Scientific Computing, Fall 2020
Assignment II: Linear Systems

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

September 18th, 2019
Due by Sunday **October 4th**

For the purpose of grading 100% will be taken to be 100 points, but 110 points are 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 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`

Note: *This problem is* ***not*** *about the Hilbert matrix per se, but about ill-conditioned matrices in general, so do not focus on this specific matrix (one reason we are using it is that we will encounter it again in a later homework on polynomial approximation). We will later talk about the Vandermonde matrix (related to polynomial interpolation) which is another example of an ill-conditioned matrix that you could use equally well.*

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*, but you can also just use the built-in *inv* function (or its equivalent in numpy if using python). Compare to the answer with that returned by the built-in exact calculation *cond* and (if using Matlab) the estimate returned by the function *rcond* (check the help pages for details).

1.2 [25 pts] Solving ill-conditioned systems

Note: *We will return to this problem in the next homework, where we will see that using the pseudo-inverse instead of inverse helps a lot. I encourage you to think a little bit whether this violates the statement that "no numerical method can solve an ill-conditioned problem accurately."*

[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 (explain what method was used to solve the system in your report) and see how many digits of accuracy you get in the solution for several n, using, for example, the infinity norm. Also compute the relative norm of the residual $\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\| \,/\, \|\boldsymbol{b}\|$ and explain how it changes with the conditioning number of the matrix.
Note: A method is called *backward stable* if it computes the *exact* solution to a nearby problem, i.e., if the residual is small.

[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?

[5pts] Now do the same but solve the system using the Cholesky factorization of $\boldsymbol{A}$, and compare the results and report if anything has changed.

[5pts] Now try computing the solution by using the numerically-computed matrix inverse, $x = inv(A) * b$, and compute the relative errors in the solution and in the residual and see how they behave. What difference do you see between using the LU or Cholesky factorization versus using matrix inverse? Which one is better and why?
Note: Do not use *invhilb* since this is specific to the Hilbert matrix only.

[5pts] Finally, if using MATLAB, try the *matinv* function posted on the course homepage instead of the built-in *inv*, and compare the two. Report what you see and try to explain your observations. If using numpy, try to find a direct (naive) implementation of the Gauss-Jordan method for computing matrix inverse.

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) with mean zero and unit variance, 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 some small value of ϵ 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 [40 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,

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

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 [10pts] 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 (you will need it for later parts of this problem). Also generate a right-hand-side (rhs) vector $\boldsymbol{b}$ and solve $\boldsymbol{Ax} = \boldsymbol{b}$.

[5pts] 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 [30pts] 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`. The SMW formula (1) can be used to compute a new solution $\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{A}}^{-1}\boldsymbol{b}$ without actually computing $\tilde{\boldsymbol{A}}^{-1}$ or factorizing $\tilde{\boldsymbol{A}}$.

[10pts] 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 as in part 3.1.

[20pts] Implement the SMW formula (1) carefully to compute $\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{A}}^{-1}\boldsymbol{b}$ as robustly and efficiently as you can, that, is, not actually calculating matrix inverses but rather (re)using the LU factorization of $\boldsymbol{A}$ [*Hint: You only need to solve two triangular systems to update the solution once you have the factorization of* $\boldsymbol{A}$]. Explain how you computed $\tilde{\boldsymbol{x}}$ and estimate *using pen and paper* how expensive this is in terms of number of operations (FLOPS). Explain how much faster it is then solving $\tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}} = \boldsymbol{b}$ directly as a function of n. Note: *Do not try to use empirical timing in MATLAB or other numerical methods to estimate the number of operations. Estimate it analytically assuming a "perfect computer" (which real computers are not!).*