# Data-Driven Kinodynamic RRT

Junghwan Lee[1] Heechan Shin[1] Sung-eui Yoon[2]

*Abstract*— **We present a novel, data-driven kinodynamic motion planner. Our sampling-based planner is based on using a physics simulator as a black box to compute a trajectory considering dynamics, even when we cannot derive exact propagation functions. To improve its overall efficiency, we pre-compute a motion database containing different motions simulated with different controls and states defined in the local frame of a robot. We then use the motion database to efficiently estimate the simulated trajectory during iterations of our planner. When the planner requests the best control to reach a desired state from a query state, we retrieve nearby motions that are close to the query state and pick the motion that is closest to the desired state for the tree extension. To control accuracy of our planner with a high efficiency, we lazily validate retrieved motions. The pre-constructed motion database contains modular trajectories and thus can be reused for other test cases, where we have different composition of obstacles or different start/goal states.**

## I. Introduction

The robot motion planning problem has been actively researched and applied to various fields including robotics, computer graphics, biology, and so on [1], [2]. Among many prior techniques, sampling-based motion planning algorithms have been widely used to solve a variety of problems in high dimensional spaces, thanks to its effectiveness with the probabilistically completeness.

Planning for real world robots that move and interact with environments under the physical laws requires another class of constraints, known as kinodynamic constraints [3]. These kinodynamic constraints include non-holonomic constraints for underactuated systems and differential constraints for dynamical systems. Kinodynamic motion planning can provide valuable solutions to a wide variety of applications such as space robots, mobile robots with wheels, etc.

There are two main approaches in the robotics community for solving the kinodynamic planning problem. The first approach decouples the problem into two phases by computing a basic path in the robot's configuration space and then adjusting it with proper controls such that the modified path satisfies the dynamics and other constraints [4], [5], [6]. Unfortunately, it is possible in this approach that the computed path in the first step cannot be adjusted in a way that we can track it under kinodynamic constraints in the second step.

The second approach is a generalization of planning in the configuration space by employing the state space that includes both configuration space and its velocity components. Based on this simple generalization, many sampling-based motion planning techniques can be used for the kinodynamic problems [7], [8] with a suitable metric and a propagation function under dynamics systems, and resultant planners become probabilistically complete. A proper propagation function that is a set of motion equations should be derived and integrated for these approaches. Unfortunately, the explicit derivation of the propagation function can be sometimes impossible or approximated for complex systems with dynamics [9].

Recently, a sampling-based planner with a physics-based simulator [10] is proposed to solve the kinodynamic planning problem with complex dynamics by using the simulator, i.e., numerical integrations with dynamic systems, instead of integration of motion equations. Many other kinodynamic planners spend a high portion on integrating motion equations in practice. Similarly, the planner using the simulator takes a high portion of its overall running time (about 85% in our tested benchmarks). We observe that many similar or even redundant simulations occur during the overall RRT process, when objects in the environment have the same dynamics properties. Eliminating these redundant simulations by precomputing several simulations, and utilizing them can improve the overall efficiency of the planner.

**Main contributions.** In this paper, we focus on sampling based kinodynamic motion planning methods, particularly, kinodynamic RRT planner, that uses simulation as integration of propagation functions, since it can handle a wide variety of dynamical systems, even where we cannot define exact propagation functions. In this context, we propose a novel, data-driven kinodynamic motion planner for such environments. Our method pre-computes a motion database containing various motions generated from different local states (Sec. IV-A). We then use the motion database to estimate simulations and efficiently perform the integration of propagation functions. We propose an efficient tree extension method using the motion database (Sec. IV-B). Our motion database has a finite number of entries and cannot contain all the requested motions during the planning iterations. To avoid any inaccuracy on the solution path, we propose lazy validation methods to efficiently update the retrieved motions from the database, whiling achieving high efficiency (Sec. IV-C). In our tested benchmarks, our planner DDK-RRT achieves up to 2.43 times runtime performance improvement over the kinodynamic RRT planner using the simulator.

[1]Junghwan Lee and Heechan Shin are at Dept. of Computer science, KAIST, Daejeon, South Korea [2]Sung-eui Yoon is at Dept. of Computer science, KAIST, Daejeon, South Korea goolbee@gmail.com, shn4438@gmail.com, sungeui@gmail.com

## II. RELATED WORK

Among the sampling-based algorithms, Probabilistic Roadmap Method (PRM) and Rapidly-exploring Random Tree (RRT) are the most widely used techniques [11]. Particularly, RRT has been applied to various single query problems, and many variations for higher performance have been developed in different directions [12], [13].

The classic approach for kinodynamic motion planning in robotics is to decouple the problem as two stages: firstly compute an initial path by solving a basic path planning problem with a geometric planner, and then compute a global solution near the initial path, while satisfying the dynamics and other constraints [14], [15]. When intractable paths are generated from the first stage, this approach, unfortunately, can result in a susceptibility to local minima, because of limits on forces and torques for the robot in the environments.

In order to cope with the local minima problem, a number of approaches have been proposed. Svestka and Overmars [16] proposed an extension of randomized holonomic planning technique to the non-holonomic planning assuming a proper steering method for a system, and LaValle and Kuffner [7] extended a sampling-based method with the configuration space into the state space. Randomized kinodynamic planning has been extended to improve the performance [17], [18], [19], [10]. Since the dimensionality of the state space is increased two times from that of its configuration space, the complexity of sampling based kinodynamic planners is higher than that of holonomic planning. Furthermore, selecting a suitable metric and designing effective extension methods still remains to be challenging.

Additionally, directly considering physical constraints and effectively generating samples under them have been also studied [20]. Recently, propagation types used for kinodynamic planners and their properties have been investigated [21], [22].

## III. BACKGROUND

In this section, we give our problem definition followed by the kinodynamic RRT planner. We then motivate our approach.

### A. Problem Definition

In the kinodynamic planning problem, the state space is used for the same purpose as the configuration space for the holonomic planning problem, to represent geometric and differential states of a robot. The state space for the kinodynamic problem is defined as a set of a state $x = (q, \dot{q})$, where $q$ denotes a configuration. Differential or non-holonomic constraints can be described by a forward propagation function $f : X \times U \rightarrow \dot{X}$, where $X$ and $U$ represent the state and control spaces, respectively. A solution path of the planning problem consists of a sequence of controls, its time durations, and states that can be obtained by sequentially integrating the propagation function $f$.

In order to integrate the propagation function, a set of motion equations need to be defined. Additionally, the dynamics in-between environments and a robot (e.g., friction, gravity), and the physical property (e.g., torque limits) of the robot should be derived. For complicated problems considering many dynamic properties, the propagation function, unfortunately, is hard to derive explicitly [9], because of the complexity of dynamics and various physical properties of the robot. In some cases, the function is provided in a simplified form, but can result in inaccuracy. In order to provide more generality and higher accuracy, a realistic physics-based simulation engine, which works as a black box to the motion planner, is used for generating motions instead of integrating the propagation function [10]. A downside of this approach is that the physics-based simulation is more computationally expensive than the integration of propagation function.

Many physics-based simulation libraries (e.g., ODE, bullet, and Nvidia PhysX) are available. These libraries support rigid and even deformable objects, and rely on various numerical integration of differential equations. In physics-based simulators, a simulation time is discretized and the propagation function is evaluated only in this discretized simulation interval (e.g., 1 ms). This discretization interval, called a simulation step, normally does not change during the whole simulation. Simulating for a specific period (e.g., 1 s) is achieved by iteratively executing simulation with the simulation step. The simulation step has a trade-off between the quality of solution and the running time of the simulation; larger simulation steps get simulation results faster, but with lower accuracy. We simply use the default value suggested by the simulator community.

### B. Kinodynamic RRT planner

Kinodynamic RRT planners have been derived from RRT designed for holonomic planning [7]. Given a random tree initialized from the start state, the kinodynamic RRT planner incrementally expands the tree by randomly generating samples in the state space and attempting to connect nodes of the tree to those random samples. Specifically, given a randomly generated sample, $x_r$, in the state space, we identify the nearest neighbor, $x_n$, in the tree from the random state $x_r$. Among a set of possible controls, the best control, $u_{best}$, is then selected as the one that pulls the node $x_n$ most closely toward the randomly chosen state $x_r$.

In order to determine such a control, integration of the propagation function is required. In this work, the propagation of motion is computed by running the physics-based simulator as we discussed earlier (Sec. III-A), instead of deriving a set of motion equations. We then determine the best control $u_{best}$ after trying all available controls and choosing the one that extends the state $x_n$ as close as possible to the $x_r$. In practice, searching the best control $u_{best}$ is achieved iteratively by randomly generating $n$ different controls and

**Algorithm 1** *n*-control Kinodynamic RRT

```
Require: tree $T$, $n$
  $T$.AddVertex ($x_{init}$)
  repeat
    $x_r \leftarrow$ RandomStateInStateSpace()
    $x_n \leftarrow$ NearestNeighbor($x_r$, $T$)
    $u_{best} \leftarrow$ FindBestControl($x_n$, $x_r$, $n$)
    $x_{new} \leftarrow$ Extend($x_n$, $u_{best}$)
    if CollisionFree($x_{new}$, $x_n$) then
      $T$.AddVertex($x_{new}$)
      $T$.AddEdge($x_n$, $x_{new}$, $u_{best}$)
    end if
  until a collision-free path between $x_{init}$ and $x_{goal}$ is found
```

choosing the best one. Finally, we generate a new state, $x_{new}$, by taking $u_{best}$ from $x_n$, and add it with its control $u_{best}$ to the tree if there is no collision in the trajectory between $x_n$ and $x_{new}$. We call this algorithm a *n*-control kinodynamic RRT, where *n* denotes the number of trials to find the best control. Its pseudocode is shown in Algorithm 1. The kinodynamic RRT is a probabilistically complete algorithm [7].

### C. Motivation

The running time of each iteration of the kinodynamic RRT can be decomposed into propagation process, nearest neighbor search, and other parts including collision detection, tree expanding, etc. When a simulator is used for the propagation process, executing the simulator takes the highest portion of the running time, while we can handle a wider set of robots and dynamics. In our tested benchmarks, running the simulation takes about 87% (Table I). Note that a similar or higher portion (e.g., 90%) was reported for performing forward propagation within tree-based planners [10] that do not use simulation.

During the kinodynamic RRT process running a simulator, we observe that many similar simulations occur, when environments consist of objects that have the same dynamics property such as the friction constant. Furthermore, some of simulations turn out to be exactly identical after performing a proper transformation.

For example, when a mobile robot moves around on a large floor that has the same, isotropic material properties, trajectories of the robot in this environment are same in a local frame defined by the robot's orientation, while their counterparts in the global frame, i.e., workspace, are different. Fig. 1 shows an example of two different trajectories of two stationary mobile robots, $x_1$ and $x_2$, applied by the same control $u$ that executes the same force with each robot's heading orientation. Their states $x_1$ and $x_2$ are different in the state space; their global positions and orientations are different. When the floor has the isotropic physical property, the results of those two different simulations will be identical in the robot's local frame. As a result, a single simulation

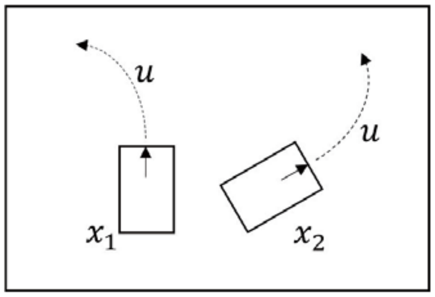

Fig. 1. Two stationary mobile robots in a floor with isotropic physical properties (e.g, the same isotropic friction). Even when the same control is applied to both states, their states in the state space are different, but the same simulation result can be utilized to both robots after a proper transformation.

result can be used for both states by transforming results from one robot's local frame to another one, instead of performing simulations repeatedly.

Eliminating these duplicated simulations can improve the overall efficiency of the planner. Furthermore, precomputing simulation results in known environments in a database and utilizing them can improve the runtime performance, and the database also can be used for other problems in the same environments with different start/goal positions or with different composition of obstacles.

## IV. Algorithm

In this section, we explain our kinodynamic sampling-based planner utilizing a precomputed motion database. Our planner is based on the RRT-based kinodynamic planner [7]. We first show the structure of the proposed motion database and how to construct it. We then explain our planning algorithm with the motion database and how to efficiently validate generated paths from the database during planning. Finally, we give an overall planning algorithm.

### A. Building a motion database

We store various motions of a robot in the motion database, where a robot motion is defined as a trajectory of the robot given a start state and a control. In order to construct such a motion database, we first define *the local state space* of a robot to be a state space in the local frame of the robot. The local state space contains various components described in the robot's local frame. They include velocities and local geometric configurations such as joint angles for an articulated robot or a steering angle of a wheeled robot.

Once the local state space is defined, we also define two conversion functions, $f_c : X \rightarrow X_{local}$ and $f_{c\_inv} : X_{local}, G \rightarrow X$, where $X$ is the global state space, $X_{local}$ denotes the local state space, and $G$ is the information (e.g., the orientation and global position of the robot) needed to position the local frame of the robot into the global space. Generally, $f_c$ and

**Algorithm 2** Data-Driven $n$-control Kinodynamic RRT

```
Require: tree $T$, $n$, and a motion database $db$
  $T$.AddVertex ($x_{init}$)
  repeat
    $x_r \leftarrow$ RandomStateInStateSpace()
    $x_n \leftarrow$ NearestNeighbor($x_r$, $T$)
    $controls \leftarrow$ RetrieveControls($db$, $x_n$)
    if True w/ $p(|controls|)$ (Eq. 1) then
      $u_{best} \leftarrow$ FindBestControlAmong($x_n$, $x_r$, $controls$)
      $x_{new} \leftarrow$ ExtendFromDatabase($x_n$, $u_{best}$)
    else
      $u_{best} \leftarrow$ FindBestControl($x_n$, $x_r$, $n$)
      $x_{new} \leftarrow$ Extend($x_n$, $u_{best}$)
    end if
    if any of two validation conditions is satisfied then
      ValidatePath ($x_{new}$)
    end if
    if CollisionFree($x_{new}$, $x_n$) then
      $T$.AddVertex($x_{new}$)
      $T$.AddEdge($x_n$, $x_{new}$, $u_{best}$)
    end if
  until a collision-free path between $x_{init}$ and $x_{goal}$ is found
```

$f_{c\_inv}$ can be easily defined by geometric transformations for robots. In general, the control space is defined in the robot's local frame. A conversion function for the control state, therefore, is not required. An example of the local state space and conversion functions we use in our experiments is explained at Sec.V-A.

In the motion database, every component of motions are defined in the local state space. Each motion entry consists of a start state, a control state, control duration, and a trajectory. Therefore, an input query consisting of a start state defined in the global state space $X$ is first converted to the local space space $X_{local}$ by applying the conversion function $f_c$. A resultant trajectory of the robot computed by performing the simulator describes a motion after applying the control to the start state of the robot. The motion trajectory is defined as a set of states from the start and goal state. When a motion retrieved from the database is applied to a global state, $x_n$, of the random tree for the tree expansion, we apply $f_{c\_inv}$ to the trajectory of the motion as the conversion process from the local to the global, and add it to $x_n$. We maintain the motion data as a lookup table indexed by a state of the local state space. One may wonder why we do not include the control in the key value of the motion database. We exclude a control value out of the key value, mainly to reduce the dimension of the database and to increase the hit ratio of identifying similar local states from the database.

In order to build the motion database, motions can be generated randomly or provided by the user. In this work, we generate motions in a random way. In other words, we randomly generate a start state and a control, and then store results of simulation for a fixed amount of time. Time duration of the control is also chosen randomly. One can build more effective motion databases by analyzing environments/robots or by domain experts to target robots/applications.

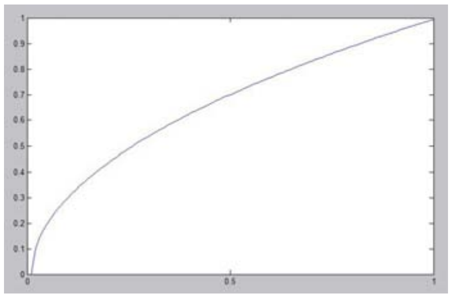

Fig. 2. An example plot of the motion utilization probability function with $\alpha = 0.5$, to decide whether the motion database is used, and the x-axis represents $n_{db}/n$.

### B. Planning with the motion database

Our planner is based on the $n$-control kinodynamic RRT planner (Sec. III-B). For each iteration, we randomly generate a sample, $x_r$, in the global state space, and find its nearest neighbor node, $x_n$, in the random tree. We then aim to choose the best control input, $u_{best}$, that yields a new node, $x_{new}$, that is the closest state to $x_r$ among different controls. We then append the new node $x_{new}$ with its control and trajectory to the random tree if there is no-collision.

In order to compute $u_{best}$, we retrieve a set of motions pre-computed with different controls from the database, instead of relying on running the simulator. Given the node $x_n$, we fetch nearby motions from the database with a query key value containing the local state of the node $x_n$. Since it is unlikely to have the motion exactly same as the query state from the database, we perform the fixed-radius near neighbor search that reports a set of states that are within a distance, $d_{sim}$, to the given query state. An example of distance metrics used for our tested benchmarks is shown at Sec. V-A. $d_{sim}$, called a displacement control parameter, determines an allowable similarity of states in the local state space. As the higher $d_{sim}$ is used, the more data are fetched, but the reliability of fetched motions becomes lower. In the following section, we explain how to validate and adjust inaccuracy caused by using such motions.

The number of retrieved motions and associated controls also varies depending on the amount of pre-computed motions in the motion database. The more motions the database has, the more number of controls are retrieved. These motions and controls are used for calculating the best control to reach $x_r$ closely. Building the database to cover the whole local state space, however, is impossible and inefficient, because

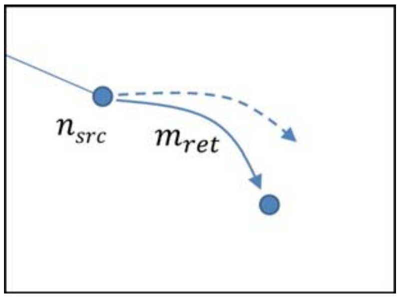

Fig. 3. This figure shows an example of a retrieval displacement between a retrieved motion (solid line) and the actual motion by running the simulation (dotted line), when a start state of $m_{ret}$ is different from a state of $n_{src}$.

the size of the local state space grows exponentially as its dimension grows. Therefore, in practice, the number of retrieved controls for a given state can be less than the required number of trials $n$ for finding the best control, or could be zero at the worst case. Searching the best control with a limited number of retrieved controls might bias the expansion of the tree in the global state space and interfere the rapid exploration of the state space, degrading the overall performance of the planner.

In order to address this issue arising with a small number of retrieved controls, we selectively use our control generation method using the motion database with a probability function, $p(n_{db})$, where $n_{db}$ represents the number of controls retrieved from the database given a query state. We define the following probability function, called motion utilization probability:

$$p(n_{db}) = (n_{db}/n)^{\alpha}, \tag{1}$$

where $n$ is the maximum number of control trials and $\alpha$ is a parameter in the range of (0,1). $\alpha$ controls a degree of utilization of the database proportional to a hit ratio of the database. When $\alpha$ value increases, the database is utilized more frequently. Fig. 2 shows a plot of the motion utilization function when $\alpha$ is 0.5. When we decide not to use our control generation method, we simply fall back to use the original technique of choosing the best control method, *FindBestControl(·)*, running the simulator (Sec. III-B).

## C. Validating retrieved motions

In this section, we explain how to control inaccuracy caused by using nearby motions retrieved from the motion database and how to validate those motions.

*1) Retrieval displacement:* A retrieved motion from the database can be incorrect, because we retrieve a motion whose source state can be different from a query state, while their distance is controlled by the parameter $d_{sim}$. Fig. 3 shows an example of such a case. Suppose that a random tree is extended from a node $n_{src}$ with a retrieved motion $m_{ret}$. When the start state of $m_{ret}$ is different from the state of $n_{src}$,

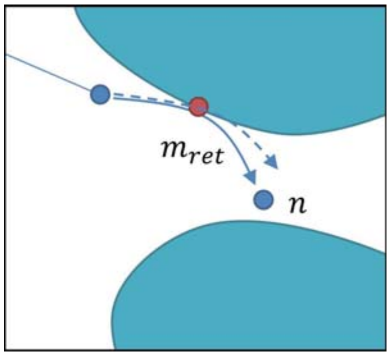

Fig. 4. This figure shows an example when a retrieved motion with slight displacement causes a collision. The retrieved motion $m_{ret}$ successfully extends the tree toward a node $n$ (solid line), its corresponding, actual motion (dotted line), however, causes a collision with obstacles. The precise trajectory to the node $n$ is computed based on our validation process.

applying the motion $m_{ret}$ to $n_{src}$ would lead to a different trajectory from the actual motion that can be generated by running the simulator to $n_{src}$. We call this displacement between the actual motion and the retrieved one *a retrieval displacement*.

Calculating the actual motions requires to perform simulation at runtime. As a result, it is infeasible to precisely measure the retrieval displacement. Therefore we approximate a retrieval displacement of a retrieved motion simply as the distance between a query state and the start state of its retrieved motion, which is bounded by the displacement control parameter $d_{sim}$.

If $d_{sim}$ is set to be small, the retrieval displacement with a few extensions would not affect the exploration of the state space in practice. The displacement, however, aggravates, since retrieval displacements are accumulated as the random tree is expanded. Moreover, even a small displacement can cause a problem when a tree is expanded near obstacle regions. Fig. 4 shows such an example. The retrieved motion $m_{ret}$ successfully extends the tree toward a node $n$ (a solid line). However, the actual motion (a dotted line) causes a collision although the retrieval displacement is small. To address this problem, we employ motion validation process.

*2) Motion validation:* In order to check whether a retrieved motion does not cause any collisions, we perform a motion validation operation by running a simulator with its start state and control. If any collision occurs in the recomputed trajectory with the simulation, we remove the motion's corresponding edge and all child nodes in the random tree. Otherwise, we update the random tree by replacing the recomputed trajectory with the prior retrieved motion.

The frequency of performing validation operations governs the accuracy and efficiency of our planner. Validating more retrieved motions reduces a probability to have inac-

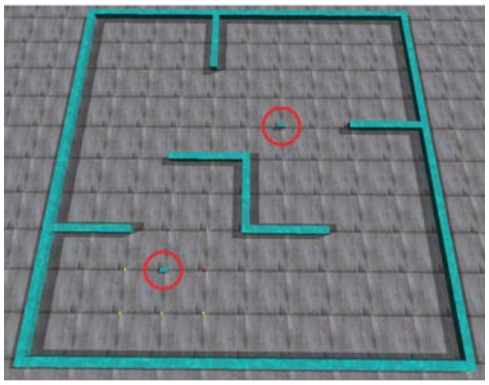

Fig. 5. Tested environment for a sled robot. The start and goal state of the robot are located at lower and upper sides of the image (circled in red).

curate nodes in the state space with a higher computational overhead. Therefore, motion validation should be carefully performed during iterations for the efficiency of the planner, while exploring the state space effectively.

Given this trade-off between the accuracy and efficiency, we use a lazy validation method that checks only motions that need to be validated immediately (e.g. motions in a solution path) or that are likely to be invalidated (e.g., motions close to obstacles). When a solution path is found, we validate every motions in the solution path in order to verify that the robot can reach to the goal with the solution. If the checked path does not reach the goal, we update it with simulated motions in the tree and then continue iterations until a new solution is found.

We define two criteria to see whether we need to perform the validation operation by considering a possibility of being invalidated in order to balance between efficiency and accuracy. When a new node is added to the tree, we check the following two criteria to decide whether we need to perform validation:

1) An accumulated displacement of the node is larger than $threshold_a$,
2) The distance from the node to the nearest obstacle is less than $threshold_o$,

If any of criteria is satisfied, we validate motions along the path from the root to the newly inserted node.

A pseudocode of the overall algorithm of our planner is shown at Algorithm 2.

## D. Probabilistic Completeness

Our planning method can efficiently explore the state space, while reducing duplicated, time-consuming propagation simulations by using the precomputed motion database.

The correctness of solutions computed by the proposed planner is same as any planning algorithm using a physics simulator, because our solution is entirely validated by the same simulator. The sub-paths in RRT tree that are not included in the solution path, however, could be incorrect, because we lazily validate motions in the tree. When there are many small obstacles in the environment, the homotopy of the solution path might be changed after performing the validation process. This, however, does not cause a problem in our application that aims to compute a feasible planning solution efficiently.

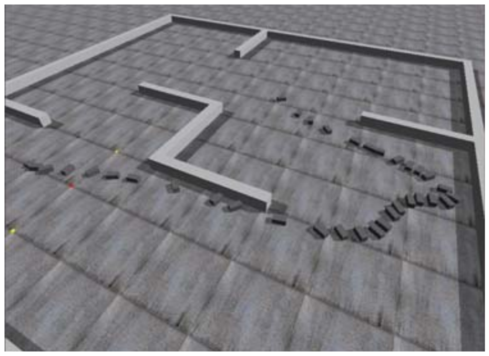

Fig. 6. A solution path computed by our method.

Nonetheless, the insufficient number of retrieved controls and displacements by using nearby retrieved motions can degrade the exploration quality of the state space. In order to guarantee the probabilistic completeness of our method, we set our planner to invoke our extension method using the database with a probability $\beta$ and use the original kinodynamic extension method with a probability $1-\beta$. We set $\beta$ to 0.85 in our tested experiments.

## V. Experimental Results

We implemented our method on an Intel i7 desktop machine that has 3.6GHz CPU and 16GB main memory. Our method is built upon an Open Motion Planning Library (OMPL [23]). We chose the Open Dynamic Engine [24] as a simulation engine among well known libraries such as PhysX, Bullet, Vortex, etc. Please note that our planner is not restricted to a specific library; therefore any library can be coupled to our method. We compared our method, denoted as DDK, to the kinodynamic RRT with $n$ control trials (Sec. III-B), denoted as $n$-RRT, against a kinodynamic planning problem for a rigid robot with dynamics.

## A. Benchmark Model

We conducted experiments for a sled robot that can apply force forward and horizontal torques on a slippery floor with static obstacles. Fig. 5 shows our experiment environment.

In our model, the global state space has the following components of the robot:

$$\mathbf{x} = (\mathbf{p}, \theta, \mathbf{v}, \omega),$$

TABLE I

PERFORMANCE COMPARISON TABLE BETWEEN $n$-CONTROL KINODYNAMIC RRT ($n$-RRT, SEC. III-B) AND OUR METHOD (DDK, SEC. IV). IN DDK 1, 40K MOTIONS ARE GENERATED AND IN DDK 2, 120K MOTIONS ARE GENERATED IN THE PREPROCESSING PHASE.

| | $n$-RRT | DDK 1<br>40k DB | DDK 2<br>120k DB |
|---|---|---|---|
| # of iterations | 12111.5 | 14841.2 | 13174.8 |
| # of nodes | 415.4 | 511.7 | 329.1 |
| DB construction time (s) | - | 38.4 | 115.3 |
| Planning time (s) | 215.88 | 131.91 | 88.84 |
| Simulation time (s) | 188.52 | 84.27 | 46.27 |
| % of simulation | 87.33 | 63.88 | 52.08 |

where $\mathbf{p}$ denotes the global position in the three dimensional space, $\theta$ denotes an orientation, $\mathbf{v}$ denotes linear velocity ($v_x$, $v_y$), and $\omega$ denotes angular velocity. The local state space, which is used in the motion database, is defined by linear velocity and angular velocity in local coordinate space of a robot:

$$\mathbf{x}_{local} = (\mathbf{v}_{local}, \omega_{local}),$$

where $\mathbf{v}_{local}, \omega_{local}$ represent the converted linear and angular velocities at the robot's local coordinate space. Conversion functions $f_c$ and $f_{c\_inv}$ are defined as $f_c = Proj(\mathbf{M_G^{-1}x})$ and $f_{c\_inv} = \mathbf{M_G}Reconst(\mathbf{x}_{local})$, where $\mathbf{M_G}$ represents a matrix containing translational and rotational parts between local and global frames. $Proj(\cdot)$ is a projection function that takes out the first two global coordinates, while $Reconst(\cdot)$ is the reconstruction function putting those two dropped global coordinates back.

We use simple, weighed Euclidean distance metrics for computing the distance between states in both global and local state spaces. These metrics are defined as:

$$d_{global}(\mathbf{x_1}, \mathbf{x_2}) = w_p(\|\mathbf{p}_1 - \mathbf{p}_2\|)^2 + w_o(1 - \mid \boldsymbol{\theta_1} \cdot \boldsymbol{\theta_2} \mid)^2$$

$$+ w_v(\|\mathbf{v}_1 - \mathbf{v}_2\|)^2 + w_\omega(\|\omega_1 - \omega_2\|)^2,$$

$$d_{local}(\mathbf{x'_1}, \mathbf{x'_2}) = w_v(\|\mathbf{v}'_1 - \mathbf{v}'_2\|)^2 + w_\omega(\|\omega'_1 - \omega'_2\|)^2,$$

where $w_p, w_o, w_v, w_\omega$ are weights for components of the state space and $x'$ denotes the variable defined in the local state space. We use $d_{global}$ for computing the nearest neighbor during RRT iterations and $d_{local}$ for the fixed-radius near neighbor search used in retrieving similar motions from the database.

Finally, the control space of the robot has two dimensions: force along the long edge of the robot to represent the forward and reverse acceleration, and yaw torque. Combining these controls, the robot can reach any place on the slippery floor with complex maneuvers.

TABLE II

BREAKDOWN OF THE RUNNING TIME

| | $n$-RRT | DDK 1 | DDK 2 |
|---|---|---|---|
| Total time (s) | 215.88 | 131.91 | 88.84 |
| Simulation time (s) | 188.52 | 84.27 | 46.27 |
| % of simulation | 87.33 | 63.88 | 52.08 |
| Retrieval time (s) | 0 | 1.92 | 7.99 |
| % of retrieval | 0.00 | 1.46 | 8.99 |
| Validation time (s) | 0 | 13.24 | 11.91 |
| % of validation | 0.00 | 10.04 | 13.41 |
| others (s) | 27.35 | 34.26 | 23.88 |
| % of others | 12.67 | 25.98 | 26.88 |

## B. Results and Comparisons

We tested our method with different sizes of the motion database. We ran the experiment 20 times for each method and report the mean of those results. Table I shows experimental results with two different settings for our method and with the conventional kinodynamic RRT with $n = 10$. In the first experiment (DDK 1), we generate 40 K motions for the database, and 120 K motions are generated in the second experiment (DDK 2). For building the database, it took 38.4 s and 115.3 s as the pre-computation time, and used 13MB and 40MB of memory. Fig. 6 shows one of solution paths computed by our planner.

In $n$-RRT, the simulation takes a very high portion (about 87%) of the running time for simulation, while our planner, DDK, uses a smaller portion for the simulation as more motions are stored in the motion database. This is mainly because we replace simulation that is computationally expensive with efficient data retrieval operations. As we use a bigger database, the planning time decreases. DDK 1 with 40K database shows 1.64 times faster, and DDK 2 with 120K database shows 2.43 times faster than $n$-RRT.

Note that the generated motion database can be reused for different problems such as different composition of obstacles and start/goal states as long as the applied dynamics to the robot is not changed, because motions in the database are constructed in its local state space. The cost of pre-computing motion databases is amortized as we use for different planning queries.

Our DDK planer can show a better performance, even if we solve a single problem and discard the database. When we compare different methods in terms of the total time including the construction time, DDK 1 shows 1.27 times faster and DDK 2 shows 5% improvement over $n$-RRT. The lower performance of DDK 2 is caused by its large construction time.

Table II shows a breakdown of the running time of $n$-control kinodynamic RRT and our method with different database sizes. We divide the running time into simulation time, retrieval time from the database, time for validations, and time spent on other parts (e.g. nearest neighbor search, maintaining graph, etc.) As we have the larger database, we reduce simulation and validation costs, while increasing the retrieval time. Since the motion retrieval from the motion database is very efficient, the overall performance with the larger database was improved.

**Limitations.** A major drawback of our method is that our planner relies on four parameters related to the validation process (Sec.IV-C). While our currently chosen parameter values worked well for our tests, further analysis is required to design robust thresholds that work with a wide variety of robots and environments. The size of the database affects the overall performance. We showed that the larger database achieves a better performance. Nonetheless, a huge database would increase not only the construction time but also the retrieval time, and could degrade the overall performance. The optimal size of the database should be chosen depending on a complexity of robots and a dimension of the database.

## VI. CONCLUSION

For planning under kinodynamic constraints, we have proposed a novel data-driven kinodynamic motion planner designed for complex dynamics. We proposed an extension method utilizing the motion database that is constructed as a preprocessing, and replaced the time-consuming integration of propagation functions by motion retrievals from the database. In order to increase the usage of a finite database in the continuous state space, we retrieved nearby motions to the query state and computed the best control and trajectory. To achieve high accuracy with high efficiency, we lazily validated retrieved motions. As a result, our proposed planner, DDK-RRT, shows meaningful improvement over the previous method. Constructed motion database can be reused for other problems such as different composition of obstacles or different start/goal state.

As future research directions, we would like to develop a better method for the database construction by reflecting the capability of a robot, in addition to addressing current drawbacks of our method. There have been machine learning approaches to utilize prior sampling information [25], [26]. It is interesting to apply this approach to our method for further improvement. Finally, we would like to apply our method to more challenging problems that deal with complicated robots such as a humanoid.

## VII. ACKNOWLEDGMENT

This work was supported in part by MI/KEIT 10070171 and MSIP/IITP [R0126-17-1108]. Corresponding author of this paper is Sung-eui Yoon.

## REFERENCES

[1] P. W. Finn and L. E. Kavraki, “Computational approaches to drug design,” *Algorithmica*, vol. 25, no. 2-3, pp. 347–371, 1999.

[2] J. Latombe, “Motion planning: A journey of robots, molecules, digital actors, and other artifacts,” *Int’l. Journal of Robotics Research*, vol. 18, pp. 1119–1128, 1999.

[3] B. Donald, P. Xavier, J. Canny, and J. Reif, “Kinodynamic motion planning,” *Journal of the ACM (JACM)*, vol. 40, no. 5, pp. 1048–1066, 1993.

[4] Z. Shiller and S. Dubowsky, “On computing the global time-optimal motions of robotic manipulators in the presence of obstacles,” *Robotics and Automation, IEEE Transactions on*, vol. 7, no. 6, pp. 785–797, 1991.

[5] J. J. Kuffner Jr, S. Kagami, K. Nishiwaki, M. Inaba, and H. Inoue, “Dynamically-stable motion planning for humanoid robots,” *Autonomous Robots*, vol. 12, no. 1, pp. 105–118, 2002.

[6] R. Diankov, N. Ratliff, D. Ferguson, S. Srinivasa, and J. Kuffner, “Bispace planning: Concurrent multi-space exploration,” *Proceedings of Robotics: Science and Systems IV*, vol. 63, 2008.

[7] S. M. LaValle and J. J. Kuffner, “Randomized kinodynamic planning,” *The International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, 2001.

[8] D. Hsu, R. Kindel, J.-C. Latombe, and S. Rock, “Randomized kinodynamic motion planning with moving obstacles,” *The International Journal of Robotics Research*, vol. 21, no. 3, pp. 233–255, 2002.

[9] I. Sucan, J. Kruse, M. Yim, and E. Kavraki, “Kinodynamic motion planning with hardware demonstrations,” in *IROS*, Sept 2008, pp. 1661–1666.

[10] I. Sucan and L. E. Kavraki, “A sampling-based tree planner for systems with complex dynamics,” *Robotics, IEEE Transactions on*, vol. 28, no. 1, pp. 116–131, 2012.

[11] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.

[12] J. Guitton, J.-L. Farges, and R. Chatila, “Cell-rrt: Decomposing the environment for better plan,” in *IROS*, 2009, pp. 5776–5781.

[13] J. Lee, O. Kwon, L. Zhang, and S. Yoon, “A selective retraction-based rrt planner for various environments,” *Robotics, IEEE Transactions on*, vol. 30, no. 4, pp. 1002–1011, Aug 2014.

[14] F. Lamiraux, S. Sekhavat, and J.-P. Laumond, “Motion planning and control for hilare pulling a trailer,” *Robotics and Automation, IEEE Transactions on*, vol. 15, no. 4, pp. 640–652, 1999.

[15] C. Stachniss and W. Burgard, “An integrated approach to goal-directed obstacle avoidance under dynamic constraints for dynamic environments,” in *IROS*, vol. 1. IEEE, 2002, pp. 508–513.

[16] P. Svestka and M. H. Overmars, “Motion planning for carlike robots using a probabilistic learning approach,” *The International Journal of Robotics Research*, vol. 16, no. 2, pp. 119–143, 1997.

[17] P. Cheng, E. Frazzoli, and S. M. LaValle, “Improving the performance of sampling-based planners by using a symmetry-exploiting gap reduction algorithm,” in *ICRA*, vol. 5. IEEE, 2004, pp. 4362–4368.

[18] A. Shkolnik, M. Walter, and R. Tedrake, “Reachability-guided sampling for planning under differential constraints,” in *Robotics and Automation, 2009. ICRA’09. IEEE International Conference on*. IEEE, 2009, pp. 2859–2865.

[19] E. Plaku, E. Kavraki, and M. Y. Vardi, “Motion planning with dynamics by a synergistic combination of layers of planning,” *Robotics, IEEE Transactions on*, vol. 26, no. 3, pp. 469–482, 2010.

[20] R. Gayle, S. Redon, A. Sud, M. C. Lin, and D. Manocha, “Efficient motion planning of highly articulated chains using physics-based sampling,” in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, 2007.

[21] T. Kunz and M. Stilman, “Kinodynamic rrts with fixed time step and best-input extension are not probabilistically complete,” in *WAFR*, 2014, pp. 233–244.

[22] Y. Li, Z. Littlefield, and K. E. Bekris, “Sparse methods for efficient asymptotically optimal kinodynamic planning,” in *Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 08/2014 2014.

[23] I. A. Sucan, M. Moll, and L. E. Kavraki, “The Open Motion Planning Library,” *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012, http://ompl.kavrakilab.org.

[24] Open Dynamics Engine. http://sourceforge.net/projects/opende/files/.

[25] M. Kalisiak and M. van de Panne, “Faster motion planning using learned local viability models,” in *ICRA*, 2007.

[26] J. Pan, S. Chitta, and D. Manocha, *Faster Sample-Based Motion Planning Using Instance-Based Learning*, 2013.