Numerical Methods I, Fall 2014
Assignment II: Linear Systems

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

September 18th, 2014
Due: **October 5th, 2014**

For the purposes of grading the maximum number of points is considered to be 125 points, but you can get more via extra credit pieces. **Please do not do extra credit problems for a grade, do them if they interest you to challenge yourself**.

Make sure to follow good programming practices in your MATLAB codes. For example, make sure that parameters, such as the number of variables n, are not hard-wired into the code and are thus easy to change. Use *fprintf* to format your output nicely for inclusion in your report.

1 [15 pts] Conditioning numbers

1. [5 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 $\kappa = 1$ in each norm, or at least give examples of such matrices?
2. [5pts] Prove that the condition number satisfies
$$\kappa(\boldsymbol{AB}) \leq \kappa(\boldsymbol{A})\,\kappa(\boldsymbol{B})$$
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 [35 + 10 pts] Ill-Conditioned Systems: The Hilbert Matrix

Consider solving linear systems with the matrix of coefficients $\boldsymbol{A}$ defined by
$$a_{ij} = \frac{1}{i + j - 1},$$
which is a well-known example of an ill-conditioned symmetric positive-definite matrix, see for example this Wikipedia article
`http://en.wikipedia.org/wiki/Hilbert_matrix`

2.1 [10 pts] Conditioning numbers

[10pts] Form the Hilbert matrix in MATLAB and compute the conditioning number for increasing size of the matrix n for the L_1, L_2 and L_∞ (the column sum, row sum, and spectral matrix) norms based on the definition $\kappa(\boldsymbol{A}) = \|\boldsymbol{A}\| \left\|\boldsymbol{A}^{-1}\right\|$ and using MATLAB's *norm* function. 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 returned by the function *rcond* (check the help pages for details).

2.2 [10pts] Solving ill-conditioned systems

[5pts] Compute the right-hand side (rhs) vector $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x}$ so that the exact solution is $\boldsymbol{x} = 1$ (all unit entries). Solve the linear system using MATLAB's built-in solver and see how many digits of accuracy you get in the solution for several n, using, for example, the infinity norm.

[5pts] Do your results conform to the theoretical expectation discussed in class? After what n does it no longer make sense to even try solving the system due to severe ill-conditioning?

2.3 [15 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.

2.4 [10pts 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 [30 points] Least-Squares Fitting

Consider fitting a data series (x_i, y_i), $i = 1, \ldots, n$, consisting of $n = 100$ data points that approximately follow a polynomial relation,

$$y = f(x) = \sum_{k=0}^{d} c_k x^k,$$

where c_k are some unknown coefficients that we need to estimate from the data points, and d is the degree of the polynomial. Observe that we can rewrite the problem of least-squares fitting of the data in the form of an overdetermined linear system

$$[\boldsymbol{A}(\boldsymbol{x})]\, \boldsymbol{c} = \boldsymbol{y},$$

where the matrix $\boldsymbol{A}$ will depend on the x-coordinates of the data points, and the right hand side is formed from the y-coordinates.

Let the correct solution for the unknown coefficients $\boldsymbol{c}$ be given by $c_k = k$, and the degree be $d = 9$. Using the built-in function *rand* generate synthetic (artificial) data points by choosing n points $0 \leq x_i \leq 1$ randomly, uniformly distributed from 0 to 1. Then calculate

$$\boldsymbol{y} = f(\boldsymbol{x}) + \epsilon\boldsymbol{\delta},$$

where $\boldsymbol{\delta}$ is a random vector of normally-distributed perturbations (e.g., experimental measurement errors), generated using the function *randn*. Here ϵ is a parameter that measures the magnitude of the uncertainty in the data points. [Hint: *Plot your data for $\epsilon = 1$ to make sure the data points approximately follow $y = f(x)$.*]

3.1 [20pts] Different Methods

For several logarithmically-spaced perturbations (for example, $\varepsilon = 10^{-i}$ for $i = 0, 1, \ldots, 16$), estimate the coefficients $\tilde{\boldsymbol{c}}$ from the least-squares fit to the synthetic data and report the error $\|\boldsymbol{c} - \tilde{\boldsymbol{c}}\|$. Do this using three different methods available in MATLAB to do the fitting:

a) [5pts] The built-in function *polyfit*, which fits a polynomial of a given degree to data points [*Hint: Note that in MATLAB vectors are indexed from 1 and thus the order of the coefficients that polyfit returns is the opposite of the one we use here, namely, c_1 is the coefficient of x^d.*]

b) [5pts] Using the backslash operator to solve the overdetermined linear system $\boldsymbol{A}\tilde{\boldsymbol{c}} = \boldsymbol{y}$.

c) [5pts] Forming the system of normal equations discussed in class,

$$\left(\boldsymbol{A}^T\boldsymbol{A}\right)\boldsymbol{c} = \boldsymbol{A}^T\boldsymbol{y},$$

and solving that system using the backslash operator.

[5pts] Report the results for different ϵ from all three methods in one printout or plot, and explain what you observe.

3.2 [10pts] Conditioning

[5pts] If $\epsilon = 0$ we should get the exact result from the fitting. How close can you get to the exact result for each of three methods? Is one of the three methods clearly inferior to the others? Can you explain your results? *Hint: Theory suggests that the conditioning number of solving overdetermined linear systems is the square root of the conditioning number of the matrix in the normal system of equations, $\kappa(\boldsymbol{A}) = \sqrt{\kappa\left(\boldsymbol{A}^T\boldsymbol{A}\right)}$.*

[5pts] Test empirically whether the conditioning of the problem get better or worse as the polynomial degree d is increased.

4 [45 + 20 points] Rank-1 Matrix Updates

In a range of applications, such as for example machine learning, the linear system $\boldsymbol{Ax} = \boldsymbol{b}$ 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.

4.1 [15pts] Direct update

[5pts] 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 $\boldsymbol{Ax} = \boldsymbol{b}$.

[7.5pts] For several n, compute and plot the time it takes to compute the solution using the particular machine you used. Can you tell from the data how the execution time scales with n [example, $O(n^2)$ or $O(n^3)$]? [*Hint: The MATLAB functions tic and toc might be useful in timing sections of code*].

[2.5 pts] Now generate random vectors $\boldsymbol{v}$ and $\boldsymbol{u}$ and obtain the updated solution $\tilde{\boldsymbol{x}}$ of the system $\tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}} = \boldsymbol{b}$. Verify the new solution $\tilde{\boldsymbol{x}}$ by directly verifying that the residual $\boldsymbol{r} = \boldsymbol{b} - \tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}}$ is small.

4.2 [15pts] SMW Formula

It is not hard to 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`.

[10pts] The SMW formula (1) can be used to compute a new solution $\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{A}}^{-1}\boldsymbol{b}$. Be careful to do this as robustly and efficiently as you can, that, is, not actually calculating matrix inverses but rather (re)using matrix factorizations to solve linear systems [*Hint: You only need to solve two triangular systems to update the solution once you have the factorization of* $\boldsymbol{A}$]. For some n (say $n = 100$), compare the result from using the formula (1) versus solving the updated system $\tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}} = \boldsymbol{b}$ directly.

[5pts] For the largest n for which you can get an answer in a few minutes, compare the time it takes to solve the original system for $\boldsymbol{x}$ versus the time it takes to compute the updated answer $\tilde{\boldsymbol{x}}$.

4.3 [15pts] 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, and explain in words how you used the symmetry (instead of cutting and pasting code). 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. [**?**] (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.

4.3.1 [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 [**?**]:

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.

5 [Up to 25 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 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} = \boldsymbol{1}$ using MATLAB.

[10 pts] 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.

[15 pts] 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.