Homework 1 (Python version)

ME570 - Prof. Tron

2023-08-18

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 Python knowledge with critical and creative thinking skills. The level of programming knowledge required by the assignments will be intermediate/advanced.

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 1b):

- If the vertices are counterclockwise ordered, they define a filled-in polygon;
- If the vertices are clockwise ordered, they define an hollow polygon (empty inside, filled outside).

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:

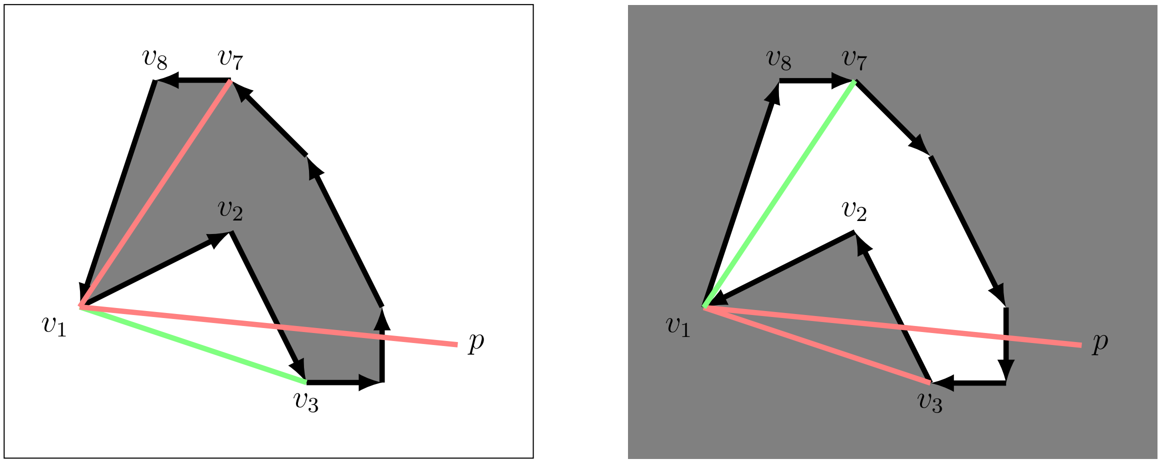
1) There is an edge blocking the line of sight (line v_1–p in both Figures 1a and 1b);

2) The line of sight falls inside the obstacle, that is, there is a *self-occlusion* (line v_1–v_7 in Figure 1a and line v_1–v_3 in Figure 1b).

Collision checking will be implemented by using the visibility functions.

Question provided 1.1. A class for polygons.

> File name: `me570_geometry.py`



(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.

> Class name: `Polygon`
> Description: Class for plotting, drawing, checking visibility and collision with polygons.
>
> > Method name: `__init__`
> > Description: Save the input coordinates to the internal attribute `vertices`.
> > Input arguments
> >
> > - `vertices` (dim. [2 × `nb_vertices`], type `nparray`): array where each column represents the coordinates of a vertex in the polygon.
>
> > Method name: `flip`
> > Description: Reverse the order of the vertices (i.e., transform the polygon from filled in to hollow and viceversa).

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

Question report 1.1. Method to plot the polygon

> File name: `me570_geometry.py`
> Class name: `Polygon`

Method name: `plot`
Description: Plot the polygon using Matplotlib.
Input arguments

- `style` (dim. [$2 \times$ `nb_vertices`], type `string`): a style specification that follows Matplotlib's standard conventions.

Requirements: Each edge in the polygon must be an arrow pointing from one vertex to the next. Use the function `matplotlib.pyplot.quiver`( ) to actually perform the drawing. The function should not create a new figure but draw on the current axes.

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.

File name: `me570_geometry.py`
Class name: `Polygon`

Method name: `is_filled`
Description: Checks the ordering of the vertices, and returns whether the polygon is filled in or not.
Output arguments

- `flag` (type `logical`): `true` if the polygon is filled in, and `false` if it is hollow.

Question provided 1.2. We will represent edges with a separate class, which stores the vertices of the edge in the same way as `Polygon` does.

File name: `me570_geometry.py`
Class name: `Edge`

Method name: `__init__`
Description: Save the input coordinates to the internal attribute `vertices`.
Input arguments

- `vertices` (dim. [2×2]): stores the coordinates of the endpoints of the edge in the internal `vertices` attribute.

Question code 1.1. A method to check if the edge is in collision with another edge.

File name: `me570_geometry.py`
Class name: `Edge`
Description: Class for storing edges and checking collisions among them.

Method name: `is_collision`

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

- `edge` : the other edge against which the collision should be checked.

Requirements: The function should be able to handle any orientation of the edges (including both vertical and horizontal). You should consider all the following cases:

- the edges do not have an intersection: `flag=False`
- the edges are parallel (whether overlapping or not): `flag=False`
- the intersection falls in the interior of both edges: `flag=True`
- the intersection falls at the endpoint of *one* edge, but no the other (i.e., they form a “T”): `flag=False`
- the intersection is an endpoint for both edges: `flag=False`

Note that the “overlap” case needs to be checked up to a tolerance due to the finite precision of floating-point representation[a].

[a] If you do not know what this means, you should find out.

Note: this is a common problem, and you can find many different solutions/tutorials online; not all of these are explained well. The most elegant solution uses parametric curves. Please try to think of the solution independently before searching for help.

Question provided 1.3. Write a function that visually tests that the edge collision function works (in most cases).

File name: `me570_hw1.py`

Method name: `edge_is_collision_test`

Description: The function creates an edge from $\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]$ to $\left[\begin{smallmatrix}1\\1\end{smallmatrix}\right]$ and a second random edge with endpoints contained in the square $[0,1] \times [0,1]$, and plots them in green if they do not overlap, and in red otherwise.

Question report 1.2. Call the provided function `edge_is_collision_test`( ) five times, check that the output is consistent with what expected from the specification, and include the five plots in your report.

Question provided 1.4. A free function to compute the counterclockwise angle between two line segments.

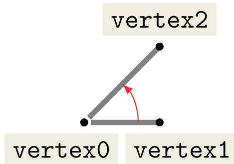
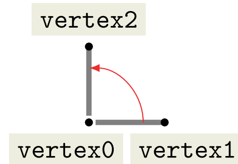
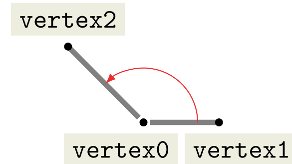
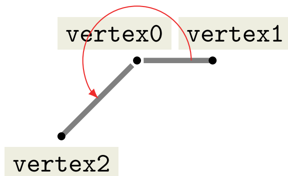
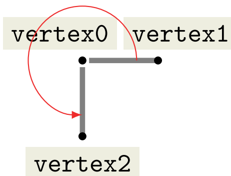
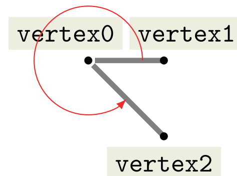
			
`edgeAngle`	`0.785`	`1.571`	`2.356`
			
`edgeAngle`	`3.927`	`4.712`	`5.498`

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

File name: `me570_geometry.py`

Function name: `angle`

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. $[2 \times 1]$, type `nparray`), `vertex1` (dim. $[2 \times 1]$, type `nparray`), `vertex2` (dim. $[2 \times 1]$, type `nparray`): coordinates of the three vertices defining the two edges.
- `angle_type` (type `string`): can be `'signed'` or `'unsigned'` to specify the range of the computed angles (defaults to `'signed'`).

Output arguments

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

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

Question **report** **1.3.** Examine the content of the function `edge_angle`( ). Explain what is the significance of the variables `s_angle` and `c_angle`, and explain how the angle `edge_angle` is computed. Include a figure illustrating your reasoning. You might have to review linear algebra to answer the question.

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.

File name: `me570_geometry.py`
Class name: `Polygon`

Method name: `is_self_occcluded`
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, we just need a vertex index `idx_vertex`. From this, one can obtain the corresponding `vertex`, and the `vertex_prev` and `vertex_next` that precede and follow that vertex in the polygon. This information is sufficient to determine self-occlusion. [a]
Input arguments

- `idx_vertex` (type `int`): Index of a vertex in the polygon with respect to which self-occlusion should be evaluated.
- `point` (dim. $[2 \times 1]$, type `nparray`): Coordinates of an arbitrary point for which visibility should be evaluated

Output arguments

- `flag_point` (type `bool`): 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 `vertex_prev` or `vertex_next` coincide with `vertex`.

Requirements: Use the function `edge_angle`( ) to check the angle between the segment `vertex` – `point` and the segments `vertex` – `vertex_prev` and `vertex` – `vertex_next`, where `vertex` corresponds to `idx_vertex`, and `vertex_prev` and `vertex_`

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

Question provided **1.5.** This function will test the previous functions for drawing polygons and checking visibility.

File name: `me570_hw1.py`
Method name: `polygon_is_self_occluded_test`
Description: Visually test the function `polygon_isSelfOccluded`( ) by picking random arrangements for `vertexPrev` and `vertexNext`, and systematically picking the position of `point`. The meaning of the green and red lines are similar to those shown in fig. 2.

Question report **1.4.** Run the provided function `polygon_is_self_occluded_test`( ) five times, and include the resulting plots in your report.
Question code **1.3.** Check visibility of points from polygon corners.

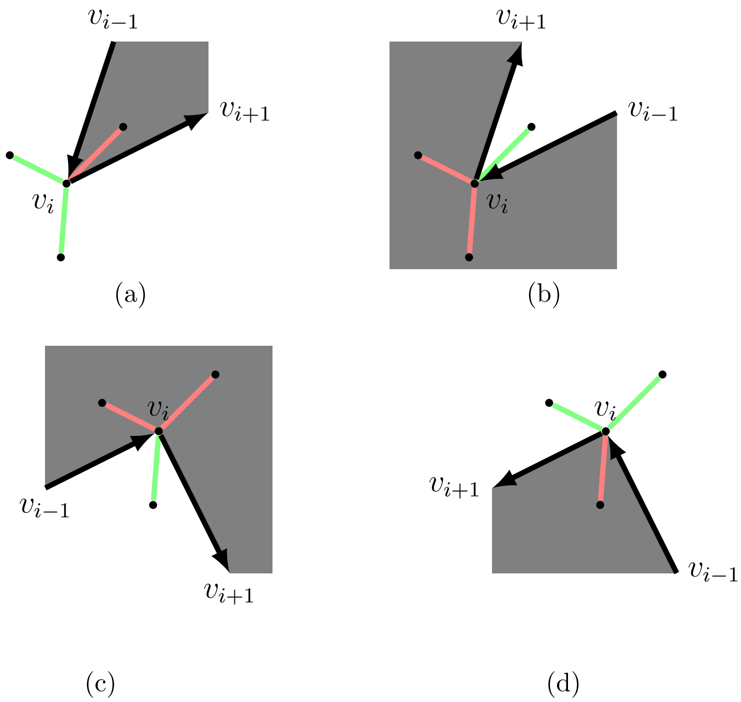


Figure 2: Examples of self-occlusions. Green and red lines mean points that, respectively, are visible and not visible from each other. Note that these figures correspond to vertices v_1 and v_2 in Figure 1.

> File name: `me570_geometry.py`
> Class name: `Polygon`

Method name: `is_visible`

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.is_self_occluded`( )).

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

Input arguments

- `idx_vertex` : a single index $1 \leq$ `indexVertex` $\leq$ `nb_vertices` identifying one of the vertices of the polygon as the specific vertex v.
- `test_points` (dim. [2 × `nb_points`], type `nparray`): array where each column represents the coordinates of a point for which visibility should be tested.

Output arguments

- `flag_points` (dim. [1 × `nbPoints`], type `bool nparray`): a vector in which each entry will be `True` if the point in the corresponding columns of `test_points` 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 1.6. The file `me570_robot.py` defines a variable `polygon=(polygon1, polygon2)`, where `polygon1` and `polygon2` are instances of the class `Polygon`, which represent the links of a 2-D manipulator. In this assignment, the polygons will be used just for testing your visibility and collision functions, while in a future assignment they will be used to "build" the two-link manipulator. The following is an example of how you can access this variable:

```
import me570_robot as robot
print(robot.polygons)
```

Question report 1.5. Write the following function, which will test the previous functions for drawing polygons and checking visibility.

File name: `me570_hw1.py`

Method name: `polygon_is_visible_test`

Description: This function should perform the following operations:

1) Create an array `test_points` with dimensions [2 × 5] containing points generated uniformly at random using `np.random.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 polygons `polygon1` and `polygon2` from `me570_robot.polygons`( ).

3) For each polygon `polygon1`, `polygon2`, display a separate figure using the following:

(a) Create the array `test_points_with_polygon` by concatenating `test_points` 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 `test_points_with_polygon` with respect to that polygon (using `Polygon.is_visible`( )).

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

4) Reverse the order of the vertices in the two polygons using `Polygon.flip`( ).

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

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.

Include the figures in your report.

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

File name: `me570_geometry.py`
Class name: `Polygon`

Method name: `is_collision`
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.is_visible`( ).

Input arguments

- `test_points` (dim. [2 × `nbPoints`], type `nparray`): array where each column represents the coordinates of a point for which collision should be tested.

Output arguments

- `flag_points` (dim. [1 × `nbPoints`], type `nparray bool`): a vector in which each entry will be `True` if the point in the corresponding columns of `test_points` is in collision from v, and `False` otherwise.

Question provided 1.7. A function to visually test the correctness of `Polygon.is_collision`( )

File name: `me570_hw1.py`
Function name: `polygon_is_collision_test`
Description: This function is the same as `polygon_is_visible_test` ( ), but instead of step *3)*c, use the following:

1) Compute whether each point in `test_points_with_polygon` is in collision with the polygon or not using `Polygon.is_collision` ( ).
2) Plot each point in `test_points_with_polygon` 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 1.6. Run the function `polygon_is_collision_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 future homework assignments, you can use the functions you will develop below, or the `queue` module in the Standard Library. For real-life applications, you should use the `queue` module.

Data structure. The queue will be stored as a simple list of `(key,value)` pairs inside the class `PriorityQueue`.

Question provided 2.1.

File name: `me570_queue.py`
Class name: `PriorityQueue`
Description: Implements a priority queue

Method name: `__init__`
Description: Initializes the internal attribute `queue_list` to be an empty list.

Question code 2.1. Inserting elements.

File name: `me570_queue.py`
Class name: `PriorityQueue`

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

Table 2: Sequence of inputs for the function `priority_test`( )

Method name: `insert`
Description: Add an element to the queue.
Input arguments

- `key` : the identifier associated with the element to be inserted.
- `cost` : the cost associated with the item to be inserted.

Question **code** **2.2.** Extracting the minimum-cost element.

File name: `me570_queue.py`
Class name: `PriorityQueue`

Method name: `min_extract`
Description: Extract the element with minimum cost from the queue.
Output arguments

- `key` : the identifier associated with the element in the queue having minimum cost; return `None`.
- `cost` : the cost associated with the item of minimum cost, return `None` if the length of the internal list `queue_list` is `0`.

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

File name: `me570_queue.py`
Class name: `PriorityQueue`

Method name: `is_member`
Description: Check whether an element with a given key is in the queue or not.
Input arguments

- `key` : The key to search for.

Output arguments

- `flag` (type `bool`): `True` if any one of the elements in the queue has the key field equal to the input key, otherwise returns `False`.

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

File name: `me570_hw1.py`
Function name: `priority_test`
Description: The function should perform the following steps. Make sure to print the contents of the queue after each step.

1) Initialize an empty queue as the object `p_queue`.
2) Add three elements (as shown in Table 2 and in that order) to that queue.
3) Extract a minimum element. Print the key and cost of such element.
4) Add another element (as shown in Table 2).
5) Check if the following keys are present: `'Apples'`, `'Bananas'`, `'(1,5)'`. Print the result after each check.
6) Remove all elements by repeated extractions. Print the extracted key and cost after each extraction.

After each step, display the content of `p_queue`.

Include a copy of the outputs from the command window into your report.

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 class `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.

Problem 3: Parametric curves

Consider the following parametric curve:

$$x(t) = \begin{bmatrix} cos(t^2) \\ sin(t^2) \end{bmatrix}. \tag{1}$$

Question report 3.1. Write the expression for $\dot{x}$, and then show that $x(t)$ and $\dot{x}$ are perpendicular for any value of t by using the inner product between the two.

> "If you want to learn something, read about it. If you want to understand something, write about it. If you want to master something, teach it."
> — *Yogi Bhajan*

Hints

Hint for question code 1.1: The most robust implementation transforms the two lines (to which the edges belong)) to parametrized curves. The intersection is then defined by finding the values of the parameters that give equal points. By comparing the values of the parameters of the intersections with those of the endpoints, it is then possible to detect if the intersection falls inside, outside, or the boundary of the edge segments.