OPTIMIZATION & NONLIN EQS (MA 784) — HW 3

Nonlinear equations, quasi-Newton

1. **(Experiments with CG/GMRES.)** In this problem, we will experiment with the effect of CG/GMRES on different matrices with different preconditioning. For reproducibility (and to make sure you have access to all the preconditioning tools), this is likely better done in Matlab. (It requires minimal coding.)

   Our test problems will come from the SparseSuite matrix collection at `http://sparse.tamu.edu/`. For each of the matrices below, try the following: solve $Ax = b$, where A is the $n \times n$ test matrix and $b = (1, \ldots, 1)/\sqrt{n}$, with tolerance 10^{-9}.

   (a) with no preconditioning;

   (b) with diagonal preconditioning;

   (c) with the "zero-fill" incomplete Cholesky preconditioner depending on the definiteness of the matrix (look up the details, but just use a canned implementation such as Matlab's `ichol`);

   The Cholesky preconditioners have additional options that you can play with. See if you can get to the point where you can solve each system within the given number of iterations and the given tolerance. (You might need close to 2000 iterations in some cases, and in other cases, you might see that it's hopeless to let it run longer than that–in that case, feel free to terminate and report the results.)

   The test problems are: `bcsstk03`, `mhd3200b`, `gyro`, `parabolic_fem` (use tolerance 10^{-7} for this one), `thermal2` (use tolerance 10^{-6} for this one).

   Include your code in your report (PDF). Additionally, report the number of iterations required with the different preconditioners and the `semilogy` plot of the sequence of residuals. (One plot per matrix that shows the comparison between the evolution of residuals.) Were there any preconditioners where you had to try additional options because the default did not work? Were there any examples where preconditioning did not improve on the convergence?

2. **(Some theory.)** Prove that if $\mathbf{b}$ is a linear combination of k of the eigenvectors of $\mathbf{A}$ (where $\mathbf{A}$ is a diagonalizable matrix), then GMRES started from $x^0 = 0$ terminates after at most k iterations. (We stated this in class, and it even appears in the notes, but didn't go over the details.) This is slightly different from the cases we detailed because the runtime depends not only on properties of $\mathbf{A}$ (like "clustered eigenvalues") but on the relationship between $\mathbf{A}$ and $\mathbf{b}$.

3. **(Some more theory.)** Every descent direction is a quasi-Newton direction if you try hard enough: suppose that at $\mathbf{x}^k$ the derivative of our objective function is $\mathbf{g}_k$, and let $\mathbf{d}$ an arbitrary descent direction. Define the matrix $\mathbf{B}$ as
$$\mathbf{B} = \mathbf{I} - \frac{\mathbf{d}\mathbf{d}^{\mathrm{T}}}{\mathbf{d}^{\mathrm{T}}\mathbf{d}} + \frac{\mathbf{g}_k\mathbf{g}_k^{\mathrm{T}}}{\mathbf{g}_k^{\mathrm{T}}\mathbf{d}}.$$
   Show that $\mathbf{B}$ is positive definite and that $\mathbf{d} = \mathbf{B}^{-1}(-\mathbf{g}_k)$, meaning that our chosen $\mathbf{B}$ would yield $\mathbf{d}$ as the search direction if we used it as a Hessian replacement.

4. **(For extra credit only: BFGS.)** In order to efficiently implement the BFGS method, it is useful to not store the Hessian estimate $\mathbf{B}$ or its inverse $\mathbf{B}^{-1}$ explicitly in the memory. Instead, we store only the n-vectors $\mathbf{s}_i$ and $\mathbf{y}_i$ used to estimate the Hessian inverse, and have a subroutine to compute (for the current iterate $\mathbf{x}^k$) the matrix-vector product $\mathbf{B}_k^{-1}(-\mathbf{g}_k)$ directly, using *only vector operations*, and without forming any matrices (let alone solving any linear equations).

   Recall that $\mathbf{B}_k^{-1}$ is defined by the recursion
$$\mathbf{B}_0^{-1} = \mathbf{I}, \quad \mathbf{B}_{i+1}^{-1} = (\mathbf{I} - \rho_i\mathbf{s}_i\mathbf{y}_i^{\mathrm{T}})\mathbf{B}_i^{-1}(\mathbf{I} - \rho_i\mathbf{y}_i\mathbf{s}_i^{\mathrm{T}}) + \rho_i\mathbf{s}_i\mathbf{s}_i^{\mathrm{T}} \quad (i = 0, \ldots, k-1)$$
   where $\rho_i = (\mathbf{s}_i^{\mathrm{T}}\mathbf{y}_i)^{-1}$ for every $i = 0, \ldots, k-1$.

Show how to arrange this computation in such a way that the result is computed in $\mathcal{O}(kn)$ time using $\mathcal{O}(n)$ working memory (in addition to the memory already used to store the input vectors). We can assume $k \leq n$. Note that this means that we can never form or store an $n \times n$ matrix such as $\mathbf{B}_i^{-1}$ or $\mathbf{s}_i\mathbf{y}_i^{\mathrm{T}}$, we cannot store a new n-vector per iteration, and we cannot afford computing any matrix-vector products. Every step must be a vector operation using the input vectors $\mathbf{s}_i$ and $\mathbf{y}_i$ and $\mathcal{O}(n)$ number of constants or $\mathcal{O}(1)$ number of n-vectors computed and stored during the algorithm.

For partial credit, reduce the problem to a recursive computation, where each iteration consists of *only one* recursive call to multiply a vector with $\mathbf{B}_{i-1}^{-1}$, plus a fixed small number of vector operations (independent of i, k, n, etc.) This does *not* fulfill the memory requirements (although superficially it might look like so).

For full credit, avoid recursion altogether: only for loops and vector operations are allowed, storing only $\mathcal{O}(n)$ scalars in addition to the memory used to store the vectors $\mathbf{s}_i$ and $\mathbf{y}_i$.