# Homework 1

ME 570 - Prof. Tron

Wednesday 6th September, 2017

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, you will have to combine your Matlab knowledge with critical thinking skills, and the ability to independently find external learning resources. Update: fixed incomplete sentences, reorganized question numbers, added restrictions on the use of external libraries, fixed inconsistency between description of Problem 1 and Figure 1.

## General instructions

**Homework report (required).** Solve each problem, and then prepare a small PDF report containing:

- Instructions on how to run your code.
- One or two sentences of comments for each question.
- All analytical derivations if required.
- Embedded figures and outputs that are representative of those generated by your code.

**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).

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

**Submission.** Keep the report and the code all in one directory (do not use subdirectories). Compress all the files into a single ZIP archive with name `<Last><First>-hw<Number>.zip`, where `<First>` and `<Last>` are your first and last name, and `<Number>` is the homework number (e.g., `TronRoberto-hw1.zip`). Submit the ZIP archive through Blackboard. Please refer to the Syllabus on Blackboard for late homework policies.

**Grading.** Each question is worth 1 point unless otherwise noted. Questions marked as **optional** are provided just to further your understanding of the subject, and not for credit. If you submit an answer I will correct it, but it will not count toward your grade. See the Syllabus on Blackboard for more detailed grading criteria.

**Maximum possible score.** The total points available for this assignment are 17.0 (16.0 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.

## 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 1):

- 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 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 1.1.** Make a function `polygon_draw(vertices,color)` that draws the polygon described by `vertices` using the color specified by the input string `color` (which follows Matlab's standard style conventions, e.g. `'r'` for red). in the current figure. 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.

**Question 1.2 (optional).** Implement a function `[flag]=polygon_isFilled(vertices)` that checks the ordering of the vertices, and returns `true` if the polygon is filled in, and `false` if it is hollow. Modify the previous function `polygon_draw()` to use the `patch()` (which draws solid polygons) if the polygon is filled in.

**Question 1.3 (2 points).** Make a function

```
[flag]=edge_isCollision(vertices1,vertices2)
```

where `vertices1` and `vertices2` are [2 × 2] variables containing the coordinates of the endpoints of two edges, and returns `flag=true` if the two edges intersect, and `false` otherwise. *Note:* if the two edges overlap but are colinear, they are not considered as intersecting (`flag=false`).

**Question 1.4 (3 points).** Implement a function

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

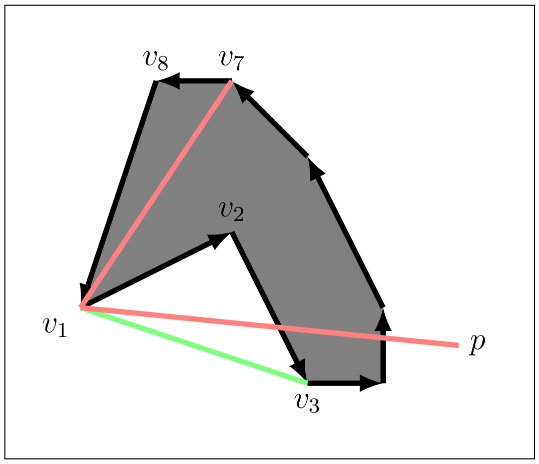
that returns `flag=true` if the line of sight between points with coordinates `vertex` and `point` is blocked due to self-occlusion (not edge intersection). The variables `vertexPrev` and `vertexNext` contain the coordinates of the vertices that, respectively, precede and follow, in the original polygon ordering, the one stored in `vertex`. Each one of the variables `vertex`, `vertexPrev`, `vertexNext`, and `point` has dimension [2 × 1]. 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 (for instance, see the points $v_1$,$v_3$,$v_8$ and $v_7$ in Figure 1); this is the reason why we pass only three vertices to the function, and not all the vertices at once.

**Question 1.5 (optional).** 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`.

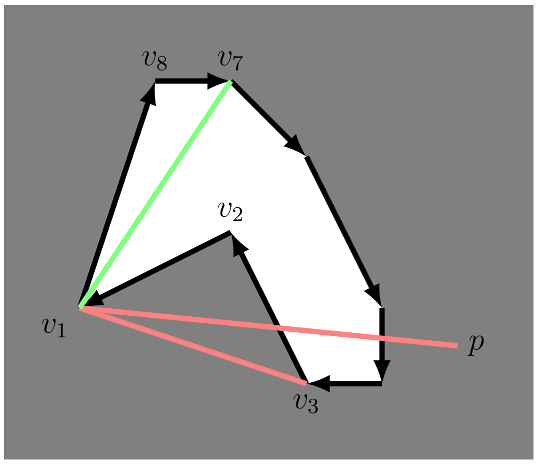
**Question 1.6 (2 points).** Make a function

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

that takes as input the [2 × `NVertices`] array `vertices` describing a polygon, a single index 1 ≤ `indexVertex` ≤ `NVertices` identifying a the column in `vertices` corresponding to a specific vertex $v$, and a [2 × `NPoints`] variable `points` containing the coordinates of `NPoints` points to test. The entries in the [1 × `NPoints`] output vector `flagPoints` will be



(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.

`true` if the points in the corresponding columns of `points` are visible from $v$, and `false` otherwise. In practice, for each point to test, you will need to use the previous function `polygon_isSelfOccluded`(), and combine its result with the result of `edge_isCollision`() called over each edge of the polygon.

**Question 1.7.** Implement a function `polygon_isVisible_testRandTriangles`() that displays a random triangle (polygon with three vertices), samples 5 random points, and plots all the lines from the vertices to the points, each line with a color that depends on the visibility (green if the point is visible from the vertex, red otherwise, similarly to Figure 1). Implement a function `polygon_isVisible_testPolygons`() that does the same for *1)* a (non-random) polygon of your choice, and *2)* the same polygon with the vertices in reverse order. The polygon you choose must have at least six vertices and be non-selfintersecting.

**Question 1.8.** Make a function

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

that uses `polygon_isVisible`() to return `true` if a point is in collsion with a polygon (that is, inside for a filled in polygon, and outside for a hollow polygon), and `false` otherwise. The semantics of the input and output arguments are the same as those having the same name in `polygon_isVisible`().

**Question 1.9.** Make a function `polygon_isCollision_testRandTriangles`() that displays a random triangle, samples 100 random points, and plots each point with a color that depends on the collision check (green if it is outside the triangle, red if it is outside).

## 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 `pq`, where each element of the array is a struct with fields

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

**Question 2.1.** Make a function [`pq`]=`priorityPrepare`() that returns a vector `pq` of structs with fields `key` and `cost`. The vector `pq` should have dimension [`0` × `1`] (i.e., it should have length zero).

**Question 2.2.** Make a function [`pq`]=`priorityPush`(`pq`,`key`,`cost`) that adds a structure with the arguments `key` and `cost` in their corresponding fields to the vector `pq`.

**Question 2.3.** Prepare a function [`pq`,`key`,`cost`]=`priorityMinPop`(`pq`) that returns the `key` and `cost` of the element in `pq` with minimum value in `cost`, and then remove that element from `pq`. The function should return an empty `key` if the priority queue is empty.

**Question 2.4.** Make a function `priority_test`() that generates a vector `v` of 10 keys associated to 10 random integers, pushes them one by one into a priority queue, extracts them again one by one, and then checks that *1)* the extracted key/value pairs all belong to the original vector; *2)* the values extracted are non-decreasing.

## Problem 3: Homework feedback

**Question 3.1 (1 point).** Indicate an estimate of the number of hours you spent on this homework (possibly broken down by problem).

**Question 3.2 (optional).** Explain what you found hard/easy about this homework, and include any suggestion you might have for improving it.

**Hint for question 1.2:** 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 $2\pi$, the vertices are ordered in a counterclockwise manner; if the sum is $-2\pi$, then the vertices are ordered in a clockwise manner.

**Hint for question 1.4:** 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 1.4:** Transform each line `vertex`–`point`, `vertex`–`vertexNext`, and `vertex`–`vertexPrev` into an angle with respect to the horizontal axis. Then, compute the differences (modulo $2\pi$) between the angle for `vertex`–`point` and the other two. Finally, determine the relative ordering of the lines from difference of the differences (again, modulo $2\pi$).

**Hint for question 1.6:** The answer for this question will end up calling the function `edge_isCollision`() a total of `NVertices` $\times$ `NPoints` times.

**Hint for question 1.8:** A point is outside a polygon if it is visible from at least one vertex.