Homework 1 (Matlab version)

ME570 - Prof. Tron

2021-09-01

The goal of this homework is to warm up your programming and analytical skills. This homework does not use any material specific to path planning, but the problems you will encounter here will *1)* give you an idea of the structure, difficulty and scope of future homework assignments, and *2)* prepare tools (functions) that will be useful to learn path planning concepts. In order to successfully complete this (and future) homework assignments, you will have to combine your Matlab knowledge with critical and creative thinking skills.

General instructions

Programming For your convenience, together with this document, you will find a `zip` archive containing Matlab files with stubs for each of the questions in this assignment; each stub contains an automatically generated description and header of the function . You will have to complete these files with the requested code. The goal of this files is to save you a little bit of time, and to avoid misspellings in the function or argument names. The files for the parts marked as **provided** (see also the *Grading* paragraph below) contain already the body of the function.

Homework help For best coding practices, please refer to the guidelines on Blackboard under Class content/Programming Tips & Tricks/Matlab. For questions specific to the content of the homework, please post on the Blackboard discussion board.

Homework report Along the programming of the requested functions, prepare a PDF report containing one or two sentences of comments for each question marked as **report**, and including: embedded figures and outputs that are representative of those generated by your code. Include comments on the questions marked as **code** only to explain any difficulty you might have encountered.

A small amount of *beauty points* are dedicated to reward reports that present their content in a professional way (see the *Grading criteria* section in the syllabus).

Analytical derivations To include the analytical derivations in your report you can type them in LaTeX(preferred method), any equation editor or clearly write them on paper and use a scanner (least preferred method).

Submission

The submission will be on Gradescope through three separate assignments: one for the questions marked as **code**, and one for those marked as **report**, and one for providing feedback. Further details are explained below. You can submit as many times as you would like, up to the assignment deadline. Each question is worth 1 point unless otherwise noted. Please refer to the Syllabus on Blackboard for late homework policies.

Report Upload the PDF of you report, and then indicate, for each question marked as **report**, on which page it is answered (just follow the Gradescope interface). Note that some of the questions marked as **report** might include a coding component, which however will be evaluated from the output figures you include in the report, not through automated tests. In general, these questions are intended as checkpoints for you to visually check the results of your functions.

Code questions Upload all the necessary `.m` files, both those written by you, and those provided with the assignment. The questions marked as **code** will be graded using automated tests: green and red mean that the test has, respectively, passed or not; if a test did not pass, check for clues in the name of the test, the message provided on Gradescope, and the text of this assignment. *Note:* The automated tests use Octave, a open-source clone of Matlab. For the purposes of this class, you should not encounter any specific compatibility problem. If, however, you suspect that some tests fail due to this incompatibility, please contact the instructor.

Optional and provided questions. Questions marked as **optional** are provided just to further your understanding of the subject, and not for credit (if submitted, I will provide comments but it will not count toward your grade).

Hints

Some hints are available for some questions, and can be found at the end of the assignment (you are encouraged to try to solve the questions without looking at the hints first). If you use these hints, please state so in your report (your grading will not change based on this information, but it is a useful feedback for me).

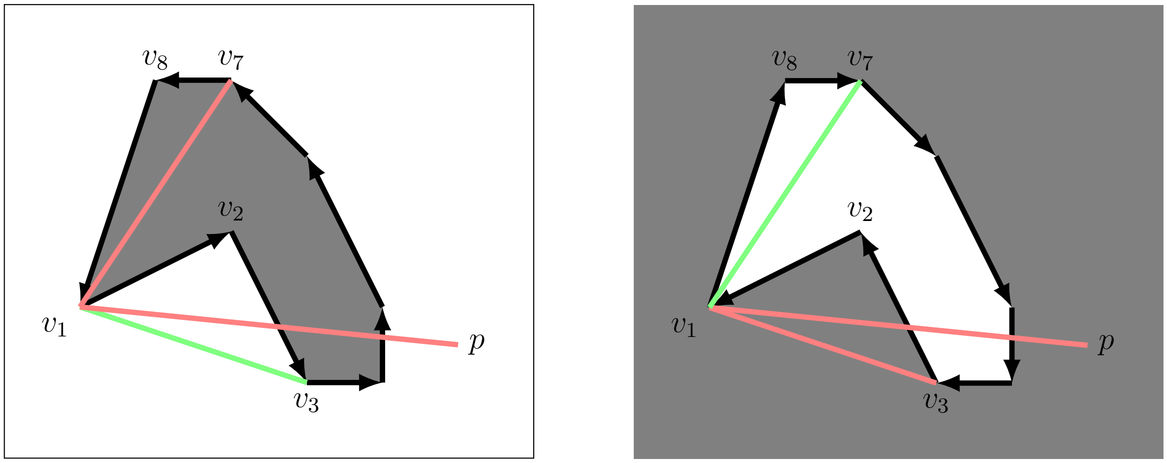
Use of external libraries and toolboxes You are **not allowed** to use functions or scripts from external libraries or toolboxes (e.g., mapping toolbox), unless specifically instructed to do so (e.g., CVX).

Problem 1: Drawing, visibility and collisions for 2-D polygons

In this problem you will write functions to draw a 2-D polygon, test if a vertex is visible from an arbitrary point, and test if a given point is inside or outside the boundary (collision checking). These functions will be useful in later homework assignments.

Data structure. We represent the polygon using a matrix `vertices` with dimensions [2 × `NVertices`], where `NVertices` is the number of points in the polygon; the first and second row of the matrix represents, respectively, the x and y coordinates of the boundary of the polygons. The polygons are assumed to not be self-intersecting. We will use the ordering of the vertices with respect to an internal point to distinguish the solidity of the polygon (see Figure 3b):

- If the vertices are counterclockwise ordered, they define a filled-in polygon;
- If the vertices are clockwise ordered, they define an hollow polygon.



(a) Filled polygon (counterclockwise ordering)

(b) Hollow polygon (clockwise ordering)

Figure 1: Examples of visibility. Green and red lines mean points that, respectively, are visible and not visible from each other. Line v_1–v_3 in (a) and v_1–v_7 in (b): visible. Line v_1–p in both (a) and (b): edge intersections. Line v_1–v_7 in (a) and v_1–v_3 in (b): self-occlusions.

As part of this problem, you will be asked to program functions that determine the visibility of a point from a vertex of the polygon. There are two reasons for which the two points might fail to be visible from each other:

1) There is an edge blocking the line of sight (line v_1–p in both Figures 3a and 3b);

2) The line of sight falls inside the obstacle, that is, there is a *self-occlusion* (line v_1–v_7 in Figure 3a and line v_1–v_3 in Figure 3b).

Collision checking will be implemented by using the visibility functions.

Question **report** **1.1.** Drawing polygons.

> `polygon_plot` (`vertices,style`)
>
> Description: Draw a closed polygon described by `vertices` using the style (e.g., color) given by `style` .
>
> Input arguments
>
> - `vertices` (dim. [2 × `nbVertices`]type): array where each column represents the coordinates of a vertex in the polygon.
> - `style` (type `string`): a style specification that follows Matlab's standard conventions (e.g., `'r'` for red).
>
> Requirements: Each edge in the polygon must be an arrow pointing from one vertex to the next. In Matlab, use the function `quiver` ( ) to actually perform the drawing. The function should *not* create a new figure.

In the report, include two figures with the plots of a filled-in polygon and a hollow polygon of your choice.

Question **optional** **1.1.** Check if a polygon is filled-in or hollow.

> [`flag`]=`polygon_isFilled`(`vertices`)
> Description: Checks the ordering of the vertices, and returns whether the polygon is filled in or not.
> Input arguments
>
> - `vertices` (dim. [2 × `nbVertices`]type): array where each column represents the coordinates of a vertex in the polygon.
>
> Output arguments
>
> - `flag` (type `logical`): `true` if the polygon is filled in, and `false` if it is hollow.

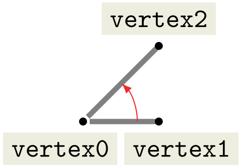
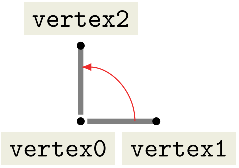
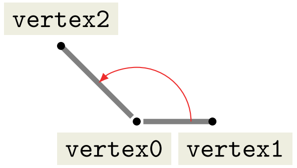
Modify the previous function `polygon_plot`( ) to use the `patch`( ) (which draws solid polygons) if the polygon is filled in (the behavior should be the same for hollow polygons). If you complete this question, add some comments in your report about the method you used.

Question code 1.1. Check for collision between edges.

> [`flag`]=`edge_isCollision`(`vertices1,vertices2`)
> Description: Returns `true` if the two edges intersect. *Note:* if the two edges overlap but are colinear, or they overlap only at a single endpoint, they are not considered as intersecting (i.e., in these cases the function returns `false`). If one of the two edges has zero length, the function should always return the result that edges are non-intersecting.
> Input arguments
>
> - `vertices1` (dim. [2 × 2]type), `vertices2` (dim. [2 × 2]type): coordinates of the endpoints of two edges
>
> Output arguments
>
> - `flag` (type `logical`): `true` if the edges intersect, `false` otherwise.
>
> Requirements: The function should be able to handle any orientation of the edges (including both vertical and horizontal). For the case where only one endpoint overlaps (i.e., the edges form a "T"), in the context of this homework, you can decide; nonetheless, if you want to specifically consider this case, it is recommended that the edges are considered overlapping (this choice will fix some very rare corner cases in future homework assignments). Note that the "overlap" case needs to be checked up to floating-point precision.

provided Compute the counterclockwise angle between two line segments.

> [`edgeAngle`]=`edge_angle`(`vertex0,vertex1,vertex2,angle_type`)
> Description: Compute the angle between two edges `vertex0`–`vertex1` and `vertex0`–`vertex2` having an endpoint in common. The angle is computed by starting from the edge `vertex0`–`vertex1`, and then "walking" in a counterclockwise manner until the edge `vertex0`–`vertex2` is found.
> Input arguments
>
> - `vertex0` (dim. [2 × 1]type), `vertex1` (dim. [2 × 1]type), `vertex2` (dim. [2 × 1]type): coordinates of the three vertices defining the two edges.

			
'signed'	0.785	1.571	2.356
'unsigned'	0.785	1.571	2.356

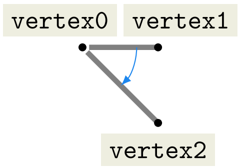
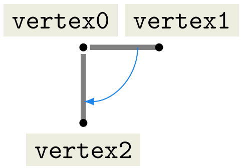
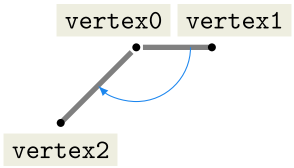
			
'signed'	-0.785	-1.571	-2.356
'unsigned'	3.927	4.712	5.498

Table 1: Examples of the input and outputs for the function `edge_angle`

- `type` (type `string`): can be `'signed'` or `'unsigned'` to specify the range of the computed angles. Defaults to `'signed'`.

Output arguments

- `edgeAngle` : angle expressed in radians. If `'unsigned'` is specificed, the angle is in the interval $[0, 2\pi)$; if `signed` is specified, the angle is in the interval $[-\pi, \pi)$.

See Table 1 for some illustrative examples of the input and outputs of this function.

Question report **1.2.** Examine the content of the function `edge_angle`( ). Explain what is the significance of the variables `sAngle` and `cAngle`, and explain how the angle `theta` is computed. Include a figure illustrating your reasoning.

Question code **1.2.** Check if a point is self-occluded by a corner of a polygon. See Figure 2 for examples of the expected results.

[`flagPoint`]=`polygon_isSelfOccluded`(`vertex,vertexPrev,vertexNext,point`)

Description: Given the corner of a polygon, checks whether a given point is self-occluded or not by that polygon (i.e., if it is "inside" the corner's cone or not). Points on boundary (i.e., on one of the sides of the corner) are not considered self-occluded.

Note that to check self-occlusion from one vertex, knowing the two lines formed with the previous and next vertices (and their ordering) is sufficient.[a]

Input arguments

- `vertex` (dim. [2 × 1]type), `point` (dim. [2 × 1]type): Coordinates of the endpoints of a line starting from a `vertex` of a polygon and an arbitrary `point`.

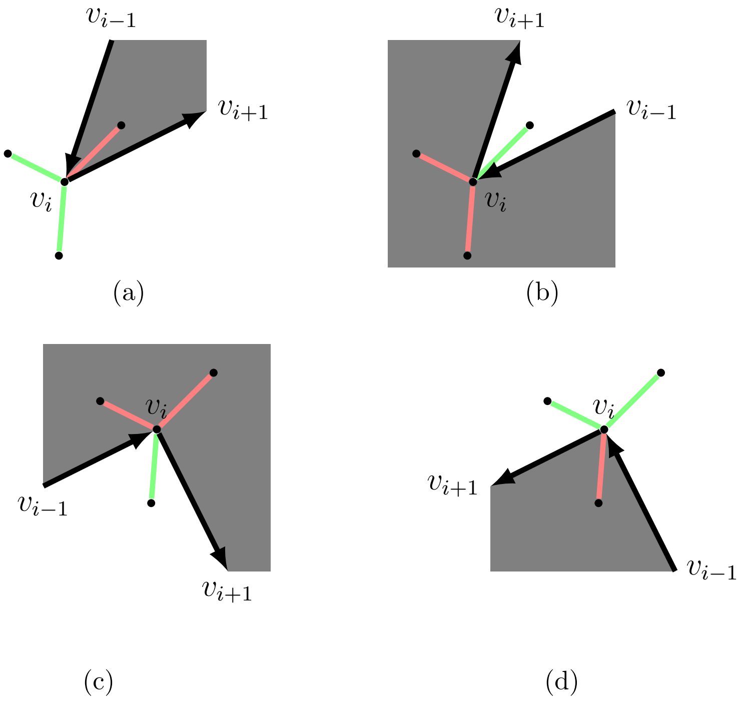


Figure 2: Examples of self-occlusions. Green and red lines mean points that, respectively, are visible and not visible from each other. Note that these figures correspond to vertices v_1 and v_2 in Figure 3.

- `vertexPrev` (dim. [2 × 1]type), `vertexNext` (dim. [2 × 1]type): Coordinates of the vertices preceding and following the `vertex` above in the original polygon.

Output arguments

- `flagPoint` (type `logical`): The output `flag` is equal to `true` if the line of sight between points with coordinates `vertex` and `point` is blocked due to self-occlusion (not edge intersection). The function returns `false` if `vertexPrev` or `vertexNext` coincides with `vertex0`.

Requirements: Use the function `edge_angle` ( ) to check the angle between the segment `vertex` – `point` and the segments `vertex` – `vertexPrev` and `vertex` – `vertexNext`.

[a] To convince yourself, try to complete the corners shown in Figure 2 with clockwise and counter-clockwise polygons, and you will see that, for each example, only one of these cases can be consistent.

Question optional 1.2. Vectorize the function `polygon_isSelfOccluded` ( ) above, so that `point` can be a [2 × `NTestPoints`] array, and `flagPoint` is a [1 × `NTestPoints`] array containing the result for each column of `point`.

Question code 1.3. Check visibility of points from polygon corners.

[`flagPoints`]= `polygon_isVisible` (`vertices,indexVertex,testPoints`)

Description: Checks whether a point p is visible from a vertex v of a polygon. In order to be visible, two conditions need to be satisfied:

1) The point p should not be self-occluded with respect to the vertex v (see `polygon_isSelfOccluded`( )).

2) The segment p–v should not collide with *any* of the edges of the polygon (see `edge_isCollision`( )).

Input arguments

- `vertices` (dim. [2 × `nbVertices`]type): array where each column represents the coordinates of a vertex in the polygon.
- `indexVertex` : a single index $1 \leq$ `indexVertex` $\leq$ `NVertices` identifying the column in `vertices` corresponding to a specific vertex v.
- `testPoints` (dim. [2 × `nbPoints`]type): array where each column represents the coordinates of a point for which visibility should be tested.

Output arguments

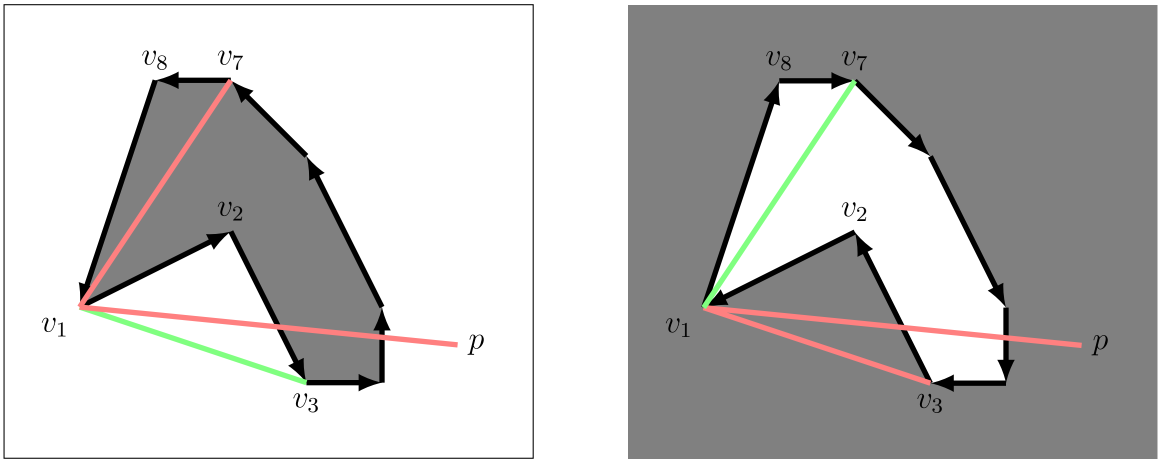
- `flagPoints` (dim. [1 × `nbPoints`], type `logical`): a vector in which each entry will be `true` if the point in the corresponding columns of `testPoints` is visible from v, and `false` otherwise.

Requirements: Note that, with the definitions of edge collision and self-occlusion given in the previous questions, a vertex should be visible from the previous and following vertices in the polygon.

Question **provided** **1.1.** Function to generate polygons that you will use for testing.

[`vertices1`,`vertices2`]=`twolink_polygons`( )

Description: Returns the vertices of two polygons that represent the links in a simple



(a) Filled polygon (counterclockwise ordering)

(b) Hollow polygon (clockwise ordering)

Figure 3: Examples of visibility. Green and red lines mean points that, respectively, are visible and not visible from each other. Line v_1–v_3 in (a) and v_1–v_7 in (b): visible. Line v_1–p in both (a) and (b): edge intersections. Line v_1–v_7 in (a) and v_1–v_3 in (b): self-occlusions.

2-D two-link manipulator.

Output arguments

- `vertices1` (dim. [2 × 4]type), `vertices2` (dim. [2 × 12]type): Coordinates of the vertices of the polygons, one for each link of the manipulator.

Question **optional** **1.3.** This function is very simple and strictly not necessary for this assignment. However, it can be reused in other assignments, and it will simplify the answer to the next question.

`plotLinesFlag` (`pointsStart,pointsEnd,flags`)

Description: Plot lines going from `pointsStart` to `pointsEnd` with a color that depends on `flags`. Each element in `flag` corresponds to a column in `pointsStart` and one in `pointsEnd` (i.e., to the endpoints of a line). If an element in `flag` is `true`, the corresponding line should be plotted in red, while if it is `false`, it should be plotted in green.

Input arguments

- `pointsStart` (dim. [2 × `nbPoints`]type), `pointsEnd` (dim. [2 × `nbPoints`]type): each column corresponds to the coordinates of, respectively, the start and end of a point.
- `flags` (dim. [1 × `nbPoints`], type `logical`): each element `flags(iPoint)` indicates the color to use to plot `points(:,iPoint)`.

Question **report** **1.3.** This function will test the previous functions for drawing polygons and checking visibility.

`polygon_isVisible_test` ( )

Description: This function should perform the following operations:

1) Create an array `testPoints` with dimensions [2 × 5] containing points generated uniformly at random using `rand` ( ) and scaled to approximately occupy the rectangle $[0, 5] \times [-2, 2]$ (i.e., the x coordinates of the points should fall between 0 and 5, while the y coordinates between -2 and 2).

2) Obtain the coordinates `vertices1` and `vertices2` of two polygons from `twolink_polygons` ( ).

3) For each polygon `vertices1`, `vertices2`, display a separate figure using the following:

(a) Create the array `testPointsWithPolygon` by concatenating `testPoints` with the coordinates of the polygon (i.e., the coordinates of the polygon become also test points).

(b) Plot the polygon (use `polygon_plot` ( )).

(c) For each vertex v in the polygon:

i. Compute the visibility of each point in `testPointsWithPolygon` with respect to that polygon (using `polygon_isVisible`( )).

   ii. Plot lines from the vertex v to each point in `testPointsWithPolygon` in green if the corresponding point is visible, and in red otherwise.

4) Reverse the order of the vertices in `vertices1`, `vertices2` using the function `fliplr`( ).

5) Repeat item *3)* above with the reversed edges.

Requirements: The function should display four separate figures in total, each one with a single polygon and lines from each vertex in the polygon, to each point.

Question code 1.4. Check if points are inside a given polygon (i.e., in collision).

[`flagPoints`]=`polygon_isCollision`(`vertices`,`testPoints`)

Description: Checks whether the a point is in collsion with a polygon (that is, inside for a filled in polygon, and outside for a hollow polygon). In the context of this homework, this function is best implemented using `polygon_isVisible`( ).

Input arguments

- `vertices` (dim. [2 × `nbVertices`]type): array where each column represents the coordinates of a vertex in the polygon.
- `testPoints` (dim. [2 × `nbPoints`]type): array where each column represents the coordinates of a point for which collision should be tested.

Output arguments

- `flagPoints` (dim. [1 × `nbPoints`], type `logical`): a vector in which each entry will be `true` if the point in the corresponding columns of `testPoints` is in collision from v, and `false` otherwise.

Question optional 1.4. This function is very simple and strictly not necessary for this assignment. However, it can be reused in other assignments.

`plotPointsFlag`(`points`,`flags`)

Description: Plot `points` with a color that depends on `flags`. Each element in `flag` corresponds to a column in `points` (i.e., to a point). If an element in `flag` is `true`, the corresponding point should be plotted in red, while if it is `false`, it should be plotted in green.

Input arguments

- `points` (dim. [2 × `nbPoints`]type): each column corresponds to the coordinates of a point.
- `flags` (dim. [1 × `nbPoints`], type `logical`): each element `flags(iPoint)` indicates the color to use to plot `points(:,iPoint)`.

Question provided 1.2. A function to visually test the correctness of `polygon_isCollision`( )

> `polygon_isCollision_test` ( )
> Description: This function is the same as `polygon_isVisible_test` ( ), but instead of step *3)*c, use the following:
>
> *1)* Compute whether each point in `testPointsWithPolygon` is in collision with the polygon or not using `polygon_isCollision` ( ).
>
> *2)* Plot each point in `testPointsWithPolygon` in green if it is not in collision, and red otherwise.
>
> Moreover, increase the number of test points from 5 to 100 (i.e., `testPoints` should have dimension [2 × 100]).

Question **report** **1.4.** Run the function `polygon_isCollision_test` ( ), and include the resulting images in your report.

Problem 2: Poor-man's priority queue

For this problem, you will write functions that implement a priority queue. For the purposes of this homework, a naïve implementation based on $O(n)$ operations is required and sufficient. For real life applications you should use an efficientimplementation based on heaps.

Data structure. We will store our queue in an array `pQueue`, where each element of the array is a struct with fields

- `pQueue(i).key` is an identifier of some type (e.g., could be a string or numeric).
- `pQueue(i).value` is a numerical value associated with the key.

provided A function to initialize the data structure to an empty queue.

> [`pQueue`]= `priority_prepare` ( )
> Description: Create an empty structure array for the queue.
> Output arguments
>
> - `pQueue` (dim. [0 × 1], type `struct`): Zero-length (i.e., empty) array of structs with fields `key` and `cost`.

Question **code** **2.1.** Inserting elements.

> [`pQueue`]= `priority_insert` (`pQueue,key,cost`)
> Description: Add an element to the queue.
> Input arguments
>
> - `pQueue` (dim. [nbElements × 1], type `struct`): the struct array containing the queue with fields `key` and `value` as described.
> - `key` : the identifier associated with the element to be inserted.
> - `cost` : the cost associated with the item to be inserted.
>
> Output arguments

- pQueue (dim. [nbElements+1 × 1], type struct): same struct as the corresponding input, but with the new element added.

Question code 2.2. Extracting the minimum-cost element.

[pQueue,key,cost]=priority_minExtract(pQueue)
Description: Extract the element with minimum cost from the queue.
Input arguments

- pQueue (dim. [nbElements × 1], type struct): the struct array containing the queue with fields key and value as described.

Output arguments

- pQueue (dim. [nbElements-1 × 1], type struct): same struct as the corresponding input, but with the element having minimum cost removed. If nbElements was 1 or 0 (i.e., after the extraction, the queue should be empty), the result should be the same as the output of priority_prepare( )
- key : the identifier associated with the element in the queue having minimum cost; return the [0 × 0] empty array if nbElements==0.
- cost : the cost associated with the item of minimum cost, return the [0 × 0] empty array if nbElements==0.

Question code 2.3. Finding out if a given key is in the queue.

[flag]=priority_isMember(pQueue,key)
Description: Check whether an element with a given key is in the queue or not.
Input arguments

- pQueue (dim. [nbElements × 1], type struct): the struct array containing the queue with fields key and value as described.
- key : the identifier associated with the element of which we need to check the presence.

Output arguments

- flag (type logical): true if any one of the elements in pQueue has the key field equal to the input key, otherwise returns false.

Requirements: Remember that the key argument could be a number, a vector of numbers, a string, or any other arbitrary type. As such, you should use the Matlab function isequal( ) to check for equality between keys (this function works for arbitrary types of variables, run doc isequal on the Matlab prompt for details).

Question report 2.1. A function to test the priority queue.

Key	Cost
'Oranges'	4.5
'Apples'	1
'Bananas'	2.7
'Cantaloupe'	3

Table 2: Sequence of inputs for the function priority_test

> Header: `priority_test()`
> Description: The function should perform the following steps:
>
> *1)* Initialize an empty queue.
>
> *2)* Add three elements (as shown in Table 2 and in that order) to that queue.
>
> *3)* Extract a minimum element.
>
> *4)* Add another element (as shown in Table 2).
>
> *5)* Check if an element is present.
>
> *6)* Remove all elements by repeated extractions.
>
> After each step, print the content of `pQueue`.

Include a copy of the outputs from the command window in your report. Try to use elements with keys of different types, and include short comments on why the outputs you get are as expected.

Question report 2.2 (3 points). Imagine that you have a grid (a simple 2-D array) of elements, where each element in the grid is identified by a pair of coordinates, and each element is associated to a cost. Explain how you could use the functions `priority_*()` to display all the elements in the grid in order of descending cost. You can either include commented code in your report, or explain the high-level idea using plain English, without writing any specific code.