Planning and Resilient Execution of Policies For Manipulation in Contact with Actuation Uncertainty

Calder Phillips-Grafflin[1] and Dmitry Berenson[2]

[1]Worcester Polytechnic Institute, [2]University of Michigan

Abstract. We propose a method for planning motion for robots with actuation uncertainty that incorporates contact with the environment and the compliance of the robot to reliably perform manipulation tasks. Our approach consists of two stages: (1) Generating partial policies using a sampling-based motion planner that uses particle-based models of uncertainty and simulation of contact and compliance; and (2) Resilient execution that updates the planned policies to account for unexpected behavior in execution which may arise from model or environment inaccuracy. We have tested our planner and policy execution in simulated $SE(2)$ and $SE(3)$ environments and Baxter robot. We show that our methods efficiently generate policies to perform manipulation tasks involving significant contact and compare against several simpler methods. Additionally, we show that our policy adaptation is resilient to significant changes during execution; e.g. adding a new obstacle to the environment.

1 Introduction

Many real-world tasks are characterized by uncertainty: actuators and sensors may be noisy, and often the robot's environment is poorly modelled. Unlike robots, humans effortlessly perform everyday tasks, like inserting a key into a lock, which require fine manipulation despite limited sensing and imprecise actuation. We observe that humans often perform these tasks by exploiting *contact*, *compliance*, and *resilience*. Using compliance to safely make contact and move while in contact allows us to reduce uncertainty. We also exhibit resilience: when an action fails to produce the desired result, we may withdraw and try again. Seminal motion planning work by Lozano-Pérez et al. [1] shows that incorporating contact and compliance is critical to performing fine motions like inserting a peg into a hole. Building from this work and our observations of human motions, we have developed a motion planner that incorporates contact, compliance, and resilience to generate behavior for robots with actuation uncertainty.

Motion in the presence of actuation uncertainty is an example of a continuous Markov Decision Process (MDP), adding in sensor uncertainty, the problem becomes a Partially-Observable Markov Decision Process (POMDP). Solving an MDP or POMDP is often framed as the problem of computing an optimal policy

π^* that maps each state to an action a that maximizes the expected reward (e.g. the probability of reaching the goal). This paper focuses on motion planning with actuation uncertainty, and thus we frame the problem as an MDP. This MDP formulation is representative of the challenges face by low-cost and compliant robots such as Baxter or Raven, which have accurate sensing but noisy actuators.

Instead of planning in the configuration or state-space of the robot, we represent the uncertainty of the state of the robot as a probability distribution over possible configurations, and plan in the space of these distributions—the *belief space*[1]. The computational expense of optimal motion planning leads us to adopt a thresholding approach from *conformant* planning [2]. Instead of attempting to find a global optimal policy, we seek to generate a *partial* policy that allows a robot with actuation uncertainty to move from start configuration q_{start} to reach goal q_{goal} within tolerance ϵ_{goal} with at least planning threshold P_{goal} probability. A partial policy, which maps a subset of possible states to actions rather than a global policy that maps all states to actions, simplifies the problem and is appropriate for the single-query planning problems we seek to solve.

The complexity of robot kinematics and dynamics preclude analytical modeling of compliance and contact for practical, high-dimensional problems, and thus we rely on the ability to forward simulate the state of the robot given a starting state and action. In the presence of uncertainty, individual actions may have multiple distinct outcomes: for example, when trying to insert a key into a lock, some attempts will succeed in inserting the key, while some will miss the keyhole. In advance of performing such an action, we cannot *select* between desired outcomes (as is assumed in [3]). However, we can *distinguish* between the outcomes after the action is executed. We directly incorporate this behavior into our planner using *splits* and *reversibility*. Splits are single actions that produce multiple distinct outcomes, which we distinguish between using a series of clustering algorithms. Reversibility is the ability of a specific action and outcome to be "undone" and return to the previous state, which allows the robot to attempt the action again. Of course, the planner may not accurately model the outcomes of every action, so we incorporate an online adaptation process to update the planned policy during execution to reflect the results of actions.

Our primary contributions are thus 1) incorporating contact and compliance into policy generation, thus allowing contacts that other planners would discard but that, in fact, can be used to reduce uncertainty; and 2) introducing resilience into policy execution and thus significantly increasing the probability of successfully completing the task. Our experiments with simulated test environments suggest that our planner efficiently generates policies to reliably perform motion for robots with actuation uncertainty. We apply our methods to problems in $SE(2)$, $SE(3)$, and a simulated Baxter robot ($\mathbb{R}^7$) and show performance improvements over simpler methods and the ability to recover from an unanticipated blockage.

[1] The term *belief* is borrowed from POMDP literature, which assumes that the state is partially-observable. Though this paper considers only MDPs, we nevertheless use "belief" as it is a convenient and widely-used term for a distribution over states.

2 Related Work

Planning motion in the presence of actuation uncertainty dates back to the seminal work of Lozano-Pérez et al. [1], which introduced pre-image backchaining. A pre-image, i.e. a region of configuration space from which a motion command attains a certain goal recognizably, was used in a planner that produced actions guaranteed to succeed despite pose and action uncertainty. However, constructing such pre-images is prohibitively computationally expensive [4, 5].

In its general form, belief-space planning is formulated as a Partially-Observable Markov Decision Process (POMDP), which are widely known to be intractable for high-dimensional problems. However, recent developments of general approximate point-based solvers such as SARSOP [6] and MCVI [7] have made considerable progress in generating policies for complex POMDP problems. For some lower-dimensional robot motion problems like [8], the POMDP can be simplified by extracting the part of the task that incorporates uncertainty (e.g. the position of an item to be grasped) and applying off-the-shelf solvers to the problem. Others have investigated learning approaches [9] for similar problems; however, we are interested in planning because we want our methods to generalize to a broad range of tasks without collecting new training data.

Several sampling-based belief-space planners have been developed [3, 10–13]. Others have evaluated the belief-space distance functions [14] and show that the selection of distance function greatly impacts the performance of the planner. Additionally, approaches using LQG and LQR controllers [12, 15, 16] and trajectory optimizers [17, 18] have been proposed. These approaches use Gaussian distributions to model uncertainty, but such a simple distribution cannot accurately represent the belief of a robot moving in contact with obstacles, where belief may lose support in one or more dimensions, or the state may become trans-dimensional. Other approaches like [3] use a set of particles to model belief like a particle filter; while we also use a particle-based representation, our approach more accurately captures the behavior of splits and also includes resilience during execution.

The importance of compliance has long been known, with [1] demonstrating the important role of compliance in performing precise motion tasks. Sampling-based motion planning for compliant robots has been previously explored in [19], albeit limited to disc robots with simplified contact behavior. We draw from these methods, but our approach differs significantly from previous work by incorporating contact and compliance directly into the planning process by using forward simulation like the kinodynamic RRT [20]. A major advantage over existing methods is that the policies we generate are not fixed; instead, we update them online during execution, which allows us to reduce the impact of differences between our planning models and real-world execution conditions.

3 Problem Statement

We consider the problem of planning motion for a *controlled compliant* robot R with configuration space $\mathcal{Q}$ in an environment with obstacles E. For given start

(q_{start}) and goal (q_{goal}), we seek to produce motion which allows the robot to reach q_{goal} within tolerance ϵ_{goal} with at least P_{goal} probability.

The robot is assumed to have actuation uncertainty modelled by $q_{t+1} = q_t + (\Delta q_t + r_{\Delta q})$ in which the next configuration q_{t+1} is the result of the previous configuration q_t, control input Δq and actuation error $r_{\Delta q}$. We assume that a function $\mathbf{F}$, which models the probability distribution of the uncertainty, is available from which to sample $r_{\Delta q} \sim \mathbf{F}(\Delta q)$ for a given Δq. Due to this actuation uncertainty, when executing actions in our planner the result is a belief distribution b. The robot is compliant, meaning that for a motion from collision-free $q_{current}$ to colliding $q_{desired}$, the resulting configuration q_{result} will be in contact and the robot will not damage itself or the environment.

Since the motion of the robot is uncertain, a path τ that is a discrete sequence of configurations may not be robust to errors. Instead, we wish to produce a partial policy $\pi : \mathcal{Q}' \rightarrow A$ that maps $\mathcal{Q}' \subseteq \mathcal{Q}$ to actions A such that for a configuration $q \in \mathcal{Q}'$, the policy returns an action to perform. Even π may not always be robust to unexpected errors, therefore during execution we wish to detect actions that do not reach their expected results; i.e. when an action produces $q_{result} \notin \mathcal{Q}'$. In such an event, we wish to adapt $\mathcal{Q}'$ and π such that $q_{result} \in \mathcal{Q}'$ and continue attempting to complete the task.

4 Methods

We have developed a motion planner consisting of an anytime RRT-based global planner and a local planner that uses a kinematic simulator to model robot behavior. Together, they produce a set of solution paths S, where each solution $s \in S$ is a sequence of nodes $n_i = (b_i, a_i)$, in which b_i is the belief distribution for n_i and a_i is the action that produced b_i. Using this set of solution paths, we construct a single partial policy π. As π is queried during execution, we update the policy to reflect the "true" state observed during the execution process.

Because it is difficult to model the belief state in contact using a parametric distribution, we use a particle-based approach similar to [3] in which we represent the belief b_i of node n_i with a set of configurations $q_1, q_2, ..., q_n$ that are forward-simulated by the local planner. Like previous work [3], we expect that performing some actions will result in multiple qualitatively different states as illustrated in Figure 1 (e.g. in contact with an obstacle some particles will become stuck on the obstacle while others slide along the surface). These distinct parts of the belief state, which we refer to as *splits*, are distinguished in our planner by a series of clustering operations. To ensure that all actions are adequately modeled, a fixed number of particles $N_{particles}$ is used to simulate every action; since splits reduce the number of particles at a given state, a new set of particles must be resampled for these states to avoid particle starvation.

It is important to understand that we cannot select between the different result states of a split when performing the action; however, we can *distinguish* using our clustering methods if we have reached an undesirable result. To be *resilient* to such errors, we incorporate the ability to reverse the action back

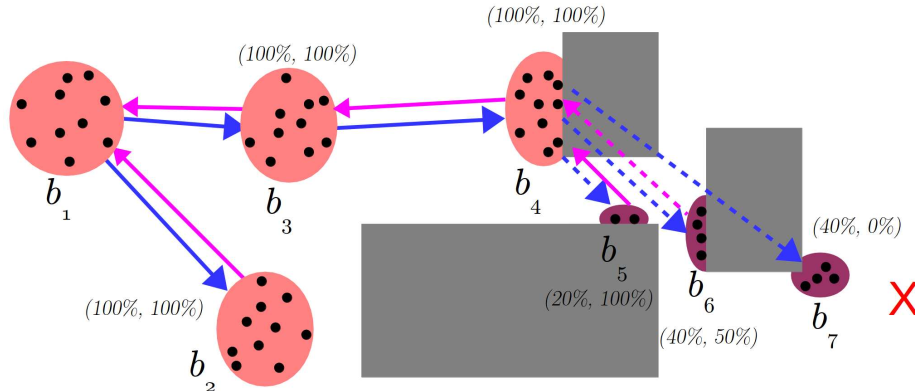


Fig. 1. Our belief-space RRT extending toward a random target (red X) from b_4. Due to compliance, the particles (dots) can slide along the obstacles (gray). Solid blue edges denote 100% probability edges, dashed edges denote a split resulting in multiple states; solid magenta edges denote 100% reversible edges, while dashed edges denote lower reversibility. Because the extension is attempting to move through a narrow passage, particles separate and a split occurs, resulting in three distinct states (b_5, b_6, b_7).

to the previous state and try the action again. Clearly, not all actions will be reversible, so we perform additional simulation to estimate the ability to reverse each action after identifying the resulting states.

We first introduce our RRT-based global planner that uses a simulation-based local planner and a series of particle clustering methods to generate policies incorporating actuation uncertainty, and then discuss our online policy execution and adaptation that enables resiliency to unexpected behavior encountered during execution.

4.1 Global planner

Until it reaches time limit $t_{planning}$, our global planner iteratively grows a tree T using the local planner to extend the tree towards a sampled configuration q_{target}. Like the RRT, q_{target} is either a uniformly sampled $q_{rand} \in \mathcal{C}$, or with some probability, the goal q_{goal}. Each time we sample a q_{target}, we select the closest node in the tree $n_{near} = \text{argmin}_{n_i \in T} \textsc{Proximity}(b_i, q_{target})$. The local planner plans from n_{near} towards q_{target} and returns new nodes $\mathcal{N}_{new}$ and edges $\mathcal{E}_{new}$ that grow the tree. We check each new node $n_{new} \in \mathcal{N}_{new}$ to see if it meets the goal conditions, and if so, add the new solution path to S.

We also incorporate several features distinct from the RRT. First, using Proximity we consider more than distance when selecting the nearest neighbor node n_{near}. We want to bias the growth of the tree toward nodes that can be reached with higher probability and have more concentrated b_i, so we incorporate weighting using $P(n_{start} \rightarrow n_i)$, the probability the entire path from n_{start}

to n_i succeeds, and $\mathrm{var}(n_i)$, the variance of b_i. The proximity of a node n_i to a configuration q is given by the following equation:

$$\begin{aligned}\textsc{Proximity}(n_i, q) &= \mathrm{dist}(\mathrm{expect}(b_i), q) \\ &* [(1 - P(n_{start} \rightarrow n_i)) * \alpha_P + (1 - \alpha_P)][\mathrm{erf}(|\mathrm{var}(b_i)|_1) * \alpha_V + (1 - \alpha_V)] \quad (1)\end{aligned}$$

Here, $\mathrm{expect}(b_i) = q_{expected}$ is the expected value of the belief distribution b_i and $\mathrm{dist}(q_{expected}, q)$ is the $\mathcal{C}$-space distance function. Two weights α_P and α_V control the effect of the probability and variance weighting, respectively. Values of α_P and α_V closer to 1 increase the effect of the weighting, while values closer to 0 increase the effect of the $\mathcal{C}$-space distance.Using the error function $\mathrm{erf}(x) = 2/\sqrt{\pi} \int_0^x e^{-t^2} dt$ maps variance in the range $[0, \inf)$ to the range $[0, 1)$ to simplify computation. Previous work in belief-space planning has used a range of distance functions, such as L1, Kullback-Leibler divergence, Hausdorff distance, or Earth Mover's Distance (EMD) [14]; however, many of these choices only provide useful distances between belief states with overlapping support. While EMD encompasses both the C-space distance and probability mass of two beliefs, it is expensive to compute. Since most of our distance computations are between beliefs with non-overlapping support, the $\mathcal{C}$-space distance between expected configurations is an efficient approximation [14].

Second, we cannot simply test if $n_{new} = q_{goal}$, since the $P(n_{start} \rightarrow n_{new})$ may be low; instead, we check if a new solution has been found. To be a solution, the probability n_{new} reaches the goal must be greater than P_{goal}, i.e. the product of $P(n_{start} \rightarrow n_{new})$ and $|q \in b_{new}|\mathrm{dist}(q, q_{goal}) \leq \epsilon_{goal}|$. Finally, once a path to the goal has been found, we continue planning to find alternative paths. We want to encourage a diverse range of solutions, so once a solution path has been found, we remove nodes on solution branches from consideration for nearest neighbor lookups. This process recurses towards the root of the planner's tree T until it either reaches the root node n_{start} or a node n_i which is the result of a split. Once the base of the solution branch is found, we remove the branch from nearest neighbors consideration and continue planning until reaching $t_{planning}$.

4.2 Local planner

Our local planner grows the planner tree T from nearest neighbor node n_{near} towards a target configuration q_{target} by forward-propagating belief using Extend to produce one or more result nodes $n_{new} \in \mathcal{N}_{new}$ and edges $e_{new} \in \mathcal{E}_{new}$ (recall that splits may occur). To improve the time-to-first-solution, the local planner operates like RRT-Connect, repeatedly calling Extend, until a solution is found, whereupon it switches to RRT-Extend, calling Extend only once, to improve coverage of the space and encourage solution diversity. Note that the RRT-Connect behavior is stopped if an extension results in a split.

Extend forward-simulates particles $Q_{initial}$ from node n_{near} towards q_{target}, clusters the resulting particles $Q_{results}$ into new nodes $\mathcal{N}_{new}$, and computes the transition probabilities. As previously discussed, we simulate every action with

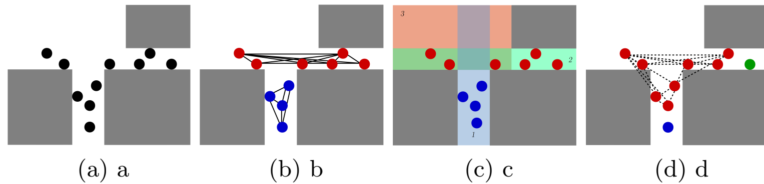
the same number of particles. If node n_{near} is *not* the result of a split, and thus b_{near} contains a full set of particles, then we simply copy b_{near} to use for simulation. If, instead, n_{near} is the result of a split, then we uniformly resample $N_{particles}$ particles from b_i. We then simulate the extension toward q_{target} for each particle. Any simulation engine that simulates contact and compliance could be used, but the simulation should be as fast as possible to minimize planning time. In our experiments, we used an approximate kinematic simulator described in Appendix A. The resulting particles are then grouped into one or more clusters using ClusterParticles, which we describe in Section 4.3. For each cluster $Q_{cluster}$, we form a new node $n_{new} = (b_{new}, a_{new})$ with belief $b_{new} = Q_{cluster}$ and action $a_{new} = q_{target}$. In the case of splits, where multiple nodes are formed, we assign $P(n_{near} \rightarrow n_{new,i}) = |b_{new,i}|/N_{particles}$. We then estimate the probability that action a_{new} can be reversed from node n_{new} by simulating $N_{particles}$ particles back towards node n_{near}. Note that some particles may become stuck while reversing, and thus the probability of reversing the action may not be 1.

The ability to reverse an action allows us to detect an undesired outcome, reverse to the parent node, and retry the action until we either reach the desired outcome or become stuck. Thus, we estimate the *effective* probability $P(n_{near} \rightarrow n_{new})_{effective}$ for each node n_{new} by estimating the probability that a particle has reached n_{new} after $N_{attempt}$ attempts, where at each attempt, particles that have not reached the n_{new} try to return to n_{near} and try again.

Analysis – The planner always stores $N_{stored} = N_{actions} N_{particles}$ particles. For every action $N_{particles}$ particles are forward-simulated, and all of them are stored in $\mathcal{N}_{new}$. In the worst case, where every action produces $N_{particles}$ distinct nodes, the number of particles that must be simulated $N_{simulated} = N_{nodes}(N_{particles} + 1)$, as each node itself is the product of one initial simulation and $N_{particles}$ simulations are required to estimate reverse probability. In practice, as we discuss in Section 5.1, the space requirements to perform complex tasks are low as most actions produce a small number of nodes, and the time cost can be reduced by simulating particles in parallel.

4.3 Particle clustering

Intuitively, we want every configuration in a cluster to be reachable from every other one using the local planner. However, testing this directly is computationally expensive, so we also consider two approximate methods. All clustering methods use two successive passes to cluster the configurations resulting from forward simulation: first, a spatial-feature-based pass that groups configurations based on their relationship to different parts of the workspace, and second, a distance-based pass that refines the initial clusters. All of our clustering methods use complete-link hierarchical clustering, as it produces smaller, more dense clusters, while not requiring the number of clusters to be known in advance [21, 3]. Below we discuss the ideal approach and our two approximations, shown in Figure 2. We compare the performance of these methods in Section 5.1.



(a) a (b) b (c) c (d) d

Fig. 2. Our proposed spatial-feature particle clustering methods. (a) The positions of particles after an extension of the planner. (b) Actuation center clustering, with clusters (red, blue) and the straight-line paths for each cluster. (c) Weakly Convex Region Signature clustering, with the three convex regions shown and labeled. (d) Particle movement clustering, with successful particle-to-particle motions shown dashed for the main cluster (red) and two unconnected particles (blue, green).

Particle Connectivity (PC) Clustering We run the local planner from every configuration to every other configuration and record which simulations reach within ϵ_{goal} of the target. For a pair of configurations q_1, q_2, going from q_1 to q_2 may fail while the opposite succeeds; however, to be conservative, we only record success if both executions succeed. We then perform clustering using the complete-link clustering method with distance threshold 0, where successful simulations correspond to distance 0 and unsuccessful simulations correspond to distance 1. Note that this method is very expensive, since it requires simulating $N^2 - N$ particles for N configurations considered.

Weakly Convex Region Signature (WCR) Clustering Intuitively, in many environments a robot can move freely from q_1 to q_2 if both configurations reside entirely in the same convex region of the workspace. This is also true for some slight concave features, so long as the features do not block the robot. Conversely, for configurations in clearly distinct regions, it is less likely that the robot can move from one configuration to the other.

Illustrated in Figure 2c, we capture this intuition by recording the position of the robot relative to *weakly convex regions* of the free workspace, to form what we call the *convex region signature*. These regions form a weakly convex covering: individual regions may contain slight concavity, and multiple regions overlap. Techniques such as [22] exist to automatically compute these regions, but for simple environments these regions can be directly encoded. The convex region signature of a configuration q, $WCR(q)$, records the region(s) occupied by every point of the robot at q. Distance metric D_{WCR} between two region signatures $WCR(q_1)$ and $WCR(q_2)$ is the percentage of points in the robot that do *not* share a common region between the signatures. Using this metric, we perform complete-link clustering. We test different thresholds for D_{WCR} in Section 5.1. This method allows configurations with some points in a shared region to be clustered together, while separating configurations that share no regions. At runtime, this method requires N computations of $WCR(q)$ and ${(N^2-N)}/{2}$ evaluations of D_{WCR} to compute all pairwise distances.

Actuation Center (AC) clustering We observe that many successful motions in contact occur when the actuation (or joint) centers of the starting and ending configuration can be connected by collision-free straight lines, so this method checks the straight-line path from the joint centers of one configuration to those of the other configuration. As with the particle movement clustering approach, configurations with successful (collision-free) paths have distance 0, while those with unsuccessful (colliding) paths have distance 1. Like the previous approach, clusters are then produced using complete-link clustering with threshold 0. At runtime, this method requires $(N^2-N)/2$ checks of the straight-line paths.

4.4 Partial policy construction

Once the global planner has produced a set of solution paths S, we construct a partial policy π. Policy construction consists of the following steps:

1. Graph construction – An explicit graph is formed, in which the vertices of the graph are nodes $n_i \in S$, and the edges correspond to the edges forming the paths in S. An edge $n_i \rightarrow n_{i+1}$ is assigned an initial cost $1/P(n_i \rightarrow n_{i+1}))$. This means that likely edges receive low cost, which is necessary to compute maximum-probability paths through the graph.
2. Edge cost updating – The edge costs are updated to reflect the estimated number of attempts needed to successfully traverse the edge by multiplying the cost of the edge by the estimated number of attempts required to reach $P(n_i \rightarrow n_{i+1}) \geq P_{goal}$. This estimate is the complement of the effective probability discussed in Section 4.2; instead of computing the probability of reaching a node after a fixed number of attempts, we compute the number of attempts needed to reach the node with P_{goal} probability. The fewer attempts necessary to traverse the edge, the faster the policy can be executed, and thus this cost represents an expected execution time.
3. Dijkstra's search – The optimal path from every vertex in the graph to the goal state is computed using Dijkstra's algorithm. This determines the optimal next state (and thus action to perform) for every state in the graph.

4.5 Partial policy execution and adaptation

At every step during execution, the partial policy π is queried for the next action to perform. While we could simply find the "closest" node in the policy using a distance function like Equation 1, doing so would discard important information. Not only do we know the configuration $q_{current}$ that results from executing an action, but we also know the action $a_{performed}$ we attempted to perform. Using this information, we know exactly which nodes(s) in π the robot should have reached. As shown in Algorithm 1, we first collect all potential result nodes (i.e. those nodes n_i with actions $a_i = a_{performed}$). We then use our particle clustering method to cluster $q_{current}$ with the belief b_i of each n_i. This clustering tells us if the robot reached a given state (if a single cluster is formed) or not (multiple clusters). In the unlikely (but possible) event that $q_{current}$ clusters with multiple

Algorithm 1 Partial policy query algorithm

```
procedure PolicyQuery(S, π, q_current, a_performed)
    N_potential ← {n_i ∈ S | a_i = a_performed}
    N_matching ← {n_i ∈ N_potential | |ClusterParticles(b_i ∪ q_current)| = 1}
    if N_matching ≠ ∅ then
        n_reached ← argmin_{n_i ∈ N_matching} DijkstraDistance(n_i)
        IncreaseProbability(n_reached, a_performed);
        for n_i ∈ N_potential | n_i ≠ n_reached do
            ReduceProbability(n_i, a_performed)
        π ← ConstructPolicy(π)
        if P(n_reached → q_goal) ≥ P_goal then
            a_next ← π(n_reached)
            return a_next
        else
            return failure
    else
        n_observed ← {{q_current}, a_performed}
        S ← S ∪ n_observed
        return PolicyQuery(S, π, q_current, a_performed)
```

potential result nodes, we select the "best" matching node $n_{reached}$ using the distance-to-goal computed via Dijkstra's algorithm.

The key contribution of our policy execution is that we adapt the policy π to reflect the results of actual execution. If a matching node $n_{reached}$ is found, we then update π to increase the probability that $n_{reached}$ is the result of $a_{performed}$. We assign a constant $A_{importance} \in \mathbb{N}$ that reflects how much we value the results of executing an action compared to the results of simulating a particle during planning. To update the probability, we increase the counts of attempted $N_{attempts}$ actions and successful $N_{successful}$ actions, then recompute probability:

$$P(n_{previous} \rightarrow n_{reached}|a) = \frac{N_{successful} + A_{importance}}{N_{attempts} + A_{importance}} \tag{2}$$

Likewise, we reduce the probability for other potential result states:

$$P(n_{previous} \rightarrow n_{other}|a) = \frac{N_{successful}}{N_{attempts} + A_{importance}} \tag{3}$$

This update process allows us to learn online, during execution, the true probabilities of reaching states given an action. In effect, the probabilities computed by the global planner serve as an initialization for this online learning. Once updated, we rebuild policy π to reflect the new probabilities. If the probability of reaching the goal $P(n_{reached} \rightarrow q_{goal})$ is at least P_{goal}, we query π for the next action to take. If the probability of reaching the goal has dropped below P_{goal}, policy execution terminates.

However, sometimes no matching node $n_{reached}$ exists. This means a split occurred during execution that was not captured in S during planning (e.g. an obstacle that is not accurately modelled in E, or where the behavior of the simulator diverges from the true robot). To handle this case, we insert a new node $n_{observed}$ with belief $b_{observed} = \{q_{current}\}$ into S, and then retry the policy query (which will now have an exactly matching state). To incorporate reversibility, we initially assign new nodes a reverse probability $N_{attempts} = N_{successful} = 1$. Thus, the next action selected by the policy will be to return to the previous node. Together with updating probabilities by inserting new states in this manner, we can thus extend the policy to reflect behavior observed during execution that was not captured during the planning process.

Analysis – In the worst case, a policy π cannot be executed successfully, and performing every action a results in a new node $n_{observed}$. For any $A_{importance} \in \mathbb{N}, P_{goal} > 0$, adapting the policy will detect failure and terminate in this case.

Proof – For every action $a_{i+1}, ...,$ node $n_{observed}$ will be created with a reverse prior $P(n_{observed} \rightarrow n_{previous}) = {}^{1}/_{1}$. If reversing to $n_{previous}$ fails, we update $P(n_{observed} \rightarrow n_{previous}) = {}^{1}/_{1+A_{importance}}$. For the ith successive failed reverse and $n_{observed,i}$ generated, $P(n_{observed,i} \rightarrow n_{previous}) = \Pi_i \frac{1}{1+A_{importance}}$. As the number of failed actions increases $P(n_{observed,i} \rightarrow n_{previous}) \rightarrow 0$, and thus $P(n_{observed,i} \rightarrow q_{goal}) \leq P(n_{observed,i} \rightarrow n_{previous}) \rightarrow 0$. Thus eventually $P(n_{observed,i} \rightarrow q_{goal})$ will fall below $P_{goal} > 0$ and execution will terminate. □

5 RESULTS

We present results of testing our planner in simulated $SE(2)$ and $SE(3)$ environments and a simulated Baxter robot. For dynamic simulation during execution, we use the Gazebo simulator. As our kinematic simulator does not consider friction, we use *contact motion controllers* to reduce contact forces (see Appendix B). We present statistical results over a range of actuation uncertainty and clustering methods and show that our planner produces policies that allow execution of tasks incorporating contact and robot compliance. We also present statistical results showing that our online policy updating adapts to unexpected behavior during execution. All planning and simulation testing was performed using 2.4 GHz Xeon E5-2673v3 processors. Likewise, all planning was performed with Proximity weights $\alpha_P = \alpha_V = 0.75$ (see Equation 1), $N_{attempt} = 50$ attempted reverse/repeats of each action, and planning threshold $P_{goal} = 0.51$, such that solutions must be more likely than not to reach the goal.

5.1 $SE(3)$ simulation

Peg-in-hole In $SE(3)$*PegInHole*, a version of the classical peg-in-hole task [1] shown in Figure 3, the free-flying 6-DoF robot "peg" must reach the bottom of the hole. This task is difficult for robots with actuation uncertainty, as the hole is only 30% wider than the peg. Even without uncertainty, attempting to avoid contact greatly restricts the motion of the robot entering the hole. Instead, as

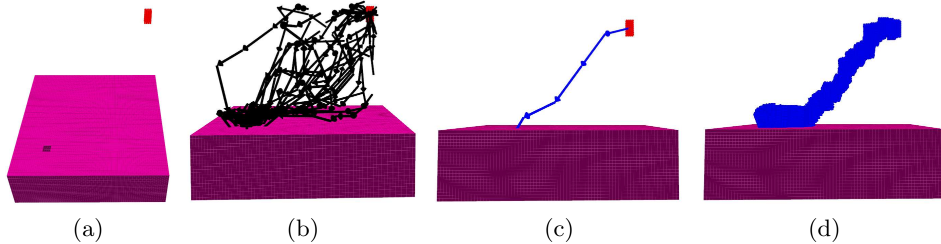


Fig. 3. (a) The $SE(3)PegInHole$ task involves moving from the start (red) to the bottom of the hole. (b) An example policy produced from 296 solutions, the (c) initial action sequence (blue arrows), actions the policy will return if every action is successful, and (d) the swept volume of the peg executing the policy. Note that the peg makes contact with the environment to reduce uncertainty, then slides into the hole.

	AC	PC	WCR with D_{WCR} =				
			0.125	*0.25*	*0.5*	*0.75*	*0.99*
P_{plan}	1.0	1.0	1.0	0.97	0.97	0.97	0.97
P_{exec}	0.97 [0.17]	0.89 [0.19]	0.73 [0.42]	0.95 [0.18]	0.84 [0.34]	0.99 [0.02]	0.96 [0.18]

Table 1. $SE(3)PegInHole$ particle clustering performance comparison (mean [std.dev.]) of P_{exec}, the probability of reaching the goal with 300 seconds, between policies produced using our planner with different clustering methods. P_{plan} is the probability that a policy is planned within 5 minutes, averaged over 30 plans, and P_{exec} is averaged over 40 executions on each successfully-planned policy.

	Simplified				Planned policies (WCR, $D_{WCR} = 0.75$)			
	Simple RRT		Contact RRT		24 particles		48 particles	
γ	P_{plan}	P_{exec}	P_{plan}	P_{exec}	P_{plan}	P_{exec}	P_{plan}	P_{exec}
0	0	0 [0]	1	0.78 [0.38]	1	0.42 [0.48]	0.97	0.59 [0.47]
1/16	0	0 [0]	1	0.78 [0.39]	1	0.60 [0.43]	1	0.625 [0.43]
1/8	0	0 [0]	1	0.79 [0.38]	1	0.99 [0.18]	0.93	0.81 [0.37]
1/4	0	0 [0]	1	0.50 [0.37]	1	0.90 [0.28]	0.86	0.72 [0.41]

Table 2. $SE(3)PegInHole$ policy performance comparison between simplified planners and our planner with 24 and 48 particles and actuation uncertainty γ.

shown in [1], the best strategy is to use contact with the environment and the compliance of the robot to guide the peg into the hole. We assess the performance of a policy approach in terms of P_{exec}, the probability that executing the policy reaches the goal within a time limit of 300 seconds. For a given value of γ, linear velocity uncertainty $\gamma_v = \gamma$ (m/s) and angular velocity uncertainty $\gamma_\omega = 1/4\gamma$ (rad/s). Linear and angular velocity noise is sample from a zero-mean truncated normal distribution with bounds $[-\gamma_{v,\omega}, \gamma_{v,\omega}]$ and standard deviation $1/2\gamma_{v,\omega}$. While this differs from zero-mean normal distributions conventionally used to model uncertainty, we believe the bounded truncated distribution better reflects

the reality of robot actuators, which do not exhibit unbounded velocity error. Goal distance threshold ϵ_{goal} was set to $1/2$ the length of the peg.

We first compared the performance of our planner at a fixed $\gamma = 1/8$ and $N_{particles} = 24$ using the clustering approaches introduced in Section 4.3, including several thresholds for $D_{WCR} = 0.125, 0.25, 0.5, 0.75, 0.99$, with 30 plans per approach (5 minutes planning time) and 40 executions of each planned policy. As seen in Table 1, WCR clustering with $D_{WCR} = 0.75$ clearly outperformed the others in terms of policy success, reaching the goal in 99% of executions. Planning time is overwhelmingly dominated by simulation, accounting for approximately 99.9% of the allotted time. Using WCR and $D_{WCR} = 0.75$, we then compared the performance of our planner against two simplified RRT-based approaches:

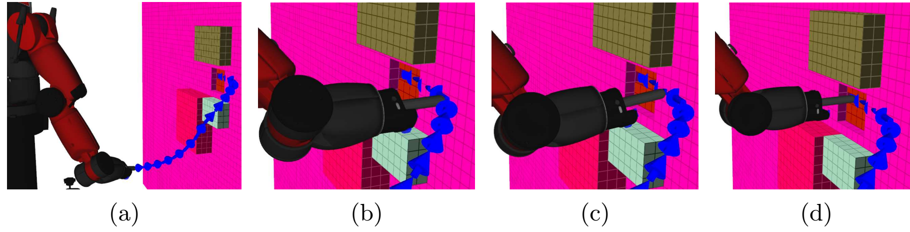
1. Simple RRT – Does not model uncertainty or allow contact, but like our planner produces multiple solutions in the allotted planning time.
2. Contact RRT – Incorporates contact and compliance but does not model uncertainty. Equivalent to planning with $\gamma = 0$ and one particle.

In addition, we tested our planner with both 24 and 48 particles to show the effects of increasing the number of particles used. As before, we planned 30 policies for each, and executed each planned policy 40 times. Note that the Simple RRT was unable to produce solutions in 5 minutes due to the confined narrow passage. Results are shown in Table 2. With low actuator error the Contact RRT performs better, as it does not expend planning time on simulating multiple particles and instead produces more solutions. As error increases, our planner clearly outperforms the alternatives. Note that increasing particles does not improve performance, indicating that 24 particles is sufficient without requiring unnecessary simulation. Low P_{exec} overall, in particular when planning with low γ, is due to the mismatch between the planning simulator and the dynamics of Gazebo (i.e. motions that are possible in the planner, but not in Gazebo) which disproportionally affects motions near the entrance to the hole. In particular, the planner at low values of γ overestimates how successfully motions at the edge of the hole can be performed and thus results in a lower-than-expected P_{exec}.

In terms of the number of particles stored, the worst case was $N_{particles} = 48, \gamma = 0$, with an average of 148894 (std.dev. 25015) particles stored. The worst case for simulated particles was $N_{particles} = 24, \gamma = 1/4$, with an average 268634 (std.dev. 16856) particles simulated. In practice, the storage and computational expense is limited; the worst-case for particles stored requires a mere 15 megabytes, while for a planning time of 300 seconds and using eight threads, the planner evaluated more than 100 particles per second per thread.

5.2 Baxter simulation

In addition to $SE(3)$ and $SE(2)$ tests, we apply our planner and policy execution to a simulated Baxter robot shown in Figure 4, with the robot reaching into a confined space. We compare the performance of our planner with $N_{particles} = 24$ and WCR clustering method with $D_{WCR} = 0.1$ with the simplified Contact



(a) (b) (c) (d)

Fig. 4. An execution of the Baxter task, from start (a) to goal (d), and environment with confined space around the goal. The planned policy is shown in blue. Note the use of contact with the environment to reduce uncertainty and reach the target passage.

RRT in terms of success probability P_{exec} for uncertainty $\gamma = 0.1$. To simulate Baxter's actuation uncertainty γ defines a truncated normal distribution with $\sigma = 1/2\gamma\dot{q}_i$ and bounds $[-\gamma\dot{q}_i, \gamma\dot{q}_i]$ for each joint i with velocity $\dot{q}_i$. Goal distance threshold $\epsilon_{goal} = 0.15$ radians. We generated 10 policies using each approach with a planning time of 10 minutes to ensure both approaches would produce multiple solutions, then executed each 8 times for up to 5 minutes. As expected, Contact RRT finds solutions faster; on average 8.42s (std.dev. 2.61) versus 65.4s (std.dev. 32.9) and policies contain more solutions; on average 17.2 (std.dev. 16.5) versus 6.33 (std.dev. 3.97) since each solution requires less simulation time. However, our planner incorporating uncertainty outperforms the Contact RRT baseline with $P_{exec} = 0.79$ (std.dev. 0.30) versus $P_{exec} = 0.70$ (std.dev. 0.30). This suggests that, while planning with uncertainty does help in this environment, our approach to policy execution and resilience also works well when uncertainty is not accounted for in the planner, but we have a diverse policy.

5.3 Policy adaptation

We performed tests in a planar $SE(2)$ (3-DoF) environment to show that our policy adaptation recovers from unexpected behavior during execution. As shown in Figure 5, the L-shaped robot attempts to move from the start (upper left) to the goal (lower right). Due to the obstacles present, there are three distinct horizontal passages. Using the same controllers and uncertainty models as the $SE(3)$ tests with uncertainty $\gamma = 0.125$ and WCR clustering method with $D_{WCR} = 0.75$, 24 particles, and a planning time of 2 minutes, we generated 30 policies using our planner. Goal distance threshold ϵ_{goal} was set to $1/8$ the length of the robot.

We evaluated the performance of the planned policies in the unmodified environment and an environment in which we blocked the horizontal passage used by the initial path of each policy. Each was executed 8 times for a total of 240 policy executions, for a maximum of 600 seconds. In the unmodified environment, 97% of policies were executed successfully, with an average of 15.4 actions (std.dev. 9.62). In the modified environment with policy execution importance

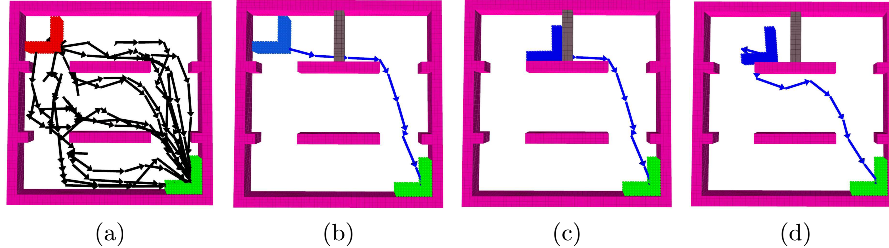


Fig. 5. (a) Our planar test environment, in which the robot must move from upper left (red) to lower right (green), with an example policy produced by our planner, with solutions through each of the horizontal passages. (b) The initial action sequence (blue arrows), showing actions the policy will return if every action is successful. (c) Following the policy, the robot becomes stuck on the new obstacle (gray). (d) Once the policy detects the failed action, it adapts to avoid the obstacle.

$A_{importance} = 500$ (this high value results in rapid policy adaptation), 73% of policies were executed successfully, with an average of 26.7 actions (std.dev. 12.3). This result shows that adapting the policy using our methods allows us to circumvent the new obstacle, however, doing so may result in following a path that is less likely to succeed.

6 Conclusion

We have developed a method for planning motion for robots with actuation uncertainty that incorporates environment contact and compliance of the robot to reliably perform manipulation tasks. First, we generate partial policies using an RRT-based motion planner that uses particle-based models of uncertainty and kinematic simulation of contact and compliance. Second, we adapt planned policies online during execution to account for unexpected behavior that arises from model or environment inaccuracy. We have tested our planner and policy execution in simulated $SE(2)$ and $SE(3)$ environments and on the simulated Baxter robot and show that our methods generate policies that perform manipulation tasks involving significant contact and compare against two simpler methods. Additionally, we show that our policy adaptation is resilient to significant changes during execution; e.g. adding a new obstacle to the environment.

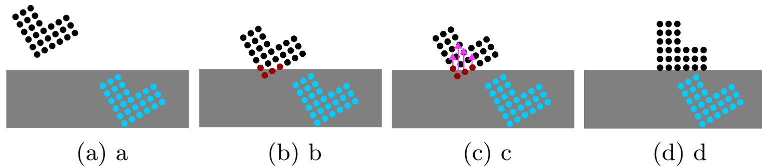
Acknowledgements This work was supported in part by NSF grants IIS-1656101 and IIS-1551219.

References

1. Lozano-Prez, T., Mason, M.T., Taylor, R.H.: Automatic synthesis of fine-motion strategies for robots. IJRR **3**(1) (1984) 3–24

2. Goldman, R.P., Boddy, M.S.: Expressive planning and explicit knowledge. In: Artificial Intelligence Planning Systems. (May 1996)
3. Melchior, N.A., Simmons, R.: Particle rrt for path planning with uncertainty. In: ICRA. (April 2007)
4. Canny, J.: On computability of fine motion plans. In: ICRA. (May 1989)
5. Erdmann, M.: Using backprojections for fine motion planning with uncertainty. The International Journal of Robotics Research **5**(1) (1986) 19–45
6. Kurniawati, H., Hsu, D., Lee, W.S.: Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces. In: RSS. (2008)
7. Bai, H., Hsu, D., Kochenderfer, M., Lee, W.S.: Unmanned aircraft collision avoidance using continuous-state pomdps. In: RSS. (June 2011)
8. Koval, M., Pollard, N., Srinivasa, S.: Pre- and post-contact policy decomposition for planar contact manipulation under uncertainty. In: RSS. (July 2014)
9. Levine, S., Wagener, N., Abbeel, P.: Learning contact-rich manipulation skills with guided policy search. In: ICRA. (May 2015)
10. Roy, N., Prentice, S.: The belief roadmap: Efficient planning in belief space by factoring the covariance. IJRR **28**(11-12) (2009) 1448–1465
11. Bry, A., Roy, N.: Rapidly-exploring random belief trees for motion planning under uncertainty. In: ICRA. (May 2011)
12. Agha-mohammadi, A.a., Chakravorty, S., Amato, N.M.: Firm: Sampling-based feedback motion planning under motion uncertainty and imperfect measurements. The International Journal of Robotics Research (2013)
13. Alterovitz, R., Simon, T., Goldberg, K.: The stochastic motion roadmap: A sampling framework for planning with markov motion uncertainty. In: RSS. (June 2007)
14. Littlefield, Z., Klimenko, D., Kurniawati, H., Bekris, K.E.: The importance of a suitable distance function in belief-space planning. In: ISRR. (September 2015)
15. Berg, J.V.D., Abbeel, P., Goldberg, K.: Lqg-mp: Optimized path planning for robots with motion uncertainty and imperfect state information. In: RSS. (June 2010)
16. Huynh, V.A., Karaman, S., Frazzoli, E.: An incremental sampling-based algorithm for stochastic optimal contro. In: ICRA. (May 2012)
17. Davis, B., Karamouzas, I., Guy, S.J.: C-opt: Coverage-aware trajectory optimization under uncertainty. IEEE Robotics and Automation Letters **1**(2) (July 2016) 1020–1027
18. Lee, A., Duan, Y., Patil, S., Schulman, J., McCarthy, Z., van den Berg, J., Goldberg, K., Abbeel, P.: Sigma hulls for gaussian belief space planning for imprecise articulated robots amid obstacles. In: IROS. (Nov 2013)
19. Nieuwenhuisen, D., van der Stappen, A.F., Overmars, M.H.: Pushing using compliance. In: ICRA. (May 2006)
20. LaValle, S.M., Kuffner, J.J.: Randomized kinodynamic planning. IJRR **20**(5) (2001) 378–400
21. Sneath, P.H.A., Sokal, R.R.: Numerical taxonomy: the principles and practice of numerical classification. Freeman (1973)
22. Asafi, S., Goren, A., Cohen-Or, D.: Weak convex decomposition by lines-of-sight. Computer Graphics Forum **32**(5) (2013) 23–31

Appendix A Fast kinematic simulation



(a) a (b) b (c) c (d) d

Fig. 6. The collision resolution process used by our lightweight simulator. From left to right, (a) a robot represented by points (black) moving towards a target (light blue) and an obstacle (gray) (b) collides with the object, triggering the collision resolution. (c) point corrections Δp_n for each colliding point of the robot are computed from the surface normals of the object and applied (d) so the robot complies to produce an in-contact state.

The global and local planners rely on the presence of a computationally-efficient simulator for the behavior of our controlled compliant robot. While progress has been made in the performance of dynamic simulators, we require a simulator capable of evaluating hundreds, if not thousands, of robot motions per second to grow the planner's tree in reasonable time. To improve computational performance, we limit ourselves to kinematic simulation, though our planning framework is agnostic to the simulator being used. Kinematic simulation is only an approximation of true robot motion; however, we mitigate the discrepancy between simulated and real dynamic behavior using policy adaptation to update the planned policy with the results of real-world executions. For this work, the robot is controlled via PD feedback controller with gains K_p, K_d, for error $e_q = q_{desired} - q$ the resulting control input is $\Delta q = K_p e + K_d \dot{e_q}$; however, different controllers can be used with the simulator. For a fixed time limit $t_{simulate}$, we forward simulate the motion of the robot from the current configuration q_t to the next configuration q_{t+1} using the equation below.

$$e_q = q_{target} - q_t \tag{4}$$

$$\Delta q = K_p e_q + K_d \dot{e_q} \tag{5}$$

$$q_{t+1} = q + \Delta q + \mathbf{F}(\Delta q) \tag{6}$$

$$q'_{t+1} = \textsc{ResolveCollisions}(q_{t+1}) \tag{7}$$

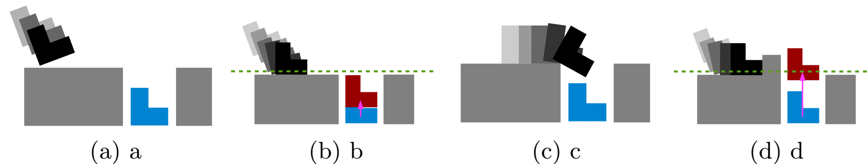
Collision resolution and robot compliance are modelled by ResolveCollisions, which iteratively corrects colliding configurations q_{t+1} until an in-contact configuration q'_{t+1} is reached. For performance purposes, the environment E is modelled using a voxel grid that stores the surface normals for all obstacles in E, and the robot R is modelled using a set of points for each link. Collision checking of a configuration q is performed by transforming every point of the robot into the

environment and checking if any of the corresponding voxels belong to an obstacle. If any voxels belong to an obstacle, the collision is resolved by iteratively applying corrections Δq as shown in Figure 6. Each correction Δq is the product of the individual point corrections Δp_n for each colliding point, where Δp_n is the product of the surface normal of the collided obstacle and penetration distance of point p_n, and the Jacobian $\mathbf{J}$ pseudoinverse of the robot for configuration q and point p_n as shown below:

$$\Delta q = \mathbf{J}(q, p_1, p_2, ...)^+ [\Delta p_1^T, \Delta p_2^T, ...]^T \tag{8}$$

Intuitively, this computes the change in configuration necessary to move points $p_1, p_2, ...$ out of collision, where the correct direction to move of out of collision is approximated by the surface normal of the collided object. Note that this approximation is only valid if the maximum penetration of an obstacle is small; thus we use small timesteps in both ForwardSimulate and ResolveCollisions to ensure that the workspace motion of the robot is small. While our kinematic simulation does not consider surface friction which could hamper sliding motions, we address this using a simple controller discussed in Section B and our simulation results show that this limitation does not overly impair the performance of our planner, though unexpected jamming could still occur.

Appendix B Execution controllers



(a) a (b) b (c) c (d) d

Fig. 7. Our contact motion controller helps mitigate the effects of contact friction. (a) The robot approaches contact while moving towards the goal in blue. (b) The robot makes contact and becomes stuck on the surface, from which we estimate a plane (green) that locally approximates the surface and adjust the goal by ϵ_{adjust} shown in magenta to reduce contact force until (c) the robot resumes moving. (d) Alternatively, the robot remains stuck for i iterations until $i\epsilon_{adjust} = 1$ and the controller terminates.

We use the Gazebo dynamics simulator in both planar and 3D environments to simulate execution of robot motion including contact with obstacles. In the kinematic simulator used during planning, a PD position controller attempts to reach a target configuration q_{target} by commanding velocities to the robot. Likewise, we control the simulated robot in Gazebo using a position controller

that receives q_{target} and commands velocities. To safely achieve those velocities in collision and contact, a velocity controller commands forces and torques that move the simulated robot. Unlike the kinematic simulator, which ignores friction and dynamic effects to achieve faster runtime, the dynamic simulator incorporates friction between the robot and the environment. To mitigate the effects of friction in execution, we use a *contact motion controller* illustrated in Figure 7 which adjusts q_{target} to reduce contact forces that cause the robot to become stuck.

When the contact motion controller receives a new target position, it first commands q_{target} without modification. For the duration of execution t_{exec}, at each iteration the controller records the trajectory of the robot and checks if the robot has become stuck, i.e. if the total motion over a sliding window of the trajectory is below a threshold ϵ_{stuck}. If the robot is stuck, we assume that the surface on which the robot is stuck can be locally approximated as a plane, which we can estimate from the recent motion of the robot. Once the robot is stuck, the controller then fits a plane defined by point P_{plane} and normal vector $\overrightarrow{N_{plane}}$ to the sliding window of the trajectory and projects q_{target} towards the plane:

$$q'_{target} = q_{target} + \left(\frac{\overrightarrow{q_{target}, P_{plane}} \cdot \overrightarrow{N_{plane}}}{\overrightarrow{N_{plane}} \cdot \overrightarrow{N_{plane}}} \overrightarrow{N_{plane}}\right)(i\epsilon_{adjust}) \tag{9}$$

Here, on the ith stuck iteration of the controller (i.e. the robot has be stuck for i consecutive iterations of the controller), the controller computes $\overrightarrow{q_{target}, P_{plane}}$, the vector from q_{target} to P_{plane}, and projects it onto $\overrightarrow{N_{plane}}$ to compute an adjustment vector. The target configuration is then moved along the adjustment vector towards the plane by $i\epsilon_{adjust}$, where ϵ_{adjust} is the amount to adjust at each step. If the controller exceeds the time limit t_{exec} or $i\epsilon_{adjust} \geq 1$, the controller reports that the robot has become "completely stuck". Intuitively, this controller reduces contact forces (and thus the effects of friction) by moving q_{target} towards the surface of the obstacle approximated by fitting a plane to the trajectory. If the robot continues to move, or if the robot resumes moving after being stuck, the controller commands the original q_{target}. For both $SE(2)$ and $SE(3)$ simulation tests, we used $\epsilon_{adjust} = 0.01$ (i.e. it will attempt 100 stuck iterations before terminating the motion).

A structurally similar contact motion controller is used with the Baxter robot; however, instead of fitting a plane in $R(7)$ and projecting the target towards it, we use the kinematic simulator to predict the next adjusted target configuration. At a stuck configuration q_{stuck}, we forward-simulate using the kinematic simulator towards q_{target} for a brief timestep, and record the resulting configuration $q_{simulated}$. We then interpolate between q_{target} and $q_{simulated}$ by $i\epsilon_{adjust}$ to produce the new adjusted target q'_{target}.