18.335 Midterm Solutions, Spring 2015

Problem 1: (10+20 points)

(a) If we Taylor-expand around $x_{\min}$, we find $f(x_{\min}+\delta) = f(x_{\min}) + \delta^2 f''(x_{\min})/2 + O(\delta^3)$, where the $O(\delta)$ term vanishes at a minimum. Because $\tilde{f}$ is forwards-stable, $\tilde{f}(x_{\min}+\delta) = f(x_{\min}+\delta) + |f(x_{\min}+\delta)|O(\varepsilon) = f(x_{\min}) + |f(x_{\min})|O(\varepsilon) + O(\delta^2)$ (via Taylor expansion). This means that the roundoff-error $O(\varepsilon)$ term can make $\tilde{f}(x_{\min}+\delta) < f(x_{\min})$ for $\delta \in O(\sqrt{\varepsilon})$. Hence searching for the minimum will only locate $x_{\min}$ to within $O(\sqrt{\varepsilon})$, and it is *not* stable. (Note that the only relevant question is forwards stability, since there is no input to the exhaustive minimization procedure.)

(b) Here, we consider $f(x) = Ax$ and $f_i(x) = a_i^T x$:

(i) The problem with the argument is that there is in general a *different* $\tilde{x}$ for each function f_i, whereas for f to be backwards stable we need $\tilde{f}(x) = f(\tilde{x})$ with the *same* $\tilde{x}$ for all components.

(ii) Consider $A = (1, \pi)^T$, for which $f(x) = (x, \pi x)^T$. In floating-point arithmetic, $\tilde{f}(x)$ will always give an output with two floating-point components, i.e. two components with a rational ratio, and hence $\tilde{f}(x) \neq f(\tilde{x})$ for any $\tilde{x}$.

Problem 2: (10+10+10 points)

Here, we are comparing $\hat{Q}^* b$ with the first n components of the $(n+1)$-st column of $\breve{R}$. The two are equal in exact arithmetic for any QR algorithm applied to the augmented matrix $\breve{A} = (A, b)$. To compare them in floating-point arithmetic, for simplicity I will assume that dot products $q^* b$ are computed with the same algorithm as the components of matrix-vector products $Q^* b$. The main point of this problem is to recognize that the augmented-matrix approach *only makes a big difference for MGS* in practice.

(a) In CGS, the first n components of the last column of $\breve{R}$ will be $q_i^* b$. This is identical to what is computed by $\hat{Q}^* b$, so the two will be *exactly* the same even in floating-point arithmetic. (Of course, CGS is unstable, so the two methods will be equally *bad*.)

(b) In MGS, to get the first n components of the last column of $\breve{R}$, we first compute $r_{1,n+1} = q_1^* b$ exactly as in the first component of $\hat{Q}^* b$, so that component will be identical. However, subsequent components will be different (in floating-point arithmetic), because we first subtract the q_1 component from b before dotting with q_2, etcetera. This will be more accurate for the purpose of solving the least-squares problem because the floating-point loss-of-orthogonality means that $\hat{Q}^* b$ may not be close to $(\hat{Q}^* \hat{Q})^{-1} \hat{Q}^* b$ in the presence of roundoff error; in contrast, the backwards-stability of MGS will guarantee the backwards stability of $\hat{R}\hat{x} = \hat{Q}^* b$ as stated (but not proved) in Trefethen. (The proof, as I mentioned in class, proceeds by showing that MGS is exactly equivalent, even with roundoff error, to Householder on a modified input matrix, and Householder by construction—as proved in homework—provides a backwards-stable product of $\breve{A}$ by $\breve{Q}^*$. You are not expected to prove this here, however.)

(c) In Householder, we get the first n components of the last column of $\breve{R}$ by multiplying b by a sequence of Householder reflectors: $I - 2v_k v_k^*$. However, because of the way Q is stored implicitly for Householder (at least, for the algorithm as described in class and in Trefethen lecture 10), this is *exactly* how $\hat{Q}^* b$ is computed from the Householder reflectors, so we will get the same thing in floating-point arithmetic. (Note that there is a slight subtlety because $\hat{Q}$ is $m \times n$ while the product of the Householder reflections gives the $m \times m$ matrix Q. But we would just multiply $Q^* b$ and then take the first n components in order to get $\hat{Q}^* b$.)

[On the other hand, not all computing environments give you easy access to the implicit storage for Q. For example, the `qr(A)` function in Matlab or Julia returns Q as an ordinary dense matrix, making it somewhat preferable to use the augmented scheme if only because the explicit computation of Q is

unnecessarily expensive—it is still backwards stable as argued in Trefethen. However, the `qrfact(A)` function in Julia returns a representation of the Householder reflectors in the "compact WY" format (such that, when you multiply the resulting $\hat{Q}^* b$ in Julia, it effectively performs the reflections—this is a more cache-friendly version of the reflector algorithm in Trefethen). NumPy also provides partial support for this via the `qr(A, mode='raw')` interface. If you call LAPACK directly from C or Fortran, of course, you have little choice but to deal with these low-level details.]

Problem 3: (10+20+10 points)

(a) If $Ax_k = \lambda_k Bx_k$, then $x_k^* Ax_k = \lambda_k x_k^* Bx_k = (A^* x_k)^* x_k = (Ax_k)^* x_k = (\lambda_k Bx_k)^* x_k = \overline{\lambda_k} x_k^* Bx_k$ (where we have used the Hermiticity of A and B). Hence $(\lambda_k - \overline{\lambda_k}) x_k^* Bx_k = 0$, and since B is positive definite we have $x_k^* Bx_k > 0$, so $\lambda_k = \overline{\lambda_k}$ and λ_k is real. Similarly, if $i \neq j$ and $\lambda_i \neq \lambda_j$, we have $x_i^* Ax_j = \lambda_j x_i^* Bx_j = \lambda_i x_i^* Bx_j$, and we obtain $x_i^* Bx_j = 0$. (See below for an alternate proof: we can change basis to show that $B^{-1}A$ is *similar* to a Hermitian matrix. A third proof is to show that $B^{-1}A$ is "Hermitian" or "self-adjoint" under a modified inner product $\langle x, y \rangle_B = x^* By$, but we haven't generalized the concept of Hermiticity in this way in 18.335.)

(b) Naively, we can just apply MGS to $B^{-1}A$, except that we replace $x^* y$ dot products with $x^* By$ (this also means that $\|x\|_2$ is replaced by $\|x\|_B = \sqrt{x^* Bx}$). Hence we replace q_j with s_j (the j-th column of S). The only problem is that this will require the $\Theta(m^2)$ operation Bv_j in the innermost loop, executed $\Theta(m^2)$ times, so the whole algorithm will be $\Theta(m^4)$. Instead, we realize that $v_j = (B^{-1}A)_j = B^{-1}a_j$ (where a_j is the j-th column of a) and cancel this B^{-1} factor with the new B factor in the dot product. In particular, let $\check{v}_j = Bv_j$ and $\check{s}_j = Bs_j$. Then our MGS "SR" algorithm becomes, for $j = 1$ to n:

(i) $\check{v}_j = a_j$

(ii) For $i = 1$ to $j-1$, do: $r_{ij} = s_i^* \check{v}_j$, and $\check{v}_j \leftarrow \check{v}_j - r_{ij}\check{s}_i$.

(iii) Compute $v_j = B^{-1}\check{v}_j$ (the best way is via Cholesky factors of B; as usual there is no need to compute B^{-1} explicitly).

(iv) Compute $r_{jj} = \|v_j\|_B$, and $s_j = v_j / r_{jj}$.

Alternatively, we can compute $s_i^* Bv_j = (Bs_i)^* v_j = \check{s}_i^* v_j$, and work with v_j rather than $\check{v}_j$. Another, perhaps less obvious, alternative is:

(i) Compute the Cholesky factorization $B = LL^*$.

(ii) Compute the ordinary QR factorization (via ordinary MGS or Householder) $L^{-1}A = QR$.

(iii) Let $S = (L^*)^{-1}Q = L^{-*}Q$. Hence $L^{-*}L^{-1}A = (LL^*)^{-1}A = B^{-1}A = SR$, and S satisfies $S^* BS = Q^* L^{-1}(LL^*)L^{-*}Q = Q^* Q = I$.

The basic reason why this Cholesky approach works is it represents a *change of basis* to a coordinate system in which $B = I$. That is, for any vector x, let $x' = L^* x$, in which case $x'^* y' = xLL^* y = x^* By$. In the new basis, $B^{-1}A$ becomes $L^* B^{-1} AL^{-*} = L^{-1} AL^{-*}$ ... which is now Hermitian in the usal sense, so we get the results of part (a) for "free" and we can apply all our familiar linear-algebra algorithms for Hermitian matrices. If we form the QR factorization in this basis, i.e. $L^{-1}AL^{-*} = Q'R'$, and then change *back* to the original basis, we get $B^{-1}A = L^{-*}Q'R'L^* = SR$ where $S = L^{-*}Q$ (satisfying $S^* BS = I$ as above) and $R = R'L^*$ (note that R is upper triangular). [Instead of the Cholesky factorization, we could also write $B = B^{1/2}B^{1/2}$ and make the change of basis $x' = B^{1/2}x$; this gives analogous results, but matrix square roots are much more expensive to compute than Cholesky.]

(c) It should converge to $S \to X$ (the matrix of eigenvectors, in descending order of $|\lambda|$), up to some arbitrary scaling. If we normalized $X^* BX = I$ (as we are entitled to do, from above), then $S = X\Phi$ where Φ is a diagonal matrix of phase factors $e^{i\phi}$. The argument is essentially identical to the reasoning from class that we used for the ordinary QR algorithm (or, equivalently, for simultaneous power iteration).

The j-th column of $(B^{-1}A)^k$ is $(B^{-1}A)^k e_j = \sum_\ell \lambda_\ell^k c_\ell x_j$, where $e_j = \sum_\ell c_\ell x_j$, i.e. $c_\ell = x_j^* B e_j$ (assumed to be generically $\neq 0$). As $k \to \infty$, this is dominated by the λ_1^k term, so we get $s_1 \sim x_1$. To get s_2, however, we first project out the s_1 term, hence what remains is dominated by the λ^{2k} term and we get $s_2 \sim x_2$. Similarly for the remaining columns, giving $s_j \sim x_k$.

Note that it is crucial that we use the B inner product here in the SR factorization, not the ordinary QR factorization. The QR factorization would give the Schur factors of $B^{-1}A$ because $x_i^* x_j \neq 0$. For example, consider the second column $v_2 = (B^{-1}A)^k e_2 \approx \lambda_1^k c_1 x_1 + \lambda_2^k c_2 x_2$, with $q_1 = x_1/\|x_1\|_2$. When we perform e.g. Gram-Schmidt on this column to project out the q_1 component (for ordinary QR), we take $v_2 - q_1 q_1^* v_2 \approx \lambda_2^k c_2 (x_2 - q_1 q_1^* x_2)$, where the λ_1^k term exactly cancels. Notice that what remains is *not* proportional to x_2, but also has an x_1 component of magnitude $\sim x_1^* x_2$, which is not small. In general, the j-th column will have components of $x_1, \ldots, x_j$ with similar magnitudes, and hence we would get a Schur factorization $T = Q^* A Q$ from Q. (This was a question on an exam from a previous year.) Instead, by using the SR factorization, we take $v_2 - s_1 s_1^* B v_2$ etcetera and the cross terms vanish.

[As with the QR algorithm, this procedure would fail utterly in the presence of roundoff errors, because everything except for the λ_1^k components would be lost to roundoff as $k \to \infty$. But one could devise a QR-like teration to construct the same S implicitly without forming $(B^{-1}A)^k$. In practice, there are a variety of efficient algorithms for generalized eigenproblems, including the Hermitian case considered here, and LAPACK provides good implementations. The key thing is that one should preserve the Hermitian structure by keeping A and B around, rather than discarding it and working with $B^{-1}A$. In practice, one need not even compute B^{-1}, e.g. in Lanczos-like algorithms for $Ax = \lambda Bx$.]

18.335J Introduction to Numerical Methods

Spring 2019