Holonomic Motion planning using RRT*

Robot Motion planning ITCS 6152 – Poripsa Chakrabarty

Summary: In this project a robot motion planning algorithm was implemented using python programming language and its libraries. The robot motion planning algorithm used was - rapidly exploring random tree star (RRT*). Given a starting point (node) and a goal point the program finds the most optimum path to the goal from start. The point robot traces the optimum path which is shown in pink in the following images. Properties of the project and environment are as follows:

1) Motion Planning Algorithm: RRT*
2) Data Structure used: List for storing nodes, python dictionary for storing parent of each node
3) Obstacle: 2D rectangular object
4) Dimension of Configuration space: 2 (2 Dimensional)
5) Type of robot: Point robot
6) Constraints (Holonomic/Non-Holonomic) : Holonomic
7) Collision Detection: The collision detection method comprises of a function which determines if the robot is with the object bounds or not and returns 1 or 0 for true or false.
8) Collision Avoidance: Collision avoidance generates nodes only in C-free

1) Motion Planning Algorithm: RRT*

The motion planning algorithm used is RRT* which determines the most optimum path to the goal. The RRT star algorithm modifies RRT algorithm by including cost of path taken by robot. It rewires the tree such that:

a. The path to the newly generated node is via a nearby node through which path cost from start to new node is minimum.
b. The path to the nearby nodes, of the newly generated node, from start node is checked to see if the path when modified to go via the new node instead, whether cost is minimized.

The following image shows the RRT* algorithm:

Algorithm 6: RRT*.

```
1  V ← {x_init}; E ← ∅;
2  for i = 1, ..., n do
3      x_rand ← SampleFree_i;
4      x_nearest ← Nearest(G = (V, E), x_rand);
5      x_new ← Steer(x_nearest, x_rand);
6      if ObtacleFree(x_nearest, x_new) then
7          X_near ← Near(G = (V, E), x_new, min{γ_RRT*(log(card(V))/card(V))^(1/d), η});
8          V ← V ∪ {x_new};
9          x_min ← x_nearest; c_min ← Cost(x_nearest) + c(Line(x_nearest, x_new));
10         foreach x_near ∈ X_near do        // Connect along a minimum-cost path
11             if CollisionFree(x_near, x_new) ∧ Cost(x_near) + c(Line(x_near, x_new)) < c_min then
12                 x_min ← x_near; c_min ← Cost(x_near) + c(Line(x_near, x_new))
13         E ← E ∪ {(x_min, x_new)};
14         foreach x_near ∈ X_near do        // Rewire the tree
15             if CollisionFree(x_new, x_near) ∧ Cost(x_new) + c(Line(x_new, x_near)) < Cost(x_near)
               then x_parent ← Parent(x_near);
16             E ← (E \ {(x_parent, x_near)}) ∪ {(x_new, x_near)}
17 return G = (V, E);
```

The implementation of the above algorithm in this project is as follows:

1) A random node is generated such that it falls in the collision free path. This is done by looping till a random node falls outside the object boundary.
2) The collision check of the object and random node is done by checking whether the x and y coordinates of the random node lie within the coordinates of the object.

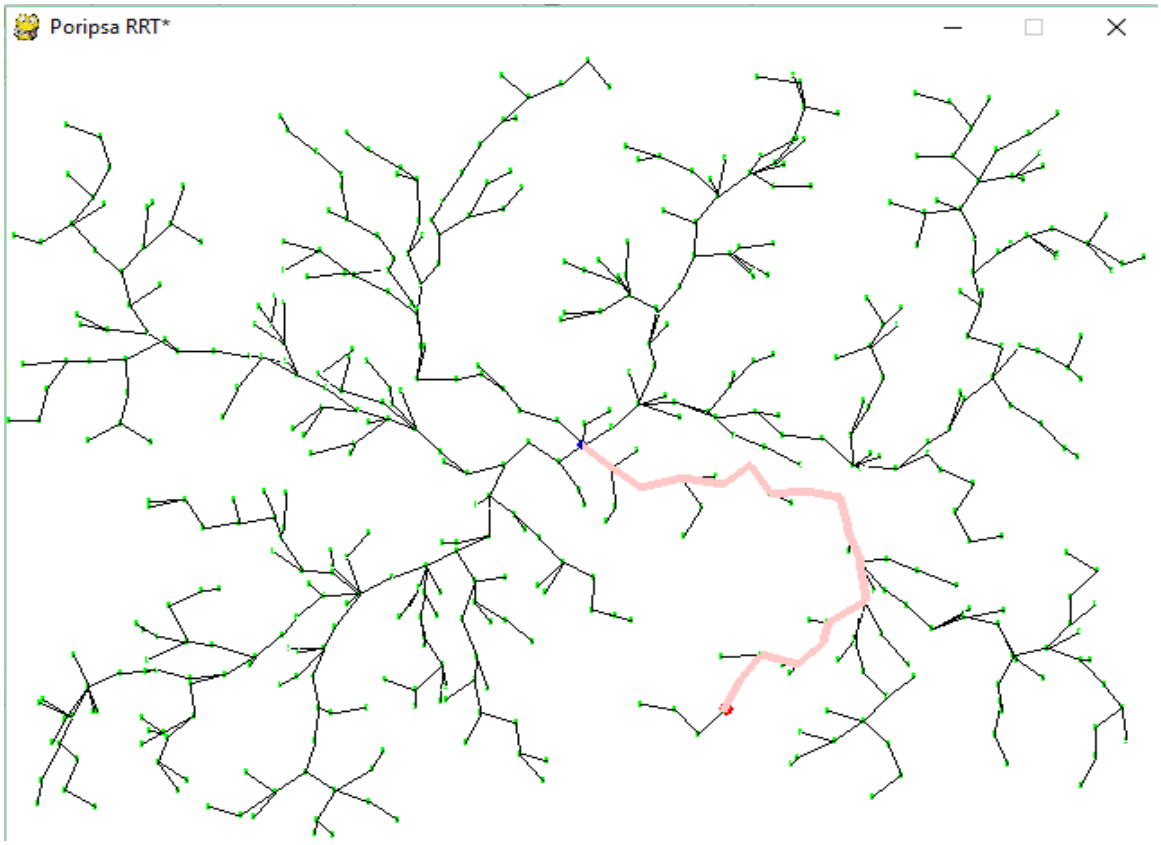
3) The nearest node to the random node is then found by comparing its distance to all nodes in the list.
4) Once the nearest node is found the random node is then steered to a new node using function steer, such that the new node lies in the direction of random node from its nearest node at a distance of epsilon (eps variable in code). This is done if the distance between the random node and its nearest neighbor exceeds epsilon otherwise the random node is returned as is, if the distance is less that epsilon.
   *New nodes are marked as green points in the simulation.
5) This nearest node is then saved as xmin and the cost of this nearest node from the start node added to the cost(distance) of the nearest node to the new node and is saved as cmin.
6) Then the program checks whether any other node fall inside the radius of:
   $$\min\{\gamma_{RRT*}(\log(\text{card}(V)) / \text{card}(V))^{1/d}, \eta\})$$
   If it does then those nodes are appended to the list of nodes termed 'near'. However the following images are generated using a fixed radius so as to include more near nodes to check the functionality and speed of the code.
7) Once near nodes are found the cost of going to new node via near nodes is found. The path of least cost is then cmin and its node is xmin.
8) The parent of the newnode is thus xmin (which is stored in a dictionary) and its path is stored in the edge list termed as 'edge'.
9) Next the path to the xmin node (from start node) via the new node is computed. If the cost of the path is less than that through its previous parent then its parent is changed to new node and the new edge is saved in the edge list.
10) Once the optimum path is found the function drawMotionPath is called which finds the nearest node to the goal node and traces it back to its parent until start node is reached.

The following images shows the simulation of the above program with different iterations:

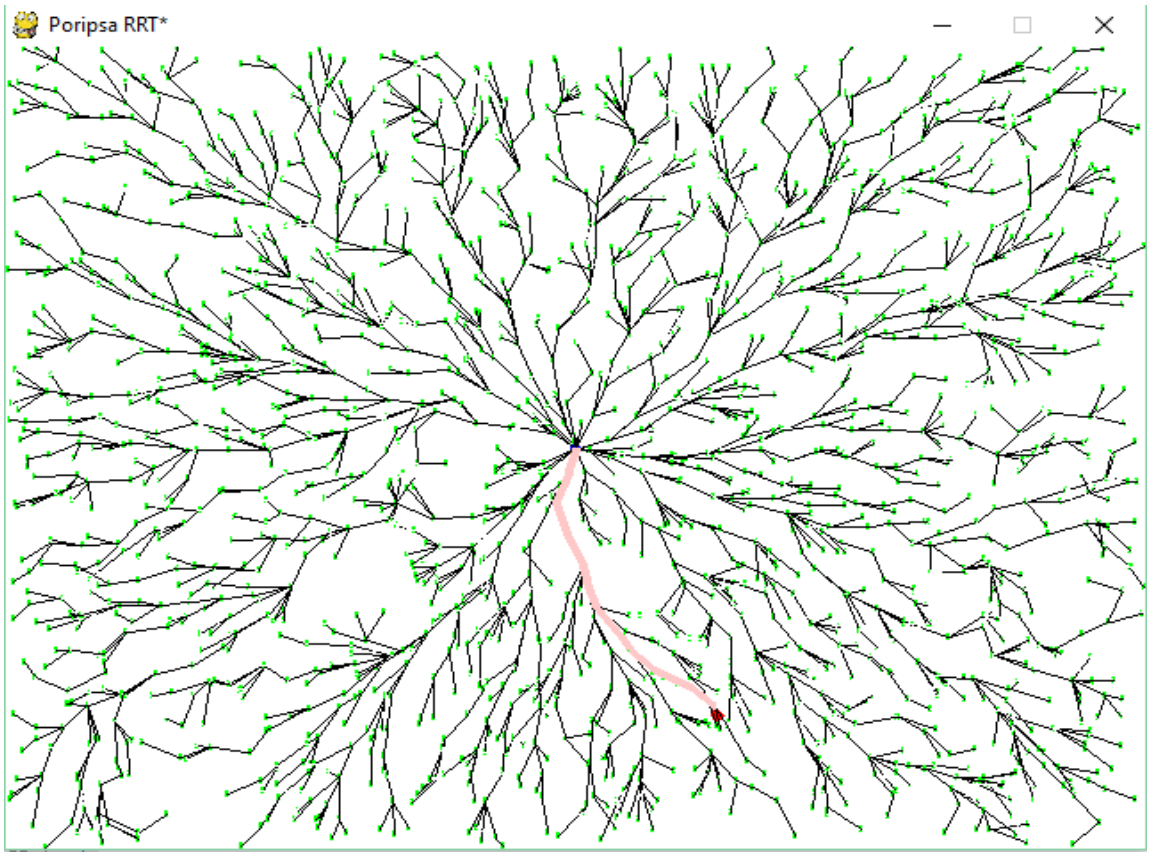
**The start node is marked in blue and the goal node is marked in red in the simulation. All other nodes are marked in green.

**The final path is marked in pink.

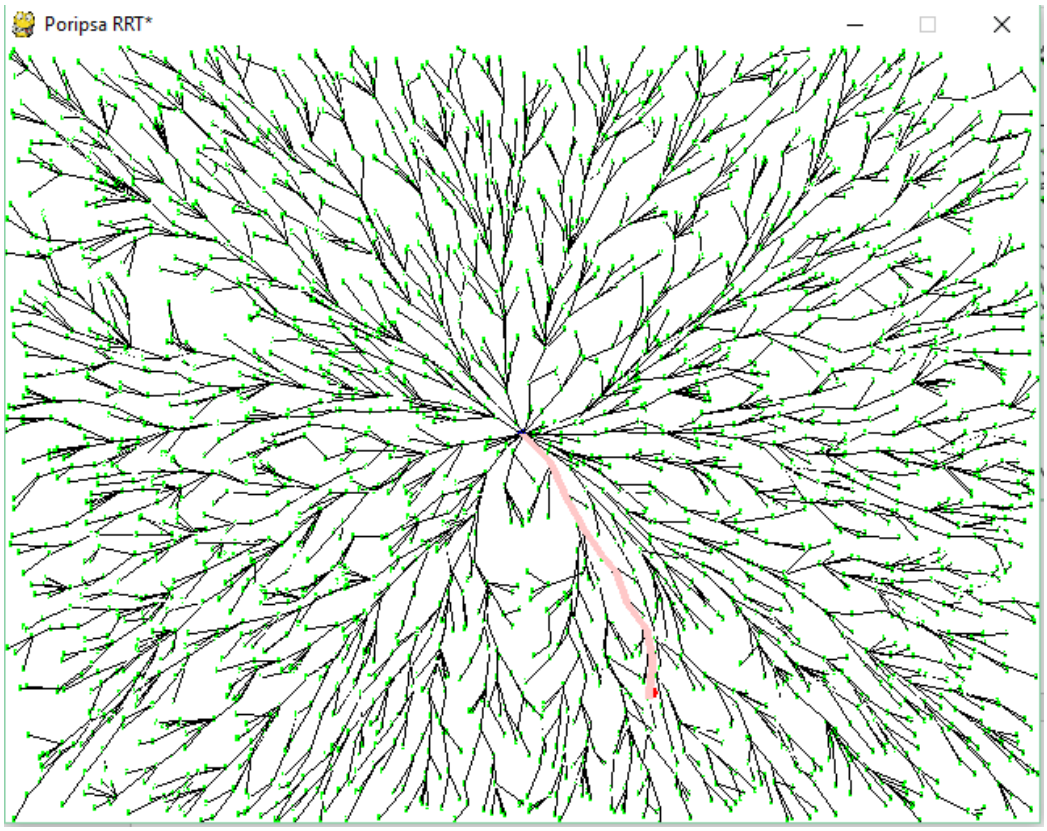
1) 500 Iterations: Time taken: 1.7secs



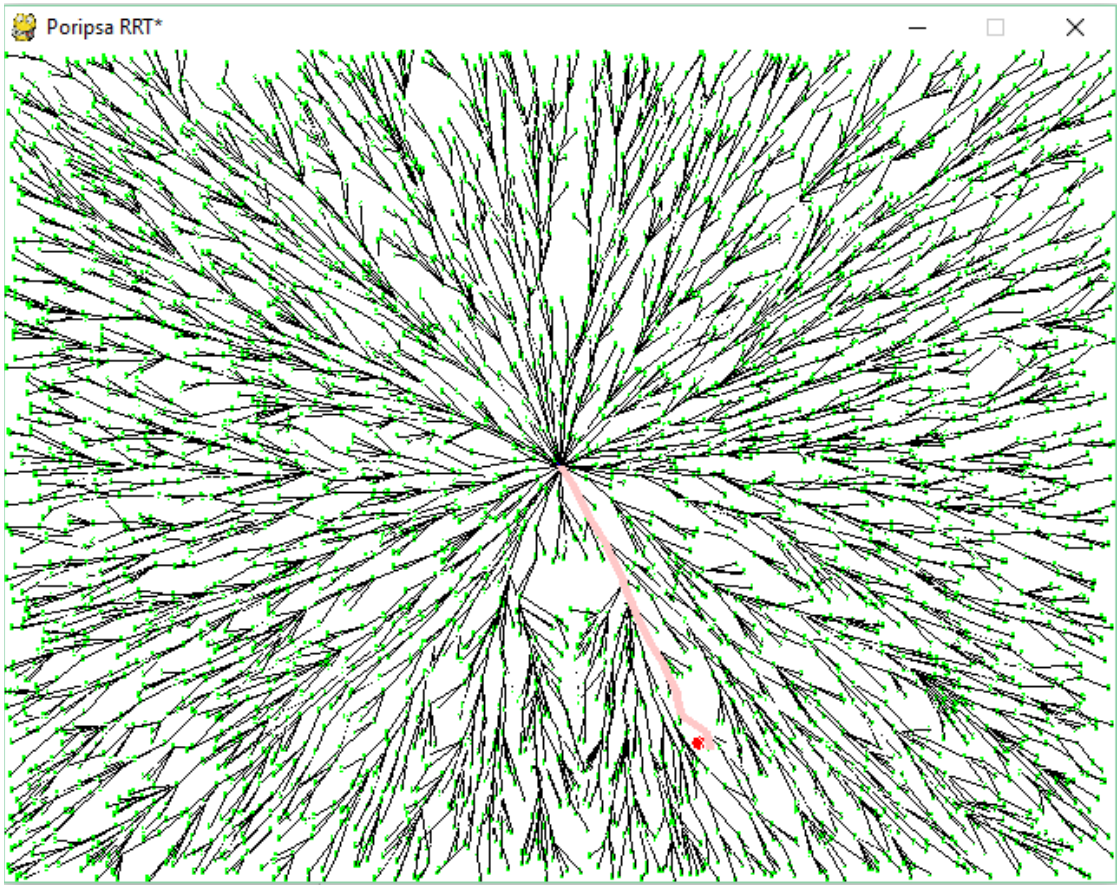
2) 1000 Iterations : Time taken: 3.5 secs



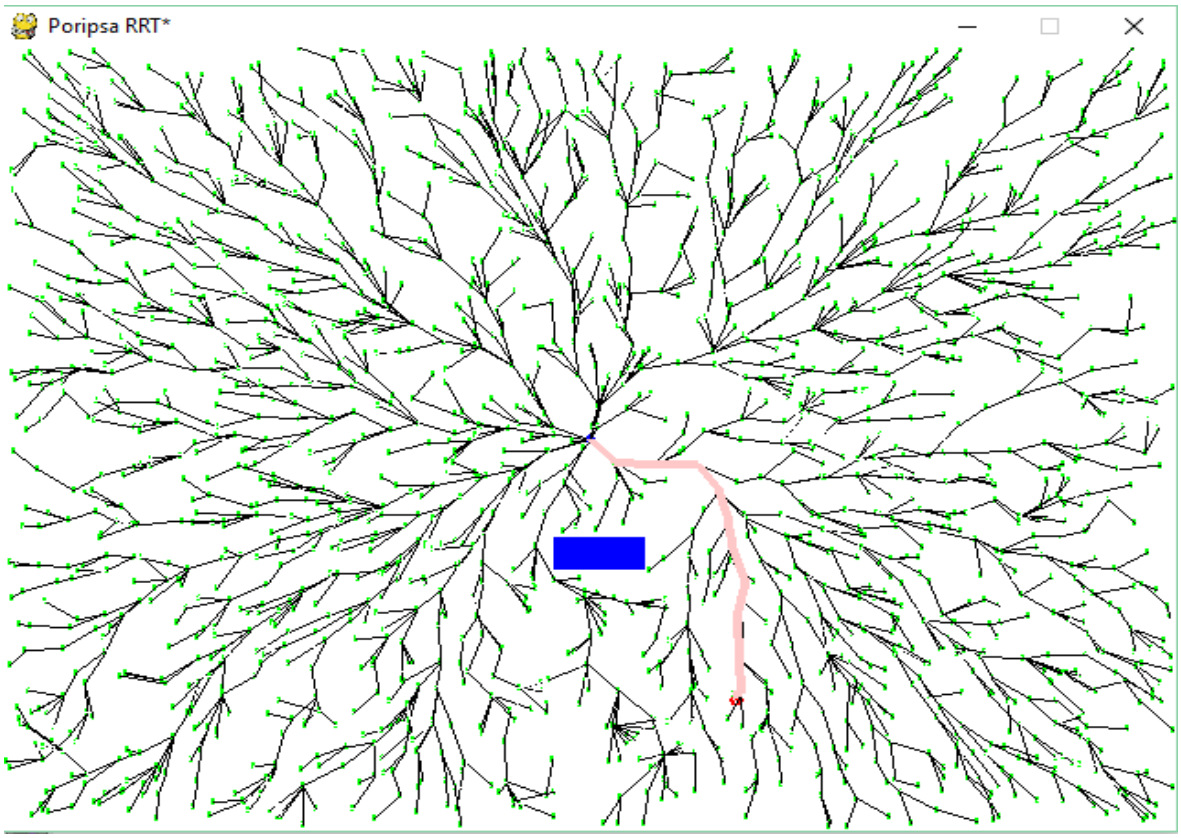
3) 3000 iterations : Time taken: 23 secs



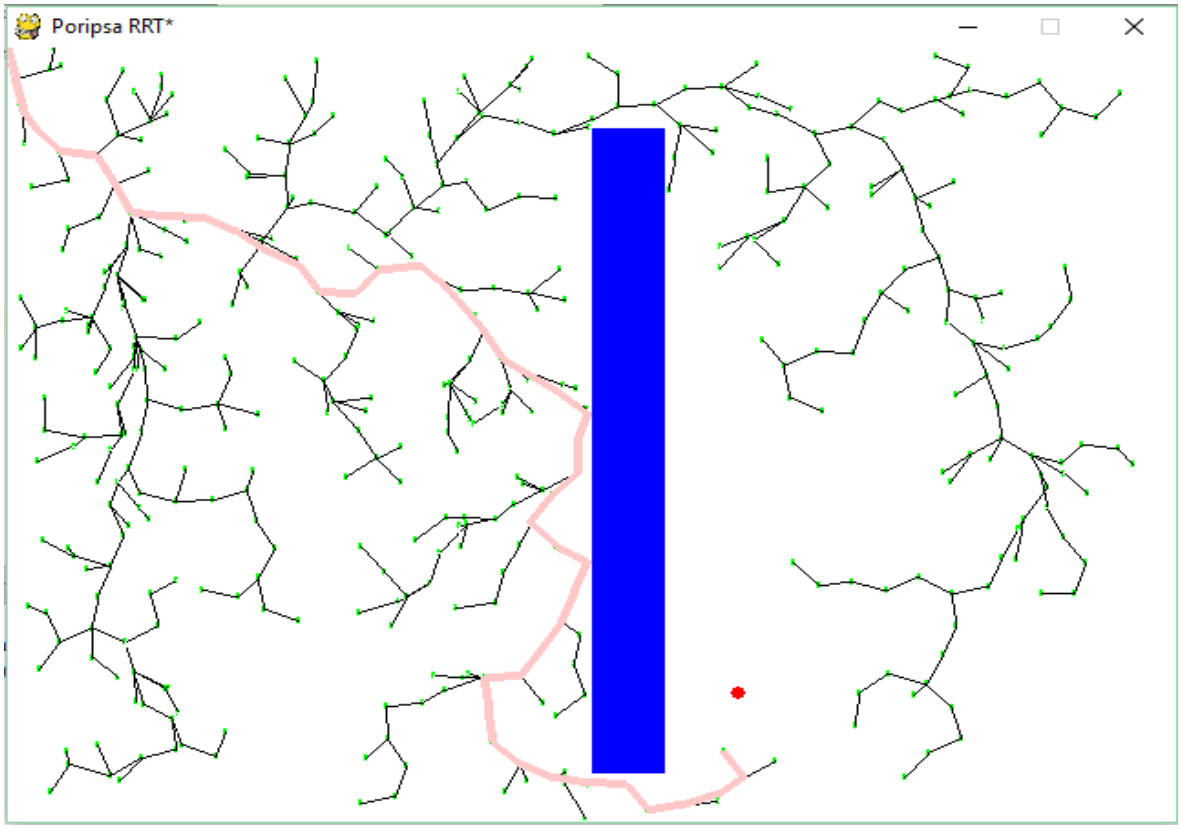
4) 5000 Iterations: Time taken: 70 secs (approx)



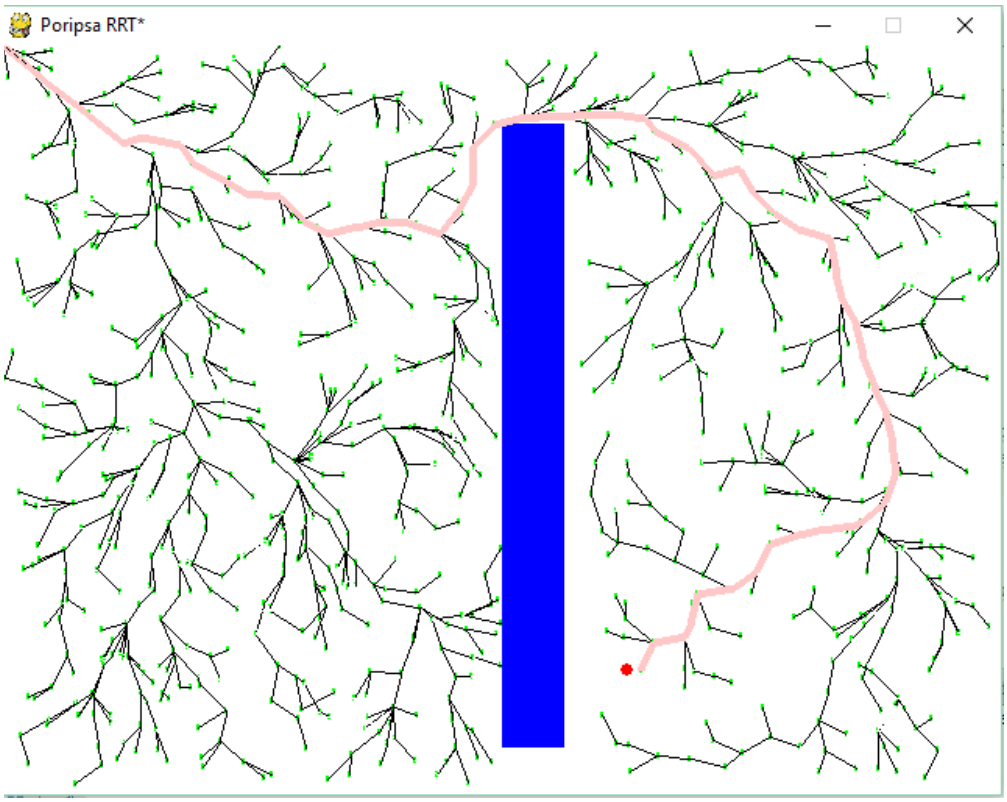
5) 2000 iterations : Collision Avoidance of small object :



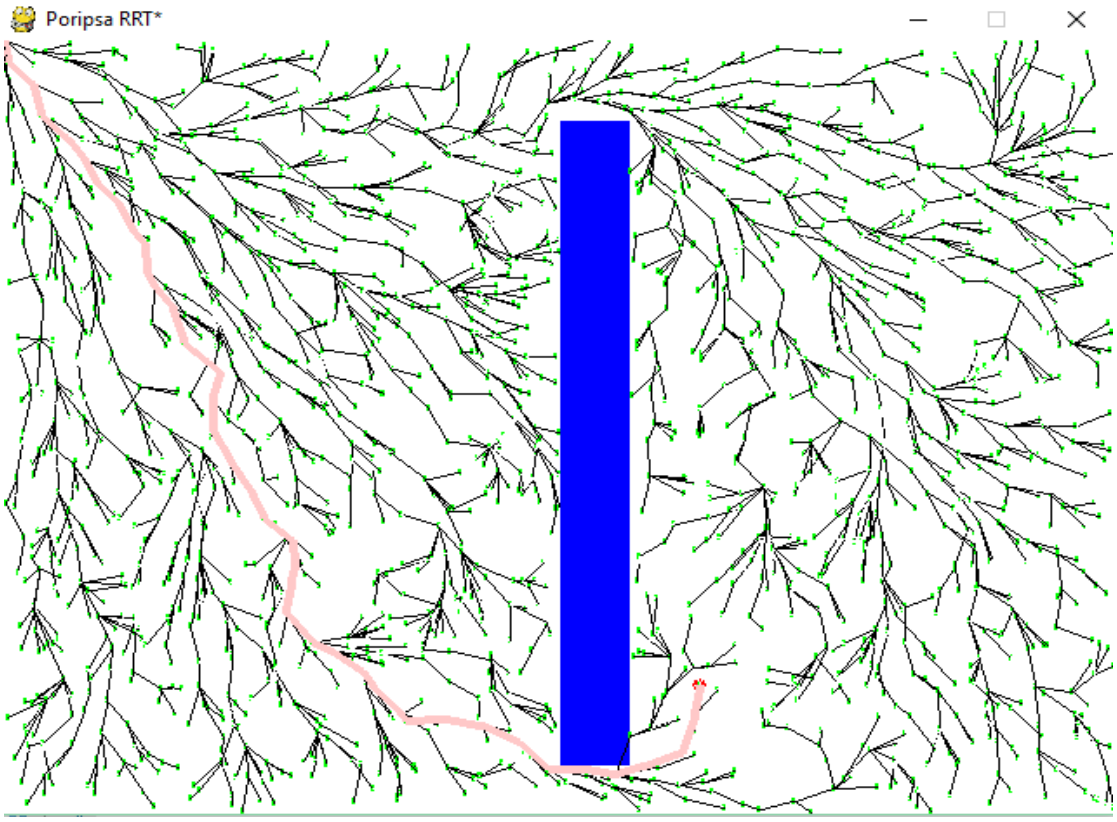
6) 500 iteration : Collision avoidance of large object: Goal changed to 0,0:



7) 1000 iteration : Collision Avoidance of large object:



8) 2000 iterations: Collision avoidance of large obstacle:



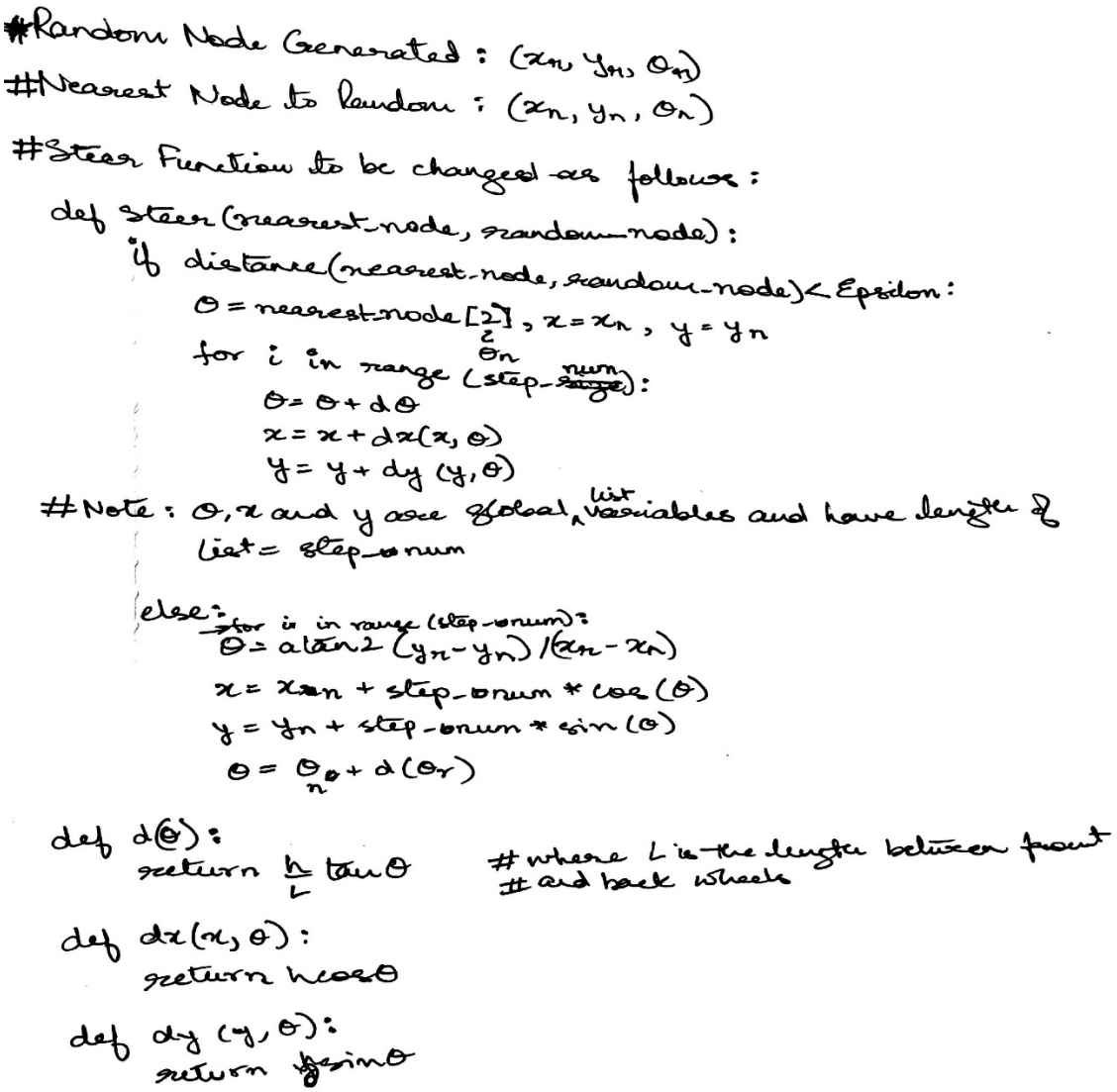
2) Collision Avoidance:

The collision was detected and avoided using the following ways:-

1) Random nodes were generated in c-free by looping inside a while loop (while collision keep on generating random nodes).
2) New node generated is checked again for collision. As once a random node is generated its distance to its nearest neighbor is checked and if the distance is greater than epsilon a new node is generated in its direction, the new node could fall in c-obstacle. Hence a check is done at steer function to see if node is inside the obstacle.
3) If the node is inside the obstacle space then the iteration is looped through (present iteration is commutated).

3) Non Holonomic Constraint:

The non-holonomic constraint implementation was worked out as:



4) Challenges:

1) Although I would have liked to implement the final path on turtlesim, that is, have the simulated turtlebot move along the final (goal) path, the challenge was more complicated than it seemed. The main problem was to convert the pixel to meter sizes of the pygame module (the graph generated using pygame) and the turtlesim simulation. Further more turtlesim simulation as far as I know can take in only angular and linear velocity commands (if not using OMPL libraries or motion planning already generated for turtlebot)whereas the nodes generated are digital. Integrating all this was a challenge.

2) Since I did not want to define nodes as an object of a class (there is no particular reason to this except that almost all available sample motion planning programs define a class and nodes as their objects), I got to experiment with dictionaries, something I had not used before. Even though the journey to the final version was took a lot of effort and research I am happy to have implemented RRT* using dictionaries and lists.

3) Some things which I could not complete to my satisfaction was generating RRT* with non-holonomic constraints. Though the code is developed, it could not be successfully implemented under present time constraints.

5) Conclusion:

In conclusion there was a lot of things to learn in this project more so from the immense lot of research which went into concepts which were eventually discarded for the scope of this project and time. Trying to implement path planning on turtlebot resulted in me learning a lot of things about, specially on how the turtlebot itself plans its course using OMPL libraries. Moreover having never developed a code from an algorithm using data structures, I certainly learned a lot in terms of programming as well, specially when it was done using a language I had not known previously. I think more than anything this project helped me learn about data structures and algorithm implementation as well as python.

References:

1. https://docs.python.org/,
2. Class documents on RRT*, RRT* RSS 2010 conference paper,
3. RRT* IJRR2011 journal paper,
4. Planning Algorithms by Lavalle,
5. Motion Planning Algorithm RRT star(RRT*) Python Code Implementation by Md Mahbubur Rahman,
6. www.pygame.org