Numerical Methods I, Fall 2010
Assignment II: Square Linear Systems

Aleksandar Donev
Courant Institute, NYU, donev@courant.nyu.edu

September 23rd, 2010
Due: October 7th, 2010

Choose the problems that interest you, including any of the extra credit ones. Anything above 70 points is excellent. "Extra credit" simply marks problems that are more free-style and do not come with very specific directions.

1 [Up to 20pts pen-and-pencil] Conditioning numbers

1. [Up to 10 pts] What is the conditioning number for a diagonal matrix with diagonal entries $d_1, \ldots, d_n$ in the row sum, column sum, and spectral norms? Can you prove a theorem about which matrices have unit condition number in each norm, or at least give examples of such matrices?
2. [5pts] Prove that the condition number satisfies

$$\kappa\left(\boldsymbol{A}\boldsymbol{B}\right) \leq \kappa\left(\boldsymbol{A}\right)\kappa\left(\boldsymbol{B}\right)$$

   for square nonsingular matrices $\boldsymbol{A}$ and $\boldsymbol{B}$.
3. [5pts] Prove that if the perturbed matrix $\boldsymbol{A} + \delta\boldsymbol{A}$ is singular, then

$$\kappa(\boldsymbol{A}) \geq \frac{\|\boldsymbol{A}\|}{\|\delta\boldsymbol{A}\|},$$

   meaning that a well-conditioned matrix is "far" from singular.

2 [Up to 55 points] Ill-Conditioned Systems: The Hilbert Matrix

Consider solving linear systems with the matrix of coefficients

$$a_{ij} = \frac{1}{i+j-1},$$

which is a well-known example of a very ill-conditioned symmetric positive-definite matrix, see for example this Wikipedia article
`http://en.wikipedia.org/wiki/Hilbert_matrix`

2.1 [15 pts] Conditioning numbers

Form the Hilbert matrix in MATLAB and compute the conditioning number for increasing size of the matrix n for the column sum, row sum, and spectral matrix norms based on the definition $\kappa(\boldsymbol{A}) = \|\boldsymbol{A}\| \left\|\boldsymbol{A}^{-1}\right\|$. Note that the inverse of the Hilbert matrix can be computed analytically, and is available in MATLAB as *invhilb*. Compare to the answer with that returned by the built-in exact calculation *cond* and the estimate *rcond* (check the help pages for details). Use the intrinsic function *fprintf* for format your output nicely so it is readable and submit that output as part of your solution, in addition to the code itself.

Compute the right-hand side (rhs) $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x}$ so that the exact solution is $\boldsymbol{x} = 1$ (all unit entries). Solve the linear system and see how many digits of accuracy you get in the solution for several n. Do your results conform to the theoretical expectation? After what n does it no longer make sense to even try solving the system due to severe ill-conditioning?

2.2 [20 pts] Iterative Improvement

For this problem, fix n at the value for which the Hilbert matrix is barely well-conditioned enough to handle in single precision.

In the previous problem we attempted to solve a linear system $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x}$ for which the exact solution is $\boldsymbol{x} = 1$. What is really meant by "exact" is that given an exactly-represented matrix $\boldsymbol{A}$ and the exactly-represented vector $\boldsymbol{x} = 1$, we have that $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x}$ *exactly*, meaning, if calculated and represented exactly using infinite precision arithmetic. Since we cannot represent either the Hilbert matrix or the vector of ones *exactly* in MATLAB (without using the symbolic toolbox), lets instead assume that $\boldsymbol{A}$ is whatever the single-precision approximation to the Hilbert matrix is, and $\boldsymbol{x} = fl(1)$ to be a vector where each entry is whatever the single-precision floating-point approximation to 1 is. In this way, we can use double precision arithmetic as a proxy for infinite-precision arithmetic, namely, we can assume that double-precision calculations for all practical purposes give an exact answer. The MATLAB code

```
A=hilb(n,'single');
x_exact=ones(n,1,'single');
b_exact=double(A)*double(x_exact);
b=single(b_exact);
```

can thus be taken to form a linear system in single precision for which the solution is x_{exact} to "infinite" precision.

The numerical solution $x = A\backslash b$ will be a bad approximation since for this n the system is too ill-conditioned for single precision and b is a single-precision approximation to b_{exact}. Verify that indeed the solution MATLAB computes with $x = A\backslash b$ (or an explicit Cholesky factorization) is far from the exact solution.

Interestingly, there is a way to compute a much better approximation by using a method called *iterative refinement*. In iterative refinement, we take an initial guess $\boldsymbol{x}$ to the solution and repeat the following steps iteratively:

1. Compute the residual $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$ using higher-precision arithmetic. That is, in MATLAB, use

   ```
   r=single(b_exact-double(A)*double(x));
   ```

   so that the residual is computed to 16 and not just 8 decimal places *before* it is rounded to single precision.
2. Solve the linear system $\boldsymbol{A}(\Delta\boldsymbol{x}) = \boldsymbol{r}$ for $\Delta\boldsymbol{x}$, paying attention to efficiency [Hint: *You can reuse a Cholesky factorization of* $\boldsymbol{A}$ *many times*].
3. Correct the solution $\boldsymbol{x} \leftarrow \boldsymbol{x} + \Delta\boldsymbol{x}$.

Note that only step 1 requires using double precision, all the rest can be performed without calling *double* at all. Implement this in MATLAB and verify that the improved solution indeed converges to the exact solution to within full single precision (meaning $7 - 8$ decimal places). By approximately what factor does the error $\|\boldsymbol{x} - \boldsymbol{x}_{exact}\|_\infty$ decrease every iteration (numerically or maybe you have some theoretical estimate)? Increase n by one and try again, until something breaks and even iterative refinement does not work.

2.3 [20pts Extra Credit] Iterative methods

Try an iterative method for the ill-conditioned Hilbert system and see if it performs better or worse than GEM with respect to roundoff errors.

3 [Up to 70 points] Rank-1 Matrix Updates

In a range of applications, such as for example machine learning, the linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ needs to be re-solved after a rank-1 update of the matrix,

$$\boldsymbol{A} \to \tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{u}\boldsymbol{v}^T,$$

for some given vectors $\boldsymbol{v}$ and $\boldsymbol{u}$. More generally, problems of *updating a matrix factorization* (linear solver) after small updates to the matrix appear very frequently and many algorithms have been developed for special forms of the updates. The rank-1 update is perhaps the simplest and best known, and we explore it in this problem. From now on, assume that $\boldsymbol{A}$ is invertible and its inverse or $\boldsymbol{LU}$ factorizations are known, and that we want to update the solution after a rank-1 update of the matrix. We will work with random dense matrices for simplicity.

3.1 [10pts] Sherman-Morrison-Woodbury (SMW) formula

Show that $\tilde{\boldsymbol{A}}$ is invertible if and only if $\boldsymbol{v}^T\boldsymbol{A}^{-1}\boldsymbol{u} \neq -1$, and in that case

$$\tilde{\boldsymbol{A}}^{-1} = \boldsymbol{A}^{-1} - \frac{\boldsymbol{A}^{-1}\boldsymbol{u}\boldsymbol{v}^T\boldsymbol{A}^{-1}}{1 + \boldsymbol{v}^T\boldsymbol{A}^{-1}\boldsymbol{u}}. \tag{1}$$

This is the so-called Sherman-Morrison formula, a generalization of which is the Woodbury formula, as discussed on Wikipedia:
`http://en.wikipedia.org/wiki/Sherman-Morrison-Woodbury_formula`.

3.2 [20pts] SMW in MATLAB

In MATLAB, generate a random (use the built-in function *randn*) $n \times n$ matrix $\boldsymbol{A}$ for some given input n and compute its *LU* factorization. Also generate a right-hand-side (rhs) vector $\boldsymbol{b}$ and solve $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. Then generate random vectors $\boldsymbol{v}$ and $\boldsymbol{u}$ and update the previous solution $\boldsymbol{x}$ using 1 to obtain the solution $\tilde{\boldsymbol{x}}$ to $\tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}} = b$. Be careful to do this as efficiently as you can, that, is, not performing unnecessary operations [*Hint: You only need to solve two triangular systems to update the solution*]. Verify the new solution $\tilde{\boldsymbol{x}}$ in two ways:

1. By computing a new factorization $\tilde{\boldsymbol{A}} = \tilde{\boldsymbol{L}}\tilde{\boldsymbol{U}}$ and solving $\tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}} = \boldsymbol{b}$.
2. By directly verifying that the residual is small. How does the norm of the residual compare to that obtained by solving $\tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}} = \boldsymbol{b}$ directly (question 1 above).

For several n, compare the time it takes to compute the updated solution directly versus the time it takes to use the Sherman-Morrison-Woodbury formula. Make an estimate of how these times should scale with n [example, $O(n^3)$] and see if the numerical data approximate conforms to the expectation [*Hint: The MATLAB functions tic and toc might be useful in timing sections of code*].

3.3 [20pts] Cholesky factorization updates

Modify your code from the previous problem 1.2 to make the matrix and the update symmetric (i.e., $\boldsymbol{u} = \boldsymbol{v}$), and to use the Cholesky instead of the *LU* factorization. Make sure to use the symmetry to make the code simpler and more efficient. A quick way to generate a symmetric positive-definite $n \times n$ matrix in MATLAB is:

$$A = gallery('randcorr', n).$$

As a new alternative to computing $\tilde{\boldsymbol{x}}$, consider computing the Cholesky factor $\tilde{\boldsymbol{L}}$ by updating the previous factor $\boldsymbol{L}$ instead of recomputing the whole factorization anew, as discussed at length in Ref. [1] (no need to read this paper) and implemented in the MATLAB function *cholupdate*. Compare the accuracy and speed of using *cholupdate* to your own implementation of the Sherman-Morrison-Woodbury formula.

3.3.1 [20pts Extra Credit] Your own *cholupdate*

While the actual algorithms used internally by MATLAB are based on some more sophisticated linear algebra software, it is not too hard to implement your own version of *cholupdate*. Here is one idea [1]:

From the identity

$$\tilde{\boldsymbol{A}} = \tilde{\boldsymbol{L}}\tilde{\boldsymbol{L}}^T = \boldsymbol{A} + \boldsymbol{v}\boldsymbol{v}^T = \boldsymbol{L}\left(\boldsymbol{I} + \boldsymbol{p}\boldsymbol{p}^T\right)\boldsymbol{L}^T = \boldsymbol{L}\boldsymbol{L}_p\boldsymbol{L}_p^T\boldsymbol{L}^T,$$

where $\boldsymbol{L}\boldsymbol{p} = \boldsymbol{v}$ and $\boldsymbol{L}_p$ is the Cholesky factor of $\boldsymbol{I} + \boldsymbol{p}\boldsymbol{p}^T$, we see that $\tilde{\boldsymbol{L}} = \boldsymbol{L}\boldsymbol{L}_p$ can be generated from the previous factorization by factorizing the very special matrix $\boldsymbol{M} = \boldsymbol{I} + \boldsymbol{p}\boldsymbol{p}^T = \boldsymbol{L}_p\boldsymbol{L}_p^T$. Can you come up with a formula/algorithm for factorizing $\boldsymbol{M}$? Hint: Try the following form for the elements of the matrix $\boldsymbol{L}_p$:

$$l_{ij}^{(P)} = \begin{cases} \alpha_i p_i & \text{if } i = j \\ \beta_j p_i & \text{if } i > j \\ 0 & \text{otherwise} \end{cases}$$

and find an algorithm for computing the unknown multipliers α and β. Verify that this works by testing it on an example.

4 [30 points Extra Credit] The Laplacian Matrix

Sparse matrices appear frequently in all fields of scientific computing. As emphasized in the lectures, their properties and the associated linear algebra depend very strongly on the origin of the matrix. Here I suggest working with a matrix that arises in solving the Poisson equation in a two-dimensional domain, simply because this matrix is available in MATLAB directly. If you want to choose some other sparse matrix to explore, including some of the MATLAB *gallery* ones, that is OK too.

The matrix representation of the Laplacian operator $\Delta \equiv \boldsymbol{\nabla}^2$ that appears when solving Poisson's PDE (details in Numerical Methods II) inside a bounded domain

$$\boldsymbol{\nabla}^2 x = b = 1 \text{ in the interior and } x = 0 \text{ on the boundary,}$$

can be generated in MATLAB for some simple two-dimensional domains using the *delsq* function (see MATLAB help topic "Finite Difference Laplacian"). For example:

```
A = delsq(numgrid('S',n));
```

generates a positive-definite discrete Laplacian matrix $\boldsymbol{A}$ for a square domain of $n \times n$ grid points. This exercise is about exploring the properties of this matrix and solving the linear system $\boldsymbol{Ax} = \boldsymbol{b} = \boldsymbol{1}$ using MATLAB.

Generate some sample Laplacian matrices and visualize their sparsity structure using *spy*. Can you see any connections between the sparsity structure and the problem (domain) geometry? Solve this system using direct methods, such as Cholesky factorization, and report the solution time and the fill-in that appears. Explore how these scale with the size of the domain n and try some matrix reordering algorithm that is built into MATLAB.

Now also try an iterative method, for example, the preconditioned conjugate gradient (PCG) method available in MATLAB via the function *pcg* [*Hint: The help page will be very useful*], and report how fast the method converges (or does not). Try some preconditioner and see if it speeds the convergence and decreases the overall solution time or not.

References

[1] P.E. Gill, G.H. Golub, W. Murray, and M.A. Saunders. Methods for modifying matrix factorizations. *Mathematics of Computation*, 28(126):505–535, 1974.