Homework 8

Michael Overton, Numerical Computing, Spring 2017

April 27, 2017

Please remember that you are supposed to do your own work. So, while it is fine to consult together with other students, as long as this is acknowledged, you are not supposed to be submitting multiple copies of collaborative work!

1. *Any three* of the five QR exercises in my notes: *Eigenvalues, Singular Values and QR*. Use the notation in the notes.

2. Download the file `WallOfWindows.jpg` on the course web page and read it into MATLAB using `A=imread(WallOfWindows.jpg,'jpg')`. Display it with `image(A)`. If you type `whos A` you will see that A is a $3456 \times 4608 \times 3$ three-dimensional "matrix" (or tensor) of unsigned 8-bit integers. Split these into red, green and blue components, each a 3456×4608 matrix of unsigned integers, via `A_red=A(:,:,1)`, etc, and convert these to double precision matrices using `double`. Then compute the economy-sized SVD of each component matrix separately. (If you like you can transpose the component matrix first so it has more rows than columns, as in the notes, but this is actually not necessary; we made this assumption only for convenience.) Then, for each component matrix, compute its nearest rank r approximation (see p. 6 of the notes *Eigenvalues, Singular Values and QR*), for $r = 10, 20, 30$ and 100. Recombine them into the $3456 \times 4608 \times 3$ format, and then display the resulting four pictures using `image`, labelling them appropriately.

   This scheme allows us to *store* low rank approximations to pictures by saving only the relevant left and right singular vectors and values: *state how much storage is required for rank r*. However, computing the SVD of such big matrices takes a long time. So, write another code to obtain *approximate* singular values and left and right singular vectors instead, using the block power method and exploiting the eigenvector-singular vector equivalences you found in HW5, either using those in question 5 (a) and (b) or the one in question 6. If you use the equivalences in question 5, avoid computing A^TA and AA^T explicitly since this is *also* expensive for such large matrices, which is possible since you can compute products such as $(A^TA)X$ via `A'*(A*X)`, where X has just r columns. If you use the equivalence in question 6, you can also do the relevant products efficiently. We don't need to compute the singular values and vectors to high accuracy, so you don't need to run many iterations of the block power method: ideally, we want to get pictures that are as good as the previous ones but that are computed faster than using `svd`. State whether you are able to accomplish this or not, for $r = 10, 20, 30$ and 100 separately, displaying the resulting images as before.

3. Consider the footnote at the bottom of p. 86 of A&G. Here $t_i = ih$ for $i = 1, 2, \ldots$ and a fixed small number h. So, writing $x = t_i$, we have $t_{i+1} = (i+1)h = x + h$. Hence the formula states that

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2).$$

Assuming that f is four times continuously differentiable, i.e., that the fourth derivative f'''' exists and is continuous, prove that this formula for f'' is correct as follows:

- write down a formula for $f(x+h)$ in terms of h, $f(x)$, $f'(x)$, $f''(x)$, $f'''(x)$ and $f''''(\xi_1)$ for some $\xi_1 \in [x, x+h]$, using the Taylor formula on p. 5 of A&G
- write down another similar formula for $f(x-h)$, with a remainder term $f''''(\xi_2)$ for some $\xi_2 \in [x-h, x]$
- combine these formulas by either adding them or subtracting them (one will work, one will not) and divide through by h^2 to obtain the desired result.

This derivation also tells you the constant in the $O(h^2)$ (this involves the two remainder terms). What is it? Although $\xi_1 \neq \xi_2$, by continuity of f'''', you can combine these two remainder terms to obtain just one term, using the intermediate value theorem on p. 10 of A&G.

4. Write a MATLAB function `findroot` to find a root of a function f of one variable using a hybrid of the bisection method and Newton's method. The inputs should be two numbers a and b with $a < b$ and $f(a)f(b) \leq 0$ and two "anonymous functions" `f` and `fderiv`, which are to be used to evaluate f and its derivative f' (see below for an example of anonymous function usage). Your `findroot` should return an output r, which is supposed to be a root of f with the highest accuracy you can find, with $a \leq r \leq b$, along with a vector `iterates` of *all* the iterates x_k where f was evaluated, (excluding the inputs a and b), **each of which is either a Newton iterate x^N or a bisection iterate x^B, with the last iterate equaling r,** as well as another vector `info` with components set to 0 when the corresponding iterate x_k was computed by bisection or 1 when it was computed by Newton.

- Start by evaluating $f(a)$ and $f(b)$, quitting with r set to a or b respectively if $f(a)$ or $f(b)$ is exactly zero, or quitting with r set to `nan` if $f(a)f(b) > 0$.
- **Your program should update the left end point a and the right end point b so that they always satisfy $a < b$ and $f(a)f(b) < 0$, and it should save the values $f(a)$ and $f(b)$ so f does not have to be evaluated at any point more than once. Also keep track of whether f was evaluated most recently at a or at b: initially, let's say this is a.**
- **At each iteration, start by computing the Newton update $x^N = x - f(x)/f'(x)$, where x is the latest point where f was**

evaluated (either a or b). Reject x^N if it lies outside the interval $[a,b]$ and replace it with the bisection point $x^B = (a+b)/2$ instead. Evaluate f at the new point x^N (if it lies in $[a,b]$) or x^B (otherwise) and, depending on the sign of f at the new point, update either a or b (not both!) to this new point, so that f still has opposite signs on a and b. Notice that in the computation of x^N, you don't have to check whether $f'(x)$ is zero since, if it is zero, the Newton step will be $\pm\infty$ which will be rejected.

- Eventually, the function should be taking only Newton steps, and you should see quadratic convergence if you print the iterates (more on this below).
- Quit when `findroot` is no longer making any significant change to the current x. You could just check to see if the new x, say x_{k+1}, is the same floating point number as x_k, but this might result in an infinite loop. So, it's better to check whether $|x_{k+1} - x_k|/|x_k| \leq \tau$, where τ is a little bigger than the machine epsilon: try 10^{-15} and make it larger if necessary. Since x_k could be zero, a better test might be $|x_{k+1} - x_k|/\max(1, |x_k|) \leq \tau$, but for the examples below it should not make much difference.
- Use a maximum iteration count to ensure your code does not have an infinite loop, but try to ensure that this is never reached.

An example of anonymous function usage is:

```
f = @(x)sin(x)
fderiv = @(x)cos(x)
```

Test `findroot` on:

(a) $f(x) = x^3 - 2$, $[a,b] = [1,2]$

(b) $f(x) = \exp(x) - 2$, $[a,b] = [-10,10]$

(c) $f(x) = 2\cosh(x/4) - x$, $[a,b] = [2,4]$ (see A&G, p. 40 and 48)

(d) $f(x) = 2\cosh(x/4) - x$, $[a,b] = [8,10]$

(e) $f(x) = \sin(x)$, $[a,b] = [2\pi + 10^{-4}, 3\pi + 10^{-4}]$ (get π from `pi`)

Make sure you use the correct formula for the derivative! If you get this wrong, `findroot` should still work because of the bisection steps but it will be slow. *To make sure you have the right formula*, check it (before calling `findroot`) against the finite difference formula $(f(x+h)-f(x))/h$ (see p. 72 of my book).

For each of these five cases:

- Print all the iterates, along with a B for iterates obtained by bisection or N for Newton. Eventually, all the iterates should be Newton iterates: if not, you probably have a bug in your code. With a pen, *underline* the digits that are correct compared to the *true answer* x^*. You can easily compute x^* via a formula for cases (a), (b) and (e),

but you will have to set x^* to your final computed answer r for (c) and (d) (these are given, but only to 9 digits, on A&G p. 48). Do you observe quadratic convergence, *with the number of correct digits approximately double the number of correct digits for the previous iterate towards the end*?

- Then, for $k = 1, 2, \ldots$, print the ratios

$$\frac{|x_{k+1} - x^*|}{|x_k - x^*|^2}$$

noting that this ratio may be meaningless for the final one or two iterates because of rounding errors. Discarding these, does this ratio converge, and if so, does it converge to

$$M = \frac{|f''(x^*)|}{2|f'(x^*)|}$$

as it should according to the quadratic convergence theory? Clearly answer this question for each of the five examples, also printing the value of M, for which you will need to use the formula for f''.

Finally, test `findroot` on other examples that you make up, and once you are satisfied it is working, *send an email to s70421g2@cs.nyu.edu* with `findroot.m` attached. The grader will test it some more on functions unknown to you.

5. Consider the five-point Laplacian matrix with the column-wise ordering on p.170-172. This matrix can be computed by:

```
%  G is an N+2 by N+2 full matrix with border entries set to zero
G = numgrid('S',N+2);  % 'S' means square with columnwise ordering
% A  is the n by n five-point Laplacian sparse matrix, where n=N^2
A = delsq(G);  % "del-squared" is a common term for the Laplacian
```

   (a) For some value of N that you choose, between 10 and 100, compute G and A and view the nonzeros of A with `spy(A,'bx')`, using the zoom tool if necessary, verifying that the structure is as claimed on p. 171-172. Also compute its lower triangular Cholesky factor via `L=chol(A,'lower')` and *superimpose its nonzeros on the same plot*, via `hold on, spy(L,'ro')`. Observe how much L is filled in compared to A. The number of nonzeros is shown at the bottom of the spy plot, but you can also compute this directly with `nnz`. What are:

   - the number of nonzeros in A, as an *approximate formula in terms of N*? This is easy, just looking at the definition or at the spy plot. You don't need to give an *exact* formula.
   - the number of nonzeros in L as an *approximate formula in terms of N*? You can estimate this from the spy plot. Again, you don't need to give an exact formula.

   (b) Reduce the fill-in in L by using `ichol` to compute the *incomplete Cholesky factorization* of A (see p. 188–189 of A&G). Experiment

with `opts.type='ict'`, `opts.droptol=.....`, `L=ichol(A,opts)`. For $N = 100$, generate a log-log plot with the drop tolerance on the x-axis (e.g., $10^{-6}, 10^{-5}, \ldots, 1$), showing both

- the number of nonzeros in the incomplete Cholesky factor L
- the norm of the "discrepancy" $LL^T - A$, which would be zero if L were the true Cholesky factor of A (since this is a sparse matrix, use the 1-norm; computing the 2-norm of a sparse matrix is more difficult to do efficiently, since it is the maximum singular value).

6. Implement the Conjugate Gradient (CG) method as described in A&G, p. 184. Notice that the termination criterion in the while loop compares the square of the residual norm, $\delta_k = r_k^T r_k = \|r_k\|^2$, to the square of `tol`. Also include a max iteration count to make sure your code does not go into an infinite loop. The denominator of the formula for α_k is $\langle p_k, s_k \rangle$ which means $p_k^T s_k$. Add a test to make sure that α_k is positive, terminating the iteration with an error if it is not. *If α_k is not positive, what does this imply about the matrix A and why?* Test your code on:

   (a) A randomly generated symmetric positive definite 10 by 10 matrix A, computed by `B=randn(10);A=B'*B;` and a random right-hand-side b. If it does not terminate with a residual that is close to zero in exactly 10 iterations, you have a bug in your code.

   (b) The discrete Laplacian matrix A in the previous question, with $N = 100$, with the right-hand side b set to the vector of all ones. Try setting the tolerance to 10^{-8} at first but if this takes too long, try increasing it to 10^{-7} or 10^{-6}. Pass the computed residual norms, $\sqrt{\delta_k}$, out of the CG code and plot them using `semilogy` against the iteration number. Obtain the running time with `tic...toc` or `timeit` and compare this with the time to compute `L=chol(A)` and the time for `x=A\b`. Since backslash uses Cholesky factorization, you would expect these last two to be about the same, but they may not be because backslash may be using pivoting to reduce fill-in which `chol` does not do by default. Also, plot the solution obtained by CG and the one obtained by backslash by "reshaping" the solutions from a vector of length $n = N^2$ to an $N \times N$ matrix and using `mesh`. They should look identical: do they?

   (c) Now we consider the matrix

$$A = I + K^{-1} B B^T K^{-1}$$

   where K is the discrete Laplacian called A earlier, I is the identity matrix of the same dimension, and B is an $N^2 \times r$ matrix, with r much less than N^2, randomly generated by `sprand(N^2,r,1)`. The final argument 1 to `sprand` implies that B is quite dense, and therefore probably has rank r.

   - Explain why this new A is positive definite.
   - Let $\mu_j, j = 1, \ldots, N^2$, denote the eigenvalues of $K^{-1} B B^T K^{-1}$. Assuming B has rank r, how many of the μ_j are nonzero? (One way to answer this is to first consider the rank of $C = K^{-1} B$,

assuming B has rank r, and then consider the rank of CC^T, remembering that eigenvalues and singular values are the same thing for symmetric positive definite matrices.) What does this tell you about the eigenvalues of A, and, in particular, how many *distinct* eigenvalues A has?

- For large enough N, it will be impossible to compute the matrix A explicitly because you will run out of memory. *However, we can still efficiently use the Conjugate Gradient method on this matrix, since all we need are matrix vector products* Av, which can be computed efficiently using the `\` operator along with `*` and `+`, along with plenty of parentheses to make sure the operations are carried out in the right order. Rewrite your CG code so that, instead of an input parameter A, it accepts an anonymous function `Avmult` that computes Av. After testing to make sure this is working properly, solve the new problem for $N = 100$ and $r = 50$, setting b to the vector of all ones and the tolerance as previously (or larger, if this takes too long). Because of the favorable eigenvalue distribution of A, CG should not require many iterations (the theoretical number is given in the middle of p. 187, namely, the number of distinct eigenvalues of A, but rounding errors may increase this substantially). Plot the computed residual norms as previously, as well as the computed solution using `reshape` and `mesh`.
- Then, experiment with how large you can make N, setting $r = N/2$, and still have CG terminate with a reasonable tolerance in a reasonable amount of time. (The definition of "reasonable" is entirely up to you!)

(d) Finally, replace I in the definition of A by a random positive diagonal matrix D generated by `D=sprand(speye(N^2))`. This destroys the favorable eigenvalue structure of A. Does this result in CG taking much longer to run? If so, try speeding it up by using D as a *preconditioner* P: see the preconditioned CG algorithm on p. 188. Does this make much difference, and if so, why?

Acknowledgment: the definition of A in (c) was suggested by Prof. Georg Stadler. It arises in connection with inverse problems in PDEs.