# A* based motion planning and particle filters

Purushottam Sharma

CS 685 Fall 2015 Final Project

## Abstract

*This paper summaries the work performed for the final project for class CS685 on robot motion control planning and localization. It addresses the problem of efficient motion control and planning of the robot to reach the goal and its localization using particle filter. The discussion includes various strategies to control the motion of the robot in order to reach the goal, finally adopting A* as the best strategy. The particle filter and various key methods in its framework are discussed, followed by the results and a brief conclusion.*

## 1. Introduction

A fundamental task in robotics is to plan motion of robot from a start to a goal position among collection of static obstacles while avoiding collisions with obstacles [1]. Before discussing the project in detail lets formulate the problem properly. Let W denote the environment, which contains a robot and obstacles. The environment is 2D world, W = R2 and O ⊂ W is the obstacle region, which has a piecewise-linear (polygonal) boundary (shown in figure 1). The robot is a point that can move through the world, but must avoid touching the obstacles.

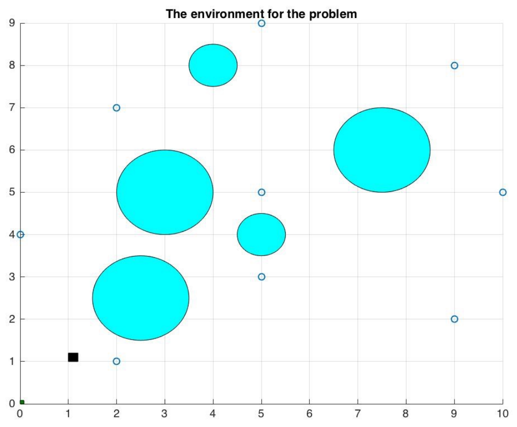

Figure 1: The environment

The path planning and localization problem can be summarized as: Given an initial position of the robot, compute, how to gradually move it to a desired goal position over an efficient/shortest path while never touching the obstacle region. And track the robot with particle filter as it moves along the path. The robot is equipped with a range sensor model to detect how far an obstacle is from it.

This next two sections, Motion planning and Particle filters. Motion planning discusses various strategies to control the motion of the robot from the start to the goal position, so that the robot takes the most efficient path (shortest path) to reach the goal. Particle filters discuses what a particle filter is and provides an overview of the working of the particle filter.

## 2. Motion Planning

Moving the robot towards a goal involves some kind of force or control strategy that directs it towards the goal position. The simplest approach towards solving the problem could be to use the distance between the robots position and goal's coordinates as an error to check whether the goal was reached after each time step. And use the control law to modify the its motion. The technique is simple, but very inefficient. And it fails to consider the obstacles the robot may encounter in its path.

Next approach that I considered uses potential fields. It assigns potential fields to the goal and obstacles. It avoids obstacles by assigning a repulsive field to them and assigns a positive field to the goal. This way there is a gradual descent of potential towards the goal and away from the obstacles. The negative gradient of the potential at any position in the environment directs the movement of the robot. The technique works well but has many pitfalls. The robot may get stuck in local minima, at which the total potential is zero. The gradient of the zero potential is zero. Thus, it cannot direct the robot anymore. However, there are methods to get out of local minima. One can assign rotating fields or random fields to the obstacles. This can avoid having local minima and keeping only one global minima at the goal. This technique guarantees the avoidance of on-line collision with obstacles and robot may eventually reach the goal. But it doesn't always give an efficient path. Hence, discarded. Next I tried some popular path finding algorithms out there like BFS and Dijkstra's algorithm. The last approach I used is the A* algorithm to find the shortest path to the goal. A* was found to be the best choice among all the approaches. The following section discusses the A* algorithm

## 2.1. A* Algorithm

A* (pronounced as "A star" ) is a computer algorithm that is widely used in pathfinding and graph traversal, the process of plotting an efficiently traversable path. A* basically uses a best-first search and finds a least-cost path from a given initial node to one goal node in a graph [5].

As A* traverses the graph, it builds up a tree of partial paths. The leaves of this tree (called the open set or fringe) are stored in a priority queue that orders the leaf nodes by a cost function, which combines a heuristic estimate of the cost to reach a goal and the distance traveled from the initial node [2]. Specifically, the cost function is

$$f(n) = g(n) + h(n) \quad (1)$$

Here, g(n) is the known cost of getting from the initial node to n; this value is tracked by the algorithm. h(n) is a heuristic estimate of the cost to get from n to any goal node.

I modified the A* algorithm to fit the problem's framework. I converted the environment into a 2D grid map, with each cell of certain size. Each cell acts as a separate node. To move across the grid from one cell to another, moves are defined to move to a neighboring cell. These moves are 2 horizontal moves( left and right), 2 vertical moves( up and down) and 4 moves along diagonal. The algorithm uses these moves to explore the grid to find the shortest path. The heuristic is the euclidean distance. h(n) is the euclidean distance between the current node and the goal node. And g(n) is the distance traveled by robot between the current node and the start node. The following is a pseudocode of the algorithm used in the project and the data structures used in the algorithm.

OpenSet [ x-coordinate, y-coordinate, cost, parentID, ID] contains the tentative nodes to be evaluated, initially containing the start node.

ClosedSet [ x-coordinate, y-coordinate, cost, parentID, ID] contains the nodes already evaluated.

Moves_possible [[-step,step]; [0,step]; [step,step]; [-step, 0]; [step,0]; [-step,-step]; [0,-step]; [step,-step]] contains 8 possible horizontal, vertical and diagonal moves that can be made on the grid. Step size should be kept smaller for more accurate path.

```
A*(start, goal, obstacles)
1.  ClosedSet ← { }
2.  OpenSet ← {start}
3.  Moves_possible ← {set of possible moves}
4.
5.  current_node ← start
6.
7.  while( distance between current_node and goal>0.7)
8.      current_node ← minimum cost node from
                OpenSet
9.      add current_node to ClosedSet
10.     remove current_node from OpenSet
11.
12.     for every move in Moves_possible
13.         new_node ← current_node + move
14.
15.         if (new_node is not in VisitedSet and is
                valid i.e doesn't touch any obstacle)
16.             evaluate cost for new_node
17.             new_node.parentID ← current_node.ID
18.             add new_node to Open_set
19.             add current_node to VisitedSet
20.
21. last_parentID ← current_node.parentID
22.
23. while ( last_parentID >0)
24.     add to path the node X from the ClosedSet
            whose ID == last_parentID
25.     last_parentID ← X.parentID
26.
27. return path
```

current_node[ x-coordinate, y-coordinate, cost, parentID, ID]: contains the current node in the algorithm

Starting with the initial node, the algorithm maintains a list of nodes to be traversed, known as the OpenSet. At each step of the algorithm, the node with the minimum cost is removed fro m the list, and assigned to the current_node and also added to the ClosedSet which maintains the list of the nodes that have been evaluated. Then, for the current node, further moves to its neighbors are evaluated to check if they have not already been visited by the algorithm( VisitedSet maintains this list and inVisited() function checks if a particular node has been visited or not) or are feasible i.e if they lie on the obstacles or not(isValid() function takes care of this). The valid moves have their cost evaluated and these valid neighbors are added to the OpenSet. The algorithm continues until the goal node has the minimum cost in the OpenSet, as the minimum cost node will be chosen as the current node and if its the goal node then the goal is reached to the current node is in certain vicinity of the goal.

The algorithm so far only explores the path. To find the actual sequence of steps, the algorithm maintains an ID for each node and its parentID. The parentID of a node is the ID of its predecessor on the path. This way each node keeps track of its predecessor. At the end, starting at the goal node, the Algorithm backtracks. The ending node will point to its predecessor, and so on, until some node's predecessor is the start node. And this sequence is our shortest path from the start to the goal which is returned as the output. The first phase of the algorithm, exploring the map for the shortest path is shown in figure 2, below. the start and the goal position are (0,0) and (9,7) respectively. Figure 3 shows the shortest path found by the algorithm.

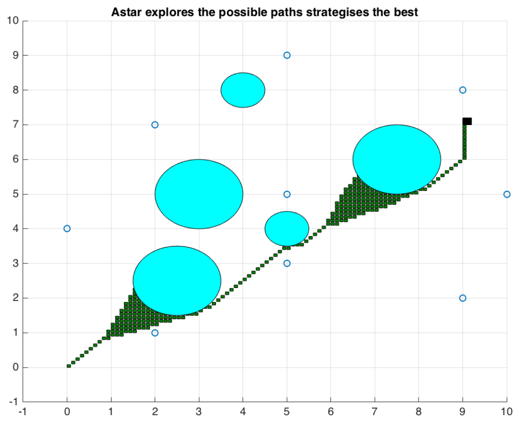

Figure 2: A* algorithm exploring for the shortest path

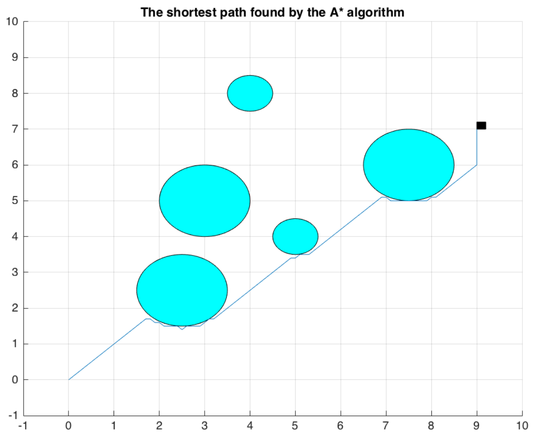

Figure 3: The shortest path found by the A* algorithm

Now that the shortest path has been found, it is now known where to move the robot to reach the goal. Each step of the path is fed to the goTo() function that was created in the homework #2. The goTo() function based on the initial and final positions uses the control law to specify the (v ,ω) for the differential drive kinematics of the robot and makes it move, traversing the shortest path given by the algorithm. The project shows the step by step exploration of the grid and building of the path.

## 3. Particle Filter

Particle filter is a sampling method used for estimating state in state space models where the state of a system changes with time and information about the state is obtained through noisy measurements made at each time step [6]. The robot tracking problem can be modeled into the particle filter estimation problem as estimating the state/configuration of the robot, $x_k$ given as:

$$x_k = f_k(x_k-1, v_k-1) \qquad (2)$$

where $x_k$ is a state of the robot at time k, $v_k-1$ is the state noise vector, $f_k$ is a non linear time dependent function. $x_k$ is unobservable and Information about $x_k$ is obtained by only the range sensor mounted on it and it provides noisy measurements , $z_k$, such that:

$$z_k = h_k(x_k, n_k) \qquad (3)$$

where $h_k$ may also be a non linear and time dependent function describing the measurement process and $n_k$ is the measurement noise.

Basically, Particle filter is a technique used to perform simulations to estimate Bayesian models. The key idea is to represent the required posterior density function by a set of random samples with associated weights and to compute estimates based on these samples and weights. A short video of 4 minutes on youtube (https://www.youtube.com/watch?v=H0G1ys1M5rc ) explains robot localization using particle filter very well in simple terms. The particle filter is visualized in the figure below.

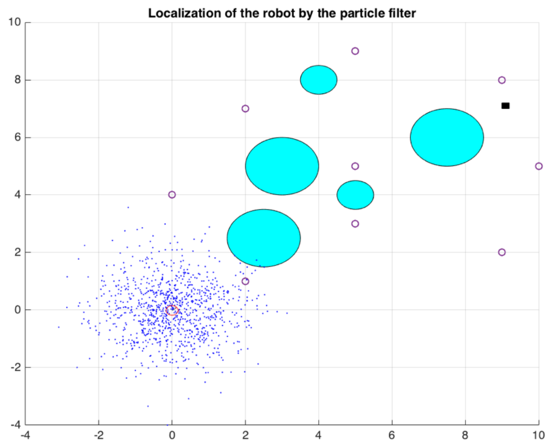

Figure 4: The particle filter tracking the robot

The Particle filter algorithm has three main tasks:

First, Sample the next generation of particles. The new particles are propagated using some motion model. Each particle's new position is calculated with the intended motion along with a noise component, which is sampled from the above distribution. This effectively samples from the motion model and provides us with what is known as the belief state at time.

Second, compute the importance weights using some measurement model. It involves calculating the likelihood of being in a given position given the measurements from the robot's sensors. For each particle error is calculated in the measurements. This error along with the noise parameters of the sensor are used to calculate the likelihood of each measurement. The probability from multiple sensor measurements are multiplied together.

Third is resampling. This step samples new particles from the existing particles. This sampling (with replacement) is with respect to the weight of each particle. High weight Particles or high likelihood values have more chances of being selected. each particle is assigned some non-zero value maintaining a broad set of particles for the belief state.

I employed the particle filter given in the project framework to track the robot. The motion model takes odometry (rotation1, translation, rotation2)sequence as the input. It was calculated from the poses (x, y, θ) returned by the goTo() function. A precomputed sensor data was provided for a set odometry sequence. As the sensor data is for a separate motion, It gets really difficult to keep track of the robot when using the filter with a path selected in the problem. For instance, when moving the robot from (0,0) to (9,7) over the shortest path( as shown in figure 2), the estimated location of the robot drifts far away from the actual location of the robot, as shown in figure 5. This is because the sensor readings are not at all consistent with the motion of the robot.

## 4. Results

The A* approach was applied to the robot's motion planning problem. The shortest path from the start to goal positions was found. There is a tradeoff between the accuracy of the shortest path and speed of computing it. It depends on how fine the grid is. Smaller the cell, more precise is the path, more time consuming is the algorithm whereas Larger the cell, less precise is the path and lesser time it takes to compute it.The computed path moves the robot towards the goal position in an efficient manner while avoiding collision with the obstacles.

I was able to integrate Particle filter with the motion planning of the robot. The odometry calculated from the robots poses was fed to the particle filter. The sensor data

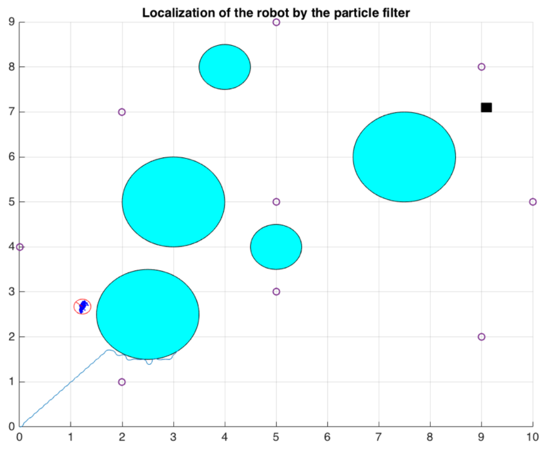

Figure 5: The particle filter fails to track the robot

was provided before hand and was for a particular motion sequence of robot. As a result the sensor readings were not at all consistent with robot's motion. As a result the robot is not properly tracked. Otherwise with the supply of the sensor data for the current motion, particle filter can track the robot with good precision.

## 5. Conclusions

The A* approach works very well to find the shortest path and guide the robot to traverse an efficient path to reach from the start to the goal position. Besides A*, there are other algorithms that provide even better results in certain situations. The D* algorithm is like A* algorithm. D* works in the dynamic environment, where the obstacles and other information is not already known. Rapidly-Exploring Random Trees (RRT) is another technique that addresses path finding problem by using a randomized approach that aims at quickly exploring a large area of the search space with iterative refinement. Particle filters offer a simple and elegant solution the the problem of robot localization. One of the great features of this algorithm is that it expends computational effort in a way that is proportional to the importance of the particles.

## 6. References

[1] Steven M. LaValle,
"Motion Planning: The Essentials" ,
http://msl.cs.uiuc.edu/~lavalle/papers/Lav11b.pdf

[2] http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

[3] Nikolaus Correll,
"Introduction to Robotics #4: Path-Planning",
http://correll.cs.colorado.edu/?p=965

[4] Harika Reddy,
"PATH FINDING - Dijkstra's and A* Algorithm's" ,
http://cs.indstate.edu/hgopireddy/algor.pdf

[5] https://en.wikipedia.org/wiki/A*_search_algorithm

[6] Emin Orhan,
"Particle Filtering" ,
http://www.cns.nyu.edu/~eorhan/notes/particle-filtering.pdf