Math 447

Due date: 2015 Feb 11 (Wednesday)

Project 1: QR Factorization and Solving Simple Linear Equations

1. Introduction

Consider the linear problem

$$A\mathbf{x} = \mathbf{b} \tag{1.1}$$

where A is a given, invertible $m \times m$ matrix, $\mathbf{b}$ is a given vector of length m, and $\mathbf{x}$ is a vector of length m to be solved for. This problem arises in various contexts in essentially every branch of science. In previous courses, you learned that the solution can be written as $\mathbf{x} = A^{-1}\mathbf{b}$. However, for large matrices, computing A^{-1} directly (e.g., by Gaussian elimination) is very costly, and can lead to instabilities. A major topic of Math 447 is to find better ways to solve (or approximate) solutions to problems in the form (1.1). We will look at two methods in this project.

1.1. Upper Triangular Systems.

Consider again problem (1.1), but imagine we are in the case where A is upper-triangular; that is, A is in the form

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1,m} \\ 0 & a_{22} & a_{23} & \cdots & a_{2,m} \\ 0 & 0 & a_{33} & \cdots & a_{3,m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{m,m} \end{bmatrix}$$

That is, $a_{ij} = 0$ whenever $i > j$. Note that the system (1.1) can then be written as

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 + \cdots + a_{1,m}x_m &= b_1 \\ a_{22}x_2 + a_{23}x_3 + a_{24}x_4 + \cdots + a_{2,m}x_m &= b_2 \\ a_{33}x_3 + a_{34}x_4 + \cdots + a_{3,m}x_m &= b_3 \\ \ddots \quad \vdots \quad \vdots \quad \vdots & \\ a_{m-1,m-1}x_{m-1} + a_{m-1,m}x_m &= b_{m-1} \\ a_{m,m}x_m &= b_m \end{aligned}$$

Notice that, if you start at the bottom, the whole system can be solved without inverting the matrix, like this:

$$\begin{aligned} x_m &= b_m/a_{m,m} \\ x_{m-1} &= (b_{m-1} - a_{m-1,m}x_m)/a_{m-1,m-1} \\ x_{m-2} &= (b_{m-2} - a_{m-2,m-1}x_{m-1} - a_{m-2,m}x_m)/a_{m-2,m-2} \\ \vdots \quad & \quad \vdots \\ x_1 &= (b_1 - a_{12}x_2 - \cdots - a_{1,m-1}x_{m-1} - a_{1,m}x_m)/a_{11} \end{aligned}$$

This method works if you follow the steps in order, since first you compute x_m, then, now that you know x_m, you use it to compute x_{m-1}. Now that you know x_{m-1} you use it and x_m to compute x_{m-2}, and so on, until you have computed all of $\mathbf{x}$.

1.2. **QR-factorization.** Notice that the QR-factorization of a matrix gives $A = QR$, where Q is a unitary matrix and R is upper-triangular. Therefore, if you want to solve $A\mathbf{x} = \mathbf{b}$, you can instead consider solving the equivalent system

$$QR\mathbf{x} = \mathbf{b}$$

You can solve this in two steps. First, invert Q, to get the upper-triangular system $R\mathbf{x} = Q^{-1}\mathbf{b}$ then solve for $\mathbf{x}$ by using back-substitution. But wait, isn't inverting a matrix supposed to be hard? Yes! However, Q is a **unitary** matrix, so $Q^{-1} = Q^*$, which is easy to compute, since Q^* is just the conjugate-transpose of Q!

2. Develop and Test your code

We will pretty much always write code in two parts: one (or more) `MATLAB` $*.m$ file(s) to compute what we want, and a separate `MATLAB` $*.m$ file to test our code.

2.1. **Part I: Upper-triangular solver.** Write a code to solve upper-triangular systems. Note the algorithm above can be written like this:

$$\begin{aligned}
&for \;\; i = m : -1 : 1 \\
&\quad y = b_i \\
&\quad for \;\; j = (i+1) : m \\
&\qquad y = y - A_{i,j}x_j \\
&\quad end \\
&\quad x_i = y/A_{i,i} \\
&end
\end{aligned}$$

NOTE! The above is "psuedo-code," meaning it is a sketch of code, and is not language-specific! It is your job to translate it into `MATLAB` code. Spend some time convincing yourself that the above two algorithms for computing $\mathbf{x}$ are the same. What's going on with that y there? Also, note that a loop over $(m+1) : m$ is an "empty loop", and therefore the inner loop won't be run the first time through.

Note that the % symbol is a "comment" symbol, used to make notes to other humans (including your future self). Anything on a line after a % will be ignored by `MATLAB`.

Your first task is to implement this algorithm as a `MATLAB` function. Try it by creating a `MATLAB` file called `upperTri.m`, and start it by something like this:

```
function x = upperTri(A,b,m)

% Your algorithm to compute x from A and b goes here.

end
```

Make a new `MATLAB` file, call it say, `testUpperTri.m`, and save it in the same folder as `upperTri.m` (note: case matters here). There are many ways to test your code, but since

this is our first project, I will give you the entire testing code so that you know what I am looking for. Here it is:

```
% ====== Begin testUpperTri.m file =====
clear all; close all;
% Test the upperTri.m program

m = 10; % Set the size of the matrix
% Create a random upper-triangular matrix:
A = triu(rand(m,m)-0.5);

x_exact = rand(m,1);
% Create a "b" from the known exact solution:
b = A*x_exact;

x = upperTri(A,b,m); % Compute our solution

% Use the 2-norm to check if the solutions are close:
error = norm(x - x_exact);
display(sprintf('The error is %g with size %d.',error,m));
% ====== End testUpperTri.m file =====
```

Some of the above commands are new. How can you find out what they are doing? Easy! Just use MATLAB's "help" command. Type into the command window (don't type the ">>"):

```
>> help triu
>> help rand
>> help norm
```

You may have to scroll up to read everything it outputs. For a more graphical version `help`, type `doc` instead of `help`.

Your code should work for arbitrary choice of the matrix size m, but the error will get worse as m grows. For $m = 10$, your errors should be around 10^{-16}, which MATLAB writes as $1e-16$.

3. Part II: QR-Factorization

Following the **modified Graham-Schmidt** algorithm in the book, write a MATLAB function to perform a QR-factorization of a given matrix. Call it `myQR.m`, and call it from another program like this:

```
[Q R] = myQR(A);
```

The first few lines in your `myQR.m` file could be something like this:

```
function [Q R] = myQR(A)
% A function to compute the QR factorization of A.
[m n] = size(A);
```

which will automatically get the size of A, where m is the number of columns, and n is the number of rows. We want to write our own QR-factorization so that we understand what's going on "under the hood", but note that MATLAB has a built-in QR-factorization as well. Use it to double-check your code. Learn more about it like this:

```
>> help qr
```

3.1. **Test your QR code.** Use the following the QR factorization

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} = QR = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{6}} & \frac{1}{\sqrt{3}} \\ 0 & \frac{2}{\sqrt{6}} & \frac{1}{\sqrt{3}} \end{bmatrix} \begin{bmatrix} \frac{2}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & \frac{3}{\sqrt{6}} & \frac{1}{\sqrt{6}} \\ 0 & 0 & \frac{2}{\sqrt{3}} \end{bmatrix}$$

to check your code. You can use the 2-norm of a matrix in MATLAB to check your results like this:

```
>> norm(Q_exact - Q_computed)
>> norm(R_exact - R_computed)
```

The resulting error should be very small, e.g., not more than 10^{-10}. Call your test code `testMyQR.m`.

3.2. **Combine both parts to solve a linear system.** Now we can put both parts together to solve a general system. (Note that in Matlab, the adjoint of a matrix Q, which we write in class as Q^*, is computed in Matlab by writing Q'.) Call the code `linearQRsolver.m`. Try something like this:

```
clear all; close all;
% Solve Ax = b by QR factorization

m = 10; % Set the size of the matrix
% Create a random matrix:
A = rand(m,m)-0.5;

x_exact = rand(m,1);
% Create a "b" from the known exact solution:
b = A*x_exact;

[Q R] = myQR(A);
y = Q'*b;  % Solve Qy = b
x = upperTri(R,y,m); % Solve Rx = y.

% Use the 2-norm to check if the solutions are close:
error = norm(x - x_exact);
display(sprintf('The error is %g with size %d.',error,m));
```

4. Instructions for turning in the project

(1) **Make sure your code runs!** Also, try as hard as possible to clear our the warnings and errors. You can see these by hovering over the little orange and red marks in Matlab's scroll bar (just to the right of your m-file). When they are all clear, a green box will appear at the top of your scroll bar. Strictly speaking, you don't have to have a green box to turn in your code, but trying for it is a good idea. To get credit for the project, your code must run!
(2) **Properly indent your code.** To do this, on all your $*.m$ files, **hit `CTRL+A` (to select all the text) and then `CTRL+I`** (to get your code properly indented). If you skip this step, I will ask you to resubmit your code.
(3) Send me a very brief email from your university account with:
    (a) The **title:** Math 447 Project 1 Submission.
    (b) Your **name**, and the names of any collaborators.
    (c) Your $*.m$ **files** attached.
    (d) ***Brief* instructions on how to run your code** (only if necessary, I know how to click "play (▶)", and so on).
    (e) **Nothing else.** If you have a question for me, send it in a separate email with a different title.
(4) Just to be clear, the codes you should submit for this project are:
    (a) `upperTri.m`
    (b) `testUpperTri.m`
    (c) `myQR.m`
    (d) `testMyQR.m`
    (e) `linearQRsolver.m`

Please send all the code in a **single email**, not compressed (i.e., don't zip it or anything).

4.1. **Collaboration.** It is OK to work with somebody else, but if you do so, **you must state it clearly in your submission email**. In general, work together to get ideas and check each other's code, but write the code yourself. This is a great way to learn. Turning in somebody else's work (whether another classmate's, something you found online, etc.) will be dealt with according to the university's academic dishonesty policy. Plus, you will miss out on learning some awesome computing skills, which would be no fun, and getting a computer to solve hard matrix problems should be fun! (I guess it is no fun for the computer, but computers are machines and have no emotions, so we probably shouldn't feel bad about making them do our computations.)

Good luck and have fun!