

Sampling-based Kinodynamic Motion Planning Using a Neural Network Controller

Dongliang Zheng^{*} and Panagiotis Tsiotras[†]
Georgia Institute of Technology, Atlanta, GA, 30332-0150

This paper proposes a sampling-based kinodynamic motion planning method with several learning components. Kinodynamic RRT* for general nonlinear systems is computational prohibitive because many two-point boundary value problems (TPBVPs) are required to be solved repeatedly. This paper exploits supervised learning and offline optimal trajectories to build a state feedback neural network controller. The neural network controller aims to steer the system from points in an initial state set to points in a goal state set, thus avoiding the need for online solving TPBVPs. We develop a learning-based kinodynamic RRT* algorithm that is explicitly designed to incorporate the learned Neural Network controller and a cost-to-go Neural Network. Finally, Multiple simulation cases are studied to evaluate the performance of the proposed method.

I. Introduction

Sampling-based motion planners, such as RRT*, are well suited for planning in obstacle-rich, high dimensional complex environments [1]. RRT* is probabilistic complete and almost surely converges asymptotically to the optimal solution. Many extensions and applications have been studied in recent years [2–5]. One limitation of RRT* is the requirement that any two points sampled in the sampling space can be connected with an optimal trajectory. For robot systems with differential constraints, this essentially requires the solution of the two-point boundary value problem (TPBVP), which is non-trivial to solve in general. Thus, many works on RRT* assume a holonomic model and connect sampled points with straight lines [6].

For motion planning of robot systems, considerations of the Kinodynamic (differential) constraints are necessary for generating safe trajectories that the robot can execute. In recent years, many works have studied the kinodynamic RRT* problem where the differential constraint of the robot is considered. The sufficient conditions that ensure asymptotic optimality of the RRT* for systems with differential constraints were established in [7]. Because every *local steering* and *distance function* requires solving a TPBVP [7], it is known that kinodynamic RRT* is computationally expensive. The closed-form solution of the TPBVP is known only for some linear systems [8], several simple nonlinear systems [7], and for some specific cost function. Infinite-horizon LQR controller is used to connect pairs of states for linearized systems in [9]. Since this controller will not reach the final state exactly or within a guaranteed neighborhood, it results in suboptimality. Reference [8] uses a fixed-final-state-free-final-time controller to achieve the exact connection of any pair of states for linear or linearized systems. Closed-form solution is not available for general linear systems. In addition, the final time is computed by solving a root finding problems. The computation of their method is still expensive. Reference [10] directly uses a numerical solver to solve the TPBVP online, thus their method can be used for nonlinear systems. However, numerically solving the optimal control problem of general nonlinear systems faces the challenge of stability and convergence. It can be sensitive to initial guesses and the computation is demanding. Because of the challenge to develop appropriate steering functions for nonlinear system in RRT*, [5] proposes a method that does not require steering function by using randomly sampling piece-wise constant control inputs of the system. The method is asymptotic optimal. However the convergence to high quality trajectories in practice can be slow by the use of random controls [10, 11].

Learning methods have been explored to replace the computationally heavy parts of kinodynamic RRT*. Reference [12] uses indirect optimal control to generate optimal trajectories. Authors use the pre-computed trajectories as training data to train a neural network. The neural network outputs a trajectory when given a boundary condition. After the neural network is trained, it is used as the steering function in the RRT algorithm. However, no detail of the accuracy of the predicted trajectories from their neural network is given. Their method is tested on an inverted pendulum system.

^{*}Graduate Student, School of Aerospace Engineering.

[†]David and Andrew Lewis Chair Professor, School of Aerospace Engineering and Institute for Robotics and Intelligent Machines; AIAA Fellow.

The results show large errors of the predicted trajectories. In this paper, instead of predicting the whole open-loop trajectory using neural network, we propose a state feedback neural network controller. We show that the neural network controller generates accurate trajectories. Also, instead of considering the RRT algorithm, we propose a learning-based kinodynamic RRT* algorithm. Using Deep Reinforcement Learning (DRL) to learn a local policy for kinodynamic RRT* is studied in [13]. They also use supervised learning to train a time-to-reach estimator and use it as the distance function of RRT*. Their policy require realtime sensor feedback, laser sensor in this case, to generate control action, thus geometric information of the environment is required at the planning stage. Using deep reinforcement learning for local steering requires a lot of training data. Because the DRL method is trained using training data of a specific environment, it is hard to generalize the trained local planner to different environments.

In this paper, we use a state feedback neural network controller to solve the local steering problem in kinodynamic RRT*. We propose a near-optimal controller synthesis using supervised learning. We train a neural network using state-action data from pre-computed optimal trajectories. The neural network is used for online control and act as the steering function in RRT*. The idea is to generate many optimal trajectories offline using (computationally expensive) numerical solvers, and then exploit the data for online optimal control. Several papers have adopted this idea to learn/approximate the optimal controller for different nonlinear systems [14–17]. Satisfactory numerical results are obtained in these papers, which demonstrate its potential. Compare to [14–17], a goal-conditioned method is introduced in this paper, which enables the steering of nonlinear systems from any initial state in an initial set to any state in a goal set. The goal-conditioned extension is necessary for combining the neural network controller with RRT*. One benefit of using data-driven learning methods is their real-time implementation potential. Once trained, the neural network makes predictions efficiently.

The contributions of this paper are: First, a goal-conditioned state-feedback near-optimal neural network controller is proposed, which enables the steering of a dynamical system from any state in an initial state set to any state in the goal state set. We modify the training data by augmenting every data point along the trajectory with the goal state of that trajectory, which results in a goal-conditioned method. The neural network is trained using state-action pairs. Given the current state, the neural network predicts the corresponding optimal control, which results in a state feedback controller. Second, a learning-based kinodynamic RRT* algorithm is developed. The RRT* algorithm is modified to explicitly incorporate the Neural Network Controller (NNC) and the cost-to-go Neural Network. We introduce a *Shink* procedure in the RRT* algorithm to explicitly require that the neural networks is used for interpolation instead of extrapolation. The performance of the NNC is evaluated using several examples, Finally, simulation results of the learning-based kinodynamic RRT* are given.

The paper is organized as follows. In Section II, we define the problem considered in this paper. The details of the neural network controller is given in Section III. The kinodynamic RRT* algorithm is introduced in Section IV. Simulations results of both the neural network controller and the learning-based kinodynamic RRT* are given in Section V. Finally, some conclusions are drawn in Section VI.

II. Problem formulation

The optimal kinodynamic motion planning problem is given below. Given the domain X , free space X_{free} , initial state x_0 , goal region X_{goal} , and system dynamics

$$\dot{x}(t) = f(x(t), u(t), t), \quad x(0) = x_0, \quad (1)$$

where f is a smooth function of its variables, find a control $u(t)$, $t \in [0, T]$ for some positive T , such that the unique solution $x(t)$ to (1) is obstacle-free, i.e. $x(t) \in X_{free}$ for all $t \in [0, T]$, reaches the goal region, i.e. $x(T) \in X_{goal}$, and minimizes the following cost functional

$$J(x, u, t) = C(x(T), T) + \int_0^T L(x(\tau), u(\tau), \tau) d\tau. \quad (2)$$

The optimal kinodynamic motion planning problem is an optimal control problem with nonconvex state constraints and possibly in high dimensional state space. Sample-based methods try to solve this problem by randomly sample intermediate states (waypoints), make optimal connections between states, and converge to the optimal solution asymptotically.

In kinodynamic RRT*, every edge connection is a TPBVP and is given as

$$u^* = \arg \min_u J(x, u, t), \quad (3)$$

$$\begin{aligned} \text{s.t. } \quad & \dot{x}(t) = f(x(t), u(t), t), \\ & x(t_0) = x_a, x(t_f) = x_b, \end{aligned} \quad (4)$$

where J is defined in (2) with time interval $[t_0, t_f]$. t_f is left free which leads to a free-final-time optimal control problem. Note that the TPBVP do not need to consider the nonconvex state constraints (obstacles).

In this paper, we develop a Neural Network Controller (NNC) to replace the computation demanding steering function (TPBVP), train a cost-to-go network to function as the distance metric of RRT*, and propose a learning-based kinodynamic RRT* algorithm to approximately solve the kinodynamic motion planning problem with reduced online computation.

III. Neural Network controller

We first generate the training data. For a dynamical system, we use numerical optimization solvers [18] to solve the free final time TPBVP, given by (3) and (4), offline. Since we are interested in steering the system from different initial states to different goal states. We generate optimal trajectories with different initial states and different goal states, see Figure 1. Each trajectory returned by the numerical solver contains a sequence of control inputs and a sequence of states indexed by time. In previous related works [14–17], all the offline trajectories have the same goal state. Thus, it is expected that the neural network trained based on this data will generate control actions that steer the system to the same goal state. If the goal state changes, one has to regenerate the trajectories correspond to the new goal state and train a different neural network.

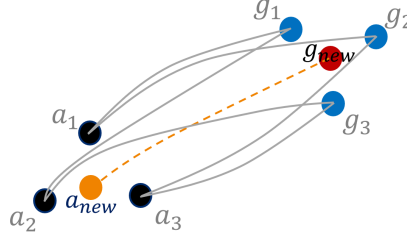


Fig. 1 Illustration of the controller synthesis procedure. Gray lines are the offline trajectories generated using a numerical solver. Black dots and blue dots are the initial states and goal states of the offline trajectories respectively. After the neural network is trained using the offline data, it is used for online feedback control. Given a new initial state (orange dot) and a new goal (red dot), the neural network controller predicts control actions that are used to propagate the system. This results in the orange dash line.

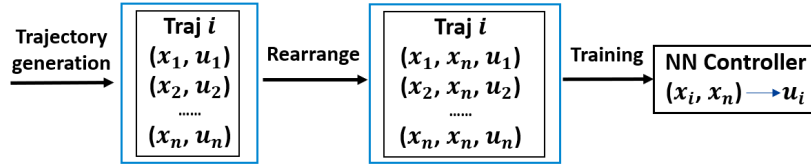


Fig. 2 Goal-conditioned neural network controller. For each offline trajectory, the final state x_n of the trajectory is augmented to every data point (x_i, u_i) of that trajectory. The neural network is trained using the augmented data. The trained neural network is a state feedback controller that maps the augmented state to the corresponding control input.

We introduce a goal-conditioned method by rearranging the training data. Specifically, for each offline trajectory, the final state of the trajectory is augmented to every data point (state-action pair) of that trajectory. For example, for any trajectory given by $\{(x_1, u_1), (x_2, u_2), \dots, (x_n, u_n)\}$ where the state and control pairs are indexed by time. x_n is the endpoint of the trajectory which is also the goal state of the trajectory. We modify the data to get the sequence $\{(x_1, x_n, u_1), (x_2, x_n, u_2), \dots, (x_n, x_n, u_n)\}$. This procedure is illustrated in Figure 2. For each trajectory, the goal state is

explicitly added as another feature to every data point of that trajectory, which enables the neural network to generate controls for different goal states. After modifying the data, we combine all the data points from all offline trajectories to form the final training dataset. Each data point is a tuple of three vectors (x_i, x_n, u_i) . The goal of the neural network is to mimic the structure of the optimal controller. The neural network controller is the mapping

$$NNController : (x_i, x_n) \rightarrow u_i,$$

where x_i is the current state. The Neural network controller predicts the control u_i based on the x_i and the goal state x_f . The idea of the goal-conditioned method has been used in [19] for path planning problems. To the best of our knowledge, this is the first time it is used to obtain a state-feedback neural network controller.

After training the neural network controller, the controller is used for online control and online trajectory generation. Provided a new boundary condition, we repetitively apply the neural network controls and simulate the system to obtain the trajectory that solves the boundary value problem. Since the final time of the trajectories is left free, we also train a time-to-reach neural network to predict the time duration that we want to simulate the system. The time-to-reach neural network is the mapping

$$NNTimeToReach : (x_i, x_n) \rightarrow t_i,$$

where t_i is the time required to go from x_i to x_f .

Finally, we train a cost-to-go neural network that functions as a distance metric in RRT*

$$NNCost : (x_1, x_n) \rightarrow J,$$

where x_1 is the initial state, J is the cost of the trajectory corresponds to (2) returned by the numerical solver [18]. In the next section, we present the kinodynamic RRT* algorithm that relies on these learned components.

IV. Learning-based Kinodynamic RRT*

The primitive procedures are summarized next before providing the main algorithms.

Sample : The sampling procedure *Sample* returns a state that is randomly sampled in free space X_{free} .

Nearest Neighbor : Given a tree $G = (V, E)$, where V is the set of nodes in the tree and E is the edge set, the function *Nearest*(V, x) returns a vertex in V that is closest to the state x in terms of Euclidean distance.

Near Neighbors : The function *Near*(V, x) returns all the vertices in V that are in the Euclidean hyper-ball of radius δ centered at x .

We use Euclidean distance to find the nearest neighbor and near neighbors, which is not the true distance. For the motion planning defined in Section II, the true distance between two states is the minimum cost J obtained by solving the TPBVP in (3) and (4) [7]. Based on the true distance, it is possible to define an ϵ -radius sub-Riemannian ball centered at x , which is the set of all states where the true distance from x to these states are less than or equal to ϵ . It is showed in [5] that there always exists a certain size Euclidean hyper-ball inside such sub-Riemannian ball under mild conditions, thus justifies the use of Euclidean norms. Euclidean distance is also used in [5]. Note that after the nearest and near vertices (in terms of Euclidean distance) are selected, we use *NNCost* as the distance metric. *NNCost* introduced in Section III is an approximation of the true distance. The Euclidean distance is used only to pre-select relevant vertices and to benefit computation.

Collision Check : Function *ObstacleFreePoint*(x) returns true if and only if the point x is in obstacle free space. Function *ObstacleFree*(τ) returns true if and only if the trajectory τ lies entirely in obstacle free space.

Parent : The function *parent*(x) returns the parent vertex of x .

Cost : The function *Cost*(x) returns the cost-to-go from the root vertex to x in the current tree G .

The complete algorithm is given by Algorithm 1, Algorithm 2, and Algorithm 3. The following procedures are specific to the proposed learning-based kinodynamic RRT* algorithm.

Shrink : The *Shrink* procedure is to ensure that the boundary condition (given by $x_{nearest}$ and x_{rand} in line 5, Algorithm 1) is within the range of the initial state set X_1 and goal state set X_2 on which the offline trajectories are generated and the neural network is trained. The boundary conditions of the offline trajectories are randomly sampled inside some bounded initial sets and bounded goal state sets.

The details of the *Shrink* procedure is shown in Figure 3. Suppose the rectangular $abcd$ space is the state space (planning space) X of the motion planning problem. To generate training data for the neural network controller, ideally, we want to sample both initial state and goal state directly on the whole space X . The boundary conditions of the offline

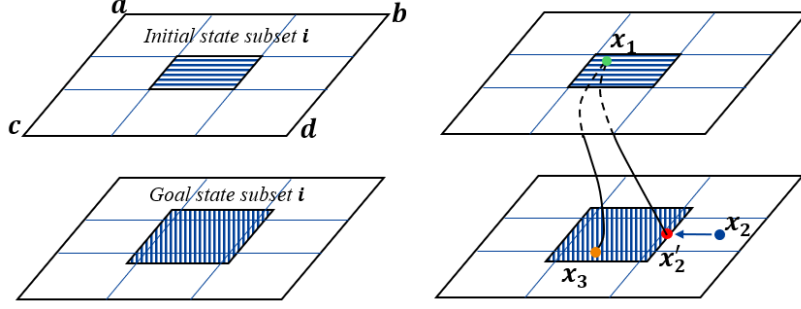


Fig. 3 *Shrink* procedure. Let the rectangular $abcd$ be the state space. Given any boundary point pair (x_1, x_2) which we want to make a connection, x_1 must belong to one of the initial state subset i . If x_2 is not in the corresponding goal state subset i , we shrink x_2 to x_2' , thus x_2' is in goal state subset i . In the figure, (x_1, x_2) is already in the range of the neural network controller and the *shrink* procedure is not needed. We say that the pairs (x_1, x_2') and (x_1, x_3) are in the range of the neural network controller, while (x_1, x_2) is not in the range of the neural network controller.

trajectories are the combination of those initial states and goal states. However, the size of the planning space X may not be known as a prior and can be varying. Even if the planning space is known and fixed, the planning space X may be very large, which requires many samples in the trajectory generation step to get a decent coverage of the state space. A larger dataset means more time required for data generation, longer training process, a more complex neural network structure, and more time required for online prediction. In our method, we divide the state space X into small subsets. The union of all subsets form the whole planning state space X and every state in the X only belong to one of the subsets. We define those subsets as the initial state subsets. For each initial state subset i , there is a corresponding goal state subset i (see Figure 3, left figure). For offline trajectory generation, the initial states are sampled from initial state subset i and the goal states are sampled from the corresponding goal state subset i . The sampling is repeated for all initial state subsets. After the neural network is trained using the offline trajectories, the RRT* algorithm uses it for control inference. Given an initial state x_1 and a goal state x_2 that we want to make connection, x_1 must belong to one of the initial state subsets i . If x_2 belong to the corresponding goal state subset i , then we say the boundary condition (x_1, x_2) is in the range of the neural network controller, and the neural network will make useful predictions. If x_2 does not belong to the goal state subset i , a *Shrink* procedure is performed (see Figure 3, right figure) such that the new state x_2' belongs to goal state subset i . The *Shrink* procedure ensures that the neural network controller is used for interpolation and avoid extrapolation.

NNSteer : Given an initial state x_1 and goal state x_2 , $NNSteer(x_1, x_2)$ repeatedly calls the feedback controller $NNController$ and forward simulate the system for a time period of $NNTimeToReach(x_1, x_2)$. $NNSteer(x_1, x_2)$ returns the trajectory τ and the endpoint of the trajectory $SegEnd$. Since the $NNSteer$ can not make an exact connection of two states, we will accept the resulting trajectory if $SegEnd$ is within a small neighborhood of goal state x_2 . This condition is given in line 7, Algorithm 2 and line 5, Algorithm 3.

NNCost : $NNCost(x_i, x_j)$ is given in Section III. It returns the cost to go from x_i to x_j .

V. Implementation Details

This section gives both the results of the neural network controller and the results of the proposed learning-based kinodynamic RRT* algorithm.

A. Results of Neural Network Controller

We evaluated the proposed method on a 2D double integrator and a planar quadrotor system. After rearranging the training data as discussed in Section III, this data is used for training. In all examples, the NNs are trained using Matlab Machine Learning Toolbox. All the implementations in this paper are done on a laptop computer with an Intel Core i5-8250U 1.6GHz CPU. The architecture of the Neural Networks is tuned for different problems. We use the Bayesian Regularization backpropagation algorithm for training in all examples.

The first example is a 2D double integrator, which has four-dimensional state space and two-dimensional control space. The offline trajectories used for training are shown in Fig. 4(a). 1264 trajectories are generated. Each trajectory contains 21 state-action pairs. After training the neural network. We test the neural network controller for randomly

Algorithm 1: Learning-based Kinodynamic RRT*

```

1  $V \leftarrow \{x_{init}\}; E \leftarrow \emptyset; \mathcal{G} \leftarrow (V, E);$ 
2 for  $i = 1 : N$  do
3    $x_{rand} \leftarrow Sample;$ 
4    $x_{nearest} \leftarrow Nearest(V, x_{rand});$ 
5   if  $x_{nearest} \times x_{rand} \notin X_1 \times X_2$  then
6      $x_{rand} \leftarrow Shrink(x_{rand});$ 
7   if  $ObstacleFreePoint(x_{rand})$  then
8      $(\tau, SegEnd) \leftarrow NNSteer(x_{nearest}, x_{rand});$ 
9     if  $ObstacleFree(\tau)$  then
10       $x_{new} \leftarrow SegEnd;$ 
11       $X_{nearby} \leftarrow Near(V, x_{new});$ 
12       $V \leftarrow V \cup \{x_{new}\};$ 
13       $x_{min} \leftarrow ChooseParent(X_{nearby}, x_{nearest}, x_{new});$ 
14       $E \leftarrow E \cup \{(x_{min}, x_{new})\};$ 
15       $E \leftarrow Rewire(X_{nearby}, E, x_{new}, x_{min});$ 
16       $\mathcal{G} \leftarrow (V, E);$ 
17 return  $\mathcal{G};$ 

```

Algorithm 2: Choose Parent Node

```

1 ChooseParent ( $X_{nearby}, x_{nearest}, x_{new}$ ):
2    $x_{min} \leftarrow x_{nearest};$ 
3    $c_{min} \leftarrow Cost(x_{nearest}) + NNCost(x_{nearest}, x_{new});$ 
4   foreach  $x_{near} \in X_{nearby} \setminus x_{nearest}$  do
5     if  $Cost(x_{near}) + NNCost(x_{near}, x_{new}) < c_{min}$  then
6        $(\tau, SegEnd) \leftarrow NNSteer(x_{near}, x_{new});$ 
7       if  $ObstacleFree(\tau) \wedge Norm(SegEnd - x_{new}) < d_1$  then
8          $x_{min} \leftarrow x_{near};$ 
9          $c_{min} \leftarrow Cost(x_{near}) + NNCost(x_{near}, x_{new});$ 
10  return  $x_{min};$ 

```

Algorithm 3: Rewire

```

1 Rewire ( $x_{nearby}, E, x_{new}, x_{min}$ ):
2   foreach  $x_{near} \in X_{nearby} \setminus x_{min}$  do
3     if  $Cost(x_{new}) + NNCost(x_{new}, x_{near}) < Cost(x_{near})$  then
4        $(\tau, SegEnd) \leftarrow NNSteer(x_{new}, x_{near});$ 
5       if  $ObstacleFree(\tau) \wedge Norm(SegEnd - x_{near}) < d_2$  then
6          $x_{parent} \leftarrow Parent(x_{near});$ 
7          $E \leftarrow E \setminus \{(x_{parent}, x_{near})\};$ 
8          $E \leftarrow E \cup \{(x_{new}, x_{near})\};$ 
9   return  $E;$ 

```

sampled novel boundary conditions. Some example trajectories obtained using the neural network controller are shown in Fig. 4(b). We test the controller with 500 new randomly sampled boundary conditions. Since we are interested in steering the system to a specific goal state, one index is the closeness between the endpoints of the trajectories and their goal states. We compute the error between the endpoint of each trajectory and its goal. 500 trajectories are generated using the neural network controller. The errors of the endpoints of those trajectories are shown in Fig 5. The Mean Squared Error (MSE) of position vector is (0.0022, 0.0019), the MSE of velocity vector is (0.0054, 0.0053).

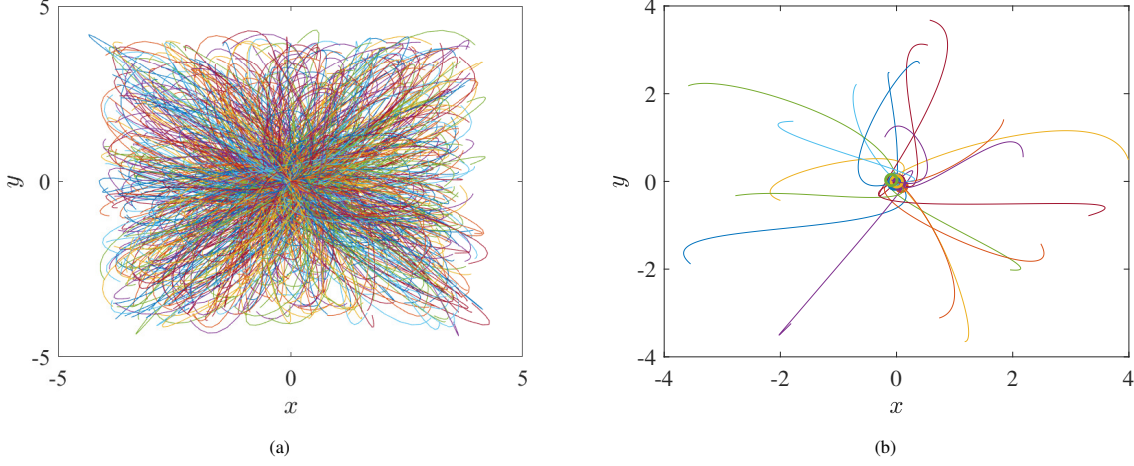


Fig. 4 Control results of a double Integrator. (a) 1264 offline trajectories. Since the double integrator is translational invariant regarding position, we can always choose the position of the goal state to be zero. Note that all trajectories go to the origin but with different final velocities. (b) example trajectories obtained using the neural network controller.

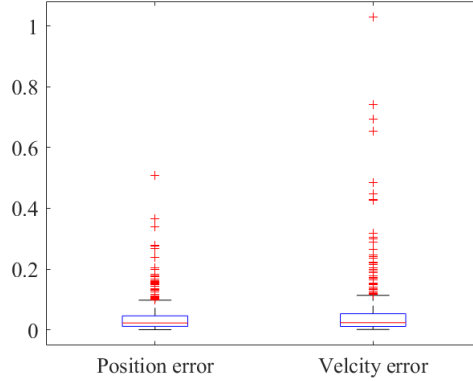


Fig. 5 Steering error of the controller for 2D double integrator. For 500 trajectories, the error between the endpoint of each trajectory and its goal state is shown. The endpoint contains a position vector and velocity vector.

The second example is a planar quadrotor [20]. Its model is shown in Fig. 6. Assume that the quadrotor is modeled by two point-masses that are connected by a rigid mass-less bar. Each point-mass is m . There are two motors, each of which applies a force orthogonal to the connecting bar. The length from each rotor to the center of the bar is d . The two forces, u_1 and u_2 from the two motors are the control inputs of the system. The planar quadrotor has three degrees of freedom, and its pose is given by the position vector (x, z) , and the rotation angle θ . The equation of motion is given by

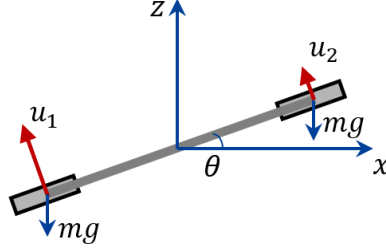


Fig. 6 Model of a planar quadrotor motion in (x, z) plane.

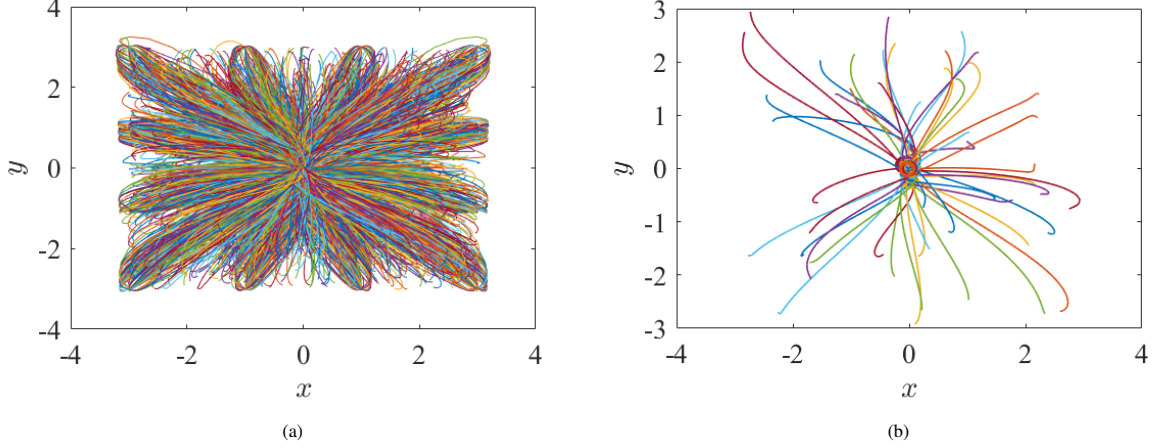


Fig. 7 Control results of a planar quadrotor. (a) 2900 offline trajectories used for training. The planar quadrotor is also translational invariant regarding position. All trajectories go to the origin with different final velocities. (b) example trajectories obtained using the neural network controller.

$$\begin{aligned}
 \ddot{x} &= -\frac{1}{m} \sin \theta (u_1 + u_2), \\
 \ddot{z} &= \frac{1}{m} (\cos \theta (u_1 + u_2) - 2mg), \\
 \dot{\theta} &= \frac{1}{2md} (u_2 - u_1).
 \end{aligned} \tag{5}$$

The pre-computed optimal trajectories used for training are shown in Fig. 7(a). The trajectories obtained using the Neural network controller are shown in Fig. 7(b).

Similar to the double integrator case, we compute the error between the endpoint of each trajectory and its goal state. The results are shown in Fig. 8. The MSE for the position vector, velocity vector, and angle are (0.0012, 0.0012), (0.0068, 0.092), and 0.0016 respectively.

B. Results of Learning-based Kinodynamic RRT*

First, we tested the learning-based kinodynamic RRT* algorithm on the 2D double integrator. The results from one run of the algorithm are given in Fig. 9. The algorithm finds the first trajectory after expanded 65 nodes, and time lapsed for finding the first trajectory is 0.51 second. The cost of the first trajectory is 105.86. We run the algorithm multiple times. The decreasing of the cost and the increasing of time as the increasing of the node expanded are given in Fig. 10.

Similarly for the planar quadrotor case, the results from one run of the algorithm are given in Fig. 11. The algorithm finds the first trajectory after expanded 95 nodes, and time lapsed for finding the first trajectory is 0.51 second. The cost of the first trajectory is 259.14. The decreasing of the cost and the increasing of time as the node expanded increasing are given in Fig. 12.

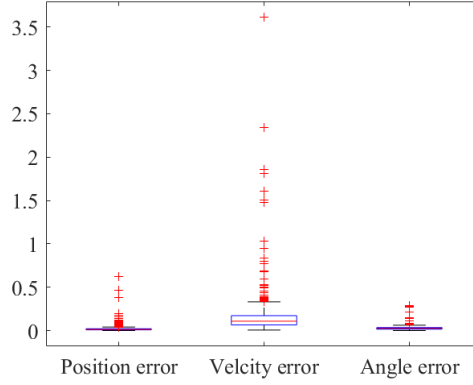


Fig. 8 Steering error of the planar quadrotor. For 500 trajectories, the position error, velocity error, and angle error of the endpoint of each trajectory is shown.

VI. Conclusion

We proposed a learning-based kinodynamic RRT* algorithm for nonlinear systems. The core of this algorithm is a near-optimal state feedback neural network controller. The neural network controller steers the system from an initial state inside a set to a goal state in another set, which is used as the steer function in RRT* algorithms. The training data generation and neural network training is done offline thus the online computation burden is reduced. The controller is suitable for real-time optimal control and trajectory generation for nonlinear systems. We developed a kinodynamic RRT* algorithm to explicitly integrate the several learned components. The proposed method is tested and preliminary results show high potential in online solving the kinodynamic motion planning problem for general nonlinear systems in obstacle cluttered complex environment.

References

- [1] Karaman, S., and Frazzoli, E., “Sampling-based algorithms for optimal motion planning,” *The International Journal of Robotics Research*, Vol. 30, No. 7, 2011, pp. 846–894.
- [2] Elbanhawi, M., and Simic, M., “Sampling-based robot motion planning: A review,” *IEEE Access*, Vol. 2, 2014, pp. 56–77.
- [3] Arslan, O., and Tsiotras, P., “Use of relaxation methods in sampling-based algorithms for optimal motion planning,” *IEEE International Conference on Robotics and Automation*, Karlsruhe, Germany, 2013, pp. 2421–2428.
- [4] Karaman, S., Walter, M. R., Perez, A., Frazzoli, E., and Teller, S., “Anytime motion planning using the RRT*,” *IEEE International Conference on Robotics and Automation*, Shanghai, China, 2011, pp. 1478–1483.
- [5] Li, Y., Littlefield, Z., and Bekris, K. E., “Asymptotically optimal sampling-based kinodynamic planning,” *The International Journal of Robotics Research*, Vol. 35, No. 5, 2016, pp. 528–564.
- [6] Gammell, J. D., Srinivasa, S. S., and Barfoot, T. D., “Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic,” *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Chicago, IL, 2014, pp. 2997–3004.
- [7] Karaman, S., and Frazzoli, E., “Optimal kinodynamic motion planning using incremental sampling-based methods,” *49th IEEE conference on decision and control*, Atlanta, GA, 2010, pp. 7681–7687.
- [8] Webb, D. J., and Van Den Berg, J., “Kinodynamic RRT*: Asymptotically optimal motion planning for robots with linear dynamics,” *IEEE International Conference on Robotics and Automation*, Karlsruhe, Germany, 2013, pp. 5054–5061.
- [9] Perez, A., Platt, R., Konidaris, G., Kaelbling, L., and Lozano-Perez, T., “LQR-RRT*: Optimal sampling-based motion planning with automatically derived extension heuristics,” *IEEE International Conference on Robotics and Automation*, Saint Paul, MN, 2012, pp. 2537–2542.

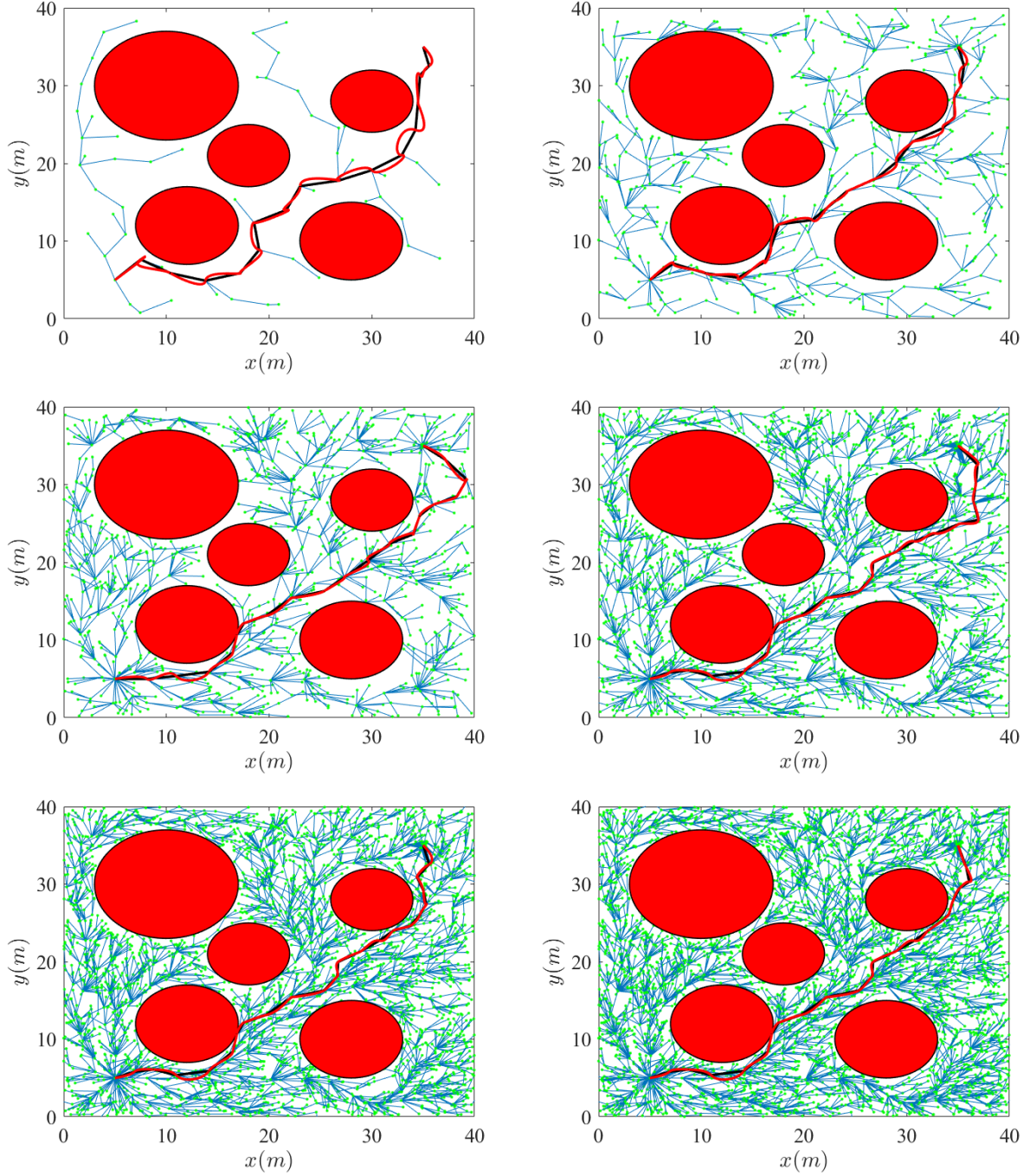


Fig. 9 Planning results of the proposed kinodynamic RRT* for the 2D double integrator. The upper left figure corresponds to the first solution found. From upper left to down right, the node expanded are 65, 500, 1000, 2000, 3000, 4000 respectively, the cost of there trajectories are 105.86, 99.97, 86.31, 78.04, 76.12, 75.39 respectively. Finally, the time for the algorithm to find those trajectories are 0.51, 7.76, 25.06, 49.73, 67.23, 87.44 seconds respectively.

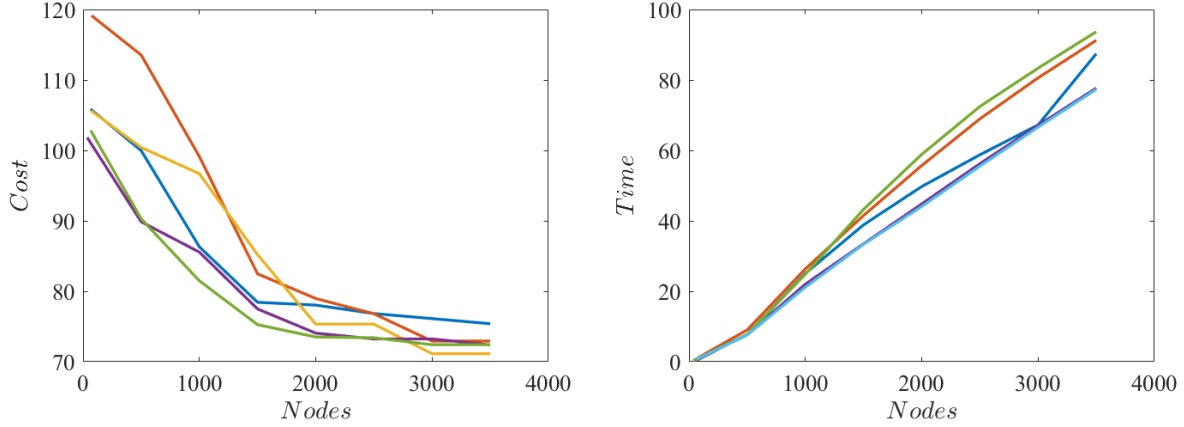


Fig. 10 Result of the algorithm for double integrator case. The lines with different colors are correspond to 5 run of the algorithm. Left figure shows the decreasing of the cost as the nodes in the tree increasing. Right figure shows the increasing in time required as the increasing of nodes.

- [10] Xie, C., van den Berg, J., Patil, S., and Abbeel, P., “Toward asymptotically optimal motion planning for kinodynamic systems using a two-point boundary value problem solver,” *IEEE International Conference on Robotics and Automation*, Seattle, WA, 2015, pp. 4187–4194.
- [11] Sivaramakrishnan, A., Littlefield, Z., and Bekris, K. E., “Towards Learning Efficient Maneuver Sets for Kinodynamic Motion Planning,” *arXiv preprint, arXiv:1907.07876*, 2019.
- [12] Wolfslag, W. J., Bharatheesha, M., Moerland, T. M., and Wisse, M., “RRT-CoLearn: towards kinodynamic planning without numerical trajectory optimization,” *IEEE Robotics and Automation Letters*, Vol. 3, No. 3, 2018, pp. 1655–1662.
- [13] Chiang, H. T. L., Hsu, J., Fiser, M., Tapia, L., and Faust, A., “RL-RRT: Kinodynamic motion planning via learning reachability estimators from RL policies,” *IEEE Robotics and Automation Letters*, Vol. 4, No. 4, 2019, pp. 4298–4305.
- [14] Ghosh, P., and Conway, B., “Near-optimal feedback strategies for optimal control and pursuit-evasion games: a spatial statistical approach,” *AIAA/AAS Astrodynamics Specialist Conference*, Minneapolis, MN, August 2012, pp. 4590–4614.
- [15] Tsiotras, P., and Diaz, R. S., “Real-time near-optimal feedback control of aggressive vehicle maneuvers,” *Optimization and Optimal Control in Automotive Systems*, edited by W. H., K. I., S. M., and del Re L., Springer, 2014, Chap. 2, pp. 109–129.
- [16] Sanchez-Sanchez, C., and Izzo, D., “Real-time optimal control via Deep Neural Networks: study on landing problems,” *Journal of Guidance, Control, and Dynamics*, Vol. 41, No. 5, 2018, pp. 1122–1135.
- [17] Taylor, D., and Izzo, D., “Learning the optimal state-feedback via supervised imitation learning,” *Astrodynamics*, Vol. 3, No. 4, 2019, pp. 361–374.
- [18] Patterson, M. A., and Rao, A. V., “GPOPS-II: A MATLAB software for solving multiple-phase optimal control problems using hp-adaptive Gaussian quadrature collocation methods and sparse nonlinear programming,” *ACM Transactions on Mathematical Software*, Vol. 41, No. 1, 2014, pp. 1–41.
- [19] Bency, M. J., Qureshi, A. H., and Yip, M. C., “Neural Path Planning: Fixed Time, Near-Optimal Path Generation via Oracle Imitation,” *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Macau, China, 2019, pp. 3965–3972.
- [20] Ritz, R., Hehn, M., Lupashin, S., and D’Andrea, R., “Quadcopter performance benchmarking using optimal control,” *IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Francisco, CA, 2011, pp. 5179–5186.

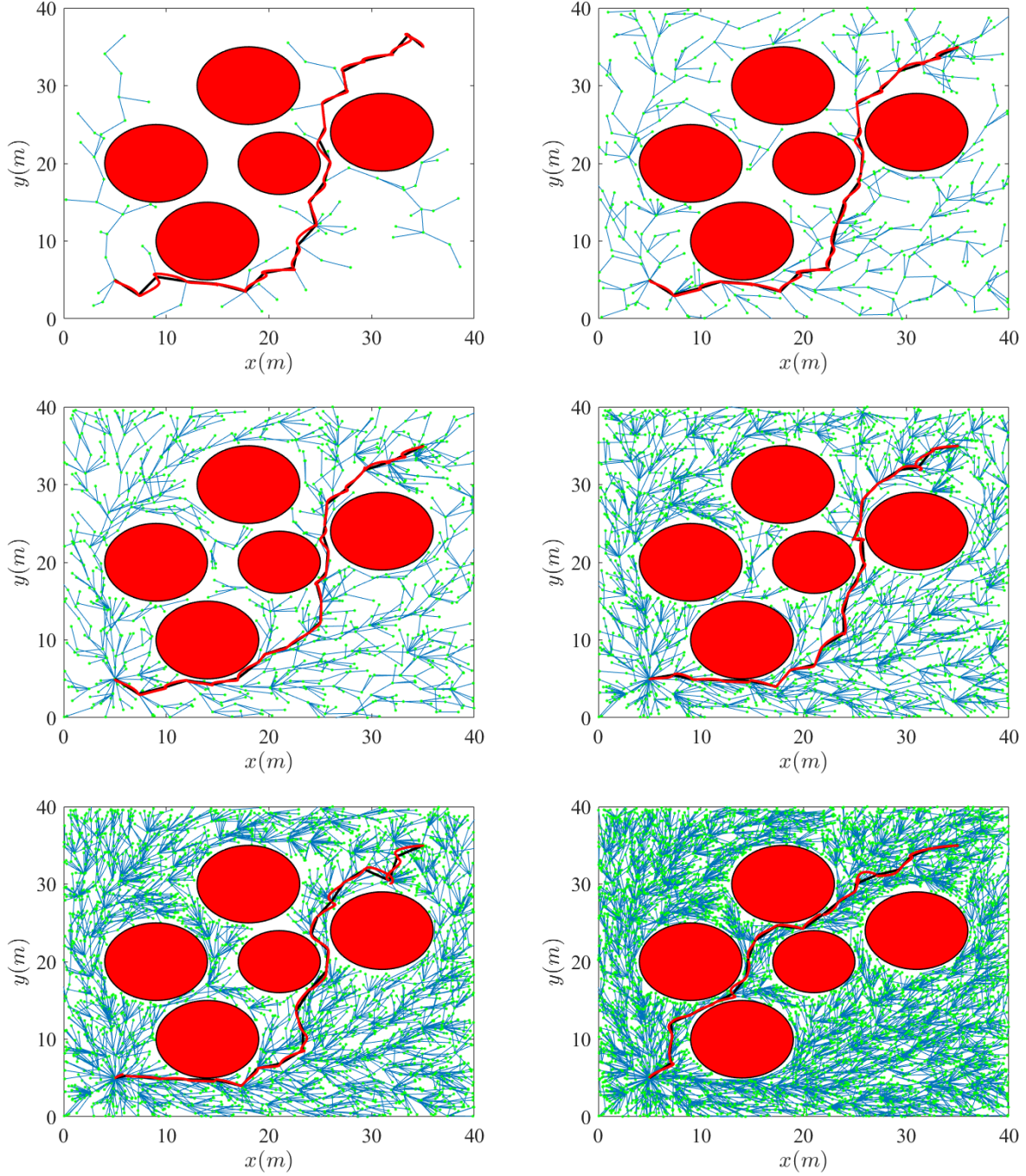


Fig. 11 Planning results of the proposed kinodynamic RRT* for the planar quadrotor. The upper left figure corresponds to the first solution found. From upper left to down right, the node expanded are 95, 500, 1000, 2000, 3000, 6000 respectively, the cost of these trajectories are 259.14, 235.56, 233.42, 216.83, 200.29, 175.61 respectively, the time for the algorithm to find those trajectories are 0.51, 4.77, 13.72, 47.33, 96.03, 229.46 seconds respectively.

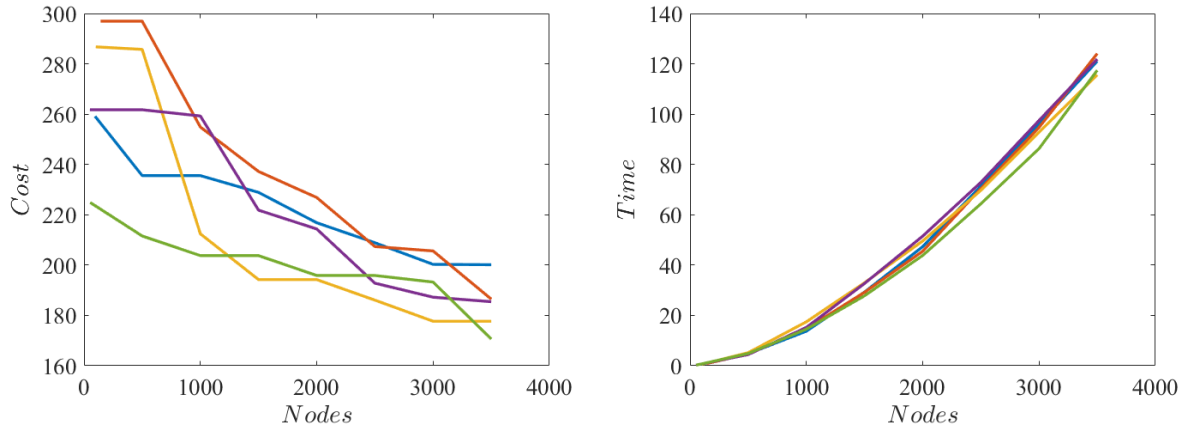


Fig. 12 Result of the algorithm for planar quadrotor case. The lines with different colors are correspond to 5 run of the algorithm. Left figure shows the decreasing of the cost as the nodes in the tree increasing. Right figure shows the increasing in time as the nodes in the tree increasing.