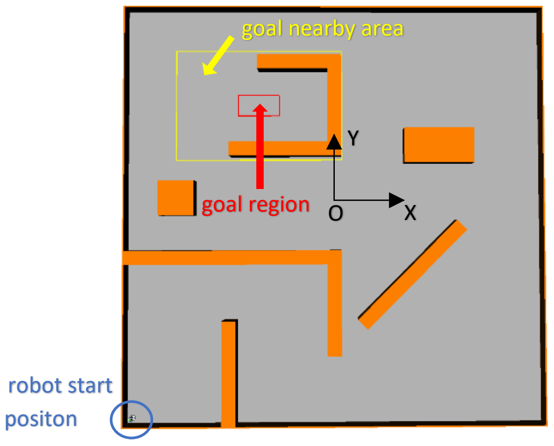
Kinodynamic RRT Implementation Report

1. Introduction

We have learned the basic form of RRT algorithm in detail during lectures and implement it in HW 3, which is very efficient in exploring the entire configuration space. However, the simplest version of RRT connect adjacent points in configuration space just simply by a straight line between them, which assumes that the robot can move in any direction at any time. From my perspective, many cases in reality do not fulfill the assumption, such as vehicles and airplanes, which cannot move in arbitrary direction because their configurations are subjected to their dynamic models, meaning that they conforms to non-holonomic constraints. Thus to solve these importance practical cases, adjacent points in the RRT tree cannot be simply connected by a straight line, and the constraints indicated by their dynamic models need to be taken into account. To get a deeper understanding of the solution of this kind of problems, I implement kinodynamic RRT algorithm, the improved version of RRT which can deal with non-holonomic constrains, on a simulated PR2 robot with a relatively simple dynamic model and an environment with some obstacles in OpenRAVE. Specifically, I want to study how the number of motion primitives influence the performance of kinodynamic RRT algorithm. The detailed descriptions of my implementation are given in the following contents.

2. Robot and environment description



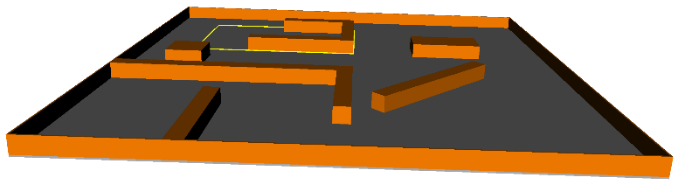


Figure 1. Top and side veiw of environment

As mentioned above, I use PR2 robot in this problem, and the values of all its joints are fixed. It is placed on the xy plane and is only allowed to move in x and y direction and rotate around z axis, denoting the angle measured counterclockwise from x axis with θ. Besides, the robot can accelerate and decelerate in x, y and θ, meaning that the velocity component in x, y direction and the angular velocity of the robot can vary, then I denote them $\dot{x}, \dot{y}$ and $\dot{\theta}$ respectively, thus the state space of the robot is constructed by the six variables mentioned above.

The environment I use in my implementation is shown in Figure 1, and its dimension is $60m \times 60m$. The origin of coordinate system is located at the center of the environment, and the start position of robot is at the lower left corner with coordinate $(-29, -29)$. As for other variables in the state space, $\theta, \dot{x}, \dot{y}, \dot{\theta}$, their values equal 0 in the beginning, which means that the initial orientation of the robot is parallel to x axis, and robot remains stopped. The goal region is the part of environment within red rectangle shown in the picture, and specifically is the region in which $-14 \leq x \leq -8$ and $14.5 \leq y \leq 17.5$. The terminal condition is that robot stops in the goal region, thus at the end robot's position is inside the red rectangular and its velocity and angular velocity $\dot{x}, \dot{y}, \dot{\theta}$ equal 0. There is a yellow rectangular area in the picture, called goal nearby area. When robot reaches this area, I want it to start to reduce velocity so that it can be prepared to stop in the goal region.

3. Model and method description

The dynamic model controlling the robot in this problem is as follows, in which u and φ denote the velocity and the steering angle of robot respectively, and L is the distance between its front and back wheels. In this problem, it is assumed that the robot can control the acceleration, denoted $\dot{u}$, and the steering angle φ at every moment, thus they establish the control space of the robot.

$$\begin{cases} \dot{x} = u\, cos\theta \\ \dot{y} = u\, sin\theta \\ \dot{\theta} = \frac{u}{L}\, tan\varphi \end{cases}$$

In the RRT algorithm, creating random points is the first step, then followed by deciding the closest points in the RRT tree and extend towards the random points. In this problem, the way I implement these steps is actually related to the position of the robot in the environment, the value of x and y specifically. On the one hand, if RRT tree has not reached the nearby area of the goal region, mentioned as goal nearby area in section 2, then I want it to explore the geometry of the map rapidly, and I don't really care about the value of other variables in the state space as long as they are subjected to the constraints exerted by my dynamic model. Thus, the random points I create in this situation are in the form (x_{random}, y_{random}) , and the distance metric I use under this circumstance is Euclidean distance $d = \sqrt{(x - x_{random})^2 + (y - y_{random})^2}$, which means that the point I choose is geometrically closest to the random point. The way I extend towards the random points will be discussed in the following paragraph. On the other hand, when some points in the RRT tree reach goal nearby region, it is necessary to think about how to satisfy the terminal condition as fast as possible. As mentioned in section 2, the terminal condition is that the robot stops within the goal region, meaning that the robot's position (x_{robot}, y_{robot}) has boundary values and its velocity u_{robot} equals zeros, notice that the angular velocity also equals zero if velocity is zero. The robot's velocity tends to be slower when exploring in this region so that it should be easier to stop in the goal region. Therefore, if the random point being created is also in the goal nearby region, then create it in the form$(x_{random}, y_{random}, u_{random})$, and set the value of u_{random} relatively low. Then use distance metric $d = \sqrt{(x - x_{random})^2 + (y - y_{random})^2 + 2(u - u_{random})^2}$ that stress the importance in the velocity and tend to decelerate the robot when exploring towards the random point.

In this paragraph, the way of extending towards a given random point is demonstrated. I choose to apply the method of computing motion primitives in this step. Firstly, I discretize the control space by assigning specific values to $\dot{u}$ and φ and then setting up an input set that includes all of their combinations. Then, I want to compute motion primitives of the robot given different inputs, and I let each motion primitive to be the trajectory of robot that is obtained by applying a pair of inputs for 1 second. In the next step, I discretize the trajectory into 10 time steps, each time steps lasts for $dt = 0.1s$, then compute the linear approximation of the segment and get the state variables using Euler integration and state variables at the last time step. To explain it in detail, let's assume that the input of a certain motion primitive are $\dot{u}$ and φ, then the state space at time step k+1 can be computed given the state space at time step k by the following equations. After 10 iterations, a complete primitive is obtain, then I repeat the process with different inputs to get all of the motion

$$\begin{cases} u_{k+1} = u_k + dt * \dot{u} \\ \dot{\theta}_{k+1} = \frac{u_{k+1}}{L}\, tan\varphi \\ \theta_{k+1} = \theta_k + dt * \dot{\theta}_{k+1} \\ \dot{x}_{k+1} = u_{k+1}\, cos\theta_{k+1} \end{cases} \quad \begin{cases} x_{k+1} = x_k + dt * \dot{x}_{k+1} \\ \dot{y}_{k+1} = u_{k+1}\, sin\theta_{k+1} \\ y_{k+1} = y_k + dt * \dot{y}_{k+1} \end{cases}$$

primitives that start at the same point. The next procedure is to choose the end point which is closest to the random point from the previously computed motion primitives. The distance metric here is exactly the same as what is mentioned in the previous paragraph. Then recursively compute new motion primitives until reach the proximity of the random point or no more new motion primitives could be obtained because of collision with obstacles.

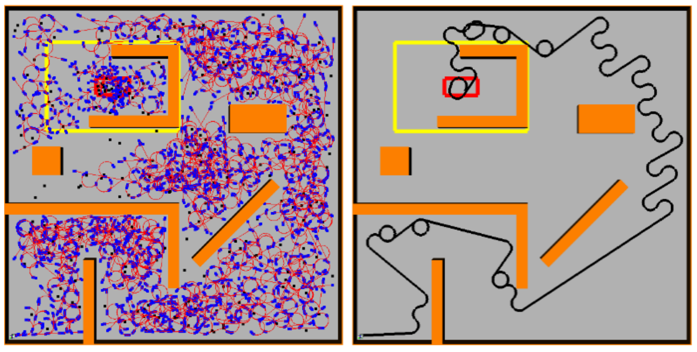
4. Implementation process description

First of all, pseudocodes are provided here to draw a general picture of my implementation.

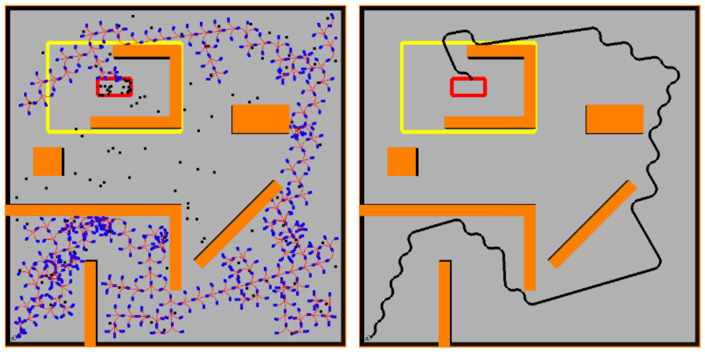
```
$tree \leftarrow initialize\ with\ start\ configuration$                      # initialize RRT tree set
$input\ set \leftarrow permutation\ of\ different\ value\ of\ \dot{u}\ and\ \varphi$      # define input set
$while\ true$
  $for\ p \in tree$
    $if\ p.x, p.y\ within\ goal\ region\ and\ p.u < threshold$           # check if the robot stops within goal region
      $current\ configuration \leftarrow p$
      $end\ RRT \leftarrow true$
      $break$
  $if\ end\ RRT\ is\ true$                                            # end RRT algorithm if goal is reached
    $break$
  $generate\ a\ random\ configuration$
  $if\ random\ configuration\ collide\ with\ obstacle$                  # discard the random point if it collides with obstacles
```

```
      $continue$
    $closest\ distance \leftarrow \infty$                                  # find the closest point to random point in RRT tree
    $for\ p \in tree$
      $if\ distance(p, random\ configuration) < closest\ distance$
        $closest\ point = p$
        $closest\ distance = distance(p, random\ configuration)$
    $if\ closest\ distance < theshold$                                      # not extend if the random point is too close
      $continue$
    $else$
      $while\ true$
        $primitives \leftarrow \emptyset$
        $for\ q \in input\ set$
          $one\ motion\ primitive \leftarrow model(closest\ point, q)$      # compute the set of points that construct a primitive using equations in section 3
          $if\ it\ collide\ with\ obstacle$                                 # discard a primitive if any of its points collides with obstacles
            $continue$
          $tree \leftarrow tree \cup\ one\ motion\ primitive$               # add a feasible primitive to the RRT tree
          $primitives \leftarrow primitives \cup one\ motion\ primitive$    # add a feasible primitive to the set of all primitives starting from one point
        $best\ distance \leftarrow \infty$
        $for\ p \in primitives$
        $if\ distance(p, random\ configuration) < best\ distance$           # find the closest end point of motion primitives to random point
          $best\ point = p$
          $best\ distance = distance(p, random\ configuration)$
        $if\ best\ distance < theshold\ or\ the\ set\ primitives\ is\ empty$  # check if extension is close enough or could not continue because of obstacles
          $break$                                                          # end extension if terminal condition is satisfied
        $closest \leftarrow best\ point$                                    # replace the closest point of last step with the one in current step
  $path \leftarrow \emptyset$
  $while\ current\ configuration\ is\ not\ start\ configuration$           # get path sequences by backtracking the parent from goal to start configuration
    $path \leftarrow path \cup current\ configuration$
    $current\ configuration \leftarrow current\ configuration.parent$
  $return\ the\ reverse\ of\ path$
```

Specifically, I use three different input sets in my implementation. The first one is constructed by $\dot{u} = \{-0.5, 0, 0.5\}$ and $\varphi = \{-\frac{\pi}{10}, 0, \frac{\pi}{10}\}$, the second one by $\dot{u} = \{-0.5, -0.25, 0, 0.25, 0.5\}$ and $\varphi = \{-\frac{\pi}{10}, -\frac{\pi}{20}, 0, \frac{\pi}{20}, \frac{\pi}{10}\}$, and the third one by $\dot{u} = \{-0.75, -0.5, -0.25, 0, 0.25, 0.5, 0.75\}$ and $\varphi = \{-\frac{\pi}{10}, -\frac{\pi}{15}, -\frac{\pi}{30}, 0, \frac{\pi}{15}, \frac{\pi}{30}, \frac{\pi}{10}\}$. Besides, I set a speed limit for the robot, $0 \leq u \leq 2.5$, meaning that the robot can neither go backward nor go in a velocity greater than 2.5m/s. There are some reasons that I set the limit. Firstly, I assume that the robot is similar to the Dubins car, meaning that it cannot go backwards, thus $u \geq 0$. Secondly, as u increases, when $\varphi \neq 0$, from the equations in section 3 we know that the absolute value of $\dot{\theta}$ also increases, and so does that of θ, then angle changes will be greater and greater so that the robot may go in a circle finally, which greatly impairs the expansion efficiency of the algorithm and causes the path to have many big curves. Given the above inputs value, set maximum of $u = 2.5$m/s is reasonable because even when using extreme values of $\dot{u}$ and φ, the angle changes is acceptable.



maximum velocity limit $3.5m/s$



maximum velocity limit $2.5m/s$

Figure 2. Comparison of different maximum velocity limits

The pictures in Figure 2 are the RRT trees and their resulting path when I set different maximum velocity limits. Thin red curves and blue dots represent motion primitives and their end points respectively, black curves represent the path, and the input set is the same for them. When limit is 3.5m/s, there are many circles in the RRT tree, which means many primitives overlap one another when exploring the map, this is actually very inefficient, and consequently the time needed to find the path is more likely to be longer than the right case. Besides, curves in the trajectory of the left case are clearly more significant, and there are even some circles in the trajectory, thus the path quality is worse than the right case.

When the terminal condition is reached, state space statistics can be obtained from tree set. The sequence of (x, y, θ) is used to generate the path and control the robot in execution. In the final execution step velocity and angular velocity statistics are not used actually due to my unfamiliar with relevant control function in OpenRAVE, but GSI told me that I could show the changes of velocity and angular velocity with graphs, so I create two plots, velocity vs time and angular velocity vs time, to indicate how $\dot{x}, \dot{y}$ and $\dot{\theta}$ changes within the trajectory. The results of my implementation will be discussed in the following section.

5. Results description

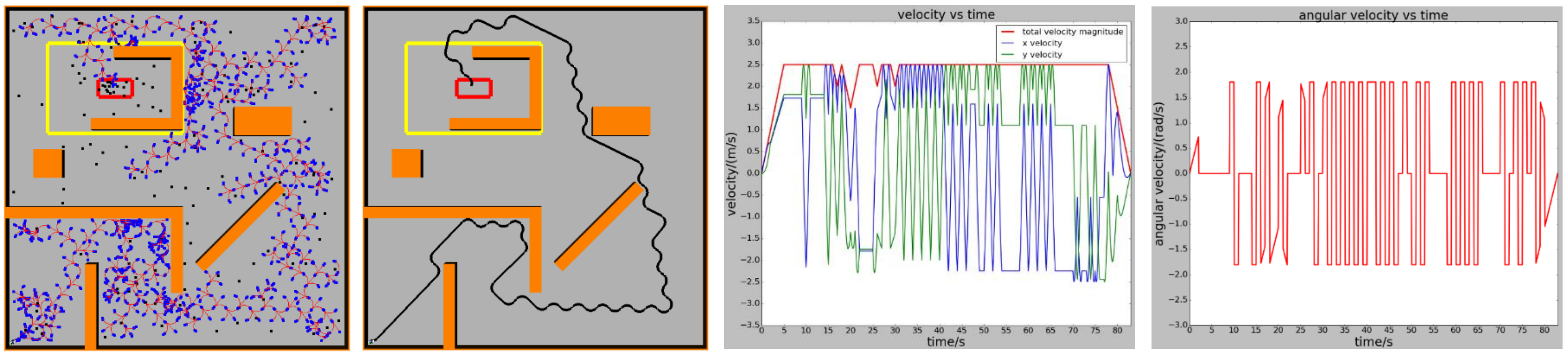


Figure 3. Results of the case with least number of motion primitives

The pictures in Figure 3 is the case given input set that is constructed by $\dot{u} = \{-0.5, 0, 0.5\}$ and $\varphi = \left\{-\frac{\pi}{10}, 0, \frac{\pi}{10}\right\}$, and since the number of inputs is the least one among three input sets mentioned in section 4, the number of motion primitives starting from a point is also the least one. Due to the probabilistic nature of RRT algorithm, the computation time of this case actually can varies from about several seconds to about 30 seconds. The results I record here take 18.23s to compute. The two pictures on the left are the RRT search tree and the path respectively, which is similar to that in section 4, so I want to pay more attention to the velocity and angular velocity graph here. In the velocity graph, the sign of x and y velocity indicates the direction, which is shown the picture in section 2. The sign of angular velocity also represents direction, and the positive direction is counterclockwise about z axis. In the velocity graph, the total velocity magnitude maintains at the maximum velocity 2.5m/s for most of the time, except for a clear acceleration process in the beginning and a deceleration process at the end, but the x and y velocity fluctuate largely most of the time. It can be seen from the path picture that a large proportion of the path is made up of consecutive curves, and these parts correspond to the large fluctuation part in x and y velocity curves. Besides, the corresponding angular velocity also fluctuate.

Note that the changing process of velocity and angular velocity are different in this problem. According to dynamic model in section 3, $\dot{x}$ and $\dot{y}$ are controlled by the total velocity u and robot's orientation θ, and because it's known that the change of u and θ are continuous, thus $\dot{x}$ and $\dot{y}$ are continuous function on time though sometimes they change sharply. As for $\dot{\theta}$, it is controlled by total velocity u and steering angle φ, then because φ is the input variable, its value is discrete and can change instantly regardless of the former value, thus angular velocity $\dot{\theta}$ is not a continuous function on time. The perpendicular segment in angular velocity curves actually means that angular velocity is not continuous here.

The pictures in Figure 4 is the case given input set that is constructed by $\dot{u} = \{-0.5, -0.25, 0, 0.25, 0.5\}$ and $\varphi = \{-\frac{\pi}{10}, -\frac{\pi}{20}, 0, \frac{\pi}{20}, \frac{\pi}{10}\}$. This case may takes about 20 to 60 seconds to get a path, and the computation time I record here is 22.40s. Due to a larger number of motion primitives compared to the case in Figure 3, the RRT search tree has more " branches " at each node. It is obvious that the path is smoother than that of the former case because the number of curves is fewer and the curvature is lower. The velocity and angular velocity graph also verify the improvement in path quality. In the velocity graph, it is similar to the former case that the total velocity magnitude remains at the maximum limit value most of the time, and we observe that the curvature of the path is generally lower in this case, thus the centripetal force needed to turn around the curve is lower, which means in reality the path in this case is easy to go through. Besides, there

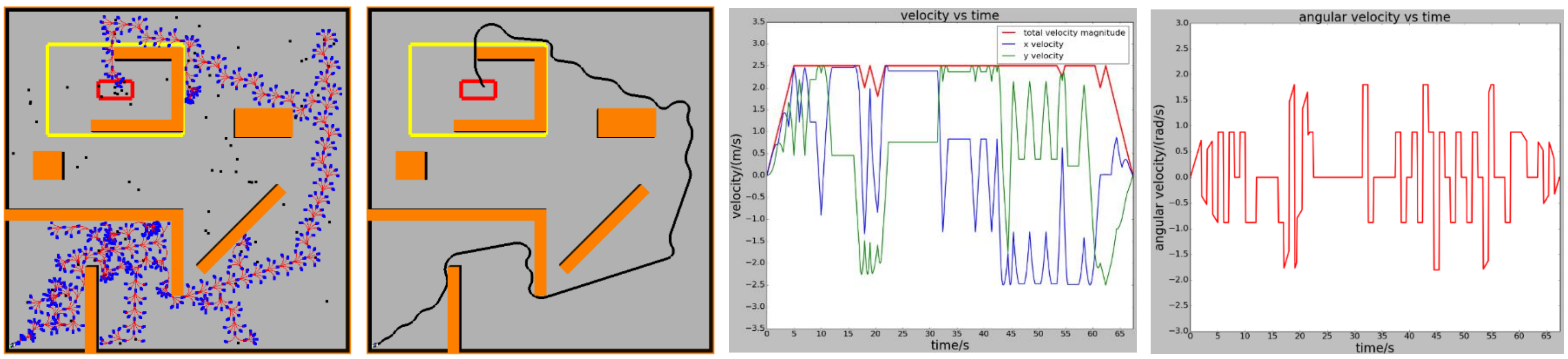


Figure 4. Results of the case with median number of motion primitives

are clearly fewer fluctuations in the $\dot{x}$ and $\dot{y}$, and the fluctuation magnitude is less than that of the case in Figure 3. In the angular velocity graph, the fluctuations frequency is also lower, and most of the angular velocity fluctuations have a smaller magnitude than that of the former case. The time cost of the trajectory in this case is also shorter than the former case. These evidences all shows that the trajectory is improved apparently by increased number of motion primitives. But the computation time is likely to be longer, I think the reason is that firstly the number of nodes in the search tree increases faster, then the search time for finding the closest point to a random point is longer, and secondly, the speed of extending towards a random point is slower since more motion primitives need to be computed from a point.

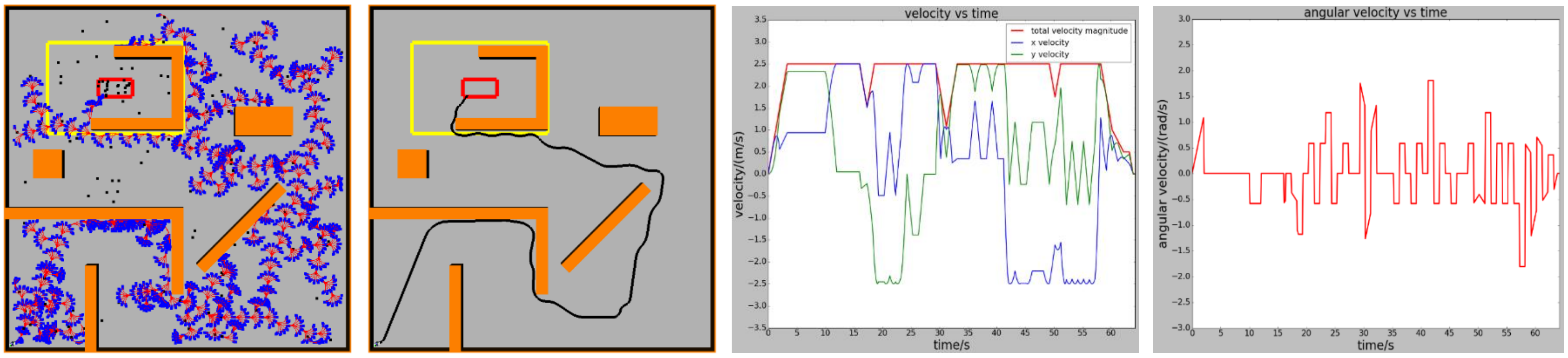


Figure 5. Results of the case with largest number of motion primitives

The pictures in Figure 5 correspond to the input set which is made up of $\dot{u} = \{-0.75, -0.5, -0.25, 0, 0.25, 0.5, 0.75\}$ and $\varphi = \{-\frac{\pi}{10}, -\frac{\pi}{15}, -\frac{\pi}{30}, 0, \frac{\pi}{15}, \frac{\pi}{30}, \frac{\pi}{10}\}$. The computation time of this case is usually between 40 to 90 seconds, and the one I record here takes 90.21s. Generally speaking, comparing to the case with median number of primitives, this case improves the trajectory to some extent, but the improvement is not as great as that of the case of Figure 4. Similar to the conclusion we get in the case of figure 4, as primitives number increases, RRT search tree is denser, and the curvature of path is a little bit lower. But in terms of the fluctuations of velocity and angular velocity, I think there is no apparent improvement, and the time cost of the trajectory is also quite close to the case of Figure 4. Thus from this case we know that the improvement here is quite small, and the computation time is quite likely to be longer.

6. Conclusion

I implement kinodynamic RRT algorithm on a PR2 robot in an environment that has some obstacles, the motion of the robot is subjected to a dynamic model, the input of which are acceleration and steering angle. The objective of this implementation is to study the how the number of motion primitives influences the performance of kinodynamic RRT algorithm. The way I change the number of motion primitives is changing the number of inputs. I choose 3 input sets to implement, and get corresponding RRT search trees, paths, velocity and angular velocity graphs. The result shows that in case of least number of motion primitives, computation time is likely to be the lowest among three cases, but the resulting trajectory is the worse with the appearance of many curves with a large curvature, the velocity components $\dot{x}$, $\dot{y}$ and angular velocity fluctuate largely and frequently, and the time cost of trajectory is likely to be the highest. The results of the case of median number of primitives improved the trajectory significantly. The path is smoother and has fewer curves and lower curvature, indicating that the path is easier to get through in reality. Besides, the frequency and magnitude of fluctuation in velocity components $\dot{x}$, $\dot{y}$ and angular velocity decreases clearly, and the time cost of trajectory tends to be shorter. But the computation time is likely to be longer. The case with largest number of primitives can only improves the performance of the median case a little bit, but the computation time needed is longer. Thus in conclusion, increasing number of primitives can improve the performance of kinodynamic algorithms at a price of more likely to have a longer computation time, and the improvement is significant when initial number of primitives is small. The improvement becomes less apparent when primitives number is larger.