pset1

September 7, 2017

1 18.06 pset 1

1.1 (due Wednesday 2/15 at 11am)

This problem set is in the form of a Julia *notebook* (using the Jupyter/IJulia browser-based interface to interactive programming). We will be using the Julia language throughout the term for simple computational explorations — practical linear algebra is not about hand computations!

You can run this without installing anything by logging in at JuliaBox. Just download the notebook file (a `.ipynb` file) by clicking the download icon at the upper right, then drag it onto the JuliaBox dashboard to upload it there.

Some of the problems are pencil-and-paper (we just happen to use the notebook to describe them), and some of them require you to run the code in the notebook to see what happens and then explain it. To **run the code** in an input cell, **just click on the cell and then type shift-return**; see also the "Help" menu in the notebook. When you submit your pset, just turn in a description and explanation of your results (no need to turn in a printout of the notebook).

1.2 Problem 1

The following code multiplies two random *lower-triangular matrices* (matrices whose entries are *zero above the diagonal*).

- What do you observe about the result?
- Explain why this always happens when one multiplies lower-triangular matrices (of any size).

```
In [ ]: L₁ = Matrix(LowerTriangular(rand(-9:9, 5,5)))
```

```
In [ ]: L₂ = Matrix(LowerTriangular(rand(-9:9, 5,5)))
```

```
In [ ]: L₁ * L₂
```

1.3 Problem 2

In this problem, we will see what happens when we think of a matrix as consisting of "blocks" that themselves are matrices ("submatrices"). In particular, we will compute the product:

$$M = XY = \underbrace{\begin{pmatrix} A & B \\ C & D \end{pmatrix}}_{X} \underbrace{\begin{pmatrix} E & F \\ G & H \end{pmatrix}}_{Y} = \begin{pmatrix} M_1 & M_2 \\ M_3 & M_4 \end{pmatrix}$$

where A, B, and so on are 2×2 submatrices of X and Y.

- The goal is to figure out how to write the entries of M in terms of matrix operations on the submatrices. In particular, if M_1 is the upper-left 2×2 submatrix of M, can we write a formula for this in terms of matrix operations on A, B and so on?

You should figure out a formula and then **try it out on** the randomly generated matrices below to see whether your formula works:

```
In [ ]: # make random 2x2 submatrices of X (with entries in -9...9)
        A = rand(-9:9,2,2)
        B = rand(-9:9,2,2)
        C = rand(-9:9,2,2)
        D = rand(-9:9,2,2)
        X = [ A B
              C D ]
```

```
In [ ]: # make random 2x2 submatrices of Y (with entries in -9...9)
        E = rand(-9:9,2,2)
        F = rand(-9:9,2,2)
        G = rand(-9:9,2,2)
        H = rand(-9:9,2,2)
        Y = [ E F
              G H ]
```

```
In [ ]: # compute the matrix M from the product XY:
        M = X * Y
```

```
In [ ]: M₁ = M[1:2, 1:2] # this is the upper-left 2x2 submatrix of M
```

Now, can you figure out a formula for M_1 in terms of matrix operations on the submatrices of M? For example, is it $A + CF - H$?

```
In [ ]: A + C*F - H # wrong formula -- fix this!
```

Nope, that doesn't match M_1. Figure out the correct formula (don't just try things at random...it might help to make a diagram of a row × column operation in computing M and see what submatrices that involves). Try out your formula in Julia and verify that it works.

1.4 Problem 3

In this problem, you will do something *like* standard Gaussian elimination, but not in quite the usual way.

Suppose we want to solve $Ax = b$ where

$$A = \begin{pmatrix} 1 & 6 & -3 \\ -2 & 3 & 4 \\ 1 & 0 & -2 \end{pmatrix}, b = \begin{pmatrix} 7 \\ 3 \\ 0 \end{pmatrix}.$$

Normally, with Gaussian elimination, you would convert A to upper-triangular form U, performing the same row operations on b to get c, and then finally solve $Ux = c$ for x by backsubstitution (starting from the last equation and working upwards).

- **Instead, for this problem,** you should convert the $Ax = b$ to the form $Lx = d$ where L is **lower triangular** (zero *above* the diagonal). Find L, find d, and then use this $Lx = d$ equation to solve for x.

For comparison, we can solve the same equation in Julia by `x = A \ b`. This is useful as a check to make sure that you got the correct answer for x in the end:

```
In [ ]: A = [ 1  6 -3
             -2  3  4
              1  0 -2 ]
```

```
In [ ]: b = [7, 3, 0]
```

```
In [ ]: x = A \ b
```

1.5 Problem 4

In class, we went over standard Gaussian elimination: you subtract rows of a matrix A, one by one, to bring it into upper-triangular form. Sometimes, if we encounter a zero pivot, we can swap rows in order to get a nonzero pivot. (If we can't do this, then the equations are *singular* and may have no solution.)

In principle, as long as we never encounter a zero pivot, this procedure will always work. In practice, however, if we apply the procedure blindly, we may get disastrous results due to **rounding errors**: a computer, a calculator, or (in olden days) a human doing hand calculation will usually only keep **a fixed number of significant digits** and will discard additional digits (*round*) during calculations.

Apply Gaussian elimination to solve the following $Ax = b$ system of equations:

$$A = \begin{pmatrix} 10^{-20} & 1 \\ 1 & 1 \end{pmatrix}, b = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

That is, convert A to upper-triangular form as usual, do the same row operations on b, and solve the resulting triangular system for x.

- What is the *exact* solution x?

- Re-do the Gaussian elimination, but *round* the result of **every** arithmetic operation ($+$, $-$, $\times$, $/$) to *16 significant digits*. What *approximate solution* $\tilde{x}$ will you get? (For example, $2 + 10^{-20} \approx 2$ if you round to 16 significant digits.) How close is it to the exact solution x?
- Do the same thing (round each operation to 16 digits), but first *swap the first and second rows of the equation* to **maximize the magnitude of the pivot**. (This is called partial pivoting.) What is the new approximate solution, and how close is it to the exact x?

(It turns out that *computer arithmetic* ordinarily rounds to about 15–16 digits, so this kind of concern is *very* important when people write computer programs to do linear algebra.)

For comparison, the Julia code below implements naive Gaussian elimination (no row re-ordering) and backsubstitution. Because this is using standard double precision computer arithmetic, it rounds to about 15–16 decimal digits (technically, 53 binary digits), so its results should be very similar to your results above. (The following code is **only for informational purposes**; you don't need it to answer the questions above.)

```
In [ ]: """
            naive_gauss(A)

        Given a matrix `A`, performs Gaussian elimination to convert
        `A` into an upper-triangular matrix `U`, and returns the matrix `U`.

        This implementation is "naive" because it *never re-orders the rows*.
        (It will obviously fail if a zero pivot is encountered.)
        """
        function naive_gauss(A)
            m = size(A,1) # number of rows
            U = copy(A)
            for j = 1:m   # loop over m columns
                for i = j+1:m   # loop over rows below the pivot row j
                    # subtract a multiple of the pivot row (j)
                    # from the current row (i) to cancel U[i,j] = U[U+1D62][U+2C7C]:
                    U[i,:] = U[i,:] - U[j,:] * U[i,j]/U[j,j]
                end
            end
            return U
        end

        """
```

```
    backsubstitution(U, c)

Given an upper-triangular matrix ‘U‘, return the solution ‘x‘ to ‘U*x=c‘ by
the backsubstitution algorithm.
"""
function backsubstitution(U, c)
    m = size(U,1)  # number of rows
    x = similar(c, typeof(c[1]/U[1,1])) # allocate the solution vector
    for i = m:-1:1 # loop over the rows from bottom to top
        r = c[i]
        for k = i+1:m
            r = r - U[i,k]*x[k]
        end
        x[i] = r / U[i,i]
    end
    return x
end
```

Let's perform naive Gaussian elimination (no row re-ordering) on the matrix A from above. We'll augment it with an extra column containing the vector b, so that the same row operations are performed on b:

```
In [ ]: A = [1e-20 1
              1     1]
        b = [1,0]
        U = naive_gauss([A b])
```

Now, let's perform backsubstitution to solve $Ux = c$ (where U is the first two columns of the augmented `U` matrix returned by `naive_gauss`, and `c` is the last column):

```
In [ ]: backsubstitution(U[:,1:2], U[:,3])
```

In comparison, the built-in \ solver is a little more clever, and may come up with a different answer:

```
In [ ]: A \ b
```