Additional programming exercises

Timoteo Dinelli[*], Marco Mehl[†]

INFO

Solutions can be found on WeBEEP and GitHub.

Exercises

1. Write a function that finds the maximum value and its position, in terms of row and column number, of the matrix M = magic(234) and compare the result obtained with the MATLAB builtin function max() and find().

2. Write a MATLAB script that proves that the magic matrix definition is correct. And compare the result with a randomly generated one.

3. Write a function that, taken as input an array A of n integers, returns its number of positive elements, without using predefined MATLAB library functions. For example:

   > **Input**: A = [1, 5, -3, -9];
   > **Output**: ans = 2

4. Given a randomly generated matrix 8×8 substitute, within the matrix, the central 4×4 submatrix with a matrix where all the elements are equal to one.

   > **Input**:
   >
   > $$A = \begin{bmatrix} 0.1981 & 0.4228 & 0.5391 & 0.5612 & 0.8555 & 0.2262 & 0.9827 & 0.2607 \\ 0.4897 & 0.5479 & 0.6981 & 0.8819 & 0.6448 & 0.3846 & 0.7302 & 0.5944 \\ 0.3395 & 0.9427 & 0.6665 & 0.6692 & 0.3763 & 0.5830 & 0.3439 & 0.0225 \\ 0.9516 & 0.4177 & 0.1781 & 0.1904 & 0.1909 & 0.2518 & 0.5841 & 0.4253 \\ 0.9203 & 0.9831 & 0.1280 & 0.3689 & 0.4283 & 0.2904 & 0.1078 & 0.3127 \\ 0.0527 & 0.3015 & 0.9991 & 0.4607 & 0.4820 & 0.6171 & 0.9063 & 0.1615 \\ 0.7379 & 0.7011 & 0.1711 & 0.9816 & 0.1206 & 0.2653 & 0.8797 & 0.1788 \\ 0.2691 & 0.6663 & 0.0326 & 0.1564 & 0.5895 & 0.8244 & 0.8178 & 0.4229 \end{bmatrix}$$
   >
   > **Output**:
   >
   > $$A^* = \begin{bmatrix} 0.1981 & 0.4228 & 0.5391 & 0.5612 & 0.8555 & 0.2262 & 0.9827 & 0.2607 \\ 0.4897 & 0.5479 & 0.6981 & 0.8819 & 0.6448 & 0.3846 & 0.7302 & 0.5944 \\ 0.3395 & 0.9427 & 1 & 1 & 1 & 1 & 0.3439 & 0.0225 \\ 0.9516 & 0.4177 & 1 & 1 & 1 & 1 & 0.5841 & 0.4253 \\ 0.9203 & 0.9831 & 1 & 1 & 1 & 1 & 0.1078 & 0.3127 \\ 0.0527 & 0.3015 & 1 & 1 & 1 & 1 & 0.9063 & 0.1615 \\ 0.7379 & 0.7011 & 0.1711 & 0.9816 & 0.1206 & 0.2653 & 0.8797 & 0.1788 \\ 0.2691 & 0.6663 & 0.0326 & 0.1564 & 0.5895 & 0.8244 & 0.8178 & 0.4229 \end{bmatrix}$$

5. Given a square matrix **A**. We want to create a matrix **B** equal to the matrix **A** while replacing only the elements on the main diagonal with the average value of the corresponding rows.

[*]timoteo.dinelli@polimi.it
[†]marco.mehl@polimi.it

> **Input**:
> $$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$
> **Output**:
> $$B = \begin{bmatrix} 2 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 8 \end{bmatrix}$$

6. Given a square matrix A. You want to create a matrix B containing below the main diagonal all null elements, above the main diagonal all elements equal to the sum of all elements of matrix A, and on the main diagonal the corresponding elements of matrix A.

> **Input**:
> $$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$
> **Output**:
> $$B = \begin{bmatrix} 1 & 45 & 45 \\ 0 & 5 & 45 \\ 0 & 0 & 9 \end{bmatrix}$$

7. Write a function that given a random vector returns the same vector but with the elements sorted in ascending order, by implementing a simple version of the *bubble sort* algorithm (reference).

> **Pseudo code**:
> 0. procedure BubbleSort(A:lista of elements to be sorted)
> 1. change is true
> 2. while scambio do
> 3. change is false
> 4. for i = 0 to length(A)-1 do
> 5. if A[i] ¿ A[i+1] then
> 6. swap(A[i], A[i+1])
> 7. change is true
>
> **Input**: v = [5, 4, 6, 8, 11];
> **Output**: ans = [4, 5, 6, 8, 11]

8. Write a function that takes two integers a and b (where $a < b$) and returns a vector containing all prime numbers in the range $[a, b]$. Implement your own logic to check if a number is prime without using built-in functions.

> **Input**: a = 10, b = 30
> **Output**: ans = [11, 13, 17, 19, 23, 29]

9. Write a function that takes a square matrix as input and returns a vector containing all elements of the matrix read in a spiral pattern (clockwise from outside to inside).

> **Input**:
> $$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$
> **Output**: ans = [1, 2, 3, 4, 8, 12, 16, 15, 14, 13, 9, 5, 6, 7, 11, 10]

10. Write a function that takes a vector $\mathbf{v}$ and a window size k as inputs and returns a new vector where each element is the average of k consecutive elements centered at that position. For edge elements, use only the available neighbors.

> **Input**: v = [1, 2, 3, 4, 5, 6, 7], k = 3
> **Output**: ans = [1.5, 2, 3, 4, 5, 6, 6.5]
>
> **Explanation:**

> - Position 1: avg(1,2) = 1.5
> - Position 2: avg(1,2,3) = 2
> - Position 3: avg(2,3,4) = 3
> - ...
> - Position 7: avg(6,7) = 6.5

11. Write a function that:
    - Takes a square matrix **A** as input
    - Creates three versions: rotated 90°, 180°, and 270° clockwise (implement rotation manually with loops, don't use `rot90`)
    - Returns which rotation (if any) makes the matrix symmetric
    - If none produce a symmetric matrix, return 0

> **Input**:
> $$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$
> Check if A rotated by 90°, 180°, or 270° is symmetric ($B = B^T$)
> **Output**: ans = 0 (none are symmetric)

Solutions

Exercise 1: Finding maximum value and position in magic matrix

```
function [max_val, row, col] = find_max_magic()
    % Create magic matrix
    M = magic(234);

    % Manual approach - initialize variables
    max_val = M(1,1);
    row = 1;
    col = 1;

    % Loop through all elements
    [n_rows, n_cols] = size(M);
    for i = 1:n_rows
        for j = 1:n_cols
            if M(i,j) > max_val
                max_val = M(i,j);
                row = i;
                col = j;
            end
        end
    end

    fprintf('Manual approach:\n');
    fprintf('Maximum value: %d at position (%d, %d)\n', max_val, row, col);

    % Using MATLAB built-in functions
    [max_cols, row_indices] = max(M);
    [max_val_builtin, col_idx] = max(max_cols);
    row_builtin = row_indices(col_idx);
    col_builtin = col_idx;

    fprintf('\nBuilt-in approach (max):\n');
    fprintf('Maximum value: %d at position (%d, %d)\n', ...
            max_val_builtin, row_builtin, col_builtin);

    % Using find
    [r, c] = find(M == max_val_builtin);
    fprintf('\nUsing find:\n');
    fprintf('Maximum value: %d at position (%d, %d)\n', ...
            max_val_builtin, r(1), c(1));
end
```

Exercise 2: Proving magic matrix definition

```
% Script to verify magic matrix properties
n = 5;  % Size of magic matrix
M = magic(n);

fprintf('Testing magic matrix of size %dx%d\n\n', n, n);

% Calculate the magic constant
magic_constant = n * (n^2 + 1) / 2;
fprintf('Expected magic constant: %.0f\n\n', magic_constant);

% Check row sums
fprintf('Row sums:\n');
row_sums = sum(M, 2);
for i = 1:n
    fprintf('Row %d: %.0f\n', i, row_sums(i));
end

% Check column sums
fprintf('\nColumn sums:\n');
col_sums = sum(M, 1);
for i = 1:n
    fprintf('Column %d: %.0f\n', i, col_sums(i));
end

% Check diagonal sums
main_diag_sum = sum(diag(M));
anti_diag_sum = sum(diag(fliplr(M)));
fprintf('\nMain diagonal sum: %.0f\n', main_diag_sum);
```

```
fprintf('Anti-diagonal sum: %.0f\n', anti_diag_sum);

% Verify all sums are equal
is_magic = all(row_sums == magic_constant) && ...
           all(col_sums == magic_constant) && ...
           main_diag_sum == magic_constant && ...
           anti_diag_sum == magic_constant;

fprintf('\nIs magic matrix valid? %s\n\n', mat2str(is_magic));

% Compare with random matrix
R = randi([1, n^2], n, n);
fprintf('Random matrix row sums: %s\n', mat2str(sum(R, 2)'));
fprintf('Random matrix column sums: %s\n', mat2str(sum(R, 1)));
fprintf('Random matrix is NOT magic!\n');
```

Exercise 3: Count positive elements

```
function count = count_positive(A)
    % Count positive elements without using built-in functions
    count = 0;
    n = length(A);

    for i = 1:n
        if A(i) > 0
            count = count + 1;
        end
    end
end

% Test the function
A = [1, 5, -3, -9];
result = count_positive(A);
fprintf('Number of positive elements: %d\n', result);
```

Exercise 4: Replace central 4x4 submatrix

```
% Generate random 8x8 matrix
A = rand(8, 8);

fprintf('Original matrix:\n');
disp(A);

% Replace central 4x4 submatrix with ones
% Central 4x4 is from rows 3:6 and columns 3:6
A(3:6, 3:6) = ones(4, 4);

fprintf('\nModified matrix:\n');
disp(A);
```

Exercise 5: Replace diagonal with row averages

```
function B = diagonal_row_average(A)
    % Create copy of matrix A
    B = A;
    n = size(A, 1);

    % Replace each diagonal element with its row average
    for i = 1:n
        row_avg = sum(A(i, :)) / n;
        B(i, i) = row_avg;
    end
end

% Test the function
A = [1 2 3; 4 5 6; 7 8 9];
B = diagonal_row_average(A);
fprintf('Original matrix A:\n');
disp(A);
fprintf('Modified matrix B:\n');
```

```
disp(B);
```

Exercise 6: Upper triangular with sum

```
function B = create_special_matrix(A)
    n = size(A, 1);
    B = zeros(n, n);

    % Calculate sum of all elements
    total_sum = sum(A(:));

    % Fill the matrix
    for i = 1:n
        for j = 1:n
            if i == j
                % Diagonal: keep original elements
                B(i, j) = A(i, j);
            elseif i < j
                % Above diagonal: use total sum
                B(i, j) = total_sum;
            else
                % Below diagonal: zeros (already initialized)
                B(i, j) = 0;
            end
        end
    end
end

% Test the function
A = [1 2 3; 4 5 6; 7 8 9];
B = create_special_matrix(A);
fprintf('Original matrix A:\n');
disp(A);
fprintf('Modified matrix B:\n');
disp(B);
```

Exercise 7: Bubble sort

```
function sorted_vec = bubble_sort(v)
    n = length(v);
    sorted_vec = v;
    change = true;

    while change
        change = false;
        for i = 1:(n-1)
            if sorted_vec(i) > sorted_vec(i+1)
                % Swap elements
                temp = sorted_vec(i);
                sorted_vec(i) = sorted_vec(i+1);
                sorted_vec(i+1) = temp;
                change = true;
            end
        end
    end
end

% Test the function
v = [5, 4, 6, 8, 11];
sorted_v = bubble_sort(v);
fprintf('Original vector: %s\n', mat2str(v));
fprintf('Sorted vector: %s\n', mat2str(sorted_v));
```

Exercise 8: Prime numbers in range

```
function primes = find_primes(a, b)
    primes = [];

    for num = a:b
        if is_prime(num)
            primes = [primes, num];
```

```
        end
    end
end

function result = is_prime(n)
    % Check if n is prime
    if n < 2
        result = false;
        return;
    end

    if n == 2
        result = true;
        return;
    end

    if mod(n, 2) == 0
        result = false;
        return;
    end

    % Check odd divisors up to sqrt(n)
    for i = 3:2:sqrt(n)
        if mod(n, i) == 0
            result = false;
            return;
        end
    end

    result = true;
end

% Test the function
a = 10;
b = 30;
primes = find_primes(a, b);
fprintf('Prime numbers between %d and %d: %s\n', a, b, mat2str(primes));
```

Exercise 9: Spiral matrix traversal

```
function spiral_vec = spiral_traversal(A)
    n = size(A, 1);
    spiral_vec = [];

    top = 1;
    bottom = n;
    left = 1;
    right = n;

    while top <= bottom && left <= right
        % Traverse right along top row
        for col = left:right
            spiral_vec = [spiral_vec, A(top, col)];
        end
        top = top + 1;

        % Traverse down along right column
        for row = top:bottom
            spiral_vec = [spiral_vec, A(row, right)];
        end
        right = right - 1;

        % Traverse left along bottom row (if still valid)
        if top <= bottom
            for col = right:-1:left
                spiral_vec = [spiral_vec, A(bottom, col)];
            end
            bottom = bottom - 1;
        end

        % Traverse up along left column (if still valid)
        if left <= right
            for row = bottom:-1:top
                spiral_vec = [spiral_vec, A(row, left)];
            end
            left = left + 1;
```

```
        end
    end
end

% Test the function
A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16];
spiral = spiral_traversal(A);
fprintf('Original matrix:\n');
disp(A);
fprintf('Spiral traversal: %s\n', mat2str(spiral));
```

Exercise 10: Running average filter

```
function result = running_average(v, k)
    n = length(v);
    result = zeros(1, n);
    half_window = floor(k / 2);

    for i = 1:n
        % Determine the valid window range
        start_idx = max(1, i - half_window);
        end_idx = min(n, i + half_window);

        % Calculate average of elements in window
        window_elements = v(start_idx:end_idx);
        result(i) = sum(window_elements) / length(window_elements);
    end
end

% Test the function
v = [1, 2, 3, 4, 5, 6, 7];
k = 3;
filtered = running_average(v, k);
fprintf('Original vector: %s\n', mat2str(v));
fprintf('Window size: %d\n', k);
fprintf('Filtered vector: %s\n', mat2str(filtered));
```

Exercise 11: Matrix rotation and symmetry check

```
function rotation_angle = check_rotation_symmetry(A)
    % Returns 90, 180, 270 if that rotation produces symmetric matrix
    % Returns 0 if none produce a symmetric matrix

    % Rotate 90 degrees clockwise
    A_90 = rotate_90(A);
    if is_symmetric(A_90)
        rotation_angle = 90;
        return;
    end

    % Rotate 180 degrees
    A_180 = rotate_90(A_90);
    if is_symmetric(A_180)
        rotation_angle = 180;
        return;
    end

    % Rotate 270 degrees
    A_270 = rotate_90(A_180);
    if is_symmetric(A_270)
        rotation_angle = 270;
        return;
    end

    rotation_angle = 0;
end

function rotated = rotate_90(A)
    % Rotate matrix 90 degrees clockwise manually
    n = size(A, 1);
    rotated = zeros(n, n);

    for i = 1:n
```

```
        for j = 1:n
            rotated(j, n+1-i) = A(i, j);
        end
    end
end

function result = is_symmetric(A)
    % Check if matrix is symmetric (A == A')
    n = size(A, 1);
    result = true;

    for i = 1:n
        for j = 1:n
            if A(i, j) ~= A(j, i)
                result = false;
                return;
            end
        end
    end
end

% Test the function
A = [1 2 3; 4 5 6; 7 8 9];
angle = check_rotation_symmetry(A);
fprintf('Original matrix:\n');
disp(A);
fprintf('Rotation that produces symmetry: %d degrees\n', angle);
if angle == 0
    fprintf('(None produce a symmetric matrix)\n');
end
```