Math 4242 Sec 40 Extra Credit

SOLVING LINEAR SYSTEMS WITH VARIABLE-PRECISION FLOATING POINT ARITHMETIC: LU-FACTORIZATION VS. MATRIX INVERSION

Grade Value: Maximum 7/100% of the total class grade.

Due date: Monday, December 1

Disclaimer: This project will require significant initiative, independence, effort on your part, as well as competence in coding.

Language: For several reasons, I suggest that you use Matlab for this project. This project specification will be written under the assumption that you are using Matlab. However, you may use whatever language you wish.

Overview

If a matrix A is invertible, then the unique solution to the system $A\mathbf{x} = \vec{b}$ is given by $\mathbf{x} = A^{-1}\vec{b}$. However, one of the basic tenants of numerical linear algebra is that one should not solve systems of linear equations by taking matrix inverses. This is both because the matrix inversion approach is computationally more expensive than the alternative approaches discussed in class, and because the matrix inversion approach has poorer stability properties.

In this project, you will write code to solve linear systems both via the LU-factorization approach, and via the computation of matrix inverses. You will then empirically compare the stability of these two approaches. To facilitate your study of stability questions, your code will implement **variable-precision floating point arithmetic**, allowing you to specify the number of significant digits used for the floating point arithmetic in the computation. In the spirit of Meyer's discussion of floating point arithmetic and computational linear algebra, we will be interested only in working with relatively small numbers of significant digits.

Assignment Details

Your code should make use of the external Matlab function **roundsd,** written by Franois Beauducel. The file containing this function is free to download at the Matlab File Exchange website. The function rounds a number in Matlab to a base-10 floating point number with user-specified precision. For example, **roundsd(3456,3)** will return 346, the base-10 floating point representation of 3456 with 3 digits of precision. **roundsd** also accepts matrices, and returns a matrix with each entry rounded to the

appropriate floating point number. [If you don't work in Matlab, you will have to build/find your own version of **roundsd** for your language.]

Using the **roundsd** function, implement each of the following functions (many of which will build on one another). Each function takes an argument t, which is a non-negative integer; it is to be understood that each function performs its arithmetic in floating point arithmetic with t digits of precision.

Note: You must write your own functions from scratch. Modifying code you found off the internet is not allowed, except for **roundsd** or its equivalent. However, you may look at code on the internet for inspiration/ideas.

(1) **new_row=ScalarTimesRow(row,sc,t)** Multiplies a real scalar **sc** by a row vector **row** of arbitrary length.

(2) **new_row=AddRows(row1,row2,t)** Adds two rows **row1** and **row2** of the same length.

(3) **sc=RowColProduct(row,col,t)** For **row** a row vector of length n and **col** a column vector of length n, takes the product $\mathbf{sc} = \mathbf{col} * \mathbf{row}$.

(4) **Prod=MatrixProduct(A,B,t)** For **A** an $l \times m$ matrix and B an $m \times n$ matrix, computes the product $\mathbf{Prod} = \mathbf{A} * \mathbf{B}$.

(5) **[L,U,P]=GEPP(A,t)** Computes the **P**=**LU** decomposition of a rectangular (not necessarily square) matrix **A**. **L** and **U** are to be returned separately, but while the code runs, their entries are to be stored in the same matrix which initially stores **A**. At the very end of the computation, **L** and **U** are to be put into separate matrices and returned. **P** is to be represented internally and returned as a "permutation vector," as in example 3.10.4 of Meyer.

(6) **x=FSolve(L,b,t)** Solves the linear system **Lx**=**b**, for **L** a lower triangular matrix, via a forward solve. **FSolve** should return an error if **L** is not lower-triangular (see the Matlab function **istril**), or if fl(**L**), the floating point representation of **L**, has a zero-diagonal entry (so that fl(**L**) is singular). Otherwise **FSolve** returns a unique floating-point solution **x**.

(7) **x=BSolve(U,b,t)** Solves the linear system **Ux**=**b**, for **U** an upper triangular matrix **U**, via a backsolve. **BSolve** should return an error if **U** if not upper-triangular (see the Matlab function **istriu**), or if fl(**U**) has a zero-diagonal entry (so that fl(**U**) is singular). Otherwise

BSolve returns a unique floating point solution **x**.

(8) **x=LUSolve(A,b,t)** Solves the $n \times n$ linear system $\mathbf{Ax} = \mathbf{b}$ using your functions **GEPP**, **FSolve**, and **BSolve**. Should return an error if the system can't be solved uniquely, whether because A is nonsingular or because of issues due to floating point arithmetic.

(9) **AInv=Invert(A,t)** Computes the inverse **AInv** of an $n \times n$ nonsingular matrix **A**. To do this, the code should solve n linear systems, as discussed in Meyer 3.7. If some system encountered during the solve does not have unique solution (because A is nonsingular or because of issues due to floating point arithmetic), the code should give an error message.

(10) **x=InvSolve(A,b,t)** Solves the $n \times n$ linear system $\mathbf{Ax} = \mathbf{b}$ using your functions **Invert**, and **MatrixProduct**. Should return an error if system can't be solved uniquely, whether because A is nonsingular or because of issues due to floating point arithmetic.

Empirical Study of Linear Algebra Using Floating-Point Arithmetic. Using your code, do the following:

(1) (Required) Conduct an empirical study of the stability of floating point solutions to linear systems, using both your LUSolve code and your InvSolve code. For each solution method, how do solutions to nonsingular linear systems $Ax = b$ deviate from the true solution as the number of significant digits t used for the floating point arithmetic changes? For fixed values of t, how do solutions change as A and b are randomly perturbed? Consider examples of both well conditioned and ill-conditioned systems $Ax = b$. Give thorough, quantitative answers. [Note: I am intentionally leaving this part a bit open-ended. It is up to you to tell a good scientific story.]

(2) (Optional, but recommended) Conduct an empirical study of the stability of matrix-vector multiplication in floating point arithmetic, analogous to what you were asked for above in 1).

(3) (Recommended) Conduct an empirical study of the stability of matrix inverse computations in floating point arithmetic, analogous to what you were asked for above in 1).

(4) (Recommended) What seems to be the connection between what you observed in 1) and what you observed in 2) and 3)?

Grading, etc.

5/7 of the credit will be for successful completion of the coding parts of the project (1)-(10) above. 2/7 of the credit will be for a well presented, thorough empirical study of stability issues.

You must submit all of your code to me in a zipped folder by email. For Matlab users, I should be able to run your code from the folder. You should present your experimental findings in a .pdf file, in the same zipped folder.

All code used to perform your the experiments should also be included in the folder.

As part of the grading process, I may ask you to come discuss your code with me. If you do not use Matlab, grading your stuff on my own will be hard, so I will ask you to throughly demonstrate the functionality of your code in person.

Grading will be strict. Please make sure you carefully test your code well and get rid of bugs and conceptual errors before submission. I will expect submitted projects to be complete, correct, and bug-free. Partial credit will be doled out only sparingly, if at all, simply because it will be too time consuming to pore over buggy code, trying to decipher what was and wasn't done correctly.

However, I will accept projects that do not include an empirical study of stability for a maximum 5/7 points.

Collaboration. You are encouraged to talk to your classmates about this project; however, **you must write all of your own code**. Sharing code for the project, especially electronically, is (emphatically) not allowed.