmetric QR iteration and the Jacobi iteration are methods that an appropriately motivated student could reasonably code[3] The divide and conquer method, on the other hand, is much more numerically subtle. Nonetheless, the ideas are quite interesting, and it is worth spending a moment describing the strategy at a high level.

Like the symmetric QR iteration, the divide-and-conquer method is preceded by a tridiagonalization process. After tridiagonalization, we think of

the initial reduction to bidiagonal — the backward error for that reduction is only small relative to the norm of A.

[2] Or on Wikipedia!

[3] I do not really recommend this, but it is not implausible.

the tridiagonal matrix as a block 2×2 matrix

$$T = \begin{bmatrix} T_{11} & \beta e_k e_1^T \\ \beta e_1 e_k^T & T_{22} \end{bmatrix} = \begin{bmatrix} \tilde{T}_{11} & 0 \\ 0 & \tilde{T}_{22} \end{bmatrix} + \beta \begin{bmatrix} e_k \\ e_1 \end{bmatrix} \begin{bmatrix} e_k \\ e_1 \end{bmatrix}^T.$$

where T_{11} and T_{22} are tridiagonal submatrices of the original tridiagonal, and $\tilde{T}_{11}$ and $\tilde{T}_{22}$ are these submatrices with β subtracted from a corner entry. We compute the eigendecompositions $Q_{11}^T \tilde{T}_{11} Q_{11} = D_1$ and similarly for $\tilde{T}_{22}$ by applying the divide-and-conquer method recursively. We then have

$$Q^T T Q = D + \beta u u^T$$

where Q is a block diagonal orthogonal matrix with diagonal blocks Q_{11} and Q_{22}, and u consists of the last row of Q_{11} and the first row of Q_{22} stacked atop each other.

Assuming we can perform this reduction, we now need to find solutions to

$$\det(D + \beta u u^T - \lambda I) = 0.$$

Without loss of generality, consider the case where u has no zero elements[4]; then none of the diagonal entries of D are eigenvalues of A, but we can write the eigenvalues of A as solutions to

$$f(\lambda) = \det(D + \beta u u^T - \lambda I) / \det(D - \lambda I) = 1 + \beta u^T (D - \lambda I)^{-1} u,$$

where we have used the identity (a good homework problem)

$$\det(I + XY^T) = \det(I + Y^T X).$$

The equation $f(\lambda) = 0$ is known as a *secular equation*, and it is a particularly nice type of rational function. The values d_i are poles of f, and there is one solution to $f(\lambda) = 0$ between each pair of poles, as well as one that is either smaller than $\min d_i$ or greater than $\max d_i$.

We can compute solutions to the secular equation very efficiently using a variant of Newton's method. Naively, this iteration would seem to require $O(n)$ time per step in order to evaluate f and its derivatives at any given point. However, the evaluation time can be reduced significantly using the *fast multipole method*, so that the overall time is close to $O(1)$ per step; as

[4] If there is a zero element in u, we have a converged eigenvalue on the diagonal, and can deflate it away

a consequence, finding all the solutions to one secular equation takes $O(n)$ time.

The difficulty in the divide-and-conquer algorithm lies primarily in obtaining accurate eigenvector estimates. The problem occurs when eigenvalues cluster together; in this case, one tends to compute eigenvector estimates which look good on their own, but which are not orthogonal to each other. Fixing this problem while retaining good performance was a technical tour de force, and we will not even attempt to do it full justice here. We suffice it to say that the divide-and-conquer algorithm is quite fast if we want all eigenvalues and eigenvectors, *particularly* if the eigenvectors are clustered.

4 Bisection with inverse iteration

Our final algorithm for the tridiagonal eigenvalue problem[5] is based on *Sylvester's inertia theorem*, which says that congruent matrices have the same inertia. In particular, that means that if we perform the symmetric factorization

$$T - \sigma I = LDL^T,$$

the number of positive, negative, and zero diagonal entries of D is the same as the number of eigenvalues of A that are respectively greater than, less than, and equal to σ. The *bisection algorithm* recursively partitions an initial interval containing all eigenvalues of T into smaller intervals that either

- Contain no eigenvalues of A;
- Contain exactly one eigenvalue of A; or
- Are smaller in length than some tolerance.

The center point of each interval containing exactly one eigenvalue is a reasonable estimate for that eigenvalue, good enough to guarantee convergence of a shift-invert iteration. And this is the bisection algorithm in a nutshell.

The main technical difficulty with the bisection algorithm lies not in the bisection step (which is remarkably well-behaved even though it involves an unpivoted factorization of an indefinite matrix), but in the computation of the eigenvectors. As with the divide-and-conquer scheme, bisection

[5] As with QR and divide-conquer, the bisection iteration is typically preceded by a tridiagonal reduction.

with inverse iteration tends to yield eigenvector estimates which individually have small residuals, but which may not be adequately orthogonal for vectors that correspond to eigenvalues in a cluster. One could use explicit re-orthogonalization, but the modern "Grail" code (the RRR routines in LAPACK nomenclature) manages stability in a much more subtle and clever way. This was the thesis work of Inderjit Dhillon before he turned his energies to machine learning and data mining. The SVD case was the prize-winning thesis work of Paul Willems. The fact that the algorithm merits multiple highly-regarded PhD theses should tell you something of the subtleties in getting it right.

The Grail code is so-called because it has optimal complexity for computing eigenvectors (given the eigenvalues); to obtain k eigenvectors requires only $O(kn)$ time. This is generally the fastest way to compute a specified subset of eigenvectors, and is often the fastest way to compute all eigenvectors.