Robot Motion Planning - Homework I

David Akhihiero

September 4, 2024

1 Introduction

In this assignment, I created MATLAB functions to allow a user to generate polygonal obstacles in a 2D configuration space, save the configuration space obstacles to a file for retrieval, and check whether a clicked point in the space is inside an obstacle.

2 Obstacle Creation and Saving

I created a function, *createObstacles*(), that uses the *drawpolygon*() MATLAB function to create obstacles in a 2D configuration space. The function interfaces with a user using the command window. To create an obstacle, a user needs to type in 'Y' to the query: "Type 'Y' to create an obstacle or 'N' to stop". To create an obstacle with m vertices, the user clicks the location of the vertices in order. After creating n obstacles, the user can enter 'N' to stop and the function saves the set of created obstacles to an 'obstacles.mat' file.

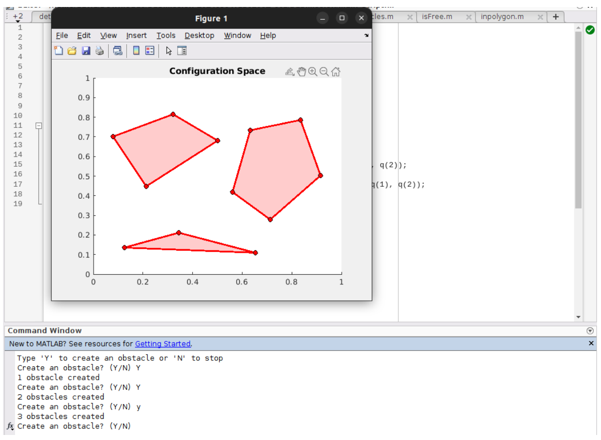


Figure 1
File Edit View Insert Tools Desktop Window Help
Configuration Space
isFree.m
inpolygon.m
Command Window
New to MATLAB? See resources for Getting Started.

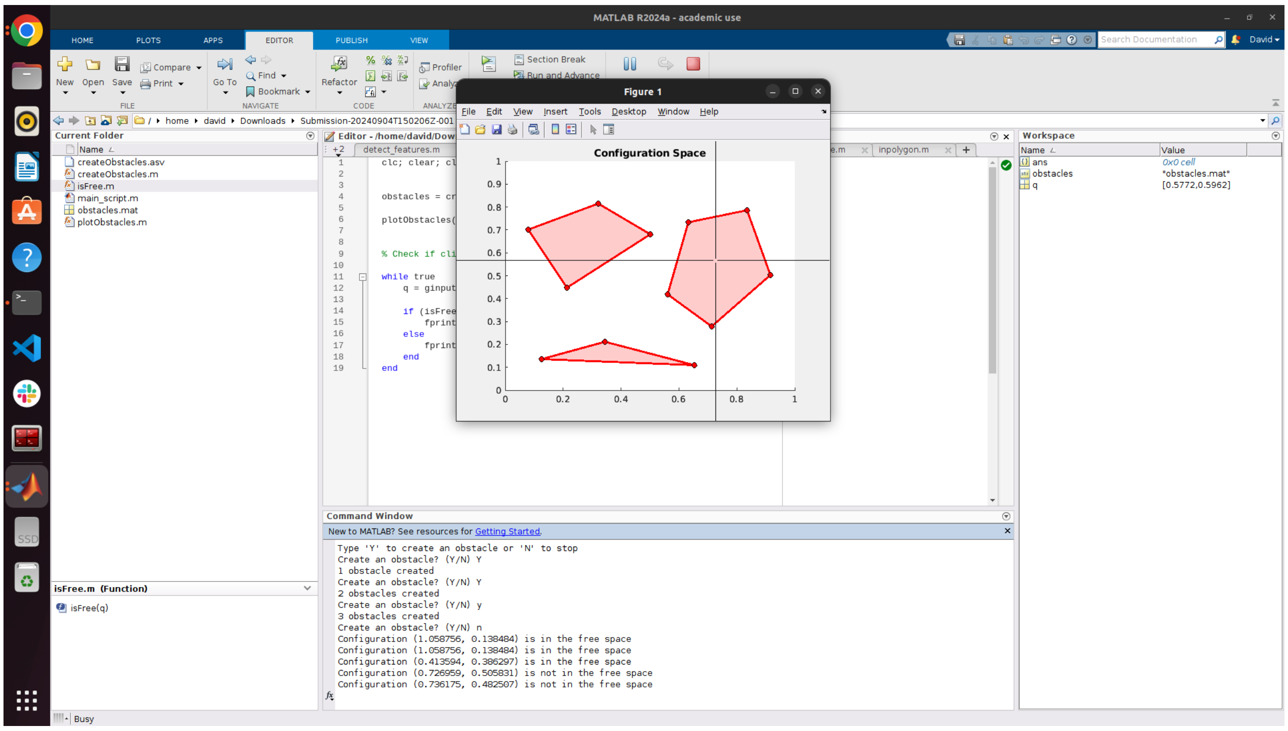
```
Type 'Y' to create an obstacle or 'N' to stop
Create an obstacle? (Y/N) Y
1 obstacle created
Create an obstacle? (Y/N) Y
2 obstacles created
Create an obstacle? (Y/N) y
3 obstacles created
Create an obstacle? (Y/N)
```

Figure 1: Obstacle Creation

I also created a function, *plotObstacles*(), that reads the 'obstacles.mat' file and plots the saved obstacles in the 2D configuration space.

3 Collision Detection

I created a function called $isFree(q)$ that takes a configuration q as input and returns true if that configuration is in the free configuration space or false if it is inside an obstacle. Part of my main script has a loop that continuously reads an input configuration from a user using the $ginput()$ function and passes that configuration to the $isFree(q)$ function. The function uses the $inpolygon()$ MATLAB function to check if a point is inside a polygonal obstacle. It does this check for every obstacle in the obstacle set. The script displays "Configuration q is in the free space" if q is not inside an obstacle otherwise it displays "Configuration q is not in the free space".



```
clc; clear; cl


obstacles = cr

plotObstacles(


% Check if cli

while true
    q = ginput

    if (isFree
        fprint
    else
        fprint
    end
end
```

```
Type 'Y' to create an obstacle or 'N' to stop
Create an obstacle? (Y/N) Y
1 obstacle created
Create an obstacle? (Y/N) Y
2 obstacles created
Create an obstacle? (Y/N) y
3 obstacles created
Create an obstacle? (Y/N) n
Configuration (1.058756, 0.138484) is in the free space
Configuration (1.058756, 0.138484) is in the free space
Configuration (0.413594, 0.386297) is in the free space
Configuration (0.726959, 0.505831) is not in the free space
Configuration (0.736175, 0.482507) is not in the free space
```

Figure 2: Collision Detection

4 Computational Complexity

4.1 Computational Complexity for Storage Space

- n = number of obstacles
- m = maximum number of vertices for an obstacle

$\implies$ Computational complexity for storage space = $\mathcal{O}(n \times m)$

4.2 Computational Complexity for Time for Creation

The computational complexity for the time to create one vertex is $\mathcal{O}(1)$, for m vertices for one obstacle is $\mathcal{O}(m)$, and for n obstacles with up to m vertices is $\mathcal{O}(n \times m)$.

4.3 Computational Complexity for Time for Collision Detection

According to Hormann and Agathos (2001), the computational complexity for the *inpolygon*() function is $\mathcal{O}(m)$ where m is the number of vertices of the polygon. For n obstacles, the computational complexity is $\mathcal{O}(n \times m)$.

4.4 Computational Complexity Comparison with a Grid-Based Representation

When using a grid-based representation with free space represented by cells with 0 and obstacle spaces represented by cells with 1, the computational complexity for storage space will be $\mathcal{O}(m^2)$ where m is the maximum number of grid cells for each dimension of the 2D space. This representation is more costly than the continuous representation because m in this representation can be much larger than n and m in the continuous representation.

The computational complexity for the time of creation with a grid-based representation is $\mathcal{O}(n \times m^2)$ where m is the maximum number of grid cells in one dimension for any arbitrary obstacle and n is the number of obstacles. This is also more expensive than the continuous representation.

For collision detection, the computational complexity for time for the grid-based representation is $\mathcal{O}(1)$. This is much faster than in the continuous representation. If the number of obstacles is few, a grid-based representation may not be best especially if the resolution is low but for a space with many obstacles and where collision checking is done often, a grid-based representation may be better since the environment is created once (the expensive part of this representation) and afterward, collision detection can be done much more cheaply compared with the continuous representation.

References

Hormann, K. and Agathos, A. (2001). The point in polygon problem for arbitrary polygons. *Computational geometry*, 20(3):131–144.