18.335 Problem Set 3

Due Friday, 13 March 2015.

Problem 1: QR and orthogonal bases

(a) Prove that $A = QR$ and $B = RQ$ have the same eigenvalues, assuming A is a square matrix. Then do a little experiment: Construct a random 5×5 real-symmetric matrix in Julia via `X=rand(5,5); A = X' + X`. Use `Q,R = qr(A)` to compute the QR factorization of A, and then compute $B = RQ$. Then find the QR factorization $B = Q'R'$, and compute $R'Q'$...repeat this process until the matrix converges. From what it converges to, suggest a procedure to compute the eigenvalues and eigenvectors of a matrix (no need to prove that it converges in general—we will discuss this in class). (There is example code to perform this computation in the pset3 Julia notebook.)

(b) Trefethen, problem 10.4.

Problem 2: Matrix addition and the CPU

Use the pset3 Julia notebook, which provides two implementations of matrix addition $C = A + B$. Unlike matrix multiplication, matrix addition has *no possibility of temporal locality*, since each matrix element is used *exactly once*. However, ordering of the operations still matters because of *cache lines* and *spatial locality*.

(a) Run the benchmark in the notebook. Which loop ordering (matmul1 vs. matmul2) is faster? Use your results to deduce whether Julia stores matrices in *row-major* or *column-major* order (google these terms if you don't know what they mean).

(b) As written, the benchmark uses `matmul1!` and `matmul2!` routines that employ pre-allocated output, whereas the built-in `A+B` operation allocates a new output array each time you call it. Modify the benchmark to compare A+B to the provided functions `matmul1` and `matmul2` (no `!`) that similarly allocate an output array each time—does this explain the performance of `A+B` in the previous part?

(c) Suppose you wanted to compute `C=A+3B+4A.^2` (noting that `.^` is element-wise squaring, not matrix squaring A^2) in Julia. Based on your results above, how/why might you make this faster (if you needed to)? (Matters would be harder to improve in Matlab or Python, where loops written in the language are slow.)

Problem 3: Schur fine

In class, we will show that any square $m \times m$ matrix A can be factorized as $A = QTQ^*$ (the *Schur factorization*), where Q is unitary and T is an upper-triangular matrix (with the same eigenvalues as A, since the two matrices are similar).

(a) A is called "normal" if $AA^* = A^*A$. Show that this implies $TT^* = T^*T$. From this, show that T must be diagonal. Hence, any normal matrix (e.g. unitary or Hermitian matrices) must be unitarily diagonalizable. Hint: consider the diagonal entries of TT^* and T^*T, starting from the (1,1) entries and proceeding diagonally downwards by induction.

(b) Given the Schur factorization of an arbitary A (not necessarily normal), describe an algorithm to find the eigenvalues and eigenvectors of A, assuming for simplicity that all the eigenvalues are distinct. The flop count should be asymptotically $Km^3 + O(m^2)$; give the constant K.

Problem 4: Caches and backsubstitution

In this problem, you will consider the impact of caches (again in the ideal-cache model from class) on the problem of *backsubstitution*: solving $Rx = b$ for x, where R is an $m \times m$ upper-triangular matrix (such as might be obtained by Gaussian elimination). The simple algorithm you probably learned in previous linear-algebra classes (and reviewed in the book, lecture 17) is (processing the rows from bottom to top):

$x_m = b_m / r_{mm}$
`for` $j = m-1$ `down to` 1
    $x_j = (b_j - \sum_{k=j+1}^{m} r_{jk} x_k) / r_{jj}$

Suppose that X and B are $m \times n$ matrices, and we want to solve $RX = B$ for X—this is equivalent to solving $Rx = b$ for n different right-hand sides b (the n columns of B). One way to solve the $RX = B$ for X is to apply the standard backsubstitution algorithm, above, to each of the n columns in sequence.

(a) Give the asymptotic cache complexity $Q(m,n;Z)$ (in asymptotic Θ notation, ignoring constant factors) of this algorithm for solving $RX = B$.

(b) Suppose $m = n$. Propose an algorithm for solving $RX = B$ that achieves a better asymptotic cache complexity (by cache-aware/blocking or cache-oblivious algorithms, your choice). Can you gain the factor of $1/\sqrt{Z}$ savings that we showed is possible for square-matrix multiplication?