ME 570 Robot Motion Planning

Homework Assignment #1

Jay Lanzafane

Problem 1:

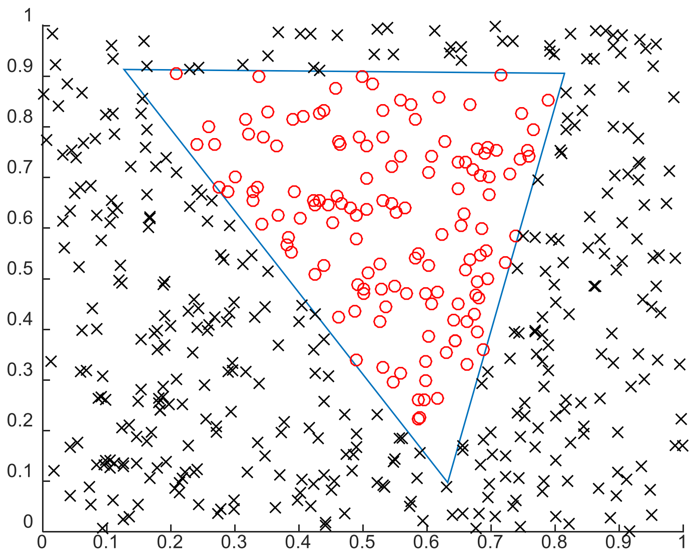
1.1) This is a pretty straightforward plot command. Since we did not use patch, we need to close the triangle and thus the plot command actually takes 4 vertices (the 1st is repeated at the end in order to close the triangle).

1.2) Here we start by writing the simplest function, points2line. This is a quick calculation to find the parameters m and b for the equation of the line y=mx+b that passes through the two provided vertices (provided as arguments to the function). Note that if the points form a vertical line, we redefine the outputs. In this case, m = inf and b = x intercept. This lets us handle vertical lines properly in the next function.

The next step is writing the function lineCheckSide. This function takes two vertices (which define the line) and a point coordinate which we will check. If the point is above or, in the case of a vertical line, to the right, the function returns a 1. Otherwise a zero is returned.

Finally, for triangleCheckCollision we harness the 2 previous functions and a bit of logic. We know that a point is inside of the triangle if the given point is on the same side as each vertex with regard to the line formed by the remaining 2 vertices.

1.3) This is a pretty straightforward script but we can see our functions are operating as expected. I bumped the number up to 500 sample points just increase the density, but obviously this is an easy change. To run, simply execute the script and a figure will be generated.



Problem 2:

2.1) To initialize the priority queue we just initiate an array (of length zero) of structures with fields 'key' and 'cost'.

2.2) I handle the priority portion of the priority queue in this function. This is NOT an optimized function, and we use a naïve approach. The priority of the queue is achieved by always arranging entries in increasing cost value, so they are inserted into the queue at the appropriate place. Note that we can change this simply by switching the < to a >. It wasn't clear in the assignment whether a low cost was higher priority or not, but because you asked to compare to a sort function I assumed that was the case. It's a pretty simple matter either way, but note that changing this WILL affect the RemoveAbove function in 2.4. If a newly inserted entry has the same cost as an existing entry, the older one will hold a higher place (first in, first out).

2.3) Here I simply return the first element in pq (with some quick error-checking), since we know that pq is properly sorted already and the lowest cost will always be the first element.

2.4) The last function is also quite simple because, again, we have pre-sorted the queue by smartly inserting new entries. We simply look at the end of the list and work our way towards the front, finding the threshold, and returning everything before that point. There are fringe cases where it removes nothing or everything, which we have to be a little careful of.

2.5) This is a pretty quick function that generates a vector v of 20 random numbers, used as the cost, and inserts them into the priority queue. We then MinPop all elements and compare the order they were removed with the sorted values of v. They should be the same, and they are.

```
1    1
1    1
1    1
2    2
2    2
2    2
2    2
3    3
3    3
4    4
4    4
4    4
5    5
6    6
```

```
  7    7

  8    8

  8    8

  9    9

 10   10

 10   10
```

2.6) This last test is slightly different. Again we make a vector v of 20 random integers and insert them into the queue. We then MinPop 5 times, which we know will remove the 5 lowers costs. From here we RemoveAbove based on a threshold, which I have set to 5. This should remove 5 from the beginning, and then ~1/2 from the end of the queue. Let's take a look at an example

```
Sort(v) = 1    3    3    3    4    4    4    5    6    6    7    7    7    8    8    9    9    9   10   10
```

We expect the first 5 MinPop to remove (1, 3, 3, 3, 4), which we will see as v2:

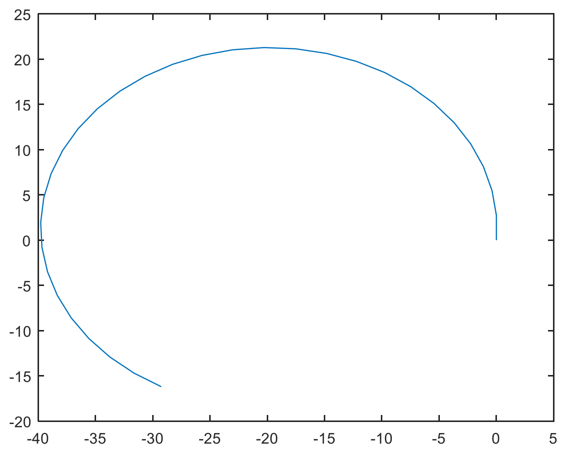
```
v2 =    1    3    3    3    4
```
(FROM MATLAB)

We then know that we are thresholding at 5, so everything greater than 5 will be removed. This should leave us with (4, 4, 5):
As we expect, pq is now length = 3 with the costs 4, 4, and 5. This is functioning as expected and as defined.

Problem 3:

3.1) The unicycle model is just a quick application of matrix math, which Matlab excels at (it's how it got its name!).

3.2) Here we test the unicycle model by stepping through the operation 5 times. Each step is of a random duration between 1 and 10 (I think this is what you meant in the instructions?). We can plot the resulting trajectories – note here we plot them in the x,y plane. This was, "for all simulated time in a Cartesian plot".


25
20
15
10
5
0
-5
-10
-15
-20
-40
-35
-30
-25
-20
-15
-10
-5
0
5