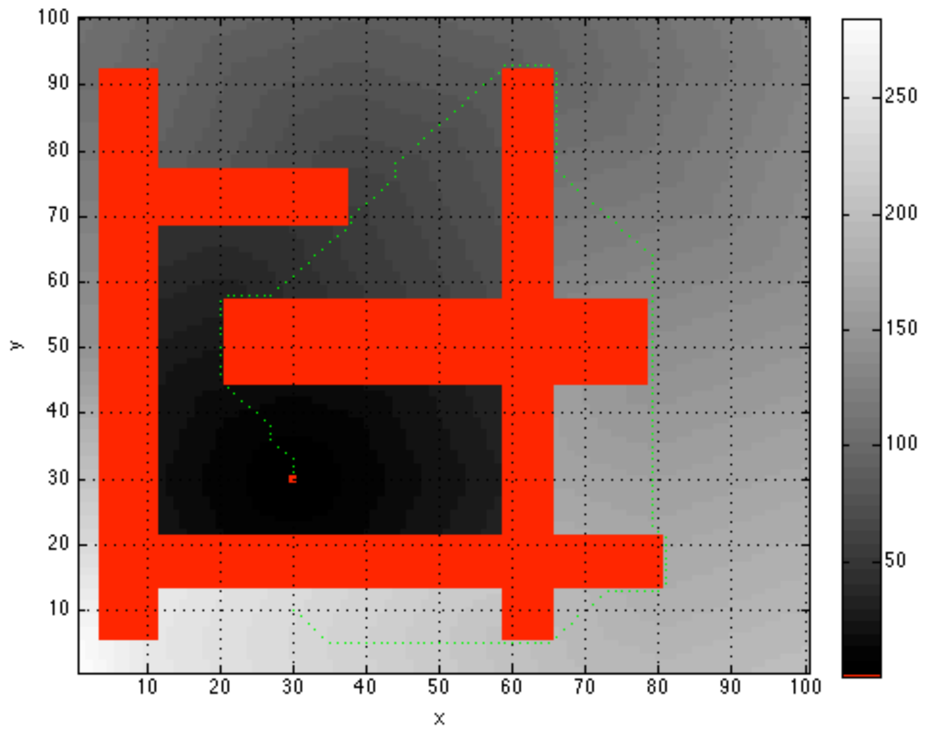
Jon Morton

Problem 1: D* Planner

Part 1. Create rough terrain and observe what happens.

Approach: I created rough terrain along the left narrow passage that was originally followed by running the following code.

```
for y=1:20
  for x=1:10
    ds.modify_cost([x,y], 2);
  end
end
```

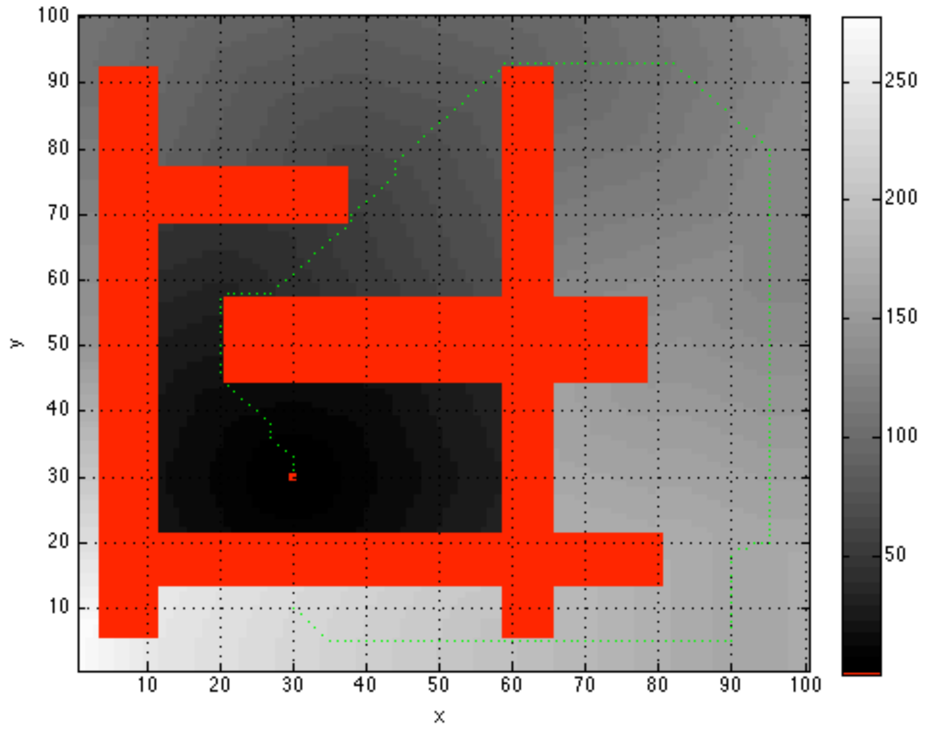


Part 2. Add a region of very low-cost terrain (less then one) and observe what happens.

Approach: In order to see how the planner would work, I added two areas with a zero cost that were slightly offset horizontally.

```
for y=1:20
  for x=90:95
    ds.modify_cost([x,y], 0);
  end
end

for y=60:80
  for x=95:99
    ds.modify_cost([x,y], 0);
  end
end
```



Observation: It is somewhat difficult to see, but the planner does plot a path through both narrow columns. Not surprisingly, the path veers back towards the goal after traversing the top most path as soon as it leaves the low cost area. I did think it was interesting (but not surprising) that the algorithm veers towards the top low-cost area *as soon as it exits* the bottom low cost area.

Jon Morton

Problem 2

1. Create a new RRT by making the integration limit and steering angle limits.

line 312…313 of robot/RRT.m defines:

```
      steermax = 1.2; % max steer angle in rads
      speed = 1;
```

…change to…

```
      steermax = 0.6; % max steer angle in rads
      speed = 1; % I want to keep this the same
```

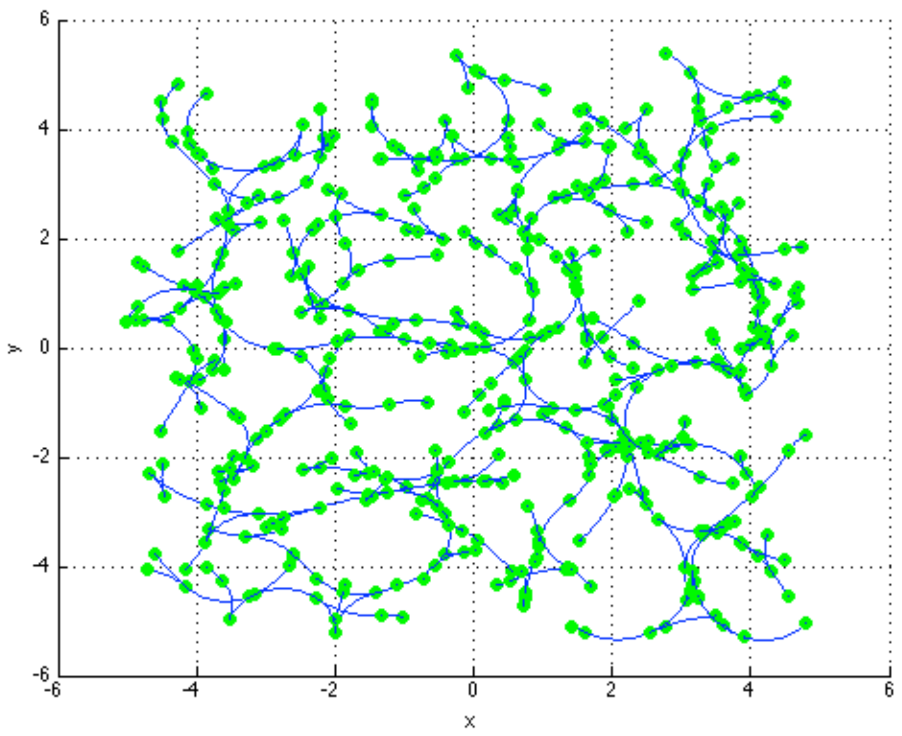
2. Current RRT chooses a streering angle from uniform distribution between streering angle limits. As people tend to drive more smoothly observe what happens of the streering angle is chosen from normal Gaussian ditribution.

line 178 of robot/RRT.m uses:

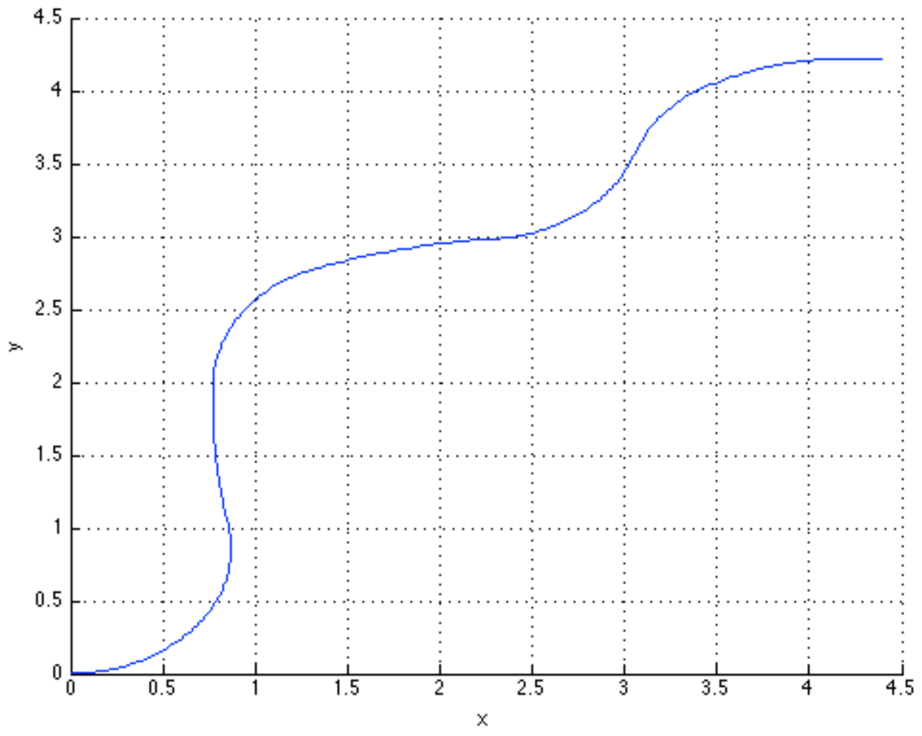
```
    theta = rand*2*pi;
```

...change to...

```
    theta = randn*2*pi; % from a normal random number generator
```



Observation: I generated an RRT using rand, and the paths were *much* more erratic. By reducing the steering range and using a normal distribution to generate theta, the steering seems much more like what I would expect to see if a person were driving the bicycle.

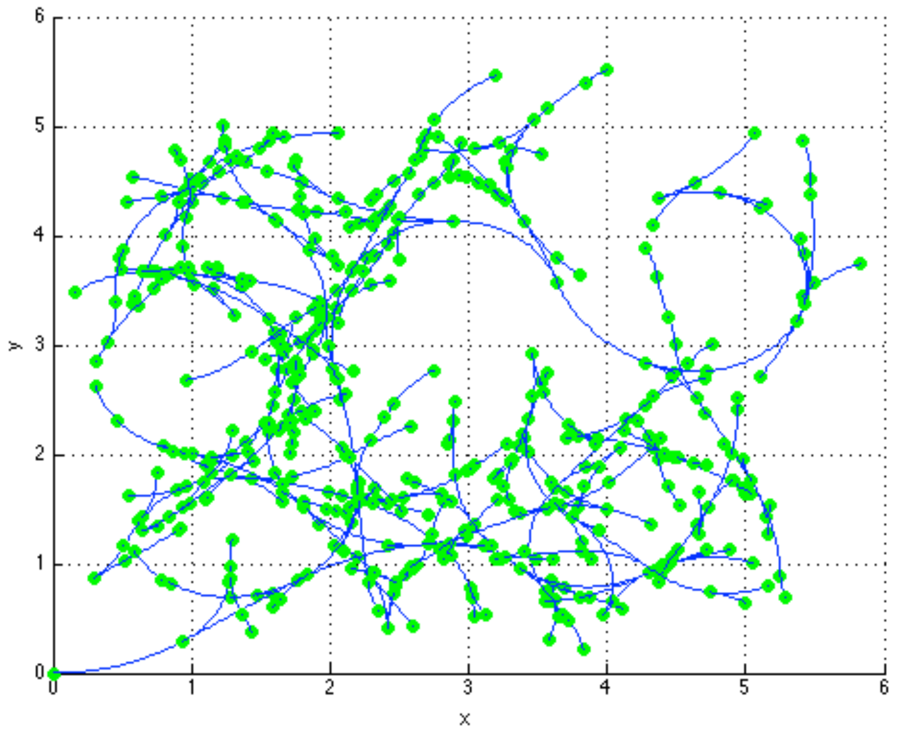


Observation: This path travels through x = 3…4, y = 3…4. Hopefully I can regenerate an RRT that has a path through this same region to see what would happen with obstacles to see how collision detection is handled between points and obstactle.

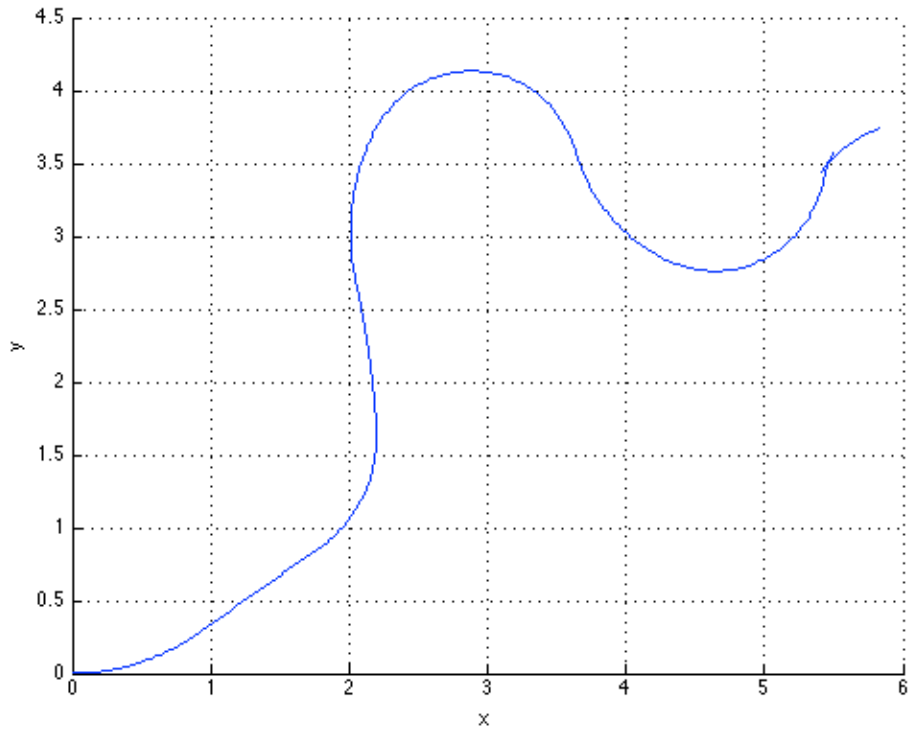
3. Add some obstacles and plot the path before and after obstacles were added.

```
        map = zeros(5,5);
        map(3:4, 3:4);
        rrt.occgrid = map;
        rrt.plan();
        rrt.path( [ 0 0 0 ], [ 5 5 0 ] );
```

In order to accomplish this task, I only had to uncomment a region detecting collisions between a plotted random point and the occupancy grid. Unfortunately, I could not come up with a way to reuse the map and omit things that were in the collision, so I did have to re-plan!



Observation: **Certain paths still cut through the obstacle despite the collision detection.** This indicates that the code I uncommented was not complete for the task at hand. It appears as though the region under certain cases ignores obstacle detection. I need to see if the lines between two random points crossing through the region containing an obstacle are selected. Unfortunately, the path to reach (4, 4) does cross through this region!



Conclusion: RVC has incomplete support for obstacle detection. Perhaps improving this would be a good project to work on for this class?

Final observation:

RRT have a steep planning cost, but are very efficient for planning a path when the collision detection algorithm is implemented correctly. I suspect that adding to an RRT can be accomplished very easily. It is important to remember that an RRT does not necessarily generate a path to an exact point. So while it might get a robot close, some final path planning might be needed to steer to a final pose.