Homework 1

ME 570 - Prof. Tron

2020-09-04

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 thinking skills, and the ability to independently find external learning resources.

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 of the function and the function header. You will have to complete these files with the body of each function requested. 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 functions from the parts marked as **provided** (see also the *Grading* paragraph below) contain already the body of the function.

Please refer to the guidelines on Blackboard under Information/Programming Tips & Tricks/Coding Practices.

Homework report Along the programming of the requested functions, prepare a small 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.
- All analytical derivations if required.

You can include comments also on the questions marked as **code**, if you feel it necessary (e.g., to explain any difficulty you might have encountered in the programming), but these will not be graded. In general, *do not* insert the listing of the functions in the report (I will have access to the source files); however, you can insert *short* snippets of code if you want to specifically discuss them.

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 done on Gradescope through two separate assignments, one for the questions marked as **code**, and one for those marked as **report**. See below for details. 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. Many of 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. Shorty after submission, you will be able to see the results for *some* of the 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. You can post questions on Blackboard at any time.

Note 1: The final grade will depend also on additional tests that will become visible only after all the grades have been reviewed as a whole. It is therefore important that you examine the results in the figures that you will generate for the report.

Note 2: 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. Questions marked as **provided** have already a solution provided in the file accompanying this assignment, which you can directly use unless you want to answer it by yourself. If you submit an answer for optional or provided questions I will correct it, but it will not count toward your grade.

Maximum possible score. The total points available for this assignment are 76.0 (75.0 from questions, plus 1.0 that you can only obtain with beauty points).

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 All the problems can be solved using Matlab's standard features. 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 2):

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

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 2a and 2b);

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

Collision checking will be implemented by using the visibility functions.

Question report 1.1 (3 points). Drawing polygons.

> `polygon_plot(vertices,style)`
> Description: Draws a closed polygon described by `vertices` using the style (e.g., color) given by `style`.
> Input arguments
> - `vertices` (dim. [2 × `nbVertices`]): 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 a figure with the plot of a 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`]): 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 (8 points).** 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]), `vertices2` (dim. [2 × 2]): 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,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]), `vertex1` (dim. [2 × 1]), `vertex2` (dim. [2 × 1]): coordinates of the three vertices defining the two edges.
- `type` (, type `string`): can be `'signed'` or `'unsigned'` to specify the range of the computed angles.

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 $[-\pi, \pi)$.

Question report 1.2 (6 points). 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 (5 points). Check if a point is self-occluded by a corner of a polygon.

`[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. $[2 \times 1]$), `point` (dim. $[2 \times 1]$): Coordinates of the endpoints of a line starting from a `vertex` of a polygon and an arbitrary `point`.
- `vertexPrev` (dim. $[2 \times 1]$), `vertexNext` (dim. $[2 \times 1]$): 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).

Requirements: Use the function `edge_angle(_)` to check the angle between the segment `vertex`–`point` and the segments `vertex`–`vertexPrev` and `vertex`–`vertexNext`. The function returns `NaN` if `vertex1` or `vertex2` coincides with `vertex0`.

[a] To convince yourself, try to complete the corners shown in Figure 1 with clockwise and counter-clockwise polygons, and you will see that, for each example, only one of these cases can be consistent.

See Figure 1 for examples of the expected results of `polygon_isSelfOccluded(_)`.

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 (5 points). 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(_)`).

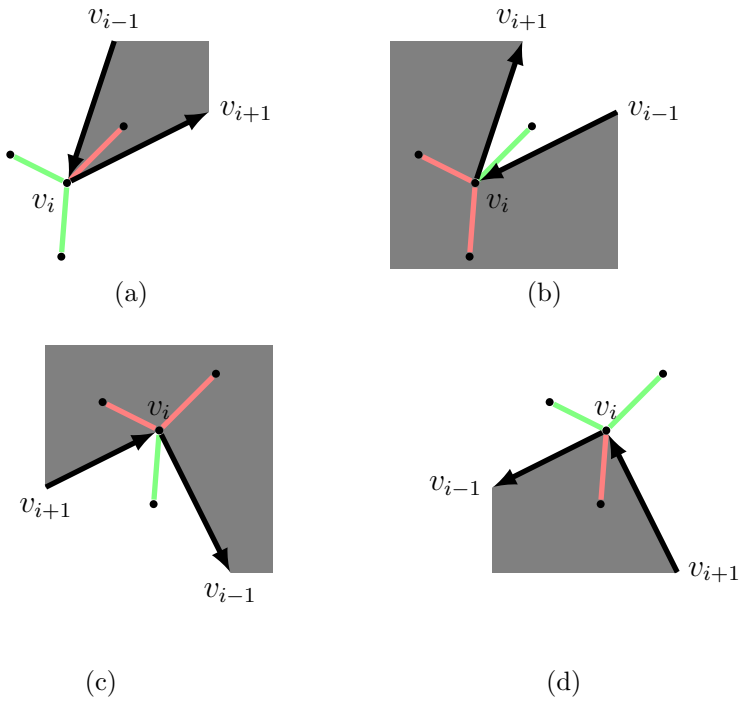


Figure 1: Examples of self-occlusions. Green and red lines mean points that, respectively, are visible and not visible from each other. Note that these figures correspond to vertices v_1 and v_2 in Figure 2.

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

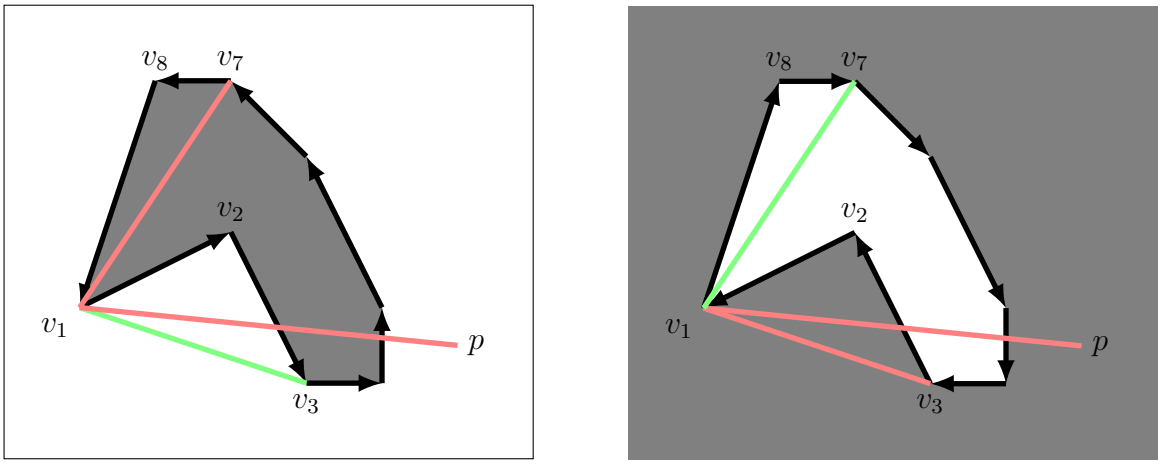
[`vertices1,vertices2`]= `twolink_polygons` ( )
Description: Returns the vertices of two polygons that represent the links in a simple 2-D two-link manipulator.
Output arguments

- `vertices1` (dim. [2 × 4]), `vertices2` (dim. [2 × 12]): 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`]), `pointsEnd` (dim. [2 × `nbPoints`]): 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)`.



(a) Filled polygon (counterclockwise ordering)

(b) Hollow polygon (clockwise ordering)

Figure 2: 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.

Question report **1.3 (5 points).** 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 $[2 \times 5]$ 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 `testPointsPolygon` 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 (5 points).** 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. $[2 \times$ `nbVertices`$]$): array where each column represents the coordinates of a vertex in the polygon.
- `testPoints` (dim. $[2 \times$ `nbPoints`$]$): 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`]): 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_isVisible(_)`

`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 (2 points). 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. However, for extra credits, you are welcome to work on efficient *real* implementations based on heaps (your own implementation or using existing implementations).

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 (3.5 points). 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 (6 points). 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==1`.
> - `cost` : the cost associated with the item of minimum cost, return the [0 × 0] empty array if `nbElements==1`.

Question code 2.3 (3.5 points). 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 (7 points). From the command line, execute the following operations:

1) Initialize an empty queue.

2) Add three elements (not in order of cost) to that queue.

3) Extract a minimum element.

4) Add another element.

5) Check if an element is present.

6) Remove all elements.

Try to use elements with keys of different types, and include 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 make a new 1-D array with all the elements in the grid in order of ascending 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: Using CVX to solve convex optimization problems

This question assumes that you have some working familiarity with linear algebra and convex optimization, and the goal is to gain familiarity with the CVX toolbox for convex optimization in Matlab.

Preparation. Please visit the following links.

- Overview of CVX: http://web.cvxr.com/cvx/doc/intro.html.
- Installation instructions: http://web.cvxr.com/cvx/doc/install.html.

- Quick start: `http://web.cvxr.com/cvx/doc/quickstart.html`
- The `quiet` option for `cvx_begin`: `http://web.cvxr.com/cvx/doc/basics.html#cvx-begin-and-cvx-end`

Question code 3.1. The function `qp_minEffort(_)` will solve the following optimization problem (a Quadratic Program, or QP for short):

$$\begin{aligned} u_{\text{opt}}, \delta_{\text{opt}} = & \underset{u\in\mathbb{R}^2, \delta\in\mathbb{R}}{\operatorname{argmin}} \|u\|^2 + m\delta^2 \\ \text{subject to } & A_{\text{attr}}u + b_{\text{attr}} + \delta \leq 0 \\ & A_{\text{barrier}}u + b_{\text{barrier}} \leq 0 \end{aligned} \tag{1}$$

> `[uOpt,DeltaOpt]=qp_minEffort(AAttr,bAttr,ABarrier,bBarrier,mPenalty)`
> Description: Solves the optimization problem in (1) for the given parameters.
> Input arguments
> - `AAttr` (dim. `[1 × 2]`), `bAttr` (dim. `[1 × 2]`): parameters of a linear constraint that involves both u and δ.
> - `ABarrier` (dim. `[nbObstacles × 2]`), `bBarrier` (dim. `[nbObstacles × 1]`): parameters of `nbObstacles` linear constraints on u; `nbObstacles` can be an arbitrary non-negative integer (even zero).
> - `mPenalty` : a weight for the cost function.
>
> Output arguments
> - `uOpt` (dim. `[2 × 1]`), `DeltaOpt` : vector and scalar with the optimal solution.

Question report 3.1 (4 points). Let $\mathbf{1} = \begin{bmatrix}1\\1\end{bmatrix}$, and consider the case $A_{attr} = \mathbf{1}^{\mathrm{T}}$, $A_{\text{barrier}} = I$ (the 2×2 identity matrix). For each one of the combinations:

1) $b_{\text{attr}} = 1$, $b_{\text{barrier}} = \mathbf{1}$, $m = 1$;

2) $b_{\text{attr}} = 1$, $b_{\text{barrier}} = -\mathbf{1}$, $m = 1$;

3) $b_{\text{attr}} = -1$, $b_{\text{barrier}} = -\mathbf{1}$, $m = 1$;

4) $b_{\text{attr}} = -1$, $b_{\text{barrier}} = -\mathbf{1}$, $m = 100$;

include in your report the following:

- A drawing of the feasible set of (1), including the vector u_{opt} obtained with the function `qp_minEffort(_)`. You can sketch the drawing by hand or a drawing program; it does not need to be in Matlab.
- A written explanation of why the solution makes sense; consider the fact that the objective function in (1) aims to find u_{opt} and δ that are as close to zero as possible (in fact, the unconstrained minimum is given by $u = 0$, $\delta = 0$).

The idea is to show an understanding of what is being optimized, and why the optimizer gives you that answer.

Problem 4: Homework feedback

Question report **4.1 (2 points).** Indicate an estimate of the number of hours you spent on this homework (possibly broken down by problem). Explain what you found hard/easy about this homework, and include any suggestion you might have for improving it.

Hint for question report 1.1: Consider the use of the function `mod(_)` to elegantly handle the last side of the polygon (going from the last vertex to the first).

Hint for question optional 1.1: To check the ordering, you can start from a vertex, and then sum (with sign) the angles formed while following the other vertices. If the sum is 2π, the vertices are ordered in a counterclockwise manner; if the sum is -2π, then the vertices are ordered in a clockwise manner.

Hint for question code 1.1: There exist different ways to do collision checking on edges (i.e., on line segments). The most robust way to solve this problem is by thinking of the two edges as parametrized lines. For instance, if the two endpoints of edge 1 are v_{1a} and v_{1b}, then edge 1 can be represented as the image of the function $x_1(t_1) = t_1 v_{1a} + (1 - t_1)v_{1b}$ for t_1 in the interval $[0, 1]$ (note how $x_1(0)$ and $x_1(1)$ give the two endpoints, and $t \in [0, 1]$ gives the points in the middle). Similarly, edge 2 can be represented as the image of a function $x_2(t_2)$. By solving the linear system given by $x_1(t_1) = x_2(t_2)$, one can recover the t_1, t_2 of the intesection point. If both t_1 and t_2 at the intersection are between 0 and 1, then there is a collision. Note that additional checks are needed to handle the case of parallel edges or edges with zero length (in these cases the linear system is not well defined).

Hint for question code 1.2: A way to solve this question is to verify whether the line `vertex` – `point` is between the lines `vertex` – `vertexNext` and `vertex` – `vertexPrev`. Note that, as a consequence, you do not need the global ordering of the vertices in the polygon, and the distinction between filled-in and hollow polygons is automatically handled.

Hint for question code 1.3: The answer for this question needs to call the function `edge_isCollision(_)` inside a `for` loop that iterates over all the edges in the polygon.

Hint for question code 1.4: A point is not in collision with a polygon if it is visible from at least one vertex.