Objectives and instructions

In this project, you will write MATLAB programs to work with sub-matrices and to find a path through a maze. You will learn about

- User-defined functions
- 2-d arrays—matricies
- Program design (organization), implementation, and testing

Question 1 is fairly straightforward and you will learn to work with sub-matrices. Question 2 requires that you learn to organize a program. Unlike previous project questions, Question 2 will have fewer explicit instructions. Rather, the question statement here explains the game, the input, and the desired outcome, and you, the programmer, need to come up with a solution (of high quality of course). At the end of this project, you will look back and realize that you haven’t written much code despite the complexity of the project. Remember that problem solving begins with understanding the problem. The challenges in this project include reading the problem and designing a program. Then of course you have to write and test the code. Testing should be done as you code—do not wait to test the entire program at the very end of the project. Careful, incremental development of your program will give you a better product and will end up saving you time!

Submit all script and function files. If you create any functions that are not specified in this question statement, they should be subfunctions, i.e., they are included in the M-file of the main function. See “Subfunctions” in Section 5.7 of the Chapman textbook.

1. Optimal location for building an observatory

Digitized relief maps show the elevations of individual points in the landscape. Typically, the landscape is uniformly sampled and the elevations at these sample points are measured and recorded. To conduct uniform sampling, a “mesh” is given and the elevation of every crossing point, or node, is measured. The result is a 2-dimensional array (matrix) of elevations.

Below are a simple uniform mesh and the corresponding elevation matrix.

\[
\begin{array}{cccccc}
0 & 1 & 3 & 4 & 5 \\
3 & 8 & 6 & 3 & 1 \\
0 & 7 & 9 & 0 & 1 \\
7 & 6 & 2 & 3 & 4 \\
6 & 4 & 5 & 2 & 1 \\
\end{array}
\]

Given such a digitalized relief map, find the best place to build an observatory. Our criterion for the optimal location is one that is highest in elevation. That way, we can gain a clear view of the starry night.

The observatory will have an area equivalent to a \( k \times k \) sub-matrix in the relief map (elevation matrix). A reasonable measure of the elevation in the sub-matrix area is the average elevation of all the sampling points in the sub-matrix. For example, in the 5\( \times \)5 elevation matrix above, the 2\( \times \)2 sub-matrix with the highest average elevation is marked by the dashed square.

Write a function \texttt{maxSubmat.m} that has the following description:
% maxSubMatrix(mat, k): Find k-by-k submatrix with the highest mean value
% Input parameters:
%   mat – matrix of numeric values
%   k – dimension of submatrix, assume 1 <= k <= smaller dimension of mat
% Output parameters:
%   aver – mean value of k-by-k submatrix
%   i, j – indices of upper left corner of the submatrix with the highest mean

For instance, if we call function maxSubmat with the 5×5 elevation matrix above and k is 2, the three returned values would be 7.5 (mean), 2 (row index), 2 (column index). Create some matrices to test your function.

2. Maze Puzzle

Have you ever played a maze puzzle? How do you make a decision when you come to a conjunction? We will apply a practical method to solve the maze puzzle problem. Below is some background information.

According to the types of passages within a maze, a maze problem can be divided into the following categories:

- **Perfect**: A "perfect" maze is one without any loops or closed circuits, and without any inaccessible areas. Also called a simply-connected maze, a perfect maze has exactly one path from one point to any other point and the maze has exactly one solution. (In Computer Science terms, such a maze can be described as a minimal spanning tree over the set of cells.)

- **Braid**: A "braid" maze, also called a purely multiply connected maze, is one without any dead ends. Such a maze uses passages that coil around and run back into each other (hence the term "braid") and causes you to spend time going in circles instead of bumping into dead ends. A well-designed braid maze can be much harder to solve than a perfect maze of the same size.

- **Unicursal**: A unicursal maze is one without any junctions. Sometimes the term *labyrinth* is used to refer to constructs of this type, while the term *maze* refers to a puzzle where choices are involved. A unicursal maze has just one long snake-like passage that coils throughout the extent of the maze. It's not really difficult unless you accidentally get turned around half way through and make your way back to the beginning.

- **Partial braid**: A partial braid maze is just a mixed maze with both loops and dead ends. The word "braid" can be used quantitatively, where a "heavily braided maze" is one with many loops or detached walls and a "slightly braided maze" is one with just a few loops or detached walls.

We introduce a practical algorithm—the Wall Follower algorithm—here. This algorithm simulates the act of always keeping a hand (either your left or right) on the wall to one side as you go through the maze. Wall following will successfully solve many but not all mazes. If your goal is to find an exit that is on the edge of the
maze, it will always work for all the types of mazes discussed above, but it will not necessarily find the shortest solution. Wall following doesn't work only when the goal is in the center of the maze and there's a closed circuit surrounding it. In that case you'll go around the center and eventually find yourself back at the beginning.

Your task is to implement the left-handed Wall Follower algorithm: at a conjunction, always turn left.

Below is an example of a maze data matrix:

```
1 1 0 1 1 1 1 1 1
1 1 0 0 0 0 0 0 1
1 1 0 1 1 0 1 0 1
1 1 0 0 1 0 1 1 1
1 1 1 1 1 0 1 1 1
```

The value 0 represents a passage and the value 1 represents a wall. The diagram on the right is the output from using the given `plotMaze` function on the maze data matrix. Note that the wall is marked with blue crosses and the passage is blank. In this case, the entrance and exit are on the edges at positions (1,3) and (5,6). Since the exit is on the edge, you are guaranteed to find a solution using the Wall Follower algorithm if there exists a solution. (If your method fails and you return to the start position, then it means there is no solution at all from the entrance to the exit.)

Using the Wall Follower algorithm, solve the maze puzzle by finding a path from the entrance (top edge) to the goal (exit). This path should have the value 2 in the maze matrix. For example, using the Wall Follower (left-turning) algorithm on the above maze, the resulting maze matrix should be

```
1 1 2 1 1 1 1 1 1
1 1 2 2 2 2 2 1 1
1 1 0 1 1 2 1 2 1
1 1 0 0 1 2 1 2 1
1 1 1 1 1 2 1 2 1
```

The diagram on the right is the output from using the given `plotMaze` function on the maze matrix. The solution path is marked with red circles.

Skeleton driver program, a given function, and data

- A skeleton driver program `maze1.m` is given. This script file (when completed) drives the program for loading maze data, computing the path, and drawing a plot of the result.
- Function `plotMaze` is given. This function produces the color diagrams above.
- **Data files:** `maze1data.mat`, `maze2data.mat`, `maze3data.mat`, `maze4data.mat`
  
Maze 1 is the maze shown above. It’s a small maze with a solution so it is suitable for you to use to develop and test your program. Mazes 2, 3, and 4 are bigger mazes that may be solvable using the Wall Follower algorithm. You’ll find out! 

When you `load` the data `mat` files, the following variables will be in the workspace memory:

- `entrance` – vector of the indices of the entrance. For the example above, the vector would be [1, 3].
- `goal` – vector containing the indices of the goal. For the example above, the vector would be [5, 6].
- `startDirection` – the direction that the player faces: 1 represents left, 2 represents down, 3 represents right, and 4 represents up.
- `maze` – the maze matrix, see example above.
To do

Write a function `findPath` that computes the solution path and “marks” the path using the value 2 in the maze matrix. Function `findPath` has the following description:

```plaintext
%findPath finds a solution path in a maze
%Input parameters:
%  matrix – the maze matrix containing 0s (passage) and 1s (wall)
%  startPosition – vector containing indices of entrance to maze
%  endPosition – vector containing indices of the goal
%  startDirection – the direction that the player faces at the start of the game.
%                   Four possible values: 1 (left), 2 (down), 3 (right), 4 (up)
%Output parameters:
%  pathMatrix – matrix containing 2s (path), 1s (wall), and 0s (passage not taken)
%  success – value 1 indicates wall follower algorithm succeeds; value 0 otherwise
```

Modify function `plotMaze` so that the entrance is marked by a black asterisk (*) and the exit is marked by a green asterisk.

Complete the skeleton driver program `maze1.m` that calls your function to compute the solution and calls function `plotMaze` to plot the result. The program displays a congratulatory message if the puzzle is solved or a sad message if the goal is not reached.

Document your program by writing a substantial comment block at the top of the driver program `maze1.m`. This comment block describes the organization of the program, lists the user-defined functions used, and briefly describes the functions. *Your program will not be graded unless you write a clear comment block to document your program.*

3. Help your friend get through the maze faster!

Remember that you may not necessarily find a shortest route in a maze using the Wall Follower algorithm—there may be some redundant route. If you mark the redundant route with stones, then you can tell your buddy who goes in after you to *not* take the path where you’ve placed stones!

In the previous example, observe that towards the right side of the maze, one reaches a dead end, turns around to go back to position (2,6), and continues on to find the goal. This redundant route is marked with the value 3 in the matrix below and with green circles in the diagram. Using the Wall Follower algorithm, you cannot prevent this from happening, but once you get back to the conjunction where the redundant route starts, **you can revisit that redundant route and mark it with stones (3s)!** (I.e., you’re being a good neighbor by preventing others from making the same mistake.)

```
  1 1 2 1 1 1 1 1 1
  1 1 2 2 2 3 3 1
  1 1 0 1 1 2 1 3 1
  1 1 0 0 1 2 1 1 1
  1 1 1 1 2 1 1 1

  x x o o o o o o x
  x x o o o o o o x
  x x x x o o o x
  x x x o o x o x
  x x x x o o x
```
To do

- Write a function `findPath2` that marks the solution path using the value 2 and any redundant route using the value 3 in the maze matrix. Besides marking the redundant route, function `findPath2` has the same description as function `findPath`. As you develop function `findPath2`, consider the following function...

- Write a function `markingStone` to mark a redundant route with stones (3s). This function is called from `findPath2` when you come back to a position where you have visited before. In order to revisit the route (to place to stones), you must know which direction you took when you were at that position last time. (Hint: consider a scheme to store the direction taken at each position.) Function `markingStone` has the following description:

  ```
  %markingStone: mark redundant route in the solution path with the value 3
  %Revisit the redundant route in the solution path and mark those positions with
  %the value 3.
  %Input parameters:
  %  routeMatrix – matrix containing 2s (path), 1s (wall), and 0s (passage not taken)
  %  position – vector of the indices of the revisited conjunction where the
  %             redundant route starts
  %  dir – direction in which to move in order to revisit the redundant route
  %Output parameters:
  %  pathMatrix – matrix containing 3s (marked redundant route), 2s (solution path),
  %              1s (wall), and 0s (passage not taken)
  ```

- Modify function `plotMaze` so that any redundant route is marked with green circles. Save this modification as function `plotLabyrinth`.

- Modify the previous driver program `maze1` to use functions `findPath2` and `plotLabyrinth`. Save this modification as script file `maze2.m`.

- Document your program by writing a substantial comment block at the top of the driver program `maze2.m`. This comment block describes the organization of the program, lists the user-defined functions used, and briefly describes the functions. *Your program will not be graded unless you write a clear comment block to document your program.*

### 4. Final words

Pay attention to how you organize your program. Whenever you notice that you are repeating a lot of very similar code, or when you notice that there are “annoying details” cluttering up your code, you should consider writing a (sub)function to replace the repetitious or very detailed code. Remember that you want to decompose the problem! Some subtasks that you identify as you decompose the problem may be good candidates for (sub)functions.