Battleships

1 The game

You will be writing a program that will play one half of a game of battleships. You only have to guess the positions of a hidden fleet of ships, but will not place any ships of your own.

In our version of battleships the game starts with the computer secretly placing a fleet of 5 ships on a 10×10 grid as shown below. The positions of the ships are fixed, but unknown to you.

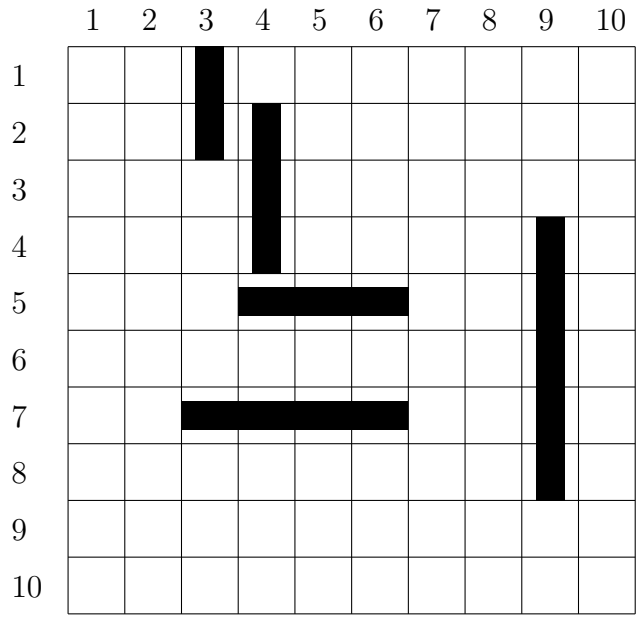

Figure 1: A valid starting grid in a game of battleships

The fleet is made up of

- one 2 block ship
- , two 3 block ships,
- , one 4 block ship
- , one 5 block ship

The ships are placed as follows:

- Ships can only lie *horizontally or vertically*, but not diagonally.
- Ships must lie *entirely within the grid*.
- Ships *cannot cross one another* (no two ships can occupy the same block)
- Different ships *may occupy adjacent blocks*.

- Ships are *placed randomly*.

As the game progresses you try to sink all of the computer's ships by guessing where they are. Each guess is called a "shot" and is aimed at a (row, column) coordinate. The computer will respond by indicating whether the shot was a hit or a miss. For example, a shot at position (7, 3) would register a hit on our example grid in figure 1, while a shot at position (9, 6) would register a miss.

If all the blocks that a ship occupies have been shot, the ship is sunk and the computer will indicate this in addition to the normal indication of a hit. Thus, if positions (1, 3) and (2, 3) were both shot, the computer would respond after the second shot by indicating that a ship of length 2 has been sunk and which blocks it occupied. As soon as the final ship is sunk, the game ends. The object of a game is to use as few guesses as possible.

2 Computer implementation

You will be supplied with a file called `battle.m`. This MATLAB function handles the placement of a fleet of ships responds to shots as described above. To start a game, run `battle('init')`. You may now start to shoot at coordinates by calling `battle(r, c)` where `r` and `c` are a row and column coordinate to shoot at. The function will return two values (`result` and `value`), the meaning of which are described in table 1:

Table 1: Meaning of output variables of the battle function

`result`	`value`	Meaning
-1	`[]`	Duplicate shot
0	`[]`	Miss
1	`[]`	Hit, no ship sunk
1	N×2 matrix	Ship of length N sunk
2	scalar	All ships sunk. `value` contains number of shots.

Calling `battle('finish')` causes the function to return two outputs – `finished`, which is zero if all the ships have not been sunk, and `shots`, which is the number of shots fired during this game.

If `battle('init', 1)` is used, a graphical representation of the state of the game as it progresses will be generated. Note that a small delay is added if the graphics are enabled. This causes an animation effect. The graphical representation uses a white dot to indicate a missed shot and a red dot to indicate a hit. When a ship is sunk, a line is drawn through all the points where that ship was.

3 Program requirements

You must write a function called `findships` that will be called without arguments *after* the game has been initialised and will play the game of battleships. An example session may look like this (the graphical output generated by this run is shown in figure 2):

```
>> battle('init', 1);
>> findships
>> [finished, shots] = battle('finish')
finished =
     1
shots =
     50
```

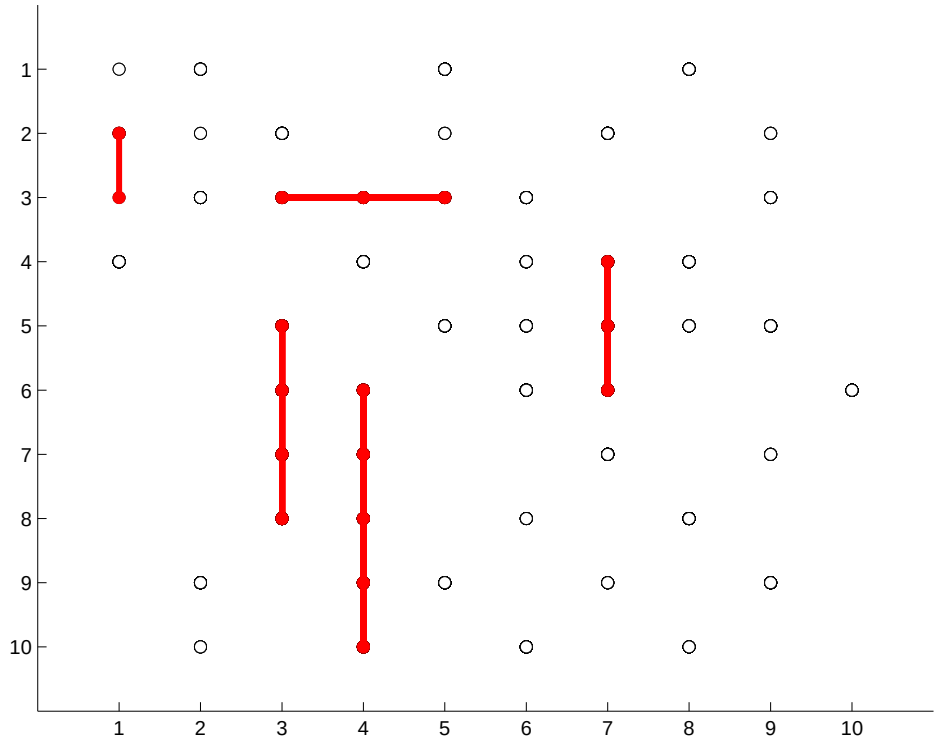

Figure 2: Sample graphics after a complete game

Note that the default behaviour of your program should be to generate *no output* to the screen. Any debugging messages should be disabled before handing the program in. Your program must terminate when it has sunk all the ships (when `battle` returns 2).

4 Testing your program

To help you determine the efficacy of your program, you will be supplied with a function called `testfindships`, which will run `findships` a certain number of times and display statistics about the runs. A sample run is shown below:

```
>> testfindships(500);
  run   min     avg   max   avg time (miliseconds)
  100    36    56.0    96       88
  200    36    55.0    96       86
  300    31    54.1    96       84
```

```
400     31     54.9     99        86
500     31     55.2     99        87
```

The function shows the number of runs completed, the minimum, the average and the maximum guesses required to find the ship, and the average time per run that the `findships` function required. Note that `testfindships` will output a vector containing the exact results for each game for further analysis.

For instance, you could use the output of `testfindships` to draw a histogram showing the distribution of your results:

```
>> t = testfindships(1000);
>> hist(t)
```

Output is shown in figure 3

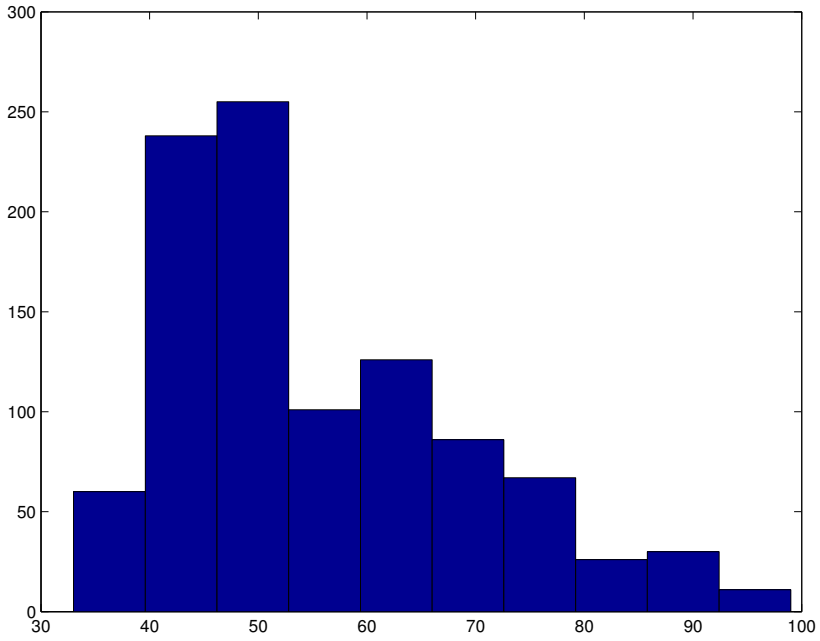

Figure 3: Histogram of guesses taken over 500 games

5 Summary

You will be supplied with the following files:

Filename	Call	Purpose
`battle.m`	`battle('init')`	set up board without graphics
	`battle('init', 1)`	set up board with graphics
	`battle(r, c)`	fire a shot at row `r`, column `c`
	`battle('finish')`	determine status of game
`testfindships.m`	`testfindships(N)`	simulate `N` games of battleships

You must submit a file called `findships.m`, which will be called without arguments and call `battle` to fire shots.