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

Programming Project: E7 Planets
Version: release

Due date: Friday April 28th 2017 at 11:59 pm.

1. Introduction and objectives

For this semester's E7 programming project, you will apply your programming skills to programming a virtual spaceship that is traveling through space, trying to collect scrap while avoiding obstacles such as asteroid fields and (space) ghosts!

Although this programming project uses a game setting, named "E7 Planets", as a learning tool (we do hope that you will have fun working on it), the primary goal of this project is to provide you with a setting and a task to accomplish that will allow and/or require you to:

- Apply many, if not all, of the programming concepts learned in this class;
- Express your creativity and ingenuity, design your own algorithms, and maybe test several of them to see which approaches work best;
- Write code that will likely be longer and more complex than the code that you are used to writing for E7 lab assignments;
- Think in a modular way, and divide your code in multiple sub-functions, each performing specific tasks; and
- Understand that there may not be a unique solution or a perfect solution, but rather different approaches that may or may not work in different situations. You will likely have to prioritize your efforts on the aspects of the game that you think are the most important.

2. The rules of E7 Planets

The rules of E7 Planets are described below:

- **The grid**. Your spaceship will move in a two-dimensional gridded domain with m rows and n columns; m and n will vary depending on which map the game is being played on. This domain will be represented in Matlab by a $m \times n$ array of class `double`, which we refer to as "the grid" for simplicity. The location of any element in the game is fully described by its row and column indices in the grid.
- **Turns and moves**. During each turn of the game, your spaceship will have the opportunity to move by one grid cell, either up, down, left, or right. Your spaceship can also stay

in place. In E7 Planets, "moving up" means moving in the grid along a given column, in the direction of decreasing row indices, "moving down" means moving in the grid along a given column, in the direction of increasing row indices, "moving left" means moving in the grid along a given row, in the direction of decreasing column indices, and "moving right" means moving in the grid along a given row, in the direction of increasing column indices.

- **Asteroid fields (*i.e.* impassable areas)**. There may be impassable areas in the domain. We call these impassable areas "asteroid fields". Each grid cell that is impassable will have a value of `Inf` in the grid.
- **Empty space and nebulae (*i.e.* slow-down areas)** Each grid cell that is not impassable will either be an "empty space" (represented by a zero in the grid) or a "nebula" (represented by a non-infinite positive integer in the grid). Whenever your spaceship moves into a different grid cell, it will have to skip a number of turns equal to the corresponding number in the grid before it can move again. For example, if your spaceship moves to an empty space (the corresponding value in the grid is zero), it will be able to move again during the following turn. In contrast, if your spaceship enters a nebula whose corresponding value in the grid is 3, then your spaceship will not be able to move during the following 3 turns (it will, however, be able to move on the fourth turn after entering the nebula).
- **Scraps and score**. The objective of the game is for your spaceship to pick up "scraps" in the domain. Your spaceship picks up a scrap by landing, after moving, in the grid cell where the scrap is located. Scraps have different values: some of them are worth more than others. The value of each scrap is a positive non-infinite integer. Your score is equal to zero at the beginning of the game. At the end of the game, your score is equal to the accumulated value of the scraps that your spaceship picked up throughout the game. Scraps do not move.
- **Wrap-around moves**. Your spaceship can use a special type of move, called a "wrap-around move". A wrap-around move consists of moving outside of the limits of the map to re-appear on the opposite side of the map. For example, if your spaceship moves up from the cell located in the first row and third column of the grid, it will re-appear in the cell located in the last row and third column of the grid. Your spaceship may not use a wrap-around move if the destination grid cell is an asteroid field.
- **Ghosts**. There may be ghosts in the domain. Ghosts can move in the domain. Your spaceship is caught by a ghost either if your spaceship and the ghost are located in the same grid cell at the end of a turn, or if your spaceship and a ghost switch locations over the course of a turn. During each turn, your spaceship and the ghosts all move simultaneously. The different types of ghosts and their attributes are described further below. None of the ghost types implemented in this version of E7 Planets ever makes use of wrap-around moves. In other words, all the ghosts view the boundaries of the domain as impassable.
- **Starting locations**. At the beginning of the game, your spaceship, each scrap, and each ghost are in different locations, and all these locations are outside of asteroid fields.

- **Turns**. Each game has a limited number of turns. This number is at least one.
- **End of the game**. The game ends if any of the following conditions is true:
  - ◇ There are no more turns.
  - ◇ Your spaceship picked up all the scraps on the map.
  - ◇ Your spaceship got caught by one or more ghost(s).

  You win the game if, at the end of the game, your spaceship has picked up all the scraps on the map, without getting caught by any of the ghosts.

Note: in this project, the names "empty space", "asteroid field" and "nebula" are not used for their rigorous physical meanings, but are just used to add a bit of flavor to this project.

3. Your task

3.1. The player function

You are tasked with writing a function, the "player function", that determines, given a map, the direction in which the spaceship should be moving next. Your function will be called once during each turn of the game, to determine the spaceship's next move, one move at a time. More precisely, write a function with the following header:

```
function [direction] = e7planets_player(map)
```

where:

- `map` is a 1×1 `struct` array that describes the map area, including the nebulae, asteroid fields, scraps, and ghosts. The structure of `map` is described in more details below.
- `direction` should be a 1×1 array of class `char` (*i.e.* a single character), and should have one of the following five values (case matters!):
  - ◇ `'U'` (upper case U). In this case, your spaceship will move up if possible this turn.
  - ◇ `'D'` (upper case D). In this case, your spaceship will move down if possible this turn.
  - ◇ `'L'` (upper case L). In this case, your spaceship will move left if possible this turn.
  - ◇ `'R'` (upper case R). In this case, your spaceship will move right if possible this turn.
  - ◇ `'.'` (period character). In this case, your spaceship will not move this turn.

  If `direction` is not one of the five possibilities listed above, or if the move is not possible (*i.e.* if your spaceship would move into an asteroid field if it were to follow the direction specified by your function's output argument), your spaceship does not move this turn.

The input argument `map` will have the following fields (and no other field):

- `grid`: a $m \times n$ array of class `double` that represents the grid, following the format described in Section 2.
- `player`: a 1×1 `struct` array with the following fields:

- ◇ `location`: a $(n_t+2)\times 2$ array of class `double`, where n_t is the number of turns of the game that have already been played. The first row of the array will always be `[NaN,NaN]`. Row number i of this array ($i > 1$) represents the location of your spaceship in the grid (as the row and column index of this location, in this order) at the start of turn number $(i-1)$. In particular, the second row of this array represents the location of your spaceship at the start of the game, and the last row of this array represents your spaceship's current location.
- ◇ `score`: a scalar of class double that represents your current score *i.e.* the accumulated value of the scraps that your spaceship has picked up so far.

• `scraps`: a $n_s \times 1$ `struct` array, where n_s is the number of scraps currently on the map ($n_s > 0$). Each element of this `struct` array represents one scrap. This `struct` array contains the following fields:
  - ◇ `location`: 1×2 array of class `double` that describes the location of the scrap in the grid, as the row and column index of this location, in this order.
  - ◇ `value`: a scalar of class `double` that represents the value of the scrap.

• `ghosts`: a $n_g \times 1$ `struct` array, where n_g is the number of ghosts on the map. Each element of this `struct` array represents one ghost. This `struct` array is empty (there can be zero ghost on the map) or contains the following fields:
  - ◇ `location`: a $(n_t+2)\times 2$ array of class `double` that describes the location of the ghost during the previous turns of the game, following the same format as `map.player.location`.
  - ◇ `type`: a row vector of class `char` (*i.e.* a character string) that represents the type of the ghost. Possible types are:
    - – `'backandforth'`: the ghost moves back and forth in a straight line. In each direction, it moves until it cannot go any further, in which case it turns around and move in the opposite direction. The first row of the location array of this ghost is not `[NaN,NaN]`, but represents the "pre-starting" location of the ghost in the grid (as the row and column index of this location, in this order). The initial moving direction of the ghost is determined by comparing the ghost's pre-starting and starting locations *i.e.* the first and second rows of the location array of the ghost, respectively. The starting location of the ghost must be either:
      - ∗ one grid cell above its pre-starting location (in which case the ghost will first move up if possible, and down otherwise); or
      - ∗ one grid cell below its pre-starting location (in which case the ghost will first move down if possible, and up otherwise); or
      - ∗ one grid cell to the left of its pre-starting location (in which case the ghost will first move left if possible, and right otherwise); or
      - ∗ one grid cell to the right of its pre-starting location (in which case the ghost will first move right if possible, and left otherwise).

      This ghost does not use "wrap-around" moves, and views the boundaries of the domain as impassable.
    - – `'towardplayer'`: the ghost always tries to move toward the player. This ghost

does not try to go around objects to get to the player, so this ghost may get stuck behind impassable areas (unless its transparency allows it to move through otherwise impassable areas). This ghost does not use "wrap-around" moves, and views the boundaries of the domain as impassable.

  ◇ `transparency`: a scalar of class `double` that is either `0`, `1`, or `2`. A value of `0` indicates that the ghost cannot go through asteroid fields, and is slowed down by nebulae, just as the player is. A value of `1` indicates that the ghost cannot go through asteroid fields, but is not slowed down by nebulae (the ghost moves through nebulae as if they were empty spaces). A value of `2` indicates that the ghost can move through all locations of the map as if they were empty spaces.

- `remaining_turns`: a scalar of class `double` that represents the maximum number of turns that remain in this game.

If your player function's header is different from the header specified above, you will receive a score of zero for each map your function is tested on.

3.2. Invalid moves

Your function must calculate and return a moving direction in one second or less. In a given turn, if your function takes more than one second to return an output, your move will not be considered valid, and your spaceship will not move. The grading will be done on the computers of the 1109 Etcheverry computer lab.

In a given turn, if your function **throws an error** when called, or **outputs something other than one of the five possibilities mentioned above**, the corresponding move will not be considered valid, and your spaceship will not move this turn.

The grading will be done on the computers of the 1109 Etcheverry computer lab. It is your responsibility to ensure that all the built-in functions that your player function relies on are installed on these computers. If you are not sure whether a particular built-in function is available on these computers, you can ask on the project-specific Discussion in bCourses.

We recommend that you do not use `while` loops anywhere in your code. **If your function enters an infinite `while` loop and/or takes more than one minute to return when tested on a map, you will receive a score of zero for this map.**

4. The engine and visualizer

We provide you with the "game engine", which is a function named `e7planets_play`. The corresponding m-file, `e7planets_play.m`, is available on bCourses. Given a map and a handle to your function `e7planets_player`, this function runs the corresponding game, determines your score, and creates an interactive Matlab figure which we call the "visualizer". The visualizer shows what happens in the game. You can use the visualizer to show the grid and the location of all game elements frame by frame (*i.e.* turn by turn) as the game proceeds, go back to previous frames, jump to the last frame, and so on. Figure 1 shows the visualizer showing the first frame of the map `spiral_ghosts` (see Section 5). Interaction with the visualizer is done through keyboard shortcuts. For example, pressing the "right arrow" key makes the visualizer advance to the next frame. Press the "h" key to display, in the command window, the list of available keyboard shortcuts. Alternatively, the available keyboard shortcuts are listed in Table 1.

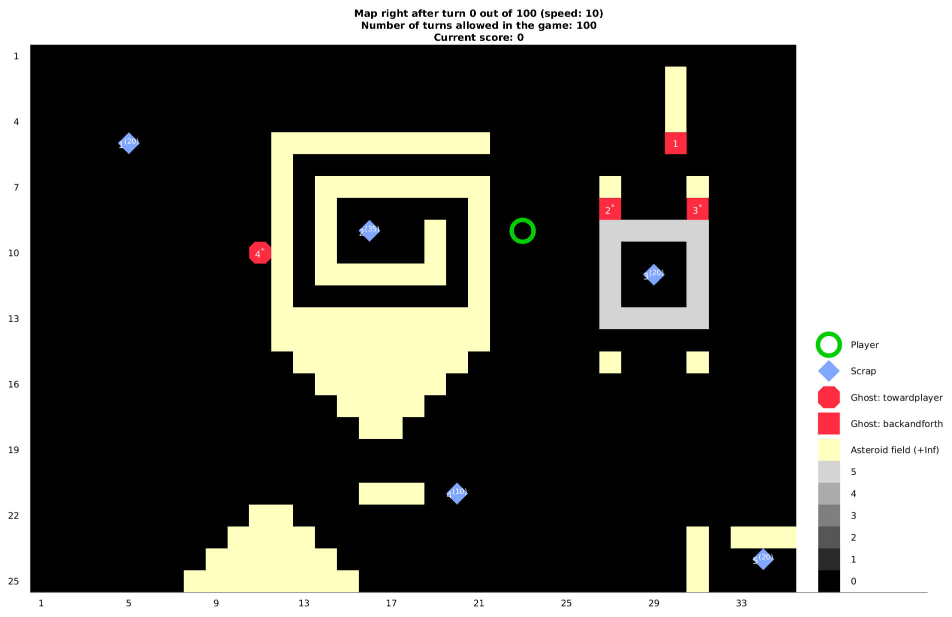

Figure 1: Visualizer showing the first frame of the map `spiral_ghosts`.

Table 1: Keyboard shortcuts that can be used in the visualizer. When several keys are bound to the same action, these keys are listed as a comma-separated list.

Key(s)	Action
right arrow, down arrow	Advance to the next frame
left arrow, up arrow	Go back to the previous frame
home key, page up	Jump to the initial frame
end key, page down	Jump to the last frame
escape key, q	Close the visualizer
enter	Run the game forward
backspace	Run the game backward
f	Increase the frame rate of the visualizer
s	Decrease the frame rate of the visualizer
h	Show this list of shortcuts in the command window

`e7planets_play` has the following header:

```
function [game_stats] = e7planets_play(map, player_function)
```

where:

- `map` is the map to be played, in its initial state. This input argument has the same structure as the input argument of the same name in your function `e7planets_player`.
- `player_function` is a function handle to a "player function" (typically the function `e7planets_player` that you will be writing for this project).
- `game_stats` is a 1×1 `struct` array that describes statistics about the game. `game_stats` contains the following fields:
  - ◇ `win`: a `logical` scalar that is true if and only if you won the game.
  - ◇ `max_turns`: a scalar of class `double` that represents the maximum number of turns that were allowed in the game.
  - ◇ `n_turns`: a scalar of class `double` that represents the number of turns in the game before the game was over.
  - ◇ `score`: a scalar of class `double` that represents your score for this game.
  - ◇ `scraps_total_value`: a scalar of class `double` that represents the total value of the scraps initially present on the map. Note that this value includes the values of scraps that are unattainable.
  - ◇ `scraps_picked_up`: a `struct` array that follows the same format as `map.scraps`, and that describes the scraps that were picked up by your spaceship during the game (in the order they were picked up).
  - ◇ `scraps_left`: a `struct` array that follows the same format as `map.scraps`, and that describes the scraps that were not picked up by your spaceship during the game.
  - ◇ `caught`: a `logical` scalar that is true if and only if your spaceship was caught by one or more ghost(s) during the game.
  - ◇ `ghosts_catch_samespot`: a row vector of class `double` that contains the indices (in `map.ghosts`) of the ghosts that caught your spaceship during the game by landing on the same spot as your spaceship.
  - ◇ `ghosts_catch_switch:` a row vector of class `double` that contains the indices (in `map.ghosts`) of the ghosts that caught your spaceship during the game by switching locations with your spaceship over the course of one turn.

Note that the order of the scraps in `game_stats.scraps_picked_up` and `game_stats.scraps_left` may differ from their order in `map.scraps`.

For example, to test your player function on the map named `spiral_ghosts` you can run the following code (see Section 5 for information about the sample maps provided to you):

```
>> load('e7planets_maps_published.mat');
>> game_stats = e7planets_play(spiral_ghosts, @e7planets_player)
```

If you find a bug in `e7planets_play.m`, please report it by email to the class' instructor. Please check the project-specific Frequently Asked Question (FAQ) Page on bCourses before reporting a bug, as the issue might already be described there.

5. Sample maps

We provide you with a number of sample maps on which to test your function. These sample maps are packaged in the `mat` file named `e7planets_maps_published.mat`, which is available on bCourses. Each map in this file comes in two versions: one without ghosts, and one with ghosts. The name of a map with ghosts is the same as the name of the corresponding map that does not have ghosts with the suffix `_ghosts` added to it. For example, the map `spiral_ghosts` is the same as the map `spiral` but with ghosts. The maps available in `e7planets_maps_published.mat` are listed in Table 2. The `mat` file named `e7planets_maps_published.mat` also contains a variable named `maps_version`, which is a character string (*i.e.* a row vector of class `char`) that indicates the version of the file. This character string will be one of `'release'`, `'revision01'`, `'revision02'`, and so on.

Table 2: List of the maps available in `e7planets_maps_published.mat`. This list is roughly sorted in order of increasing difficulty (from the top to the bottom of the table). A map with ghosts is more difficult than the corresponding map without ghosts.

		Size of the grid
`one_scrap`	`one_scrap_ghosts`	15×11
`two_scraps`	`two_scraps_ghosts`	15×11
`two_scraps_block`	`two_scraps_block_ghosts`	15×11
`two_scraps_nebula`	`two_scraps_nebula_ghosts`	6×11
`with_unreachable`	`with_unreachable_ghosts`	15×11
`three_blocks`	`three_blocks_ghosts`	15×20
`corners`	`corners_ghosts`	35×35
`spiral`	`spiral_ghosts`	25×35

6. Instructions, deliverables, due dates, and grading

6.1. General instructions and scoring

These instructions and the code contained in the file `e7planets_play.m` provided to you may be used only for the purpose of working on this final project. **Neither these instructions nor the code contained in the file `e7planets_play.m` provided to you may be shared with anyone, nor uploaded nor posted online anywhere, without prior written consent of the instructor of the class.**

For this project, you will work in a team of 2 or 3 students. Your team must consist of students in your lab section. You are allowed to write code together for this and only this assignment. You may share code only within your team, not with other students in the class.

You may not post your code online in a place that is accessible to other students of the class.

You may not use code from outside sources in your submissions. You may implement yourself algorithms whose descriptions you find online, but **you may not use code that you find online**. Ask the class' instructor and/or a GSI if you have doubts or if you are confused about this distinction. The **only** code (no exception) that is not your own code and that you can use in your project submissions is the code available on bCourses that is provided to you as part of this semester's E7 lectures and as solutions to this semester's E7 lab assignments, as well as the code in `e7planets_play.m`. We will control for plagiarism in your submissions. **If plagiarism is detected in your submissions, you will receive a score of zero for the entirety of the project.**

Your function must rely solely on the information provided in its input argument `map` to decide in which direction your spaceship should move next. Any attempt to use other information will be considered cheating. Cheating also includes exploiting a bug in the engine code (*i.e.* in `e7planets_play.m`, see Section 4) that results in the rules of the game described above not being respected, reading and/or writing data to disk, and accessing any internet resource from your player function. **If caught cheating, you will receive a score of zero for the entirety of the project.**

If your player function (`e7planets_player`) depends on other user-defined functions, these functions must be defined as nested or sub-functions of your main function `e7planets_player`, as opposed to being defined in separate m-files.

Your project will be scored out of 100 points as follows:

- **Beta test** (due Friday April 14th 2017 at 12 pm – noon): 15 points
- **Final code** (due Friday April 28th 2017 at 11:59 pm): 70 points
  - ⋄ Performance of your function on type-1 maps: 20 points.
  - ⋄ Performance of your function on type-2 maps: 30 points.
  - ⋄ Performance of your function on type-3 maps: 20 points.
- **Final write-up** (due Friday April 28th 2017 at 11:59 pm): 15 points

Each item of the grading rubric is described in more detail below.

6.2. Beta test

Beta testing is an early release of software to friendly users for testing. After spending some time working on the project, you will submit a version of your code for testing and feedback. In the beta test, your function will be tested on three different maps. None of these maps will contain asteroid fields, nebulae, nor ghosts. Each of these maps will contain only one scrap, which will be accessible within the number of turns allowed by the map without using "wrap around" moves. You will get 5 points for each map on which your function wins the game.

In your submission, you can include questions for your GSI so you can get some feedback on your progress so far. We recommend you to work hard on the project before the beta test due date, because the more you turn in by this deadline, the more feedback you can get from your GSI.

For the beta test, **submit on bCourses a single file named `e7planets_player.m` (*i.e.* your player function)**. Include questions that you have for your GSI as bCourses comments along with your submission. The due date for the beta test is Friday April 14th 2017 at 12 pm (noon), with a two-hour grace period.

6.3. Final code

Your function will be tested and graded on maps of varying difficulty, some of which are provided to you in the file `e7planets_maps_published.mat`, some of which are not. We define three types of maps: type-1, type-2, and type-3. **Type-1 maps** are maps that contain no asteroid fields, nebulae, nor ghosts, and where all the scraps can be picked up by your spaceship within the number of turns allowed by the map, without using "wrap around" moves. **Type-2 maps** are maps that contain no ghost. Certain type-2 maps can also qualify as type-1 maps. Any map qualifies as a **type-3 map**.

For a given map, even if you do not win the game, you will get partial credit for picking up scrap before the game ends. The higher your score at the end of the game, the more partial credit you will get. Note that certain maps may have scraps that are not accessible, or may not allow you enough moves for you to pick up all the scraps. These facts will be accounted for in the grading process, and you do not necessarily need to win all the games that your function will be tested on to obtain full credit for the project.

As your final code, **submit on bCourses a single file named `e7planets_player.m` (*i.e.* your player function)**. The due date for the final code is Friday April 28th 2017 at 11:59 pm, with a two-hour grace period.

6.4. Final write-up

Write (in your own words) and submit a 1- to 2-page summary of your final code, which includes an overview of your algorithm, how it works, and why you made the design decisions you did. You should highlight the key features and general structure of your code, as opposed to focusing on the details. Include a title and your team member names at the top of your summary, and make proper use of headings, sections, and figures as needed to support the text. Your write-up will be graded by your GSIs, based on the use of appropriate technical writing, and the adequacy of your description (we should be able to understand your methodology without looking at your code). Also make sure your write-up is within the page limit to receive full credit.

Submit your final write-up **on bCourses as a single PDF file named `project.pdf` (formats other than PDF will not be accepted)**. The due date for the final write-up is Friday April 28th 2017 at 11:59 pm, with a two-hour grace period.

7. Recommendations

Work on basic functionality before working on advanced features.

First, start with writing a player function that makes your spaceship always try to move up, and try running the game on different maps. Once you are comfortable with using `e7planets_play`, start implementing your player function in more detail.

Brainstorm! Spend time with your team talking about different approaches you could take. Think about all the concepts you have learned in class and how you might apply each of them to your project *e.g.,* iteration, recursion, branching, regression, root finding, series, interpolation, integration, etc. You never know what might trigger an idea for a new algorithm!

Create a modular program so you can divide the project into pieces that can be written and tested individually, and that each team member can work on separately.

Use functions and sub-functions to organize your program in a logical way. Test the pieces separately before putting them all together; otherwise, debugging might become more tedious than necessary.

Your code should be well commented, to facilitate team work.

Be creative! Remember that there is no unique or perfect solution, but rather different approaches that may or may not work in different situations; you will likely have to prioritize your efforts on the aspects of the game that you think are the most important.

We encourage you to make up your own maps to test how your function performs in a variety of situations.

Be wary of last-minute changes to your code. Always test your final code before submitting it.

You should always make sure you are working with the latest version of:

- These instructions (the version is indicated in the header of this document).
- The engine and visualizer code (the version is indicated in the comments of the top-level function in `e7planets_play.m`).
- The sample maps (the version is indicated by the variable `maps_version` in the file `e7planets_maps_published.mat`).

The versions will be kept consistent across these three resources. The name of the latest version will be indicated in the project-specific Frequently Asked Question (FAQ) Page on bCourses.