Manipulation Planning on Constraint Manifolds

Dmitry Berenson[†] Siddhartha S. Srinivasa[‡†] Dave Ferguson[‡†] James J. Kuffner[†]

[†]*The Robotics Institute, Carnegie Mellon University*
5000 Forbes Ave., Pittsburgh, PA, 15213, USA
[dberenso, kuffner]@cs.cmu.edu

[‡]*Intel Research Pittsburgh*
Pittsburgh, PA, 15213, USA
[siddhartha.srinivasa, dave.ferguson]@intel.com

***Abstract*— We present the Constrained Bi-directional Rapidly-Exploring Random Tree (CBiRRT) algorithm for planning paths in configuration spaces with multiple constraints. This algorithm provides a general framework for handling a variety of constraints in manipulation planning including torque limits, constraints on the pose of an object held by a robot, and constraints for following workspace surfaces. CBiRRT extends the Bi-directional RRT (BiRRT) algorithm by using projection techniques to explore the configuration space manifolds that correspond to constraints and to find bridges between them. Consequently, CBiRRT can solve many problems that the BiRRT cannot, and only requires one additional parameter: the allowable error for meeting a constraint. We demonstrate the CBiRRT on a 7DOF WAM arm with a 4DOF Barrett hand on a mobile base. The planner allows this robot to perform household tasks, solve puzzles, and lift heavy objects.**

I. Introduction

Our everyday lives are full of tasks that constrain our movement. Carrying a coffee mug, lifting a heavy object, or sliding a milk jug out of a refrigerator, are examples of tasks that involve constraints imposed on our bodies as well as the manipulated objects. Enabling autonomous robots to perform such tasks involves computing motions that are subject to multiple, often simultaneous, task constraints.

Consider the task of manipulating a heavy object as in Figure 1. When humans manipulate heavy objects that are difficult to lift, they often slide the object to support its weight. This enables human arms with limited muscle strength to perform a wider range of tasks without violating the maximum torque constraints imposed on their muscles. To enable robots to perform similar tasks, a motion planning algorithm is needed that can generate trajectories that obey joint torque constraints and slide objects along support surfaces when necessary. Consider the C-space of a 3-DOF manipulator shown in Figure 1(a). The large purple manifold represents configurations where torque constraints are valid (while holding a 3kg weight) and the green manifolds represent configurations of the robot where the weight is placed on either table. In order to go from configuration a to configuration e the robot must find a path along the left green manifold that will bring it near configuration b, a *bridge* configuration where the two manifolds intersect. At b, the robot may lift the weight as long as it stays within the purple manifold. Note that not all configurations which place the weight at the edge of the table allow it to be lifted, so the position of the weight alone is insufficient to determine if a configuration is valid.

Dmitry Berenson was partially supported by the Intel Summer Fellowship awarded by Intel Research Pittsburgh and by the National Science Foundation under Grant No. EEC-0540865.

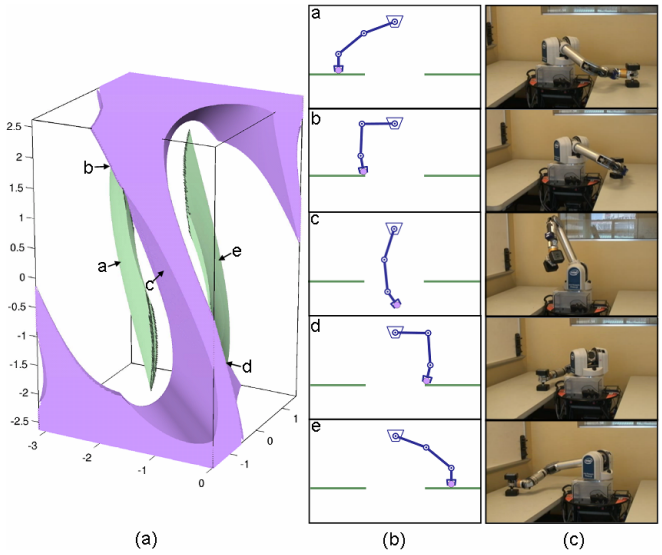

Fig. 1. (a) C-space of a 3DOF manipulator generated by exhaustive sampling. (b) 3-Link Manipulator configurations corresponding to several points along a path that moves the weight from one table to the other. (c) Snapshots from a 7DOF WAM arm with a 8.17kg end-effector mass executing a path found by the CBiRRT to move the dumbbell from one table to the other.

For a 3-DOF manipulator, we are able to compute constraint manifolds by exhaustively sampling the C-space but for a higher-DOF robot, computing these manifolds online is far too time-consuming. As a result, the planner has no prior knowledge of the shape of these manifolds nor where to find bridges (like b and d) between the manifolds in the general case. As we will show in subsequent sections, the final path is often quite complicated, traversing several manifolds and bridges. The lack of prior knowledge of the manifold structure precludes the use of task-space control[1] as a complete solution. Although controllers can be used to perform subtasks, such as sliding the weight on the table or maintaining its position in the air, a planner like the CBiRRT is required to orchestrate them, performing the C-space exploration necessary to discover bridges between manifolds and to avoid complex obstacles.

Many constraints, such as sliding constraints or pose constraints on a robot's end-effector, implicitly define manifolds that may occupy an infinitesimal volume in the C-space. Indeed the green sliding manifolds in Figure 1(a) occupy 2D surfaces in the 3D C-space. Discovering a configuration

that lies on such a manifold through randomly sampling joint-values is extremely unlikely. This fact precludes the use of standard sampling-based planners such as RRTs or Probabilistic Road Maps (PRMs) that sample C-space directly.

This paper introduces the Constrained Bi-directional Rapidly-Exploring Random Tree (CBiRRT) planner, which addresses the problem of sampling on constraint manifolds. CBiRRT first samples in the C-space and then uses projection operations to move samples onto constraint manifolds when necessary. This technique allows the planner to explore constraint manifolds efficiently and to construct paths embedded in them. CBiRRT also exploits the "connect" sampling heuristic of the RRT to find bridges between manifolds corresponding to different constraints, such as sliding an object and then lifting it. Such a slide-and-lift motion can be found using a single extension operation of the CBiRRT algorithm.

In the rest of the paper, we first give a brief overview of previous work relevant to constrained motion planning. We then introduce the CBiRRT algorithm and describe how to formulate various types of constraints. We then present several example problems and experiments which illustrate the ability of the CBiRRT to plan for tasks that were previously unachievable without special-purpose planners. The paper ends with a discussion of the advantages and limitations of our approach.

II. Background

The CBiRRT algorithm builds on several developments in motion planning and control research in robotics. In motion planning, a number of efficient sampling-based planning algorithms have been developed recently for searching high-dimensional C-spaces. Although CBiRRT is based on the Rapidly-exploring Random Tree (RRT) algorithm by LaValle and Kuffner[2], it is possible to adapt some of the ideas and techniques in this paper to other search algorithms. We selected RRTs for their ability to explore C-space while retaining an element of "greediness" in their search for a solution. The greedy element is most evident in the bidirectional version of the RRT algorithm (BiRRT), where two trees, one grown from the start configuration and one grown from the goal configuration, take turns exploring the space and attempting to connect to each other. In this paper, we demonstrate that such a search strategy is also effective for motion planning problems involving constraints when it is coupled with projection methods that move C-space samples onto constraint manifolds. Note that RRTs have also been previously extended to planning for hybrid control systems[3], which is similar to planning with constraint manifolds.

In the robotics literature, projection methods have arisen in the context of research in controls and inverse kinematics. Iterative inverse kinematics algorithms use projection methods based on the *pseudo-inverse* or transpose of the Jacobian to iteratively move a robot's end-effector closer to some desired workspace transformation (e.g. [4]). Sentis and Khatib's potential-field approach[1] uses *recursive null-space projection* to project a robot's configuration away from obstacles and toward desirable configurations. The Randomized Gradient Descent (RGD)[5] method uses random-sampling of the C-space to iteratively project a sample towards an arbitrary constraint[6]. Though [5] showed how to incorporate RGD into a randomized planner, it requires significant parameter-tuning and they dealt only with closed-chain kinematic constraints, which are a special case of the pose constraints used in this paper. Furthermore, Stilman [7] showed that when RGD is extended to work with more general pose constraints it is significantly less efficient than Jacobian pseudo-inverse projection and it is sometimes unable to meet more stringent constraints. Inspired by this result, we also use the Jacobian pseudo-inverse projection method, though our framework can use any projection method that moves samples on to constraint manifolds efficiently.

In some previous work the problem of planning for an object's motion is subdivided into planning a path in a lower-dimensional space[8][9] that lies within some manifold (like the surface of a table) or using a pre-scripted lower-dimensional path[10][11][12]. The lower-dimensional path is then followed in the full C-space of the robot. This approach suffers from feasibility problems because a lower-dimensional path may not be trackable by the robot because of joint limits or collisions. The CBiRRT algorithm therefore plans in the full C-space of the robot, which incurs a larger computational burden but allows it to handle more general types of constraints and to find paths through multiple constraint manifolds.

Algorithm 1: CBiRRT(Q_s, Q_g)

```
1  T_a.Init(Q_s); T_b.Init(Q_g);
2  while TimeRemaining() do
3      q_rand ← RandomConfig();
4      q^a_near ← NearestNeighbor(T_a, q_rand);
5      q^a_reached ← ConstrainedExtend(T_a, q^a_near, q_rand);
6      q^b_near ← NearestNeighbor(T_b, q^a_reached);
7      q^b_reached ← ConstrainedExtend(T_b, q^b_near, q^a_reached);
8      if q^a_reached = q^b_reached then
9          P ← ExtractPath(T_a, q^a_reached, T_b, q^b_reached);
10         return SmoothPath(P);
11     else
12         Swap(T_a, T_b);
13     end
14 end
15 return NULL;
```

III. The CBiRRT Algorithm

The CBiRRT algorithm (see Algorithm 1) operates by growing two trees in the C-space of the robot. During each iteration of the algorithm one of the trees grows a branch toward a randomly-sampled configuration q_{rand} using the ConstrainedExtend function. The branch grows as far as possible toward q_{rand} but may be stalled due to collision or constraint violation and will terminate at $q^a_{reached}$. The other tree then grows a branch toward $q^a_{reached}$, again growing as far

Algorithm 2: ConstrainedExtend(T, q_{near}, q_{target})

$q_s \leftarrow q_{near}$; $q_s^{old} \leftarrow q_{near}$;
while *true* **do**
  if $q_{target} = q_s$ **then**
    return q_s;
  else if $|q_{target} - q_s| > |q_s^{old} - q_{target}|$ **then**
    return q_s^{old};
  end
  $q_s \leftarrow q_s + \min(\Delta q_{step}, |q_{target} - q_s|) \frac{(q_{target} - q_s)}{|q_{target} - q_s|}$;
  $q_s \leftarrow$ ConstrainConfig(q_s^{old}, q_s);
  if $q_s \neq$ NULL ***and*** CollisionFree(q_s^{old}, q_s) **then**
    T.AddVertex(q_s);
    T.AddEdge(q_s^{old}, q_s);
    $q_s^{old} \leftarrow q_s$;
  else
    return q_s^{old};
  end
end

Algorithm 3: SmoothPath(P)

while TimeRemaining() **do**
  $T_{shortcut} \leftarrow \{\}$;
  $i \leftarrow$ RandomInt(1, $P.size - 1$);
  $j \leftarrow$ RandomInt(i, $P.size$);
  $q_{reached} \leftarrow$ ConstrainedExtend($T_{shortcut}$, P_i, P_j);
  if $q_{reached} = P_j$ ***and*** Length($T_{shortcut}$) < Length($P_i \cdots P_j$) **then**
    $P \leftarrow [P_1 \cdots P_i, T_{shortcut}, P_{j+1} \cdots P.size]$;
  end
end
return P;

as possible toward this configuration. If the other tree reaches $q^a_{reached}$, the trees have connected and a path has been found. If not, the trees are swapped and the above process is repeated.

The ConstrainedExtend function (see Algorithm 2) works by iteratively moving from a configuration q_{near} toward a configuration q_{target} with a step size of Δq_{step}. After each step toward q_{target}, the function checks if the new configuration q_s has reached q_{target} or if it is not making progress toward q_{target}, in either case the function terminates. If the above conditions are not true then the algorithm takes a step toward q_{target} and passes the new q_s to the ConstrainConfig function. The ConstrainConfig function is problem-specific, and several examples of such functions are given in the example problems. If ConstrainConfig is able to project q_s to a constraint manifold and this q_s is not in collision, the new q_s is added to the tree and the above process is repeated. Otherwise, ConstrainedExtend terminates (see Figure 2). ConstrainedExtend always returns the last configuration reached by the extension operation.

The CollisionFree(q_s^{old}, q_s) function checks collision by stepping along the interval between q_s^{old} and q_s. Collision-checking can also be treated as a constraint, and can be incorporated in to the ConstrainConfig function.

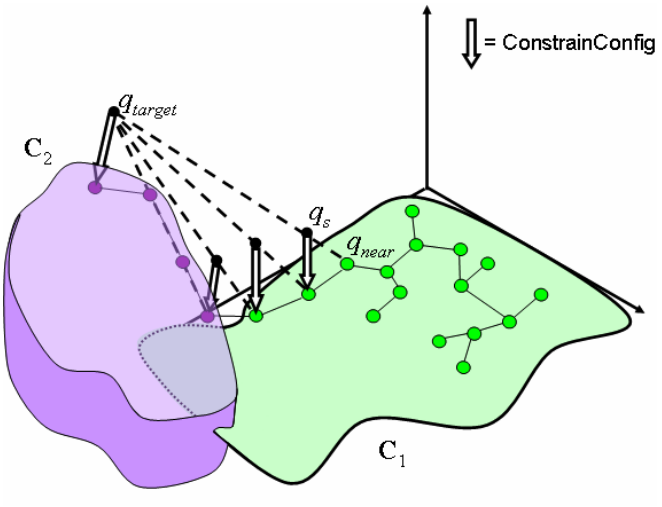

Fig. 2. Depiction of one extend operation that moves across two manifolds. The operation starts at q_{near}, which is a node of a search tree on constraint manifold $\mathbf{C}_1$ and iteratively moves toward q_{target}, which is a randomly-sampled configuration in C-space. Each step toward q_{target} is constrained using the ConstrainConfig function to lie on the closest constraint manifold.

The SmoothPath function uses the "short-cut" smoothing method to iteratively shorten the path from the start to the goal[13]. Since we use the ConstrainedExtend function for each short-cut, we are guaranteed that the constraints will be met along the smoothed path. Also, it is important to note that a short-cut generated by ConstrainedExtend between two nodes is not necessarily the shortest path between them because the nodes may have been projected in an arbitrary way. This necessitates checking whether Length($P_{shortcut}$) is shorter than the original path between i and j.

Besides handling constraints, an important difference between CBiRRT and the standard BiRRT algorithm is that multiple start and goal configurations (Q_s and Q_g, respectively) can be used to initialize the trees. This capability is important because many of the constraints that we deal with can invalidate paths between distant configurations. For instance, moving from an elbow-up configuration to an elbow-down configuration may not be possible if the end-effector is constrained to not move in a certain direction. Thus, when we run CBiRRT, we usually seed it with multiple IK solutions for both the start and goal configuration of an object we are trying to manipulate. Implementing this multiple start/goal capability is also straightforward. If a tree is stored as an array of nodes with each node containing a pointer to its parent, we define a placeholder node as the start/goal and set it as the parent of the Q_s/Q_g nodes. The algorithm then proceeds as normal.

Another key point is that the BiRRT algorithm is a special case of the CBiRRT algorithm. If the ConstrainConfig() function always returns true without modifying q_s (i.e. there are no constraints) and $|Q_s| = |Q_g| = 1$, the CBiRRT and BiRRT algorithms behave identically.

IV. CONSTRAINTS

Our planner is capable of handling tasks with multiple constraints as long as each constraint can be evaluated as a function of the robot's configuration. The algorithm can handle arbitrary strategies for dealing with these constraints by encoding these strategies in the ConstrainConfig function. In this paper, we focus on two general strategies for dealing with constraints: rejection and projection. Neither strategy requires an analytical representation of the constraint manifold.

In the rejection strategy, we simply check if a given configuration of the robot meets a certain constraint, if it does not, we deem the configuration invalid. This strategy is effective when there is a high probability of randomly sampling configurations that satisfy this constraint, in other words, the constraint manifold occupies some significant volume in the C-space.

The projection strategy is robust to more stringent constraints, namely ones whose manifolds do not occupy a significant volume of the C-space. However, this robustness comes at the price of requiring a distance function to evaluate how close a given configuration is to the constraint manifold. The projection strategy works by using gradient descent to iteratively reduce the distance to the constraint manifold and terminates when a configuration is found that is within some threshold ϵ of the manifold.

A. Object/End-Effector Pose Constraints

Many of the constraints we deal with are restrictions on the pose of an object being manipulated by a robot arm or the arm's end-effector, which were first discussed in [14]. We assume that the arm has grasped the object it is holding rigidly, effectively translating constraints on the object into constraints on the end-effector. Thus we will treat constraints on the object and constraints on the robot's end-effector as conceptually equivalent.

Throughout this paper, we will be using transformation matrices of the form $\mathbf{T}_b^a$, which specifies the pose of b in the coordinates of frame a. $\mathbf{T}_b^a$, written in homogeneous coordinates, consists of a 3×3 rotation matrix $\mathbf{R}_b^a$ and a 3×1 translation vector $\mathbf{t}_b^a$.

$$\mathbf{T}_b^a = \begin{bmatrix} \mathbf{R}_b^a & \mathbf{t}_b^a \\ \mathbf{0} & 1 \end{bmatrix} \tag{1}$$

The first step to working with constraints on the object's pose is to define a reference transform for the object $\mathbf{T}_{obj}^0$ as well as a reference transform for the constraint $\mathbf{T}_c^0$. $\mathbf{T}_c^0$ can be stationary in the world (for instance the hinge of a door) or can change depending on the pose of the object. Constraints are then defined in terms of the permissible differences between $\mathbf{T}_{obj}^0$ and $\mathbf{T}_c^0$ as in Equation 2.

$$\mathbf{C} = \begin{bmatrix} c_{x_{min}} & c_{x_{max}} \\ c_{y_{min}} & c_{y_{max}} \\ c_{z_{min}} & c_{z_{max}} \\ c_{\psi_{min}} & c_{\psi_{max}} \\ c_{\theta_{min}} & c_{\theta_{max}} \\ c_{\phi_{min}} & c_{\phi_{max}} \end{bmatrix} \tag{2}$$

The first three rows of $\mathbf{C}$ bound the allowable translation along the x, y, and z axes and the last three bound the allowable rotations about those axes, all in the $\mathbf{T}_c^0$ frame. Note that this assumes the Roll-Pitch-Yaw (RPY) Euler Angle convention.

Such a representation has several advantages. First, specifying constraints is intuitive as will be shown in the example problems. Second, this representation allows us to define a distance function for pose constraints that is very fast to compute. Given a configuration q_s, we define the DisplacementFromConstraint($\mathbf{C}$, $\mathbf{T}_c^0$, q_s) function as follows:

First compute the forward kinematics at q_s to get $\mathbf{T}_{obj}^0$. Then compute the pose of the object in constraint-frame coordinates.

$$\mathbf{T}_{obj}^c = (\mathbf{T}_c^0)^{-1}\mathbf{T}_{obj}^0 \tag{3}$$

Then convert $\mathbf{T}_{obj}^c$ from a transformation matrix to a 6-dimensional displacement vector d^c, consisting of displacements in x, y, z, roll, pitch and yaw:

$$d^c = \begin{bmatrix} \mathbf{t}_{obj}^c \\ \arctan2(\mathbf{R}_{obj_{32}}^c, \mathbf{R}_{obj_{33}}^c) \\ -\arcsin(\mathbf{R}_{obj_{31}}^c) \\ \arctan2(\mathbf{R}_{obj_{21}}^c, \mathbf{R}_{obj_{11}}^c) \end{bmatrix} \tag{4}$$

Taking into account the bounds in $\mathbf{C}$, we get the displacement to this constraint $\Delta\mathbf{x}$:

$$\Delta\mathbf{x}_i = \begin{cases} d_i^c - \mathbf{C}_{i_{max}} & \text{if } d_i^c > \mathbf{C}_{i_{max}} \\ d_i^c - \mathbf{C}_{i_{min}} & \text{if } d_i^c < \mathbf{C}_{i_{min}} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where i indexes through the six rows of $\mathbf{C}$ and six elements of $\Delta\mathbf{x}$ and d^c. The distance to the constraint is then $\|\Delta\mathbf{x}\|$. Note that this distance function is only used when projecting to pose constraints, the standard Euclidean distance function is used when selecting nearest-neighbors in the RRT.

B. Using Projection with Pose Constraints

In order to meet pose constraints, we employ a gradient-descent projection method based on the Jacobian-pseudo inverse which is similar to that used in [7] (see Algorithm 4), however any effective projection method is acceptable.

Algorithm 4: ProjectConfig(q_s^{old}, q_s, $\mathbf{C}$, $\mathbf{T}_c^0$)

```
while true do
    Δx ← DisplacementFromConstraint(C, T_c^0, q_s);
    if ‖Δx‖ < ε then return q_s;
    J ← GetJacobian(q_s);
    Δq_error ← J^T (J J^T)^{-1} Δx;
    q_s ← (q_s − Δq_error);
    if |q_s − q_s^old| > 2Δq_step or OutsideJointLimit(q_s)
    then return NULL;
end
```

The GetJacobian function returns the Jacobian of the manipulator with the rotational part of the Jacobian in the RPY convention. Converting the standard angular-velocity Jacobian

to the RPY Jacobian is done by applying the linear transformation $E_{rpy}(q)$, which is defined in the Appendix of [7].

C. Torque Constraints

Another constraint we will deal with is the constraint on joint torques when lifting heavy objects. Since we will be employing the rejection strategy with respect to torque constraints, we need only calculate the torques on the joints in a given q_s. This is done using standard Recursive Newton-Euler techniques described in [15]. To incorporate the object into the robot model, we take a weighted average of the centers of mass of the end-effector and the object and set that as the mass and center of mass of the end-effector. We will refer to the combined mass as m. Note that this formulation only takes into account the torque necessary to maintain a given q_s, i.e. it assumes the robot's motion is quasi-static.

So far this section has described how to find displacements to a given constraint and how to handle torque constraints. However when there are multiple constraints, the planner must make decisions about which constraints to project to and must sometimes use a combination of rejection and projection strategies to plan a path. In the following three sections, we describe three example problems which deal with various constraints and show effective methods for planning paths for these problems based on the above two strategies.We also describe implementation details and simulation results for an instance of each type of problem.

V. Example A: The Maze Puzzle

In this problem, the robot arm must solve a maze puzzle by drawing a path through the maze with a pen (see Figure 3(a)). The constraint is that the pen must always be touching the table however the pen is allowed to pivot about the contact point up to an angle of α. We define $\mathbf{T}^0_{obj}$ to be at the tip of the pen with no rotation relative to the world frame. We define $\mathbf{T}^0_c$ to be at the height of the table at the center of the maze with no rotation relative to the world frame (z being up). This example is meant to demonstrate that CBiRRT is capable of solving multiple narrow passage problems while still moving on a constraint manifold. It is also meant to demonstrate the generality of the CBiRRT; no special-purpose planner is needed even for such a specialized task. The ConstrainConfig function used for this example is shown in Algorithm 5.

A. Implementation and Results

The robot's base is fixed in this problem. IK solutions were generated for both the start and goal position of the pen using the given grasp and input as Q_s and Q_g. For this example we place the base roughly halfway between the start and goal positions of the object such that it does not collide with any obstacles. We then compute all IK solutions for both the start and goal positions of the object up to a 0.05rad discretization of the arm's first joint angle. The values in Table I represent the average of 10 runs for different α values. Runtimes with a "$>$" denote that there was at least one run that did not terminate before 120 seconds. For such runs, 120 was used in computing the average. $\Delta q_{step} = 0.05$ and $\epsilon = 0.001$.

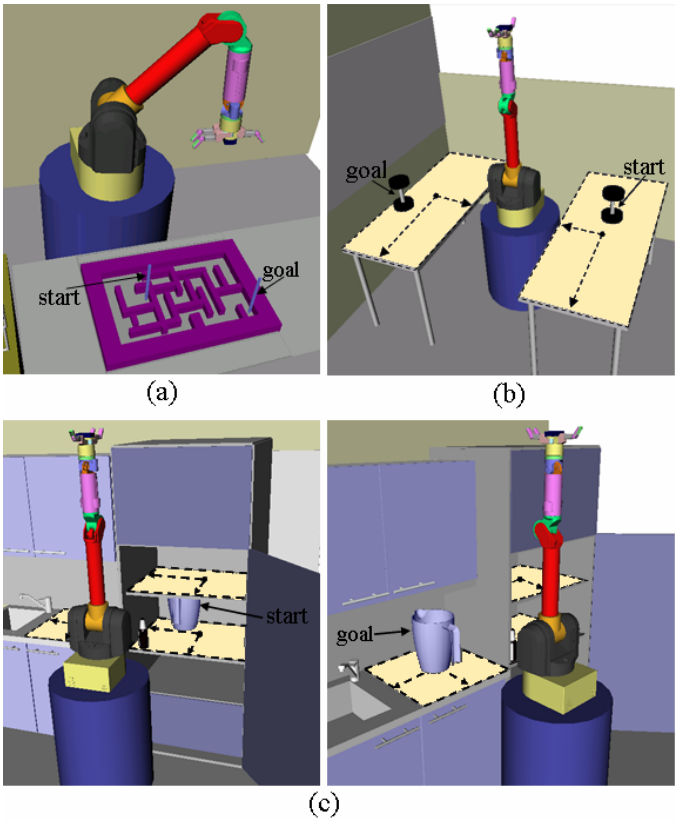

Fig. 3. Example problems. (a) Maze Puzzle (b) Heavy object with sliding surfaces (c) Heavy object with sliding surfaces and pose constraints. Yellow planes represent surfaces that can be used for sliding.

α(rad.)	0.0	0.1	0.2	0.3	0.4	0.5
Avg. Runtime(s)	$>$83.5	$>$58.8	$>$49.0	19.5	14.3	15.2
Success Rate	40%	60%	90%	100%	100%	100%

Table I: Simulation Results for Example A

The shorter runtimes and high success rates for larger α values demonstrate that the more freedom we allow for the task, the easier it is for the algorithm to solve it. This shows a key advantage of formulating the constraints as bounds on allowable pose as opposed to requiring the pose of the object to conform exactly to a specified value. For problems where we do not need to maintain an exact pose for an object we can allow more freedom, which makes the problem easier. See Figure 4 for an example trajectory of the tip of the pen.

VI. Example B: Heavy Object with Sliding Surfaces

In this problem the task is to move a heavy object (a dumbbell in this example) from a start position to a given goal position (see Figure 3(b)). It is not known a-priori whether the object is light enough to lift directly from its start position or if it can be placed directly into its goal position without sliding. Sliding surfaces are also provided so that the planner may use these if necessary. Each sliding surface is a rectangle of

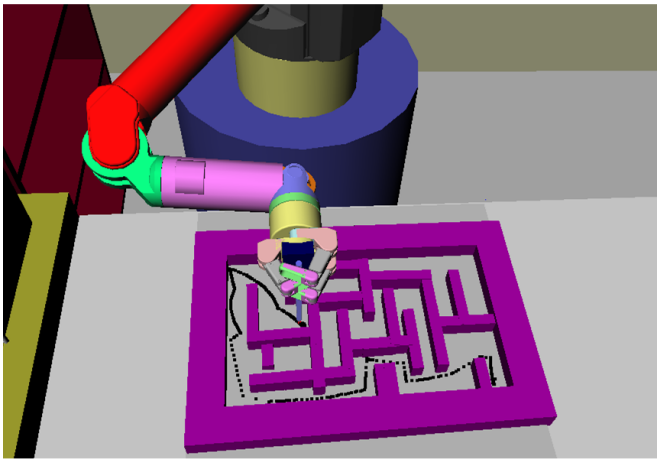

Fig. 4. A trajectory found for Example A using $\alpha = 0.4$rad. The black points represent positions of the tip of the pen along the trajectory.

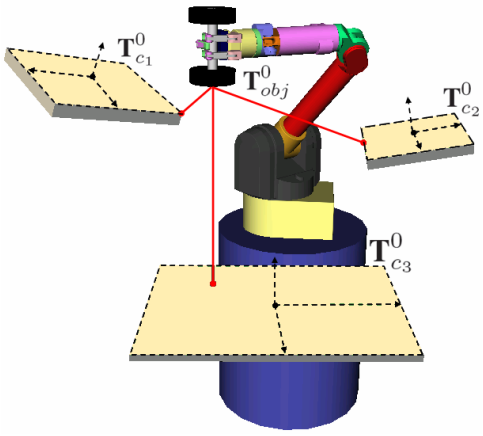

Fig. 5. Depiction of the GetNearestSlidingFrame function for choosing a sliding manifold. The shortest distance from $\mathbf{T}^0_{obj}$ to $\mathbf{T}^0_{c_i}$ (computed using the DisplacementToConstraint function) determines which surface is chosen for projection.

Algorithm 5: ConstrainConfig(q_s^{old}, q_s) for Example A

```
C = [−∞ ∞; −∞ ∞; 0 0; −α α; −α α; −∞ ∞];
T^0_c = CenterOfTable();
return ProjectConfig(q_s^old, q_s, C, T^0_c);
```

known bounds with an associated surface normal. In general, the surfaces may be slanted so they may only support part of the objects's weight. Each sliding surface gives rise to a constraint manifold and there can be any number of sliding surfaces. g is a unit vector representing the direction of gravity in homogeneous coordinates; for our problem $g = [0\ 0\ \text{-}1\ 0]^T$.

The GetNearestSlidingFrame function (based on the displacement defined in Equation 5) returns the $\mathbf{T}^0_c$ of the nearest sliding surface (see Figure 5) along with the constraint describing that surface, which will be of the form $\mathbf{C} = [-t_w\ t_w;\ -t_l\ t_l;\ 0\ 0;\ 0\ 0;\ 0\ 0;\ -\infty\ \infty]$, where t_w and t_l are the half-width and half-length of the surface. In Algorithm 6 ProjectionConfig is called with this $\mathbf{C}$ so that $\mathbf{T}^0_{obj}$ moves toward the closest point on the closest surface.

A. Implementation and Results

We ran this example for both the fixed and mobile base cases. When planning with a mobile base, we allow translation of the base in x and y to be considered as two additional DOF of the robot. No non-holonomic constraints are placed on the base's motion. For the fixed base mode, we generate Q_s and Q_g the same way as in the Maze Puzzle. For the mobile base mode, we sampled 200 random base positions in a circle around the start and goal of the dumbbell and computed all IK solutions (to a 0.05rad discretization of the first joint) for each base position. All the collision-free IK solutions were input as Q_s and Q_g. The values in Table II represent the average of 10 runs for different weights of the dumbbell. Runtimes with a ">" denote that there was at least one run that did not terminate before 120 seconds. For such runs, 120 was used in computing the average. The weight of the dumbbell was

Algorithm 6: ConstrainConfig(q_s^{old}, q_s) for Example B

```
if CheckTorque(q_s, m) then return q_s;
{T^0_c, C} ← GetNearestSlidingFrame(q_s);
m_s = m(1 − CLAMP(−g · T^0_c[0, 0, 1, 0]^T, [0 1]));
if ProjectConfig(q_s^old, q_s, C, T^0_c) and
    CheckTorque(q_s, m_s) then
    return q_s;
else
    return NULL;
end
```

increased until the algorithm could not find a path within 120 seconds in any of the 10 runs. $\Delta q_{step} = 0.05$ and $\epsilon = 0.001$.

Weight	7kg	8kg	9kg	10kg	11kg	12kg	13kg	14kg
Fixed Base								
Avg. Runtime(s)	1.89	2.06	3.84	5.51	7.29	12.4	27.5	>53.9
Success Rate	100%	100%	100%	100%	100%	100%	100%	80%
Mobile Base								
Avg. Runtime(s)	12.9	22.1	17.5	33.5	57.3	>105	>110	>120
Success Rate	100%	100%	100%	100%	100%	40%	40%	0%

TABLE II: SIMULATION RESULTS FOR EXAMPLE B

The shorter runtimes and higher success rates for lower weights of the dumbbell match our expectations about the constraints induced by torque limits. As the dumbbell becomes heavier, the manifold of configurations with valid torque becomes smaller and thus finding a path through this manifold becomes more difficult. See Figure 7 for two sample trajectories illustrating this concept. The mobile base tends to not do as well as the fixed base in this example because the addition of the base's DOF expands the size of the C-space exponentially, thus making the problem more difficult.

We also implemented this problem on our physical WAM robot. Snapshots from three trajectories for three different weights are shown in Figure 6. As with the simulation environment, the robot slid the dumbbell more when the weight

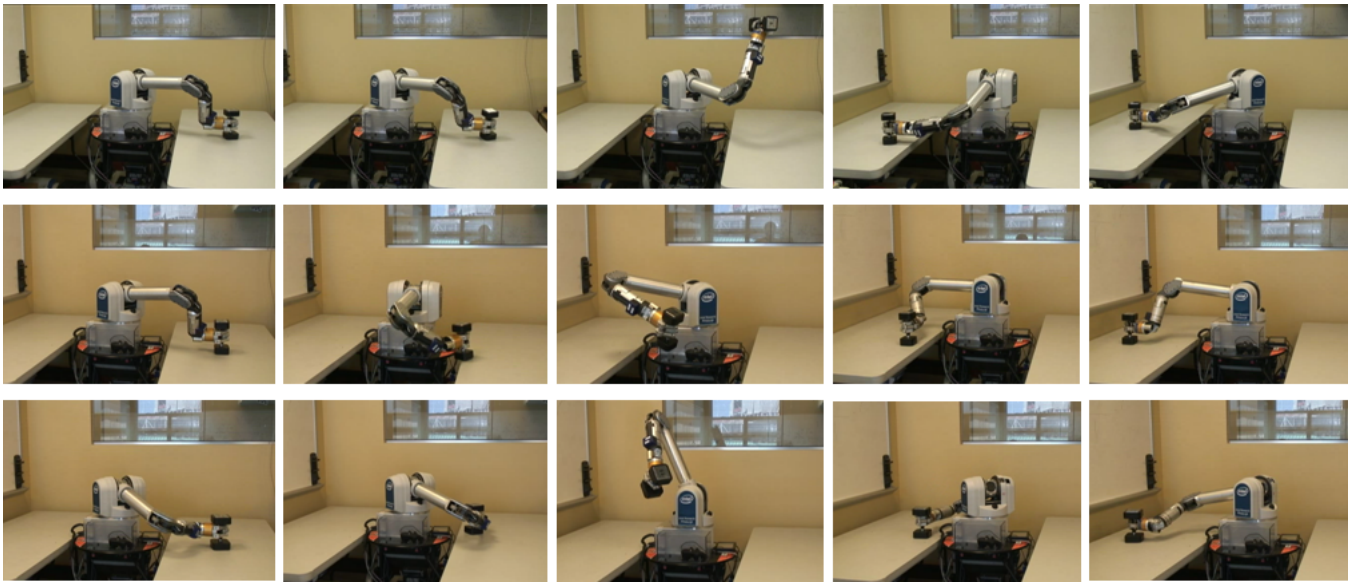

Fig. 6. Experiments on the 7DOF WAM arm for three different dumbbells. Top Row: $m = 4.98$kg. Middle Row: $m = 5.90$kg, and Bottom Row: $m = 8.17$kg. The trajectory for the lightest dumbbell requires almost no sliding, where as the trajectories for the heavier dumbbells slide the dumbbell to the edge of the table.

was heavier and sometimes picked up the weight without any sliding for the mass of 4.98kg. Note that we take advantage of the compliance of our robot to help execute these trajectories but in general such trajectories should be executed using an appropriate force-feedback controller. Please see our video at:

http://www.cs.cmu.edu/%7edberenso/constrainedplanning.mp4

VII. Example C: Heavy Object with Sliding Surfaces and Pose Constraint

This problem is similar to the previous one except that there is a constraint on the pose of the object throughout the task. The example we use for this kind of task is getting a pitcher of water out of a refrigerator and placing it on a counter (see Figure 3(c)). Since the top of the pitcher is open, we must impose a constraint on the pose of the pitcher so that the water does not spill out. Again, we do not know a priori whether the pitcher is light enough to simply lift out of its start configuration or to place directly in its goal position without sliding. While this task is more complex than the previous one, it only requires the addition of the line

if ProjectConfig($q_s^{old}, q_s, \mathbf{C}_{nt}, \mathbf{I}$)=NULL **then return** NULL;

before line 1 in Algorithm 6. $\mathbf{C}_{nt}$ = [$-\infty$ ∞; $-\infty$ ∞; $-\infty$ ∞; 0 0; 0 0; $-\infty$ ∞] specifies the no-tilting constraint bounds and $\mathbf{I}$ is the identity transform. Since we do not want to spill the water while sliding, only non-tilted sliding surfaces are considered in this problem.

A. Implementation and Results

This example was also run for the fixed base and mobile base cases. Q_s and Q_g are generated the same way as in the previous example. The weight of the pitcher is incremented and runtimes are averaged as with the previous example. The results are summarized in Table III. The center of gravity of the pitcher was set to be the same as the center of gravity of the weight to make the results of the two problems comparable. $\Delta q_{step} = 0.05$ and $\epsilon = 0.001$.

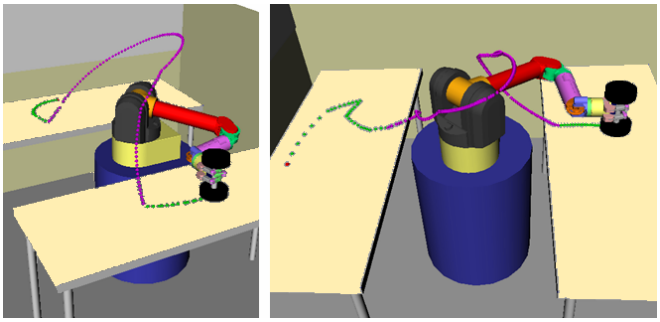

Fig. 7. Two trajectories for example B using a fixed base. Left: Trajectory found for $m = 7$kg, Right: Trajectory found for $m = 12$kg. Green nodes represent points in the trajectory where the dumbbell is sliding, purple nodes represent points where the dumbbell is not sliding. In the 7kg case, completing the task does not require much sliding of the dumbbell and it can be lifted high above the robot. In the 12kg case the weight of the dumbbell prohibits lifting it above the robot and necessitates more sliding along the tables.

Weight	5kg	6kg	7kg	8kg	9kg	10kg	11kg	12kg
Fixed Base								
Avg. Runtime(s)	2.79	15.8	18.1	>39.1	>120	>120	>120	>120
Success Rate	100%	100%	100%	90%	0%	0%	0%	0%
Mobile Base								
Avg. Runtime(s)	31.2	54.9	57.8	>78.3	>104	>80.9	>90.7	>115
Success Rate	100%	100%	100%	90%	60%	80%	60%	20%

Table III: Simulation Results for Example C

The results of this problem demonstrate the advantages of having a mobile base in cluttered environments. The fixed base is placed close to both the start and goal locations of the object so that it can pull the object close to its body while keeping it on a sliding surface, thus maintaining a low torque when it lifts the object. However, in order to get from the refrigerator

to the counter, the robot must lift the pitcher over its own body to avoid collisions, which requires more torque. This make it difficult to pick a position for the base that will work with larger weights because the base must be close enough to the refrigerator and counter to lift the object off of the sliding surfaces but must be far enough so that the robot can move the pitcher between the refrigerator and the counter without requiring a lot of torque. The mobile base is preferable in this situation because it can access both the refrigerator and the counter by moving the base closer to them and maintaining low torques for the arm. It can then drive from the refrigerator to the counter while keeping the arm in roughly the same position, thus requiring no increase in torque. See Figure 8 for sample trajectories both with and without the mobile base.

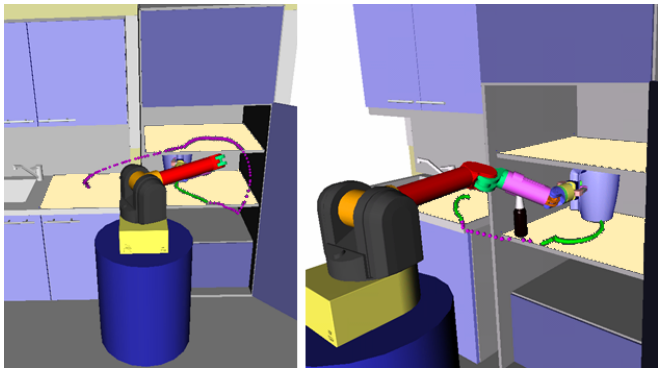

Fig. 8. Left: Trajectory found for $m = 8$kg using a fixed base. Right: Trajectory found for $m = 12$kg using a mobile base. Green nodes represent points in the trajectory where the pitcher is sliding, purple nodes represent points where the pitcher is not sliding. When $m = 8$kg, the pitcher can be lifted over the body of the robot to avoid collision but when $m = 12$kg, the fixed base mode fails because the pitcher cannot be lifted over the body of the robot.

VIII. Discussion

One of the CBiRRT's main strengths is that it is able to plan with a variety of constraints, several of which have been described in this paper. Composing these constraints has allowed us to plan for manipulation tasks that were previously unachievable in the general case. However, so far the constraints we have considered only apply to configurations of the robot so the resulting trajectory is only valid assuming the robot's motion is quasi-static. This is a common assumption when planning trajectories in high dimensional spaces but RRTs have been applied to kinodynamic (position-velocity) spaces as well[16]. Planning in kinodynamic space incurs the disadvantages of doubling the dimensionality of the space, introducing difficulty in connecting search trees, and relying on an unclear distance metric. As an alternative to kinodynamic planning, we could impose a worst-case constraint on *edges* in the CBiRRT. Assuming bounds on the velocity and acceleration of the robot, we can determine the worst-case torques the robot will need to apply to move between two configurations and determine if those torques exceed the limits. Such a strategy would be useful for sliding a heavy object on surfaces with significant friction. Applying such an edge constraint in the smoothing step can guarantee that a trajectory found by CBiRRT can be executed within a certain time.

Another limitation of the CBiRRT is that it currently assumes that an object has only one reference transform $\mathbf{T}^0_{obj}$, however this may be insufficient. The dumbbell we used could have slid on either the top or the bottom surface, however since there is only one reference transform for the end-effector/object, we are currently limited to using only one of these sliding modes. We plan to continue development of the CBiRRT and address these issues in future work.

IX. Conclusion

We have presented the CBiRRT algorithm for planning paths in C-spaces with multiple constraints, including constraints on end-effector pose, joint torques, and following workspace surfaces. We have shown how to formulate these constraints and how to include them into our algorithm using combinations of rejection and projection strategies. Finally, we have presented several example problems on the 7DOF WAM arm with a mobile base and showed that our planner is capable of solving complex problems involving diverse and stringent constraints that were previously unsolvable in the general case.

References

[1] S. Sentis and O. Khatib, "Synthesis of whole-body behaviors through hierarchical control of behavioral primitives," *International Journal of Humanoid Robotics*, 2005.

[2] S. LaValle and J. Kuffner, "Rapidly-exploring random trees: Progress and prospects," in *WAFR*, 2000.

[3] M. Branicky, M. Curtiss, J. Levine, and S. Morgan, "RRTs for nonlinear, discrete, and hybrid planning and control," in *Proc. IEEE Conf. on Decision and Control*, 2003, pp. 657–663.

[4] L. Sciavicco and B. Siciliano, *Modeling and Control of Robot Manipulators*, 2nd ed. Springer, 2000, pp. 96–100.

[5] J. Yakey, S. LaValle, and L. Kavraki, "A probabilistic roadmap approach for systems with closed kinematic chains," in *IEEE Transactions on Robotics*, 2001.

[6] Z. Yao and K. Gupta, "Path planning with general end-effector constraints: Using task space to guide configuration space search," in *IROS*, 2005.

[7] M. Stilman, "Task constrained motion planning in robot joint space," in *IROS*, 2007.

[8] Y. Koga, K. Kondo, J. Kuffner, and J. Latombe, "Planning motions with intentions," in *SIGGRAPH*, 1994.

[9] K. Yamane, J. Kuffner, and J. Hodgins, "Synthesizing animations of human manipulation tasks," in *SIGGRAPH*, 2004.

[10] S. Seereeram and J. Wen, "A global approach to path planning for redundant manipulators," *Robotics and Automation, IEEE Transactions on*, vol. 11, no. 1, pp. 152–160, Feb 1995.

[11] G. Oriolo, M. Ottavi, and M. Vendittelli, "Probabilistic motion planning for redundant robots along given end-effector paths," in *IROS*, 2002.

[12] G. Oriolo and C. Mongillo, "Motion planning for mobile manipulators along given end-effector paths," in *ICRA*, 2005.

[13] P. Chen and Y. Hwang, "SANDROS: a dynamic graph search algorithm for motion planning," *Robotics and Automation, IEEE Transactions on*, vol. 14, no. 3, pp. 390–403, Jun 1998.

[14] M. Mason, "Compliance and force control for computer controlled manipulators," *IEEE Trans. on Systems, Man, and Cybernetics*, 1981.

[15] M. Walker and D. Orin, "Efficient dynamic computer simulation of robotic mechanisms," *ASME Journal of Dynamic Systems Measurement and Control*, vol. 104, pp. 205–211, 1982.

[16] S. LaValle and J. Kuffner, "Randomized kinodynamic planning," *Internation Journal of Robotics Research*, vol. 20, Jun 2001.