CSCI 2033: Elementary Computational Linear Algebra (Spring 2020)

Assignment 1 (100 points)

Due date: February 21st, 2019 11:59pm

In this assignment, you will implement MATLAB functions to perform row operations, compute the RREF of a matrix, and use it to solve a real-world problem that involves linear algebra, namely GPS localization.

For each function that you are asked to implement, you will need to complete the corresponding `.m` file with the same name that is already provided to you in the zip file. In the end, you will zip up all your complete `.m` files and upload the zip file to the assignment submission page on **Gradescope**.

In this and future assignments, you may not use any of MATLAB's built-in linear algebra functionality like `rref`, `inv`, or the linear solve function `A\b`, except where explicitly permitted. However, you may use the high-level array manipulation syntax like `A(i,:)` and `[A,B]`. See "Accessing Multiple Elements" and "Concatenating Matrices" in the MATLAB documentation for more information. However, you are allowed to call a function you have implemented in this assignment to use in the implementation of other functions for this assignment.

Note on plagiarism A submission with any indication of plagiarism will be directly reported to University. Copying others' solutions or letting another person copy your solutions will be penalized equally. Protect your code!

1 Submission Guidelines

You will submit a zip file that contains the following `.m` files to **Gradescope**. Your filename must be in this format: **Firstname_Lastname_ID_hw1_sol.zip** (please replace the name and ID accordingly). Failing to do so may result in points lost.

- `interchange.m`
- `scaling.m`
- `replacement.m`
- `my_rref.m`
- `gps2d.m`
- `gps3d.m`
- `solve.m`

The code should be stand-alone. No credit will be given if the function does not comply with the expected input and output.

Late submission policy: 25% off up to 24 hours late; 50% off up to 48 hours late; No point for more than 48 hours late.

2 Elementary row operations **(30 points)**

As this may be your first experience with serious programming in MATLAB, we will ease into it by first writing some simple functions that perform the elementary row operations on a matrix: interchange, scaling, and replacement.

In this exercise, complete the following files:

`function B = interchange(A, i, j)`
Input: a rectangular matrix $\mathbf{A}$ and two integers i and j.
Output: the matrix resulting from swapping rows i and j, i.e. performing the row operation $R_i \leftrightarrow R_j$.

`function B = scaling(A, i, s)`
Input: a rectangular matrix $\mathbf{A}$, an integer i, and a scalar s.
Output: the matrix resulting from multiplying all entries in row i by s, i.e. performing the row operation $R_i \leftarrow sR_i$.

`function B = replacement(A, i, j, s)`
Input: a rectangular matrix $\mathbf{A}$, two integers i and j, and a scalar s.
Output: the matrix resulting from adding s times row j to row i, i.e. performing the row operation $R_i \leftarrow R_i + sR_j$.

Implementation tips:

In each of these functions, you should check that the input indices i and j are in range, i.e. $1 \leq i \leq m$ and $1 \leq j \leq m$, where m is the number of rows in the matrix (which may not be the same as the number of columns!). If any index is out of range, print an error using the built-in function `disp`, and return the original matrix. This could help you diagnose problems when you write your RREF function in the next part.

Testing:

Test your code on rectangular $m \times n$ matrices of various sizes, both when $m < n$ and when $m > n$. You can generate some simple matrices for testing on using the functions `eye(m,n)`, `ones(m,n)`, and `randi(10,m,n)`.

3 RREF (40 points)

Next, you will use these row operations to write a function that performs Gauss-Jordan elimination and compute the reduced row echelon form of any matrix. We will call the function `my_rref`, because the `rref` function already exists in MATLAB.

Specification:

`function R = my_rref(A)`
Input: a rectangular matrix **A**.
Output: the reduced row echelon form of **A**.

For full credit, your function should handle the following:

- *Partial pivoting:* At each step, you should swap the current row with the one whose entry in the pivot column has the largest absolute value.
- *Free variables:* Due to numerical error, the entries in a column corresponding to a free variable may be extremely small but not precisely zero. Therefore, you should consider an entry to be zero if its absolute value is smaller than 10^{-12}.

We suggest first implementing the algorithm without considering these two issues, then adding code to deal with them one at a time.

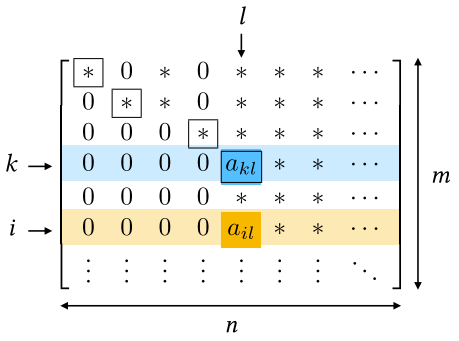
Implementation tips:

There are two different ways one can implement Gauss-Jordan elimination.

- In Section 1.2 under "The Row Reduction Algorithm", the book describes it in two phases: first do Gaussian elimination (Steps 1-4), then perform row operations equivalent to back-substitution (Step 5).
- Gauss-Jordan elimination can also be done in a single phase: every time you find a pivot, perform scaling so the pivot entry becomes 1, then perform elimination on all the other rows, both above and below the pivot row.

You may use either approach in your implementation. Below, we provide pseudocode for the latter approach.

In either case, since we want to be able to handle free variables, the pivot entry won't necessarily be on the diagonal. Instead, you'll need to keep track of both the pivot row, say k, and the pivot column, l, as you go along; see the illustration on the right.



Algorithm 1 RREF

initialize pivot row $k = 1$, pivot column $l = 1$
while $1 \le k \le m$ and $1 \le l \le n$ **do**
    among rows k to m, find the row p with the biggest* entry in column l
    $R_k \leftrightarrow R_p$
    if a_{kl} is zero** **then**
        $l \leftarrow l + 1$
    else
        $R_k \leftarrow (1/a_{kl})R_k$
        for $i = 1, \ldots, k-1, k+1, \ldots, m$ **do**
            $R_i \leftarrow R_i - (a_{il}/a_{kl})R_k$
        end for
        $k \leftarrow k + 1$, $l \leftarrow l + 1$
    end if
end while

*Here "biggest" means having the largest absolute value (`abs` in MATLAB).
**Consider a number to be zero if its absolute value is smaller than 10^{-12}.

Test cases:

`my_rref([2,3,4; 1,1,0])` should return the matrix `[1,0,-4; 0,1,4]`, i.e.

$$\begin{bmatrix} 2 & 3 & 4 \\ 1 & 1 & 0 \end{bmatrix} \to \begin{bmatrix} 1 & 0 & -4 \\ 0 & 1 & 4 \end{bmatrix}.$$

Partial pivoting: `my_rref([0,6; -3,0])` should return the matrix `[1,0; 0,1]`:

$$\begin{bmatrix} 0 & 6 \\ -3 & 0 \end{bmatrix} \to \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Numerical error: `my_rref([3,0,0,1; 0,3,0,1; 0,0,3,1; 1,1,1,1])`:

$$\begin{bmatrix} 3 & 0 & 0 & 1 \\ 0 & 3 & 0 & 1 \\ 0 & 0 & 3 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \to \begin{bmatrix} 1 & 0 & 0 & 1/3 \\ 0 & 1 & 0 & 1/3 \\ 0 & 0 & 1 & 1/3 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Free variables: `my_rref([4,4,4,4; 2,2,4,4; 2,2,4,2])`:

$$\begin{bmatrix} 4 & 4 & 4 & 4 \\ 2 & 2 & 4 & 4 \\ 2 & 2 & 4 & 2 \end{bmatrix} \to \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

You can also obtain a random $m \times n$ matrix with entries in $\{-1, 0, 1\}$ by calling `A = randi([-1,1], m, n)`, then compare your result `my_rref(A)` with MATLAB's built-in `rref(A)`.

Note: Solving linear systems **(no points)**

Once we have an RREF function, we can use it to solve linear systems $\mathbf{Ax} = \mathbf{b}$ by computing the RREF of the augmented matrix $[\mathbf{A} \mid \mathbf{b}]$. For simplicity, we will assume that the system has a unique solution. In this case, the RREF will be of the form

$$\left[\begin{array}{ccc|c} 1 & & & x_1 \\ & \ddots & & \vdots \\ & & 1 & x_n \end{array}\right]$$

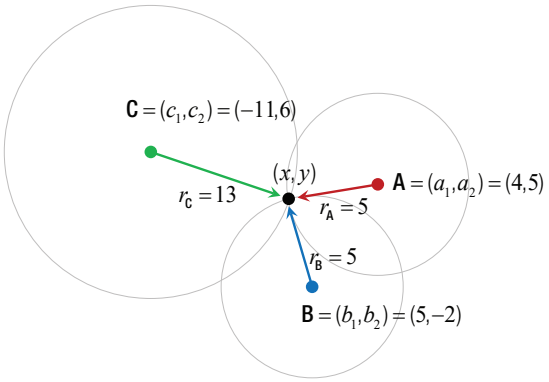
and the solution vector is just its last column. This is easy to implement in MATLAB (which is already provided to you as in `solve.m`):

```
function x = solve(A, b)
    augmented_matrix = [A b];
    R = my_rref(augmented_matrix);
    x = R(:, end);
end
```

Test this function on some simple examples of linear systems which you know to have unique solutions.

4 GPS localization **(30 points)**

One real-life application of solving linear systems is GPS localization. For simplicity, let us work in a 2D world first. Suppose your cellphone receives signals from three GPS satellites at known positions $\mathsf{A} = (a_1, a_2)$, $\mathsf{B} = (b_1, b_2)$, and $\mathsf{C} = (c_1, c_2)$, and can measure its distances $r_\mathsf{A}, r_\mathsf{B}, r_\mathsf{C}$ from all of them, as shown below.



This gives us three *quadratic* equations for our own position $\mathsf{P} = (x, y)$:

$$r_\mathsf{A}^2 = (a_1 - x)^2 + (a_2 - y)^2, \tag{1a}$$

$$r_\mathsf{B}^2 = (b_1 - x)^2 + (b_2 - y)^2, \tag{1b}$$

$$r_\mathsf{C}^2 = (c_1 - x)^2 + (c_2 - y)^2. \tag{1c}$$

Subtracting equation (1a) from equation (1b) and equation (1b) from equation (1c), then simplifying, gives us a 2×2 linear system in x and y:

$$r_{\mathsf{B}}^2 - r_{\mathsf{A}}^2 = 2(a_1 - b_1)x + 2(a_2 - b_2)y - a_1^2 - a_2^2 + b_1^2 + b_2^2, \tag{2a}$$

$$r_{\mathsf{C}}^2 - r_{\mathsf{B}}^2 = 2(b_1 - c_1)x + 2(b_2 - c_2)y - b_1^2 - b_2^2 + c_1^2 + c_2^2. \tag{2b}$$

Since this is a system of linear equations, it can be expressed in matrix form,

$$\begin{bmatrix} * & * \\ * & * \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} * \\ * \end{bmatrix}, \tag{3}$$

for some matrix and some vector on the right-hand side that you should be able to figure out. Your task in this part of the assignment is to implement this method in MATLAB (in `gps2D.m` and `gps3D.m`). That is, given the points A, B, C and distances r_{A}, r_{B}, r_{C}, you will have to:

1. construct the matrix and the right-hand side vector in (3) corresponding to the linear system (2a)-(2b),
2. pass this matrix and vector to the `solve` function (already provided in the zip file) to obtain the point P. **Note**: if you did not get your `my_rref` function to work, you can change it to the MATLAB built-in `rref` in the `solve` function in `solve.m`.

Exactly the same approach also works in 3D, except we will now need our distances from *four* known points $\mathsf{A} = (a_1, a_2, a_3)$, $\mathsf{B} = (b_1, b_2, b_3)$, $\mathsf{C} = (c_1, c_2, c_3)$, and $\mathsf{D} = (d_1, d_2, d_3)$. Your second task is to work out the analogous equations to (2a)-(2b), and implement another program that works in 3D.

Specification:

`function p = gps2d(a, b, c, ra, rb, rc)`
Input: The 2D coordinates of three points A, B, C, and their distances r_{A}, r_{B}, r_{C} from an unknown point P.
Output: The 2D coordinates of the point P.

`function p = gps3d(a, b, c, d, ra, rb, rc, rd)`
Input: The 3D coordinates of four points A, B, C, D, and their distances r_{A}, r_{B}, r_{C}, r_{D} from an unknown point P.
Output: The 3D coordinates of the point P.

Test cases:

`gps2d([4;5], [5;-2], [-11;6], 5, 5, 13)` should return the vector `[1;1]`.

`gps3d([6;-3;3], [-1;-6;5], [-5;4;7], [6;8;4], 5, 7, 9, 9)` should return `[2;0;3]`.

In general, you can create a random 2D vector with entries between say 0 and 10 by using `a = 10*rand(2,1)`. Do this four times for `a`, `b`, `c`, and `p`, and set `ra = norm(a - p)` and so on. (The `norm` function returns the length of a vector.) Then check whether `gps2d(a, b, c, ra, rb, rc)` gives you back `p`.