E7: Introduction to Computer Programming for Scientists and Engineers
University of California at Berkeley, Spring 2017
Instructor: Lucas A. J. Bastien

Lab Assignment 07: File Input/Output; Review of Part 1 of the Class
Version: release

Due date: Friday March 10th 2017 at 12 pm (noon).

General instructions, guidelines, and comments:

- For each question, you will have to write and submit one or more Matlab functions. We provide a number of test cases that you can use to test your function. The fact that your function works for all test cases provided does not guarantee that it will work for all possible test cases relevant to the question. It is your responsibility to test your function thoroughly, to ensure that it will also work in situations not covered by the test cases provided. During the grading process, your function will be evaluated on a number of test cases, some of which are provided here, some of which are not.
- Submit on bCourses one m-file for each function that you have to write. The name of each file must be the name of the corresponding function, with the suffix `.m` appended to it. For example, if the name of the function is `my_function`, the name of the file that you have to submit is `my_function.m`. **Carefully check the name of each file that you submit.** Do not submit any zip file. If you re-submit a file that you have already submitted, bCourses may rename the file by adding a number to the file's name (*e.g.,* rename `my_function.m` into `my_function-01.m`). This behavior is okay and should be handled seamlessly by our grading system. Do not rename the file yourself as a response to this behavior.
- A number of optional Matlab toolboxes can be installed alongside Matlab to give it more functionality. All the functions that you have to write to complete this assignment can, however, be implemented without the use of any optional Matlab toolboxes. We encourage you to not use optional toolboxes to complete this assignment. All functions of the Matlab base installation will be available to our grading system, but functions from optional toolboxes may not. If one of your function uses a function that is not available to our grading system, you will loose all points allocated to the corresponding part of this assignment. To guarantee that you are not using a Matlab function from an optional toolbox that is not available to our grading system, use one or both of the following methods:
  - ◇ Only use functions from the base installation of Matlab.
  - ◇ Make sure that your function works on the computers of the 1109 Etcheverry Hall computer lab. All the functions available on these computers will be available to our grading system.

- For this assignment, the required submissions are:
  - ◇ `my_read_array.m`
  - ◇ `my_destination.m`
  - ◇ `my_fill_open_space.m`
  - ◇ `my_gridcells_connected.m`
  - ◇ `my_open_spaces.m`
  - ◇ `my_buses.m`

1. Text file array reader (15 points)

In this question, you will write a function that reads specially formatted text files (that describe arrays), and creates the corresponding Matlab arrays. More precisely, write a function with the following header:

```
function [array] = my_read_array(filename)
```

where:

- `filename` is a non-empty character string (*i.e.* a row vector of class `char` that has at least one element) that represents the name of the text file to read.
- `array` is an $m \times n$ array of class `double` that represents the array read from the text file. You can assume that $m > 0$ and $n > 0$.

In this question, the text files that represent arrays are formatted as follows:

- The first line contains exactly one number (and nothing else). This number is the number m of rows in the array.
- The second line contains exactly one number (and nothing else). This number is the number n of columns in the array.
- Each subsequent line contains exactly three numbers (and nothing else), separated by space characters. If we call these three numbers `i`, `j`, and `val`, respectively, then the line read from the text file indicates that `array(i,j)` should have the value `val`. You can assume that the numbers `i` and `j` will be valid row and column indices (respectively) for the array.

The text files will not necessarily specify the values of all the elements of the arrays that they represent. Use `NaN` as the default value for elements of `array` that are not specified by the text file.

Hints:

- You can use Matlab's built-in function `str2num` to convert a number specified as a character string to the corresponding number of class `double` in Matlab.
- You can use Matlab's built-in function `strsplit` to split a character string around white

spaces. For example:

```
>> strsplit('hello there! How are you?')
ans =
  1x5 cell array
    'hello'    'there!'    'How'    'are'    'you?'
```

You can use the following template as a starting point to implement your function:

```
function [array] = my_read_array(filename)

% E7 Spring 2017, University of California at Berkeley.
% Template for the function for question 1 of Lab 07.

% Open the text file in read-only mode
fid = fopen(filename, 'r');

% Get the number of rows and columns in the array, and initialize the array
% !!! Add code here !!!

while true

    % Read one line from the text file
    line_from_file = fgetl(fid);

    % We break out of the loop if we have reached the end of the file
    if ~ischar(line_from_file)
        break
    end

    % Process the line
    % !!! Add code here !!!

end

% Close the file
fclose(fid);

end
```

We provide you (on bCourses) with two text files with which to test your function: `array01.txt` and `array02.txt`. Save these two files in your Matlab working directory before trying the following test cases. You do **not** need to submit these text files on bCourses for this assignment, as they will already be available to our grading system.

Test cases:

```
>> % Let us look at what the first text file looks like
>> type array01.txt

3
```

```
4
2 2 5
2 3 32
1 2 4
1 3 9.2
2 1 6
3 1 5.4
3 2 -4
3 4 0

>> % Try the function on the first text file
>> array = my_read_array('array01.txt')
array =
       NaN    4.0000    9.2000       NaN
    6.0000    5.0000   32.0000       NaN
    5.4000   -4.0000       NaN         0

>> % Let us look at what the second text file looks like
>> type array02.txt

5
1
1 1 10
2 1 5
3 1 5.1
5 1 -9

>> % Try the function on the second text file
>> array = my_read_array('array02.txt')
array =
   10.0000
    5.0000
    5.1000
       NaN
   -9.0000
```

2. Mazes (60 points)

In this question, you will write functions that analyze mazes. We will represent a maze in Matlab as an $m \times n$ array of class `double` that contains only zeros and ones. A `0` represents a wall and a `1` represents an open space. In this question, we will call a "**grid cell**" one individual element of the maze (*i.e.* one element of the array of class `double` that represents the maze). We will say that two grid cells are "**adjacent**" if they are directly next to each other either horizontally (*i.e.* they are in the same row of the array and one column apart) or vertically (*i.e.* they are in the same column of the array and one row apart). We will say that two grid cells are "**connected**" if both grid cells are open spaces and if it is possible to travel between these two grid cells by taking a succession of steps between adjacent open spaces. **All the grid cells located on the edge of a maze** (first and last row and first and last column of the corresponding array) **will always be walls**.

Note that:

- Solving question 2.2 may significantly help you solve the following two questions (2.3 and 2.4).
- We provide you with two functions (m-files `my_create_maze.m` and `my_show_maze.m`) that can help you visualize the mazes and test your functions:
  - ◇ `my_create_maze` creates a randomly-generated $m \times n$ maze.
  - ◇ `my_show_maze` creates a figure that shows an $m \times n$ maze.

  To use these functions, save the corresponding m-files on your computer where Matlab can find them. Use the commands `help my_create_maze` and `help my_show_maze` to see documentation on how to use these functions. Alternatively, look at the comments and code in the corresponding m-files. We recommend you to use `my_create_maze` to generate additional test cases to test the functions that you will write for this question, and to use `my_show_maze` to visualize the corresponding mazes, to help you determine whether your functions work on these additional test cases.
- On the figure created by `my_show_maze`, black squares represent walls, white squares represent open spaces, the numbers to the left of the plot represent the row indices of the corresponding grid cells and the numbers on the top of the plot represent the column indices of the corresponding grid cells.
- We also provide you with the `mat` file named `mazes.mat`. This file contains three sample mazes that you can use to test your functions with. Each variable defined in this `mat` file represents a different maze. The names of these variables are `maze_small`, `maze_medium`, and `maze_large`. Figure 1 shows an example of the figure generated by `my_show_maze` when used on the maze named `maze_medium` provided in `mazes.mat`.

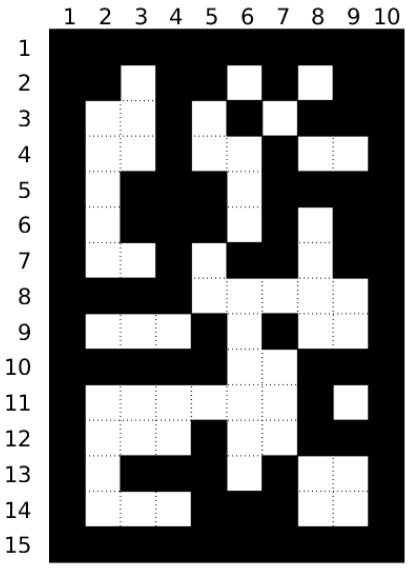

Figure 1: The maze named `maze_medium` from `mazes.mat` as plotted by `my_show_maze`. Black squares represent walls, white squares represent open spaces, the numbers to the left of the plot represent the row indices of the corresponding grid cells and the numbers on the top of the plot represent the column indices of the corresponding grid cells.

2.1. Following a path (10 points)

Write a function with the following header:

```
function [i_end, j_end] = my_destination(maze, i_start, j_start, moves)
```

where:

- `maze` is an $m \times n$ maze following the format described above.
- `i_start` and `j_start` are the row and column indices, respectively, of an open space (the "starting position") in the maze represented by `maze`.
- `moves` is a possibly empty character string (*i.e.* a possibly empty row vector of class `char`). See below for a more detailed description.
- `i_end` and `j_end` are the row and column indices, respectively, of the "ending position". See below for a more detailed description.

The character string `moves` is a set of instructions, where each character in the string corresponds to a distinct instruction:

- A `'U'` (upper-case "U") indicates "move up" (*i.e.* move to the previous row).
- A `'D'` (upper-case "D") indicates "move down" (*i.e.* move to the next row).
- A `'L'` (upper-case "L") indicates "move left" (*i.e.* move to the previous column).
- A `'R'` (upper-case "R") indicates "move right" (*i.e.* move to the next column).
- Any other character represents an invalid instruction, which should be ignored.

The "ending position" is the position reached when starting from the "starting position" and following the instructions described in `moves`, one by one, and in order. Any instruction that corresponds to moving from an open space to a wall should be ignored.

Test cases:

```
>> load mazes.mat

>> [i, j] = my_destination(maze_small, 2, 2, 'RRD')
i =
     3
j =
     4

>> [i, j] = my_destination(maze_small, 2, 2, 'RRU')
i =
     2
j =
     4

>> [i, j] = my_destination(maze_medium, 2, 3, 'DLDDDDR')
i =
     7
j =
     3

>> [i, j] = my_destination(maze_medium, 14, 4, 'URL.LUL+UR.RU!Rrl')
i =
    11
j =
     5
```

2.2. Fill an open space (20 points)

Write a function with the following header:

```
function [filled] = my_fill_open_space(maze, i, j, n)
```

where:

- `maze` is an $m \times n$ maze following the format described above.
- `i` and `j` are the row and column indices, respectively, of an open space in the maze represented by `maze`.

- `n` is a scalar of class `double` that is neither `0`, `1`, `NaN`, `Inf`, nor `-Inf`.
- `filled` is a copy of `maze` where the value of all the grid cells connected to the grid cell of index (`i`, `j`), including the grid cell of index (`i`, `j`) itself, has been changed to `n`.

There exist multiple ways to implement this function, including with and without using recursion. We propose the following algorithm (you don't necessarily have to use this algorithm) to implement the function `my_fill_open_space` without using recursion:

1. Initialize `filled` as an exact copy of `maze`.
2. In `filled`, replace the element of index (`i`, `j`) by the value of `n`.
3. Look at each element of `filled`. If you find an element whose value is `n`, replace the values of all the open spaces adjacent to it by the value of `n`.
4. If you did at least one replacement in step 3, repeat step 3.

Test cases:

```
>> load mazes.mat

>> filled = my_fill_open_space(maze_small, 2, 4, 2)
filled =
     0     0     0     0     0
     0     2     2     2     0
     0     2     2     2     0
     0     2     0     0     0
     0     0     0     0     0

>> filled = my_fill_open_space(maze_small, 2, 4, 7)
filled =
     0     0     0     0     0
     0     7     7     7     0
     0     7     7     7     0
     0     7     0     0     0
     0     0     0     0     0

>> filled = my_fill_open_space(maze_medium, 4, 2, 5)
filled =
     0     0     0     0     0     0     0     0     0     0
     0     0     5     0     0     1     0     1     0     0
     0     5     5     0     1     0     1     0     0     0
     0     5     5     0     1     1     0     1     1     0
     0     5     0     0     0     1     0     0     0     0
     0     5     0     0     0     1     0     1     0     0
     0     5     5     0     1     0     0     1     0     0
     0     0     0     0     1     1     1     1     1     0
     0     1     1     1     0     1     0     1     1     0
     0     0     0     0     0     1     1     0     0     0
     0     1     1     1     1     1     1     0     1     0
     0     1     1     1     0     1     1     0     0     0
     0     1     0     0     0     1     0     1     1     0
     0     1     1     1     0     0     0     1     1     0
     0     0     0     0     0     0     0     0     0     0
```

```
>> filled = my_fill_open_space(maze_medium, 10, 6, -4)
filled =
     0     0     0     0     0     0     0     0     0     0
     0     0     1     0     0     1     0     1     0     0
     0     1     1     0     1     0     1     0     0     0
     0     1     1     0     1     1     0     1     1     0
     0     1     0     0     0     1     0     0     0     0
     0     1     0     0     0     1     0    -4     0     0
     0     1     1     0    -4     0     0    -4     0     0
     0     0     0     0    -4    -4    -4    -4    -4     0
     0     1     1     1     0    -4     0    -4    -4     0
     0     0     0     0     0    -4    -4     0     0     0
     0    -4    -4    -4    -4    -4    -4     0     1     0
     0    -4    -4    -4     0    -4    -4     0     0     0
     0    -4     0     0     0    -4     0     1     1     0
     0    -4    -4    -4     0     0     0     1     1     0
     0     0     0     0     0     0     0     0     0     0
```

2.3. Connected cells (10 points)

Write a function with the following header:

```
function [answer] = my_gridcells_connected(maze, i1, j1, i2, j2)
```

where:

- `maze` is an $m \times n$ maze following the format described above.
- `i1` and `j1` are the row and column indices, respectively, of an open space in the maze represented by `maze`.
- `i2` and `j2` are the row and column indices, respectively, of an open space in the maze represented by `maze`. This grid cell can be the same as the one described by `i1` and `j1`.
- `answer` is a scalar of class `logical` that is true if and only if the two grid cells represented by `i1` and `j1` on the one hand, and `i2` and `j2` on the other hand, are connected.

Test cases:

```
>> load mazes.mat

>> [answer] = my_gridcells_connected(maze_small, 2, 2, 2, 2)
answer =
  logical
   1
>> [answer] = my_gridcells_connected(maze_small, 2, 2, 3, 4)
answer =
  logical
   1
>> [answer] = my_gridcells_connected(maze_medium, 11, 2, 8, 5)
answer =
  logical
```

```
   1
>> [answer] = my_gridcells_connected(maze_medium, 11, 9, 14, 8)
answer =
  logical
   0
>> [answer] = my_gridcells_connected(maze_medium, 2, 6, 4, 9)
answer =
  logical
   0
```

2.4. Open spaces (20 points)

Write a function with the following header:

```
function [n_open, n_max] = my_open_spaces(maze)
```

where:

- `maze` is an $m \times n$ maze following the format described above.
- `n_open` is the number of distinct groups of connected grid cells present in the maze described by `maze`. A group of connected grid cells contains at least one open space, and contains all the grid cells that are connected to this open space.
- `n_max` is the number of grid cells in the largest group of connected grid cells. The largest group of connected grid cells is the group of connected grid cells that contains the most grid cells.

Test cases:

```
>> load mazes.mat

>> [n_open, n_max] = my_open_spaces(maze_small)
n_open =
     1
n_max =
     7

>> [n_open, n_max] = my_open_spaces(maze_medium)
n_open =
    10
n_max =
    29

>> [n_open, n_max] = my_open_spaces(maze_large)
n_open =
    47
n_max =
    70
```

3. Bus line (25 points)

Consider a bus line with n_s stops. Assume that within a day, n_b buses stop at each of these bus stops. In this question, time of day t_d is measured in minutes (integer numbers only, no decimal part) elapsed from 12 am (midnight) of the same day. For example, 8:10 am is measured by the time of day $t_d = (8 \times 60 + 10)$ minutes $= 490$ minutes. The number of minutes in a day is $n_t = 60 \times 24 = 1440$. In this question, you will write a function that calculates the number of passengers carried by each bus throughout an entire day. More precisely, write a function with the following header:

```
function [n_travel, n_no_travel] = my_buses(schedule, passengers, n_max)
```

where:

- `schedule` is an $n_s \times n_b$ array of class `double` where each row represents a different bus stop (row i represents bus stop number i) and each column represents a different bus (column j represents bus number j). `schedule(i,j)` is an integer that represents the time of day at which bus number `j` arrives at bus stop number `i`. You can assume the following:
  - ◇ `schedule(i+1,j) > schedule(i,j)` (*i.e.* in `schedule`, the bus stops are in order of pick up along the bus route).
  - ◇ `schedule(i,j+1) > schedule(i,j)` (*i.e.* in `schedule`, the buses are in order of pick up, and there cannot be more than one bus at a given bus stop at any given time).
- `passengers` is an $n_s \times n_t$ `cell` array that describes when and where passengers arrive at bus stops with the intent of taking a bus.

  `passengers{i,j}` is a $1 \times n_p$ array of class `double`, where n_p is the number of passengers who arrive at bus stop number `i` and at time of day $j - 1$ (measured in minutes elapsed since midnight of the same day, as explained above – the "minus one" is required such that the index $j = 1$ represents midnight). n_p can be zero or positive. Each element of the row vector `passengers{i,j}` represents a passenger that arrives at bus stop number `i` at time of day $j - 1$, and the value of that element is a positive integer that represents the destination (as the bus stop number) of this passenger on the bus line. If `passengers{i,j}` is empty, its size can be 1×0 or 0×0.

  The table below illustrates with an example the structure of the `cell` array `passengers`. In this example, a passenger arrives at bus stop number 3 at time $t = 0$ (*i.e.* at 12 am midnight). This passenger intends to get off the bus at bus stop number 10. Four passengers arrive at bus stop number 2 at time $t = 2$ (*i.e.* at 12:02 am). Three of these people intend to get off the bus at bus stop number 8, while the other person intends to get off the bus at bus stop number 5. Remember that `passengers` is a $n_s \times n_t$ `cell` array, so it has $n_t = 1440$ columns.

	j = 1 (time $t = 0$)	j = 2 (time $t = 1$)	j = 3 (time $t = 2$)	...
i = 1 (Bus stop 1)	`[]`	`[]`	`[]`	...
i = 2 (Bus stop 2)	`[]`	`[]`	`[8, 8, 8, 5]`	...
i = 3 (Bus stop 3)	`[10]`	`[]`	`[]`	...
...	`[]`	`[]`	`[]`	...

- `n_max` is a $1 \times n_b$ array of class `double`. `n_max(i)` is a positive integer that represents the maximum number of passengers that bus number `i` can carry at once.
- `n_travel` is a $1 \times n_b$ array of class `double`. `n_travel(i)` is an integer that is positive or zero and that represents the number of passengers that bus number `i` picked up throughout the entire day.
- `n_no_travel` is a scalar of class `double`. `n_no_travel` is an integer that represents the number of passengers who arrived at bus stops but who could not get on a bus. Possible reasons for a passenger not being able to get on a bus are:
  - ◇ The passenger is waiting at the last bus stop (see below).
  - ◇ The passenger arrives at their bus stop after the last bus of the day has already stopped there.
  - ◇ All the buses that pass by the stop where and when the passenger is waiting are full.

Below are notes and assumptions that you can make:

- Buses are empty of passengers when they arrive at the first bus stop.
- Buses **never** pick up passengers at the last bus stop, even if passengers are waiting when the bus arrives and the bus still has room for more passengers.
- Assume that, for a bus, the process of stopping at a bus stop, unloading passengers, loading new passengers, and driving away from the bus stop happens instantaneously.
- At each bus stop, buses load as many of the passengers waiting there as the capacity of the bus will allow (after unloading passengers who get off the bus at this stop), but not more.
- Passengers wait indefinitely at their bus stop after they arrive there, until they are able to get on a bus.
- A passenger is in time to catch a bus if they arrive at the bus stop at a time of day that is **equal to or smaller than** the time of day at which the bus stops at the corresponding bus stop.
- When a bus stops at a stop, if not all the passengers that are waiting there can get on the bus, the bus should load the passengers who arrived first. If the bus has to decide who to load between passengers who arrived at the same time, the bus should load the passengers in the order in which they appear in the corresponding element of `passengers`.

- You can assume that each passenger that gets on a bus will eventually get off of it before the end of the day.

Writing this function from scratch may be difficult. We provide you with a template that you can use as a starting point to implement your own function. The template is available on bCourses and is reproduced here for convenience:

```
function [n_travel n_no_travel] = my_buses(schedule, passengers, n_max)

% E7 Spring 2017, University of California at Berkeley.
% Template for the function for question 3 of Lab 07.
%
% Version: release.

% Get the dimensions of the problem
% nb: number of buses
% ns: number of bus stops
% nt: number of time steps
[ns, nb] = size(schedule);
nt = size(passengers, 2);

% We are going to maintain the following lists of people:
%
% - people in each bus (there will be "nb" such lists)
%
% - people waiting at each bus stop (there will be "ns" such lists)
%
% Each list will be a row vector of class double. For a given list, the
% number of elements in the vector is the number of people in the list. The
% value of each element of the vector is the destination (as a bus stop
% number) that the corresponding person on the list wants to reach
%
% For example: the vector [3, 5, 5] indicates a list of 3 passengers. The
% first passenger wants to get off the bus at bus stop number 3, and the
% other two passengers want to get off the bus at bus stop number 5
%
% We will store the lists of passengers in each bus in the 1 by nb cell
% array named "people_travelling" (each cell of this cell array is one
% list of people)
%
% We will store the lists of people waiting at each bus stop in the ns by 1
% cell array named "people_waiting" (each cell of this cell array is one
% list of people)
people_travelling = cell(1, nb);
people_waiting = cell(ns, 1);

% Initialize the output vector "n_travel"
n_travel = zeros(1, nb);

% For each time step...
for it = 1:nt
```

```
    % Calculate the time of day in minute
    time = it - 1;

    % Update the lists of people waiting at each bus stop, by appending to
    % the corresponding lists the people who just arrived at the bus stop
    for is = 1:ns
        people_waiting{is} = % !!! Add code here !!!
    end

    % For each bus and for each bus stop...
    for ib = 1:nb
        for is = 1:ns

            % !!! Add code here !!!
            % This section needs to be completed!
            % !!! Add code here !!!

            % If this bus is not stopping at this bus stop at this time,
            % there is nothing to do

            % Unload passengers (update the relevant list in the cell array
            % "people_travelling")

            % Load passengers (update the relevant lists in the cell arrays
            % "people_waiting" and "people_travelling"). Also update the
            % value of n_travel accordingly. Don't load passengers if it's
            % the last bus stop

        end
    end

end

% Count the number of people who are still waiting for a bus at the end of
% the day
% !!! Add code here !!!

end
```

The test cases for this question are typed up in a script named `my_buses_testcases.m` that is available on bCourses. This script is reproduced here for convenience:

```
% E7 Spring 2017, University of California at Berkeley.
%
% This script contains the test cases provided for question 3 of lab 07.
%
% Version: release.

% Clear the workspace and the command window
clear all
clc
```

```
% Note: not all of the test cases presented below will be used for the
% grading of this question.

% First, we create a bus schedule. This schedule consist of:

% - 15 bus stops, each a one-hour drive away from the next
% - At each bus stop, a bus comes every 15 minutes over the span of one
%   hour (there are therefore 4 different buses on this line)
% - The first bus reaches the first stop at 6am
% - The last bus reaches the last stop at 8:45pm
schedule = [transpose([6:20]*60), transpose([6:20]*60)+15, ...
    transpose([6:20]*60)+30, transpose([6:20]*60)+45];

% Let us look at what this schedule looks like
schedule
% (obtain:)
% schedule =
%          360         375         390         405
%          420         435         450         465
%          480         495         510         525
%          540         555         570         585
%          600         615         630         645
%          660         675         690         705
%          720         735         750         765
%          780         795         810         825
%          840         855         870         885
%          900         915         930         945
%          960         975         990        1005
%         1020        1035        1050        1065
%         1080        1095        1110        1125
%         1140        1155        1170        1185
%         1200        1215        1230        1245

% This schedule might be easier to read if displayed in units of hours, as
% oppposed to units of minutes
schedule / 60
% (obtain:)
% ans =
%     6.0000    6.2500    6.5000    6.7500
%     7.0000    7.2500    7.5000    7.7500
%     8.0000    8.2500    8.5000    8.7500
%     9.0000    9.2500    9.5000    9.7500
%    10.0000   10.2500   10.5000   10.7500
%    11.0000   11.2500   11.5000   11.7500
%    12.0000   12.2500   12.5000   12.7500
%    13.0000   13.2500   13.5000   13.7500
%    14.0000   14.2500   14.5000   14.7500
%    15.0000   15.2500   15.5000   15.7500
%    16.0000   16.2500   16.5000   16.7500
%    17.0000   17.2500   17.5000   17.7500
%    18.0000   18.2500   18.5000   18.7500
%    19.0000   19.2500   19.5000   19.7500
```

```
%    20.0000   20.2500   20.5000   20.7500

% We configure each bus to have a different maximum capacity
n_max = [30, 35, 20, 15];

% We initialize the passenger variable such that there is no passenger in
% the system throughout the entire day
n_times = 60*24;
[n_stops, n_buses] = size(schedule);
passengers = cell(n_stops, n_times);

% Let us run our function with zero passengers in the system
[n_travel, n_no_travel] = my_buses(schedule, passengers, n_max)
% (obtain:)
% n_travel =
%      0     0     0     0
% n_no_travel =
%      0

% Let us add one passenger who arrives at the first bus stop at 6am. This
% passenger is headed toward the second bus stop. Note that the first time
% (index 1) represented in the array of passengers is time 0 (0am,
% midnight). which explains the +1 in the following line of code
passengers{1, 6*60+1} = [2];
[n_travel, n_no_travel] = my_buses(schedule, passengers, n_max)
% (obtain:)
% n_travel =
%      1     0     0     0
% n_no_travel =
%      0

% Let us add two passengers (both of whom arrive at the first bus stop at
% 5:45am -- one is headed to the third bus stop, the other one is headed to
% the last bus stop)
passengers{1, 5*60+45+1} = [3, n_stops];
[n_travel, n_no_travel] = my_buses(schedule, passengers, n_max)
% (obtain:)
% n_travel =
%      3     0     0     0
% n_no_travel =
%      0

% Let us add one person (headed to stop 4) who misses the last bus at the
% second stop by 1 minute
passengers{2, 7*60+46+1} = [4];
[n_travel, n_no_travel] = my_buses(schedule, passengers, n_max)
% (obtain:)
% n_travel =
%      3     0     0     0
% n_no_travel =
%      1
```

```
% Let us add a group of 100 people arriving at the first stop at 5:55am.
% They are all headed to the third stop. 3 out of these 100 people are
% unable to take the bus because three spots in the first bus were already
% taken by other passengers
passengers{1, 5*60+55+1} = zeros(1, 100)+3;
[n_travel, n_no_travel] = my_buses(schedule, passengers, n_max)
% (obtain:)
% n_travel =
%     30    35    20    15
% n_no_travel =
%      4

% Let us add a group of 100 people arriving at the second stop at 6:55am.
% They are all headed to the third stop. These people cannot get on any bus
% because all the buses are full when they arrive where the passengers are
% waiting
passengers{2, 6*60+55+1} = zeros(1, 100)+3;
[n_travel, n_no_travel] = my_buses(schedule, passengers, n_max)
% (obtain:)
% n_travel =
%     30    35    20    15
% n_no_travel =
%    104

% Let us add a group of 100 people arriving at the third stop at 7:55am.
% They are all headed to the fourth stop. All of them but one can get on a
% bus because all of the previous passengers get off at the third stop,
% except for one passenger, who is headed to the last stop
passengers{3, 7*60+55+1} = zeros(1, 100)+4;
[n_travel, n_no_travel] = my_buses(schedule, passengers, n_max)
% (obtain:)
% n_travel =
%     59    70    40    30
% n_no_travel =
%    105

% Let us add a passenger who arrives at the last stop (so they cannot get
% on a bus no matter what) at 12:30pm
passengers{n_stops, 12*60+30+1} = [n_stops];
[n_travel, n_no_travel] = my_buses(schedule, passengers, n_max)
% (obtain:)
% n_travel =
%     59    70    40    30
% n_no_travel =
%    106
```