pset1-sol

September 7, 2017

1 18.06 pset 1 solutions

1.1 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 [1]: L1 = Matrix(LowerTriangular(rand(-9:9, 5,5)))

Out[1]: 5x5 Array{Int64,2}:
         -7   0   0  0  0
         -5   8   0  0  0
         -3   5  -2  0  0
          3   4  -3  7  0
         -9  -3   0  0  2

In [2]: L2 = Matrix(LowerTriangular(rand(-9:9, 5,5)))

Out[2]: 5x5 Array{Int64,2}:
         -7   0   0  0  0
         -9   1   0  0  0
         -8  -9   9  0  0
          2  -6   5  3  0
         -1   5  -2  5  1

In [3]: L1 * L2

Out[3]: 5x5 Array{Int64,2}:
          49    0    0   0  0
         -37    8    0   0  0
          -8   23  -18   0  0
         -19  -11    8  21  0
          88    7   -4  10  2
```

1.2 Solution

The product of two lower triangular matrices is always lower triangular. In fact if L_1 and L_2 are lower triangular the (i, j) component of L_1L_2 is obtaining by the dot product of the i-th row of L_1 and the j-th column of L_2. But when $i < j$ (= entries *above* the diagonal), the dot product is always 0, since a nonzero component of the i-th row of L_1 is always paired with a zero component of the j-th column of L_2 and vice versa. In formulas:

$$(L_1L_2)_{ij} = \sum_{k=1}^{n}(L_1)_{ik}(L_2)_{kj} = \sum_{k=1}^{j-1}(L_1)_{ik}(L_2)_{kj} + \sum_{k=j}^{n}(L_1)_{ik}(L_2)_{kj} = \sum_{k=1}^{j-1}(L_1)_{ik}\cdot 0 + \sum_{k=j}^{n} 0\cdot (L_2)_{kj} = 0$$

In the $\sum_{k=1}^{j-1}$, we have $(L_2)_{kj} = 0$ because $k < j$, which corresponds to an entry of L_2 above the diagonal. In the $\sum_{k=j}^{n}$, we have $(L_1)_{ik} = 0$ if $i < j$ because $k \geq j > i$ and hence $(L_1)_{ik}$ is an entry of L_1 above the diagonal.

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:

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

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

- 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** a randomly generated matrix below to see whether your formula works:

```
In [4]: # make random 2x2 submatrices
        A = rand(-9:9,2,2)
        B = rand(-9:9,2,2)
        C = rand(-9:9,2,2)
        D = rand(-9:9,2,2)
        E = rand(-9:9,2,2)
        F = rand(-9:9,2,2)
        G = rand(-9:9,2,2)
        H = rand(-9:9,2,2)

        # compute the matrix M from the product:
        M = [ A B
              C D ] * [ E F
                        G H ]
```

```
Out[4]: 4x4 Array{Int64,2}:
         -82   28   33    9
         -49  -64   66  -56
          13   59  -46   90
         -36  -92   73  -71
```

```
In [5]: M₁ = M[1:2, 1:2] # this is the upper-left 2x2 submatrix of M
```

```
Out[5]: 2x2 Array{Int64,2}:
         -82   28
         -49  -64
```

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

```
Out[6]: 2x2 Array{Int64,2}:
         41   14
         31  -43
```

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 Solution

The correct formula is

```
In [7]: A*E+B*G
```

```
Out[7]: 2x2 Array{Int64,2}:
         -82   28
         -49  -64
```

This can be seen by looking at the definition of matrix multiplication: to compute the entry in the (i, j)-th position we compute the dot product of the i-th row of the first matrix by the j-th column of the second matrix. This is the sum of the dot product of the i-th row of A by the j-th column of E with the dot product of the i-th row of B with the j-th column of G. More details can be found on the textbook section on block matrices and block multiplication at page 74.

1.5 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

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

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

```
Out[8]: 3x3 Array{Int64,2}:
          1  6  -3
         -2  3   4
          1  0  -2
```

```
In [9]: b = [7, 3, 0]
```

```
Out[9]: 3-element Array{Int64,1}:
         7
         3
         0
```

```
In [10]: x = A \ b

Out[10]: 3-element Array{Float64,1}:
          -2.0
           1.0
          -1.0
```

1.6 Solution

We start with the matrix

$$(A|b) = \left(\begin{array}{ccc|c} 1 & 6 & -3 & 7 \\ -2 & 3 & 4 & 3 \\ 1 & 0 & -2 & 0 \end{array}\right)$$

In order to reduce to lower triangular form, add twice the third row to the second and remove 3/2 times the third row from the first

$$\left(\begin{array}{ccc|c} -\frac{1}{2} & 6 & 0 & 7 \\ 0 & 3 & 0 & 3 \\ 1 & 0 & -2 & 0 \end{array}\right)$$

Finally we remove twice the second row from the first

$$\left(\begin{array}{ccc|c} -\frac{1}{2} & 0 & 0 & 1 \\ 0 & 3 & 0 & 3 \\ 1 & 0 & -2 & 0 \end{array}\right)$$

So we have transformed the system to the equivalent one $Lx = d$ where

$$L = \begin{pmatrix} -\frac{1}{2} & 0 & 0 \\ 0 & 3 & 0 \\ 1 & 0 & -2 \end{pmatrix} \qquad d = \begin{pmatrix} 1 \\ 3 \\ 0 \end{pmatrix} .$$

And we can solve it via backsubstitution: we start with the first equation

$$-\frac{1}{2}x_1 = 1 \Rightarrow x_1 = -2$$

$$3x_2 = x_2 \Rightarrow x_2 = 1$$

$$x_1 - 2x_2 = 0 \Rightarrow x_2 = \frac{1}{2}x_1 = -1$$

Finally, the solution is

$$\begin{pmatrix} -2 & 1 & -1 \end{pmatrix} .$$

1.7 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?
- If you *round* the result of each operation 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.)

1.8 Solution

Let us use the Gauss elimination algorithm on the matrix

$$\left(\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 1 & 1 & 0 \end{array}\right) \Rightarrow \left(\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 0 & 1-10^{20} & -10^{20} \end{array}\right)$$

So the exact solution is

$$\begin{pmatrix} -\frac{10^{20}}{10^{20}-1} & \frac{10^{20}}{10^{20}-1} \end{pmatrix} \approx \begin{pmatrix} -1 & 1 \end{pmatrix} .$$

However if we approximate the system after row reduction we get

$$\left(\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 0 & -10^{20} & -10^{20} \end{array}\right)$$

and the solution to the approximate system is

$$\begin{pmatrix} 0 & 1 \end{pmatrix}$$

which is very different from the exact solution. This is exactly the solution that `naive_gauss` produces below, because Julia (like most computer programs) performs arithmetic rounded to about 16 decimal places ("double precision").

If we swapped the rows instead we would get

$$\left(\begin{array}{cc|c} 1 & 1 & 0 \\ 10^{-20} & 1 & 1 \end{array}\right) \Rightarrow \left(\begin{array}{cc|c} 1 & 1 & 0 \\ 0 & 1-10^{-20} & 1 \end{array}\right) \approx \left(\begin{array}{cc|c} 1 & 1 & 0 \\ 0 & 1 & 1 \end{array}\right) .$$

The solution to this approximate system is

$$\begin{pmatrix} -1 & 1 \end{pmatrix}$$

which is very close to the solution to the exact system.

Note that `A \ b` in Julia produces this (nearly) correct solution. In fact, the Julia solver algorithm (like all serious numerical linear algebra programs) swaps the rows (for *every* column step of Gaussian elimination) so as to obtain the pivot with the largest possible magnitude. This is called *partial pivoting*, and is essential for accuracy. See also the end of section 2.7 in the textbook, and section 11.1.

In [11]:
```
"""
    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 pivit 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
```

Out[11]: backsubstitution (generic function with 1 method)

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 [12]:
```
A = [1e-20 1
      1     1]
b = [1,0]
U = naive_gauss([A b])
```

Out[12]:
```
2x3 Array{Float64,2}:
 1.0e-20   1.0       1.0
 0.0      -1.0e20  -1.0e20
```

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 [13]:
```
backsubstitution(U[:,1:2], U[:,3])
```

Out[13]:
```
2-element Array{Float64,1}:
 0.0
 1.0
```

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

```
In [14]: A \ b

Out[14]: 2-element Array{Float64,1}:
          -1.0
           1.0
```