Scientific Computing, Spring 2012
Assignment II: Linear Systems

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

February 16th, 2012
Due by Thursday **March 1st**

A total of 100 points is possible. 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 [35 points] 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`

1.1 [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). Format your output nicely and submit it as part of your report.

1.2 [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?

1.3 [15pts] Solving using matrix pseudoinverse (SVD)

The solution to the system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ can be formally written as $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$, although we do not use the matrix inverse in practice. However, there is a numerically stable algorithm for computing the inverse (or solving the linear system) using the Singular Value Decomposition (SVD) of the matrix $\boldsymbol{A}$ (to be covered in Lecture 5). This works even for non-square matrices and is a way to define and compute the so-called matrix pseudoinverse $\boldsymbol{A}^\dagger$. With exact arithmetic for a square invertible matrix $\boldsymbol{A}^\dagger = \boldsymbol{A}^{-1}$, but calculations based on the SVD can be different in the presence of roundoff errors. An approximation to the pseudoinverse can be obtained in MATLAB using the SVD using the call

$$\hat{\boldsymbol{A}}^\dagger = pinv(\boldsymbol{A}, \epsilon),$$

where ϵ is a small tolerance that is used to improve the numerical stability of the computation.

[10 pts] Let the size of the Hilbert matrix $\boldsymbol{A}$ be $n = 15$, and consider the same $\boldsymbol{b}$ as in part 1.2. For several logarithmically-spaced tolerances (for example, $\varepsilon = 10^{-i}$ for $i = 1, 2, \ldots, 16$), compute the approximate

pseudo-inverse using *pinv* and then a solution $\hat{\boldsymbol{x}} = \hat{\boldsymbol{A}}^{\dagger}\boldsymbol{b}$. Plot the relative error in the modified solution versus the tolerance on a log-log scale. You should see a clear minimum error for some $\varepsilon = \tilde{\varepsilon}$. Report this optimal $\tilde{\varepsilon}$ and the smallest error possible.

[5pts] Explain what kinds of errors are traded off, that is, what kind of errors dominate for large versus for small tolerances [*Hint: You should recall similar plots of errors versus parameter from the first homework. It may also be useful to consult the help page for pinv*].

2 [35 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 $0 \le x_i \le 1$ 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 $y = f(x)$.*]

2.1 [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.

2.2 [15pts] The Best Method

[10pts] If $\epsilon = 0$ we should get the exact result from the fitting. What is the highest accuracy you can achieve with each of the 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.

3 [30 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,

$$\boldsymbol{A} \rightarrow \tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{uv}^T,$$

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 [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.

3.2 [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

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

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 using matrix factorizations to solve linear systems [*Hint: You only need to solve two triangular systems to update the solution once you factorize* $\boldsymbol{A}$]. 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 $\tilde{\boldsymbol{x}}$.