EA1 Homework Program 5: Row Echelon Reduction

Due Friday, Oct. 27, 2023, at 6:00am

Please submit everything on Canvas as a single file.

In this assignment, you will write a library of MATLAB functions. Each function will perform a single row operation on a matrix. Then, you will combine your functions into a program to perform row echelon reduction (to reduced echelon form) on a matrix. Follow the steps below.

Suggestion: make a separate file for each function, so you can easily run the tests for each one. At the end, combine all 4 of your functions in one file, with the 3 row operation functions as helper functions. Make sure your final row reduction code works after you have combined all the functions!

1. Write a MATLAB function `exchange` that takes as its input arguments

   - a matrix `M`,
   - two numbers `row1` and `row2` that refer to rows of the matrix, and
   - a boolean (true/false) `verbose` that determines if the function prints to the screen,

   and that produces as a result the same matrix except with rows `row1` and `row2` exchanged. If `verbose` is true, then the function should print the operation it just performed. Otherwise, it should do it silently. If `verbose` is not given, it should default to true. Your function should

   - include some documentation at the start of the function to explain it,
   - check that the second and third arguments are integers between (and including) one and the number of rows in the matrix argument, and if not, return with an error message,
   - if `verbose` is true, tell the user which rows are getting exchanged using an `fprintf`.

   Note that `i` is an integer if `i == round(i)` is true. Test your program with the following commands (you do not have to include their output at the end of your file):

   ```
   exchange([9 6; 4 2; 2.2 3; 3.3 2], 4, 1)
   M = eye(6)
   M = exchange(M, 2, 6, false)
   exchange([9 6; 4 2; 2.2 3; 3.3 2], 1, 4.2)
   exchange([4 3 6 7; 2 3 9 10], 0, 2)
   ```

2. Write a MATLAB function `mult` that takes as its input arguments

   - a matrix `M`,
   - any nonzero number `d`,
   - a number `row` that refers to a row of the matrix, and
   - a boolean (true/false) `verbose` that determines if the function prints to the screen,

   and that produces as a result the same matrix except with row `row` multiplied by `d`. If `verbose` is true, then the function should print the operation it just performed. Otherwise, it should do it silently. If `verbose` is not given, it should default to true. Your function should

   - include some documentation at the start of the function to explain it,

- check that the second argument is not zero, otherwise return with an error message,
- check that the third input argument is an integer between (and including) one and the number of rows in the matrix argument, and if not, return with an error message,
- if `verbose` is true, tell the user which row is getting multiplied, and by what.

Test your program with the following commands (you do not have to include their output at the end of your file):

```
mult([9 6; 2 4; 2.2 3; 3.3 2], 1/4, 2)
mult([9 6; 2 4; 2.2 3; 3.3 2], 1/4, 1.1)
mult([9 6; 2 4; 2.2 3; 3.3 2], 1/3, 3, false)
mult([4 3 6 7; 2 3 9 10], 2, 3)
mult([4 3 6 7; 2 3 9 10], 0, 2)
```

3. Write a MATLAB function `add` that takes as its arguments

   - a matrix `M`,
   - any real number `r`,
   - two numbers `row1` and `row2` that refer to rows of the matrix, and
   - a boolean (true/false) `verbose` that determines if the function prints to the screen,

   and produces as a result the same matrix except with `r` times row `row1` added to row `row2`. If `verbose` is true, then the function should print the operation it just performed. Otherwise, it should do it silently. If `verbose` is not given, it should default to true. Your function should

   - include some documentation at the start of the function to explain it,
   - check that the third and fourth arguments are integers between (and including) one and the number of rows in the matrix argument, and if not, return with an error message,
   - if `verbose` is true, tell the user which row is getting multiplied, and by what.

   Test your program with the following commands (you do not have to include their output at the end of your file):

```
add([9 6; 2 4; 2.2 3; 3.3 2], -2.0, 2, 3)
add([9 6; 2 4; 2.2 3; 3.3 2], -2.0, 0, 3)
A = [1.0 2 3; 2.5 6 9]
A = add(A, -2.5, 2, 1)
A = add(A, -2.5, 2, 2, false)
add([9 6; 2 4; 2.2 3; 3.3 2], 0, 3, 4.4)
add([9 6; 2 4; 2.2 3; 3.3 2], -2, 3.1, 2)
```

4. You can use your library of functions `exchange`, `mult`, and `add` to reduce a matrix into reduced echelon form. In this part, you will write a function to do just this.

   Write a MATLAB function `reduce` that takes as its input argument a matrix `M` and a boolean `verbose`, and produces as a result the reduced echelon form of that matrix, as well as a row vector containing the numbers of the pivot columns (i.e. the same outputs as from the built-in `rref` function). If `verbose` is true, then every row operation should be printed on the screen. Otherwise, the printing should be suppressed. If the user does not input a value for `verbose`, it should default to true. Your function should include some documentation at the start to explain it.

In this assignment, we will use a special technique to help with numerical accuracy, which is called "partial pivoting". Without this technique, we can get unexpected results due purely to roundoff errors in the programming language, which is inescapable since machines have to store numbers to a given precision (as opposed to their theoretical, infinite forms in the case of irrational numbers). While partial pivoting does not avoid this potential issue, it minimizes its occurrence. The only difference between partial pivoting and the regular approach to row reduction is that instead of using *any* nonzero number as a pivot, you always choose the largest (in absolute value) number from the column. This will be explained more below.

The way to approach this program is to think in a very structured manner about the way that row reduction is performed. Here is an outline:

(a) For the first pivot, consider the entire matrix.
   i. Use the `max` command to find the maximum absolute value in each column, which should return a row vector. (Make sure to explicitly instruct MATLAB to find the max in each column! Think about the case when the matrix only has one row; MATLAB will find the max in that single row, which is not what you want.)
   ii. Use the `find` function to find the first nonzero value in this vector, which will be your first pivot column. (Think about what the column must look like if its maximum absolute value is zero.)
   iii. Using the second output of the `max` function, swap the row with the largest value into the first row (since the second output gives you the location of the largest element in each column).
   iv. Store the pivot column number in your vector keeping track of the pivots.
   v. Use the pivot to zero out all entries below it.

(b) Once you have found your first pivot, you should only consider the submatrix whose columns are to the right of the pivot column and the rows below the pivot row.
   i. Repeat the same procedure as with the first pivot, i.e. find the first nonzero column, find the maximum absolute value in that column, swap it into the first row, store the pivot column number, and zero out all entries below it.

(c) Keep on going until you have accounted for all of your pivots. This should all be performed in a loop. Once you have found all of the pivots, you should exit the loop.

(d) To get to reduced echelon form from regular echelon form, you should go through the pivot columns you have already found in the initial reduction, and make sure every pivot is scaled to one and the entries above it are reduced to zeros.

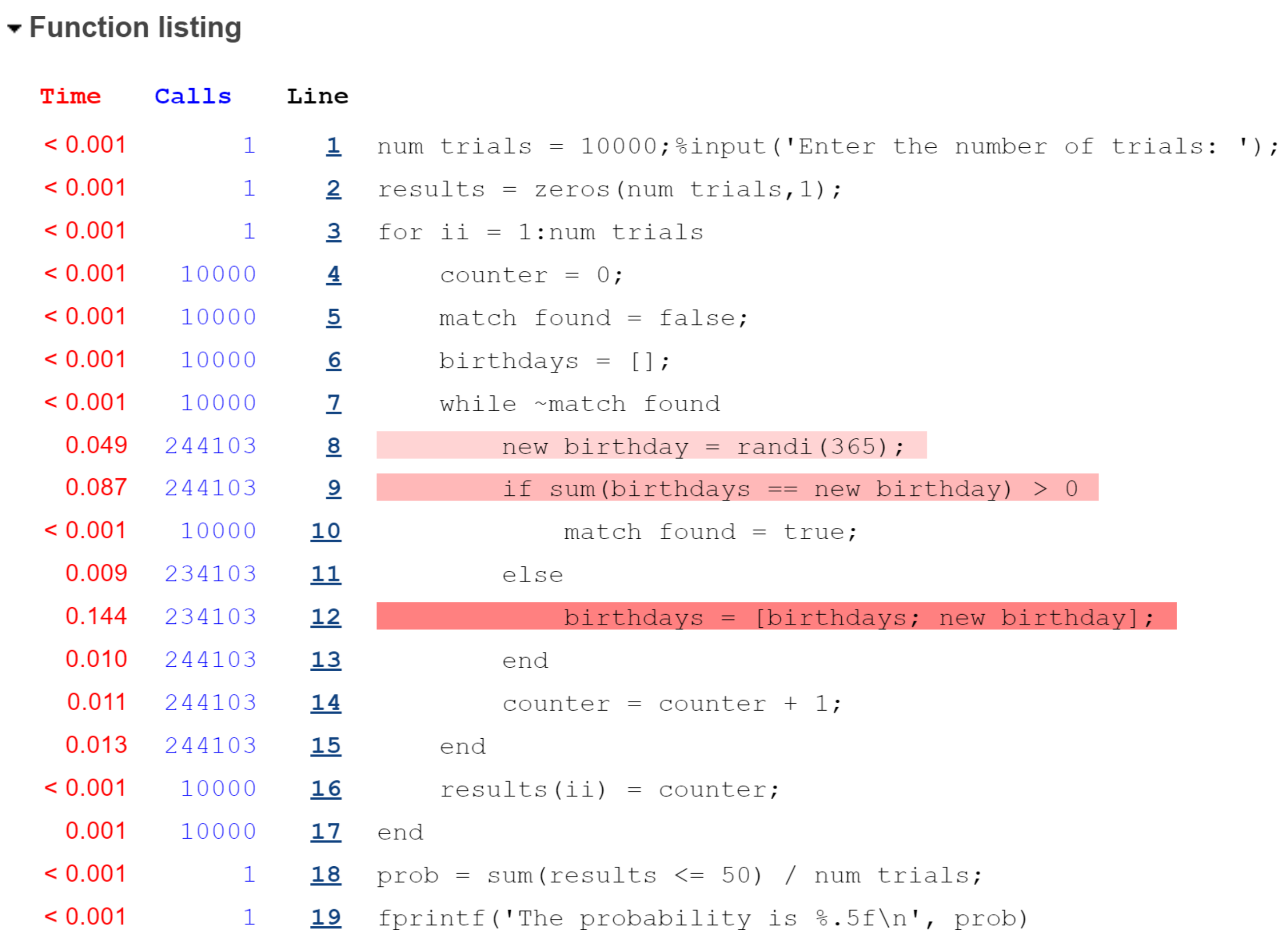
Following this kind of procedure should account for all possibilities. Below is an example run of the program. Note that there are some unnecessary operations (e.g. the second to last operation). These can be excluded by adding some more conditions to the code, but it is not wrong to include extra operations as long as you get to the correct result.

```
>> A = [0 0 0 0;1 0 0 1;0 1 3 3;3 0 0 2;6 0 0 4]

A =

     0     0     0     0
     1     0     0     1
     0     1     3     3
     3     0     0     2
     6     0     0     4

>> [R, piv] = reduce(A)
Exchanging rows 1 and 5
Adding -0.166667Row1 to Row2
Adding -0.500000Row1 to Row4
Exchanging rows 2 and 3
Multiplying row 3 by 3.000000
Adding -4.000000Row3 to Row1
Adding -3.000000Row3 to Row2
Multiplying row 2 by 1.000000
Multiplying row 1 by 0.166667

R =

     1     0     0     0
     0     1     3     0
     0     0     0     1
     0     0     0     0
     0     0     0     0


piv =

     1     2     4
```

Run your program with the following tests. **Include the results of these tests in a comment after your code. Make sure this shows the fprintf output from each row operation.**

```
A = randi([-5 5], 4, 10); [R, piv] = reduce(A)
A = randi([-5 5], 10, 4); [R, piv] = reduce(A)
A = [1 2 0 0 0;0 0 0 2 3;0 0 2 1 3]; [R, piv] = reduce(A)
a = [1 2 4 3 5]; A = [a;a;a;a]; [R, piv] = reduce(A)
```

5. One very important aspect of writing efficient code is optimization. We do not focus on optimization in this class, but it is still interesting to explore how your code performs. MATLAB has a nice utility for this called "Run and Time". To use this on a script, instead of pressing "Run" in the script, you press the dropdown and press "Run and Time". If you do this, MATLAB will generate a report showing how long each line of your code took to execute. An example of this is shown below for the last problem on the midterm. (Note that initially you will see another screen, but when you click on your file name, it should take you to the line-by-line breakdown.) However, as you know, you cannot just hit "Run" for a function. Instead, in the dropdown for "Run", there is a prompt for "type code to run". Here, you can type exactly what you would in the command window, and then when you hit "Run and Time", it will run this line of code, which allows you to call your function. Try calling "reduce(A)" using the last A matrix from the previous part, and write a brief comment about which line of your code is slowest and why.



▾ Function listing

Time	Calls	Line	
< 0.001	1	1	`num trials = 10000;%input('Enter the number of trials: ');`
< 0.001	1	2	`results = zeros(num trials,1);`
< 0.001	1	3	`for ii = 1:num trials`
< 0.001	10000	4	`    counter = 0;`
< 0.001	10000	5	`    match found = false;`
< 0.001	10000	6	`    birthdays = [];`
< 0.001	10000	7	`    while ~match found`
0.049	244103	8	`        new birthday = randi(365);`
0.087	244103	9	`        if sum(birthdays == new birthday) > 0`
< 0.001	10000	10	`            match found = true;`
0.009	234103	11	`        else`
0.144	234103	12	`            birthdays = [birthdays; new birthday];`
0.010	244103	13	`        end`
0.011	244103	14	`        counter = counter + 1;`
0.013	244103	15	`    end`
< 0.001	10000	16	`    results(ii) = counter;`
0.001	10000	17	`end`
< 0.001	1	18	`prob = sum(results <= 50) / num trials;`
< 0.001	1	19	`fprintf('The probability is %.5f\n', prob)`

6. As a final step, let's benchmark our code against the built-in `rref` function. In the command window, run the following code:

```
N = 10000;
rref_vec = zeros(N,1);
reduce_vec = zeros(N,1);
for ii = 1:N
    A = rand(randi(20), randi(20));
    tic;rref(A);rref_vec(ii) = toc;
    tic;reduce(A,false);reduce_vec(ii) = toc;
end
ratio = mean(rref_vec)/mean(reduce_vec);
fprintf('Our code is %.2f times faster than rref\n', ratio)
```

In a comment at the end of your code, briefly explain what this code is doing, and give the result of the fprintf statement at the end. As a benchmark, when I ran the code with N = 1e6 (though I recommend sticking with N=1e4), I got 6.87 times faster!

Submission Instructions
Your assignment should have the following format:

reduce.m

```
% Homework Program 5
%
% Name:
% Date:

function [M, piv] = reduce(M, verbose)

% Code for the reduce program

end %End of function reduce

function M = exchange(M, row1, row2, verbose)

% Code for the exchange operation here

end %End of function exchange

function M = mult(M, d, row, verbose)

% Code for the multiplication operation here

end %End of function mult

function M = add(M, r, row1, row2, verbose)

% Code for the add operation here

end %End of function add

% Your results as comments at the end of the file

% Your answers to Part 5 and Part 6
```