MATH 170A (WI21, Dumitriu)

Homework 1

Jiahong Long
[redacted]

14 January 2021

1. Prove that if A^{-1} exists, then there can be no nonzero y for which $Ay = 0$.

   We proceed towards a direct proof.

   Since A is invertible, it must be square. Let n denote the number of rows (ie. also the number of columns) in A. The linear equation $Ay = 0$ represents the system of equations

   $$\begin{aligned} a_{11}y_1 + a_{12}y_2 + \ldots + a_{1n}y_n &= 0 \\ a_{21}y_1 + a_{22}y_2 + \ldots + a_{2n}y_n &= 0 \\ \vdots \quad & \quad \vdots \\ a_{n1}y_1 + a_{n2}y_2 + \ldots + a_{nn}y_n &= 0 \end{aligned}$$

   Therefore, we can solve the matrix by augmenting A with the zero column vector:

   $$A' = \left[\begin{array}{c|c} A & 0 \end{array}\right] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \ldots & a_{1n} & 0 \\ a_{21} & a_{22} & \ldots & a_{2n} & 0 \\ \vdots & & \ddots & & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} & 0 \end{array}\right]$$

   Because A is invertible, we know that it is full rank. It follows, therefore, that the reduced row-echelon form of the augmented matrix A' will have n pivot positions on the diagonal populated with nonzero numbers. For $i = 1, 2, \ldots, n$, we denote the ith pivot position a_i:

   $$A' \sim \left[\begin{array}{cccc|c} a_1 & 0 & \ldots & 0 & 0 \\ 0 & a_2 & \ldots & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \ldots & a_n & 0 \end{array}\right]$$

   It follows, then that for each row i, we have the linear equation

   $$a_i y_i = 0$$

   where $a_i \neq 0$. Then y_i must be zero for all i, and therefore the only solution to $Ay = 0$ is the trivial solution. ■

2. Set up, but do not solve, a linear equation to numerically solve the ODE

   $$u'' - 3u = 3, \qquad \text{for all } x \in (0,1), \qquad \text{with } u(0) = u(1) = 0$$

   using a partition of $[0, 1]$ with $m = 5$ and $h = 1/5$. You will need to use the approximation formula

   $$u'' \approx \frac{u(x+h) - 2u(x) + u(x-h)}{h^2}$$

We seek to approximate the solution using the sequence

$$u_i = u_1, u_2, u_3, u_4$$

where

$$u_i \approx u(0 + i\frac{1}{5})$$

The Taylor polynomial expansion yields the approximation $u'' \approx \frac{u(x+h)-2u(x)+u(x-h)}{h^2}$. Expanding this, we can obtain an approximation of u'' as a linear combination of u_{i-1}, u_i, and u_{i+1}:

$$\begin{aligned} u'' &= \frac{1}{h^2}\left(u_{i+1} - 2u_i + u_{i-1}\right) \\ &= u_{i-1}\left(\frac{1}{h^2}\right) + u_i\left(\frac{-2}{h^2}\right) + u_{i+1}\left(\frac{1}{h^2}\right) \end{aligned}$$

Then the ODE may be approximated:

$$u''(x+ih) \approx u_{i-1}\left(\frac{1}{h^2}\right) + u_i\left(\frac{-2}{h^2}\right) + u_{i+1}\left(\frac{1}{h^2}\right)$$

$$u(x+ih) \approx u_i$$

hence:

$$u_{i-1}\left(\frac{1}{h^2}\right) + u_i\left(\frac{-2}{h^2}\right) + u_{i+1}\left(\frac{1}{h^2}\right) + u_i = 3$$

$$u_{i-1}\left(\frac{1}{h^2}\right) + u_i\left(\frac{h^2-2}{h^2}\right) + u_{i+1}\left(\frac{1}{h^2}\right) = 3$$

where

$$\frac{h^2-2}{h^2} = \frac{\frac{1}{25}-2}{\frac{1}{25}} = -49$$

$$\frac{1}{h^2} = 25$$

and we can write the linear system

$$\overbrace{\begin{bmatrix} -49 & 25 & 0 & 0 \\ 25 & -49 & 25 & 0 \\ 0 & 25 & -49 & 25 \\ 0 & 0 & 25 & -49 \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix}}^{x} = \overbrace{\begin{bmatrix} 3 \\ 3 \\ 3 \\ 3 \end{bmatrix}}^{b}$$

3. Consider a system

$$Ux = b$$

where U is upper triangular.

(a) Modify the solution we developed in class for lower triangular systems to upper triangular systems (in the upper triangular case, this process is called backward substitution). In particular, find the formula to compute x_k given $x_{k+1}, \ldots, x_n$.

We represent our system:

$$\overbrace{\begin{bmatrix} a_{11} & a_{12} & a_{13} & \ldots & a_{1n} \\ 0 & a_{22} & a_{23} & \ldots & a_{2n} \\ 0 & 0 & a_{33} & \ldots & a_{3n} \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & 0 & 0 & a_{nn} \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}}^{x} = \overbrace{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}}^{b}$$

Then, to compute x_k given $x_{k+1}, \ldots, x_n$, we can use the expression

$$\boxed{x_k = \frac{b_k - \left(\sum_{i=k+1}^{n} a_{ki} x_i\right)}{a_{kk}}}$$

(b) Using the model code for `lowertriangsolve.m`, write a MATLAB script that solves upper triangular systems.

```
function y = uppertriangsolve(A, b)

% Check boundaries.
if size(A, 1) ~= size(A, 2)
    error("Matrix isn't upper triangular.")
end

y = b;
n = size(A, 1);
for i = fliplr(1:n)
    for k = ((i+1):n)
        y(i) = y(i) - A(i, k)*b(k);
    end
    if A(i, i) == 0
        error("Matrix is not invertible.")
    end
    y(i) = y(i) / A(i, i)
end
```

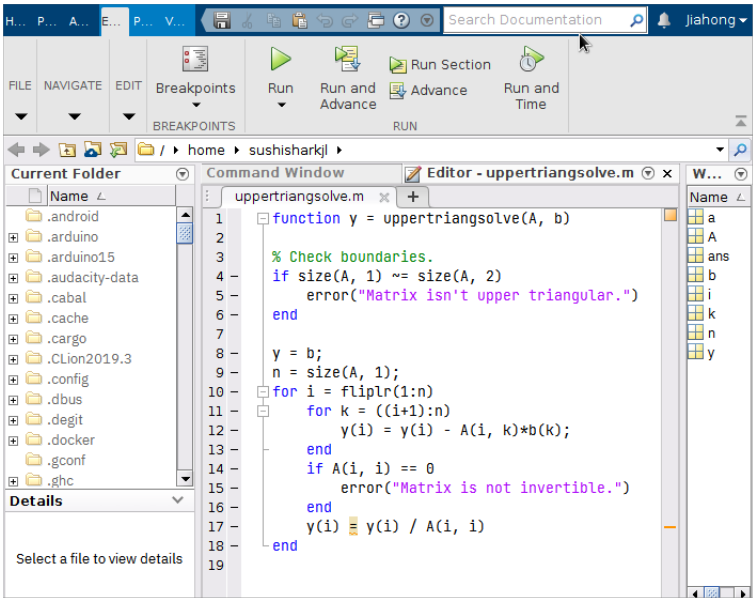

Figure 1: Script solving upper-triangular systems

4. Prove that if U is an invertible upper triangular matrix, then U^{-1} is also upper triangular.

We proceed towards a direct proof.

Since U is invertible, it must also be square; thus let n denote the number of rows (ie. the number of columns) in U. In addition, being invertible, U is also non-singular and therefore has a nonzero determinant. Then, there are no non-zero entries on the diagonal. We can write, by definition

$$UU^{-1} = I_n$$

We can further decompose U^{-1} into a block matrix:

$$U \begin{bmatrix} | & | & | & & | \\ u_1^{-1} & u_2^{-1} & u_3^{-1} & \dots & u_n^{-1} \\ | & | & | & & | \end{bmatrix} = I_n = \begin{bmatrix} | & | & | & & | \\ e_1 & e_2 & e_3 & \dots & e_n \\ | & | & | & & | \end{bmatrix}$$

where u_i^{-1} denotes the ith column of the inverse and e_i denotes the ith column unit vector. Then it follows that for all $i \in [1, n]$ we have the system

$$\overbrace{\begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & u_{in} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{bmatrix}}^{U} \overbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix}}^{u_i^{-1}} = \overbrace{\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}}^{e_i}$$

As U is upper triangular, we can use back-substitution. Recall that U is invertible, and as such (as it is also triangular) it has no 0 entries on the diagonal. We will then show that u_i^{-1} is zero for entries x_k for $k > i$ by induction, from which it immediately follows that U^{-1} is upper triangular.

- *Base case.*
  Take note of the equation represented by the last row:
  $$u_{nn}x_n = 0$$
  Since U is triangular and invertible, no entry on the diagonal can be zero. Hence, $u_{nn} \neq 0$. It follows that x_n must equal zero, which is as required.
- *Recursive case.*
  We will show x_k is zero, where $k \in [i+1, n-1]$. The equation represented by the last row is given by
  $$u_{kk}x_k + u_{k,k+1}x_{k+1} + \ldots + u_{nn}x_n = 0$$
  Assume, as the induction hypothesis, that all entries x_{k+1} through x_n are zero. It follows that the equation degenerates:
  $$u_{kk}x_k = 0$$
  and since U is triangular and invertible, u_{kk} is nonzero. Then x_k is zero.

Thus we have shown that in the ith column of U^{-1}, entries $i+1$ through n are zero. It follows as an immediate consequence that U^{-1} is upper triangular. ■

5. Using basic programming, write two MATLAB functions that both take as an input
   - $n \times n$ matrix A,
   - $n \times n$ matrix B,
   - $n \times 1$ matrix (column vector) x.

   Have the first function compute ABx through $(AB)x$, and the second through $A(Bx)$. Have both functions output the number of flops used.

   (a) Take a screenshot of the first function.

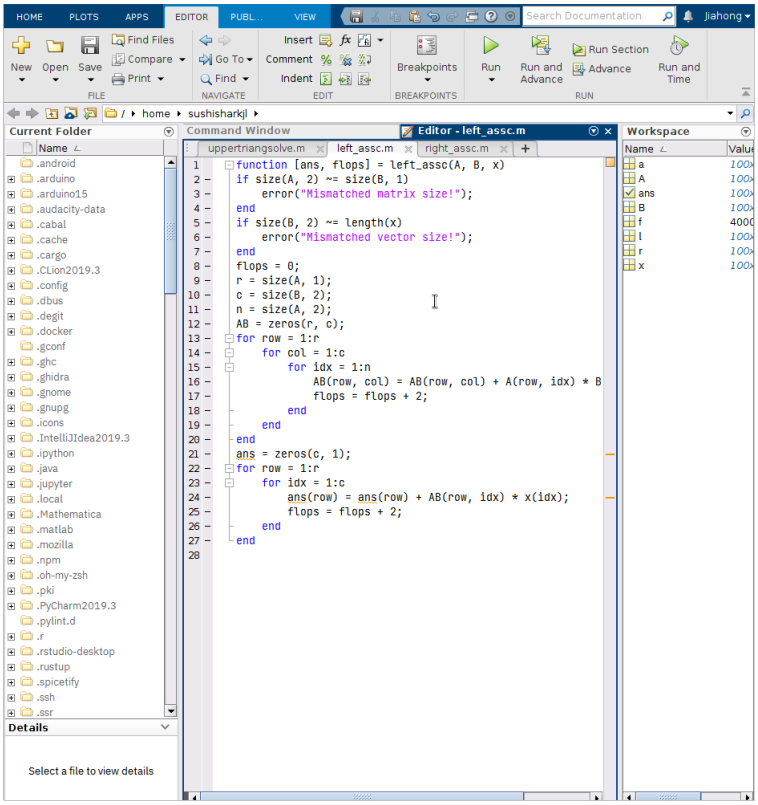

Figure 2: First function calculating ABx using left-associativity

We reproduce the code here for legibility:

```
function [ABx, flops] = left_assc(A, B, x)
if size(A, 2) ~= size(B, 1)
    error("Mismatched matrix size!");
end
if size(B, 2) ~= length(x)
    error("Mismatched vector size!");
end
flops = 0;
r = size(A, 1);
c = size(B, 2);
n = size(A, 2);
AB = zeros(r, c);
for row = 1:r
    for col = 1:c
        for idx = 1:n
            AB(row, col) = AB(row, col) + A(row, idx) * B(idx, col);
            flops = flops + 2;
        end
    end
end
ABx = zeros(c, 1);
for row = 1:r
    for idx = 1:c
```

```
            ABx(row) = ABx(row) + AB(row, idx) * x(idx);
            flops = flops + 2;
        end
    end
```

(b) Take a screenshot of the second function.

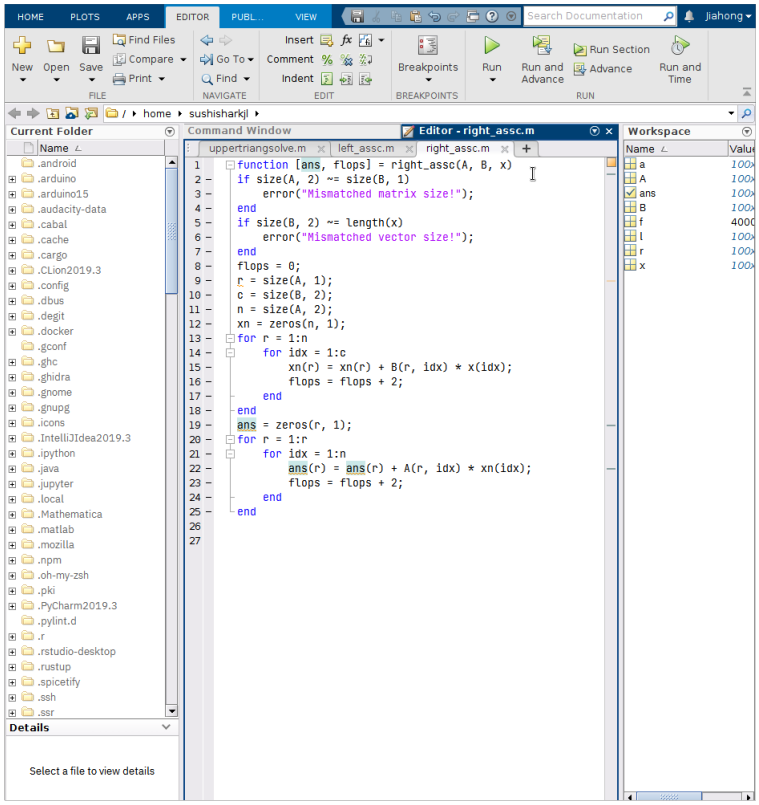

Figure 3: First function calculating ABx using right-associativity

We reproduce the code here for legibility:

```
function [ABx, flops] = right_assc(A, B, x)
if size(A, 2) ~= size(B, 1)
    error("Mismatched matrix size!");
end
if size(B, 2) ~= length(x)
    error("Mismatched vector size!");
end
flops = 0;
r = size(A, 1);
c = size(B, 2);
n = size(A, 2);
Bx = zeros(n, 1);
for r = 1:n
    for idx = 1:c
        Bx(r) = Bx(r) + B(r, idx) * x(idx);
```

```
            flops = flops + 2;
        end
    end
    ABx = zeros(r, 1);
    for r = 1:r
        for idx = 1:n
            ABx(r) = ABx(r) + A(r, idx) * Bx(idx);
            flops = flops + 2;
        end
    end
```

(c) Apply both algorithms for $n = 100, 200, 400, 800$. Which approach uses fewer flops?

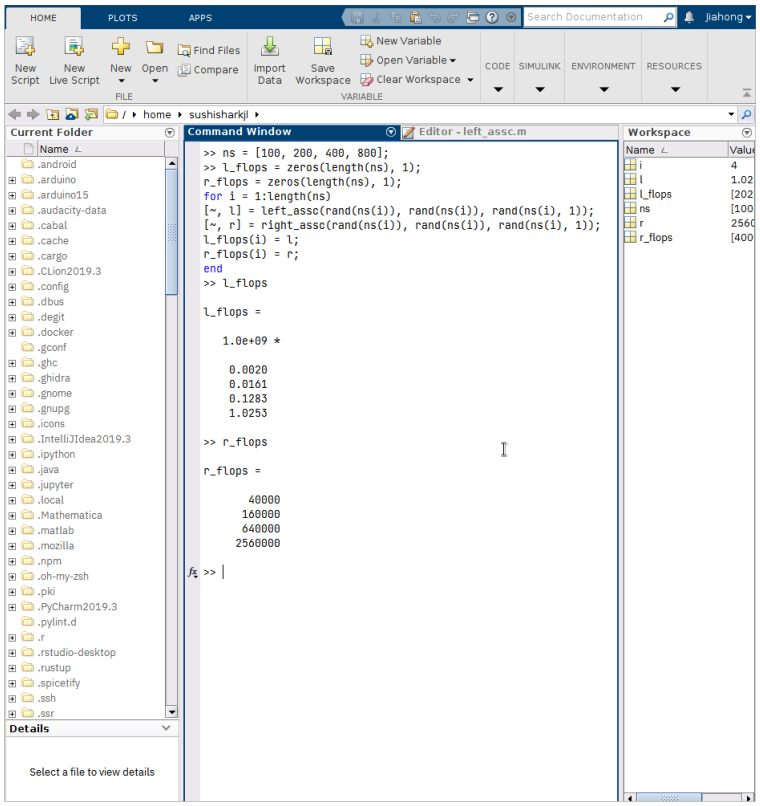

Figure 4: Comparing total FLOPS for left- and right-associative multiplication

We reproduce the code here for legibility:

```
ns = [100, 200, 400, 800];
l_flops = zeros(length(ns), 1);
r_flops = zeros(length(ns), 1);
for i = 1: length(ns)
    [~, l] = left_assc(rand(ns(i)), rand(ns(i)), rand(ns(i), 1));
    [~, r] = right_assc(rand(ns(i)), rand(ns(i)), rand(ns(i), 1));
    l_flops(i) = l;
    r_flops(i) = r;
end
```

The results are:

Input size	**Left**	**Right**
100	$2,000,000$	$40,000$
200	$16,100,000$	$160,000$
400	$128,300,000$	$640,000$
800	$1,025,300,000$	$2,560,000$

Clearly, right-associative multiplication is faster, and consumes fewer flops.