Sequential Quadratic Programming for Task Plan Optimization

Dylan Hadfield-Menell[2], Christopher Lin[1], Rohan Chitnis[1], Stuart Russell[2], and Pieter Abbeel[2]

***Abstract*— We consider the problem of refining an abstract task plan into a motion trajectory. Task and motion planning is a hard problem that is essential to long-horizon mobile manipulation. Many approaches divide the problem into two steps: a search for a task plan and task plan refinement to find a feasible trajectory. We apply sequential quadratic programming to jointly optimize over the parameters in a task plan (e.g., trajectories, grasps, put down locations). We provide two modifications that make our formulation more suitable to task and motion planning. We show how to use movement primitives to reuse previous solutions (and so save optimization effort) without trapping the algorithm in a poor basin of attraction. We also derive an early convergence criterion that lets us quickly detect unsatisfiable constraints so we can re-initialize their variables. We present experiments in a navigation amongst movable objects domain and show substantial improvement in cost over a backtracking refinement algorithm.**

I. INTRODUCTION

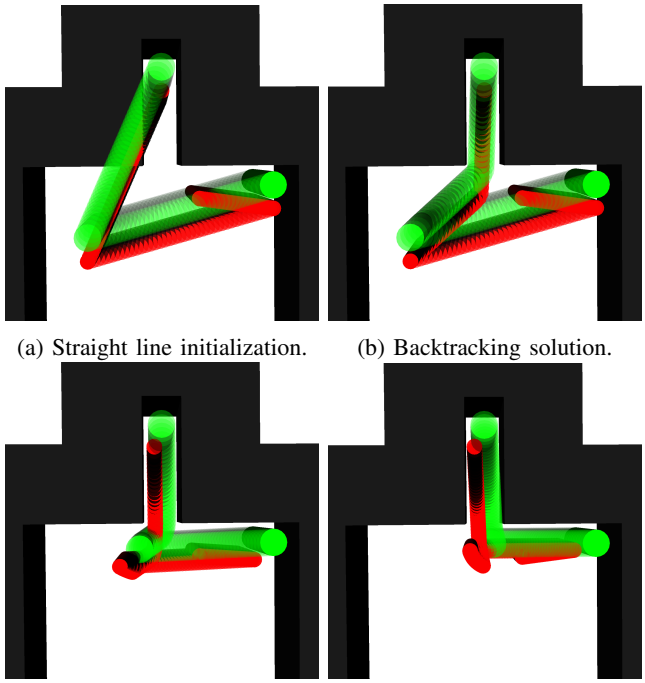
Long-horizon mobile manipulation planning is a fundamental problem in robotics. Viewed as trajectory optimization, these problems are wildly non-convex and direct motion planning is usually infeasible. Viewed as a classical planning problem, there is no good way to represent the geometry of the problem efficiently in a STRIPS or PDDL representation.

The robotics and planning communities have studied the problem of *task and motion planning* (TAMP) as a way to overcome these challenges. TAMP integrates classical task planning methods, that can handle long horizons, with motion planning approaches, that can handle complex geometry. Recent years have seen a variety of approaches to finding feasible task and motion plans [1], [2], [3].

The approach to TAMP in [1] relies on three components: a black box classical planner that ignores geometry to find an abstract task plan, a black box motion planner that can determine motion plans for a given abstract action, and an interface that shares information between the two different planners. Task plans consist of bound object references (e.g., can_1) and unbound pose references (e.g., pose_1). Pose references are continuous parameters that are characterized by a set of constraints. For example, a task plan may require that pose_1 be a grasping pose for can_1.

The process of motion planning for an abstract plan is called *plan refinement*. If plan refinement for a given task plan fails, the interface updates the task planner with information that lets it plan around the failure. In this work, we contribute a novel method for the task plan refinement component of this system. Our approach has applications to systems that use a similar decomposition and as a trajectory smoother for general TAMP algorithms.

Current approaches to task plan refinement rely on a backtracking search over the parameters of the plan and solve a sequence of independent motion planning problems. We propose an approach that jointly optimizes over all of the parameters and trajectories in a given abstract plan. This leads to final solutions with substantially lower cost, when compared with approaches that compute motion plans for each high level action independently. Figure 1 shows an example that compares the result from joint optimization with the result from a backtracking search.



(a) Straight line initialization. (b) Backtracking solution. (c) Intermediate Solution. (d) Final Solution.

Fig. 1: The robot, shown in red, moves a green can to the goal location. The backtracking solution samples and fixes a trajectory waypoint. This leads to an unnecessarily long path. (c) and (d) show an intermediate and final trajectory computed by running sequential quadratic programming on the task plan.

The optimization problems we consider are highly non-convex. We rely on randomized restarts to find solutions: if we fail to converge, we determine variables associated with infeasible constraints and sample new initial values. After a fixed budget of restarts, we return to the task planning layer and generate a new task plan. We contribute two algorithmic modifications that facilitate efficient randomized restarts.

The first modification uses a minimum velocity projection [4] of the previous solution to re-initialize trajectories. This preserves the overall global structure of the trajectories without trapping new solutions in the same basin of attraction. The second modification is an early convergence criterion that checks to see if a constraint is likely to be

unsatisfiable. This allows us to restart more frequently and reduces solution time.

Our contributions are as follows: 1) we apply sequential convex programming to jointly optimize over the trajectories and parameters in a plan refinement; 2) we show how to reuse previous solutions without trapping the optimization in a bad basin of attraction; and 3) we show how to do early convergence detection to avoid wasted effort on infeasible plans. We present experiments that compare our approach to a backtracking refinement. Our approach leads a 2-4x reduction in the total path cost of solutions at the cost of a 1.5-3x increase in running time. We verify that our proposed modifications led to reductions in refinement time.

II. RELATED WORK

Related work largely comes from plan-skeleton approaches to task and motion planning. These are approaches that search over a purely discrete representation of the problem and then attempt to refine the task plans they obtain.

Toussaint [3] also considers joint trajectory optimization to refine an abstract plan. In his formulation, the symbolic state from a task plan defines constraints on a trajectory optimization. The system optimizes jointly over all plan parameters and uses an initialization scheme similar to ours. The problems they consider are difficult because the intermediate states are complicated structures that must satisfy stability constraints. In contrast, the problems we consider are difficult because motion planning problems are hard to solve. This leads us to focus on trajectory re-use and early convergence detection.

Lozano-Pérez and Kaelbling [5] consider a similar approach. They enumerate plans that could possibly achieve a goal. For each such abstract plan, they discretize the parameters in the plan and formulate a discrete constraint satisfaction problem. They use an off-the-shelf CSP solver to find a trajectory consistent with the constraints imposed by the abstract plan. Our approach to refinement draws on this perspective, but we do not discretize the plan parameters; instead, we use continuous optimization to set them.

Lagriffoul and Andres [6] define the fluents in their task planning formulation in a similar way to ours. They use these constraint definitions to solve a linear program over the plan parameters. They then use this LP to reduce the effort of a backtracking search for plan refinement. This is similar to the first initialization step that we and [3] use, in that it only considers the intermediate states.

III. TRAJECTORY OPTIMIZATION WITH SEQUENTIAL QUADRATIC PROGRAMMING

Our approach uses sequential quadratic programming to do task plan refinement. In this section, we describe the motion planning algorithm from [7], which applies sequential quadratic programming to motion planning.

Motion Planning as Constrained Trajectory Optimization

A core problem in robotics is *motion planning*: finding a collision-free path between fixed start and goal poses. A motion planning problem is defined by:

- a *configuration space* of robot poses
- a set of obstacles O
- an initial and goal configuration.

We define configuration spaces by a set of feasible robot poses $\mathcal{X}$ and a dynamics constraint. The dynamics constraint is a Boolean function $f : \mathcal{X} \times \mathcal{X} \rightarrow \{0, 1\}$. It takes as input a pair of poses p_1, p_2 and is 1 iff p_2 is directly reachable from p_1.

Figure 1 shows a 2D motion planning problem that will serve as the starting point for a running example. The pose of the robot is represented by a pair (x,y). We let $\mathcal{X}$ be a bounding box so $x \in [0, 7]$ and $y \in [-2, 7]$. The dynamics function ensures that the distance between subsequent states of the trajectory is always less than a fixed constant: $f(p_1,\ p_2) = (p_1 - p_2 < d_{max})$.

There are three main approaches to motion planning that are used in practice: discretized configuration space search [8], randomized motion planners [9], [10], and trajectory optimization [7], [11]. In this work, we build on trajectory optimization approaches.

The downside of trajectory optimization approaches is that they are usually locally optimal and incomplete, while the other approaches have completeness or global optimality guarantees. The upside of trajectory optimization is that it scales well to high dimensions and converges quickly. The second property is useful in a task and motion planning context because it quickly rules out infeasible task plans.

Trajectory optimization generates a motion plan by solving the following constrained optimization problem.

$$\begin{aligned} \min_{\tau_t \in \mathcal{X}} \quad & ||\tau||^2 \qquad (1) \\ \text{subject to} \quad & f(\tau_t, \tau_{t+1}) = 1 \\ & SD(\tau_t, o) \geq d_{safe} \quad \forall o \in O \\ & \tau_0 = p_0, \tau_T = p_T \end{aligned}$$

We optimize over a fixed number of waypoints τ_t, with $t = 0, \ldots, T$. The objective $||\tau||^2$ is a regularizer that produces smooth trajectories. A standard choice is the *minimum velocity* regularizer

$$||\tau||^2 = \sum_t ||\tau_t - \tau_{t+1}||^2.$$

The first constraint is the dynamics constraint that ensures that the pose at time $t+1$ is reachable from the pose at time t. The second constraint is a collision avoidance constraint. It requires that the distance[1] from any robot pose to an object be larger than a fixed safety margin. The final constraint ensures that the trajectory begins (resp. ends) at the initial (resp. final) pose.

Sequential Quadratic Programming

[7] applied *sequential quadratic programing* (SQP) to trajectory optimization. Practically, this treats a robot trajectory as variables in a mathematical program and applies

[1]This is actually the signed-distance, which is negative is the robot and object overlap.

standard solution algorithms. SQP is an iterative non-linear optimization algorithm that can be seen as a generalization of Newton's method. [12] Ch. 18 describes several variants of SQP. The most important attribute of SQP for trajectory optimization is that it can typically solve problems with very few function evaluations. This is useful in trajectory optimization because function evaluation (i.e., collision checking) is a computational bottleneck.

SQP minimizes a non-linear f subject to equality constraints h_i and inequality constraints g_i.

$$\begin{aligned} \min_{x} \quad & f(x) && \\ \text{subject to} \quad & h_i(x) = 0 \qquad && i = 1, \ldots, n_{eq} \\ & g_i(x) \leq 0 \qquad && i = 1, \ldots, n_{ineq} \end{aligned} \tag{2}$$

Loosely speaking, the approach iteratively applies two steps. The first is to make a convex approximation to the constraints and objective in Equation 2. We write the approximations as $\widetilde{f}, \widetilde{h}_i, \widetilde{g}_i$. SQP makes a quadratic approximation to f and linear approximations to the constraints h_i, g_i.

Once we have obtained a convex local approximation we can minimize it to get the next solution $x^{(i+1)}$. We need to ensure that the approximation is accurate so we impose a trust-region constraint. This enforces a hard constraint on the distance between $x^{(i)}$ and $x^{(i+1)}$. Let $\widetilde{f}, \widetilde{h}_i, \widetilde{g}_i$ be convex approximations to f, h_i, g_i. The optimization we solve is

$$\min_{x} \quad \widetilde{f} + \mu \left(\sum_{1}^{n_{eq}} |\widetilde{h}_i(x)| + \sum_{1}^{n_{ineq}} |\widetilde{g}_i(x)|^{+} \right) \tag{3}$$

$$\text{subject to} \quad |x - x^{(i)}| < \delta \tag{4}$$

where δ is the trust-region size. The ℓ_1-norm to penalize constraint violations results in a non-smooth optimization, but can still be efficiently minimized by standard quadratic programming solvers. We elect to use an ℓ_1-norm, as opposed to an ℓ_2 norm, because it drives constraint violations to 0 and performs well with large initial constraint deviations. Algorithm 1 shows pseudocode for this optimization method.

As an example, consider the behavior of SQP on the motion planning problem from Figure 2. The initial pose is in the top right at location $(0, 2)$ and the target pose is around a corner at location $(3.5, 5.5)$. We initialize with an infeasible straight line trajectory. We use 20 time-steps for our trajectory. We let the x coordinate for the robot take values in $[0, 7]$ and the y coordinate take values in the range $[-2, 7]$. The corresponds to the following trajectory optimization:

$$\begin{aligned} \min_{\tau_t \in [0,7] \times [-2,7]} \quad & \sum_{t=0}^{20} ||\tau_t - \tau_{t+1}||^2 \\ \text{subject to} \quad & |\tau_t - \tau_{t+1}| \leq d_{max} \\ & SD(\tau_t, Wall) \geq d_{safe} \\ & \tau_0 = (7, 3) \\ & \tau_{20} = (3, 7) \end{aligned}$$

Algorithm 1 ℓ_1 Penalty Sequential Quadratic Programming [12].

Define: SQP$(x^{(0)}, f, \{h_i\}, \{g_i\})$
Input: initial point $x^{(0)}$, the function being minimized f, a set of non-linear equality constraints $\{h_i\}$, a set of non-linear inequality constraints $\{g_i\}$.
/* increase the penalty for violated nonlinear constraints in each iteration */
for $\mu = 10^0, 10^1, 10^2 \ldots, \mu_{max}$ **do**
  for $i = 1, \ldots,$ ITER_LIMIT **do**
    /* compute a quadratic approximation for f*/
    $\tilde{f}, \{\tilde{h}_i\}, \{\tilde{g}_i\}$ = ConvexifyProblem$(f, \{h_i\}, \{g_i\})$
    for $j = 1, 2, \ldots$ **do**
      $x = \text{argmin}$ (4) subject to (5) and linear constraints
      if TrueImprove / ModelImprove $> c$ **then**
        /* expand trust region */
        $\delta \leftarrow improve\_ratio \cdot \delta$
        break
      end if
      /* shrink trust region */
      $\delta \leftarrow decrease\_ratio \cdot \delta$
      if converged() **then**
        /* converge if trust region too small
        or current solution is a local optimum */
        return locally optimal solution x^*
      end if
    end for
  end for
end for

The first step of the algorithm makes a linear approximation to the signed distance constraint. The details of the approximation can be found in [7]. The first image shows this initialization and superimposes the local approximation to the signed distance constraint on top of it. It pushes each pose towards the outside of the walls.

The next step of the algorithm minimizes the approximation to this constraint subject to a trust region constraint. This makes progress on the objective, so we accept the move and increase the size of the trust region. After several iterations, we obtain the trajectory in the middle of the image. At termination we arrive at the motion plan in the left most image: a collision-free, locally-optimal trajectory.

IV. TASK AND MOTION PLANNING

In this section, we formulate task and motion planning (TAMP). We present an example formulation of the naviga-

(a) Initialization (b) Optimization (c) Final trajectory

Fig. 2: Trajectory optimization for a 2D robot. The gradient from the collision information pushes the robot out of collisions despite the infeasible initialization.

tion amongst moveable objects (NAMO) as a TAMP problem. We give an overview of the complete TAMP algorithm presented in [1].

Problem Formulation

Definition 1: We define a task and motion planning (TAMP) problem as a tuple $\langle T, O, F_P, F_D, I, G, U \rangle$:

- T a set of object *types* (e.g., movable objects, trajectories, poses, locations).
- O a set of *objects* (e.g., can_2, grasping_pose_6, location_3).
- F_P a set of *primitive fluents* that collectively define the world state (e.g., robot poses, object geometry). The set of primitive fluents, together with O, defines the configuration space of the problem.
- F_D a set of *derived fluents*, higher-order relationship between objects defined as boolean functions that depend on primitive fluents.
- I a conjunction of primitive fluents that defines the initial state.
- G a conjunction of (primitive or derived) fluents that defines the goal state.
- U a set of *high-level actions* (e.g., grasp, move, put-down). Each high-level action $a \in U$ is parametrized by a list of objects and defined by: 1) $a.pre$, a set of *pre-conditions*, fluents that describe when an action can be taken; 2) $a.post$, a set of *post-conditions*, fluents that hold true after the action is performed; and 3) $a.mid$ a set of *mid-conditions*, fluents that must be true while the action is being executed.

A state in a TAMP problem is defined by a set of primitive fluents. Note that this defines the truth value of all derived fluents. The solution to a TAMP problem is a plan

$$\pi = \{s^0, (a^0, \tau^0), s^1, (a^1, \tau^1), \ldots, (a^{N-1}, \tau^{N-1}), s^N\}.$$

The s^i are states, defined as a set of primitive predicates that are true. The a^i are the actions in the plan. τ^i is the trajectory for action i and is defined as a sequence of states. A valid solution satisfies the following constraints.

- The first state is the initial state: $s^0 \in I$.
- Pre-conditions are satisfied: $a^i.pre \in s^i$.
- Mid-conditions are satisfied: $a^i.mid \in \tau^i_t \ \forall t$.
- Post-conditions are satisfied: $a^i.post \in s^{i+1}$.
- Trajectories start in the states that precede them and end in the states that follow them: $\tau^i_0 = s^i$, $\tau^i_T = s^{i+1}$.
- The final state is a goal state: $G \in s^N$.

Our formulation differs from the standard formulation of TAMP in two ways. The first is that we explicitly differentiate between primitive fluents and derived fluents. We use the difference between the two types of fluents to distinguish between variables and constraints for the optimization in Section V.

The second difference is the introduction of *mid-conditions*. These are invariants: constraints that must be satisfied on every step on of a trajectory that implements a high-level actions. Mid-conditions define the space of trajectories than can implement a given high-level action. An example mid-condition is a collision avoidance constraint.

Example Domain: Navigation Amongst Movable Objects

Here, we formulate a 2D version of the *navigation amongst moveable objects* (NAMO) problem [13]. In our domain, a circular robot navigates a room full of obstructions. If the robot is next to an object, it can attach to it rigidly via a suction cup. In the top middle of our domain is a closet. The robot's goal is to store objects in, or retrieve objects from, the closet. Thus, we call the problem the 2D closet domain (CL-2D-NAMO). This domain is characterized as follows.

Object types T. There are six object types: 1) *robot*, a circular robot that can move, pick, and place objects; 2) *cans*, cylinders throughout the domain that the robot can grasp and manipulate; 3) *walls*, rectangular obstructions in the domain that the robot can not manipulate; 4) *poses*, vectors in $\mathbb{R}^2$ that represent robot poses; 5) *locs*, vectors in $\mathbb{R}^2$ that represent object poses; and 6) *grasps*, vectors in $\mathbb{R}^2$ that represent grasps as the relative position of the grasped object and robot.

Objects O. There is a single robot, R. There are N movable objects: $\text{can}_1, \ldots, \text{can}_N$. There are 8 walls that make up the unmovable objects in the domain: $\text{wall}_1, \ldots, \text{wall}_8$. Robot poses, object locs, and grasps make up the remaining objects in the domain. The are continuous values so there are infinitely many of these objects. Robot poses and object locs are contained in a bounding box around the room B. Grasps are restricted to the be in the interval $[-1, 1]^2$.

Primitive Fluents F_P. The primitive fluents in this domain define the state of the world. We define the robot's position with a fluent whose sole parameter is a robot pose: robotAt(?rp-pose). We define an object's loc with a similar fluent that is parametrized by an object and a loc: objAt(?o-can ?ol-loc).

Derived Fluents F_D. There are three derived fluents in this domain. The first is a collision avoidance constraint that is parametrized by an object, a loc, and a robot pose: obstructs(?obj-can ?loc-loc ?rp-pose). This is true when ?obj and the robot overlap at their respective locations and poses.

It is defined as a constraint on the signed distance: SD(?obj, R) $\geq d_{safe}$.

We determine if the robot can pick up a can with isGraspPose(?obj-can ?rp-pose ?loc-loc). This is true if a robot at ?rp touches the can at location ?loc. This is implemented as an equality constraint on signed distance: SD(R, can) $= \epsilon$. We use this to determine when the robot can pick up the object, and when it can put it down.

Once the robot has picked up an object, we need to ensure that the grasp is maintained during the trajectory. We do this with inManip(?obj-can ?g-grasp), which is parametrized by a can and a grasp. It is defined by an equality constraint on the respective positions of the object and the robot: (robotAt(?rp) $\wedge$ objAt(?obj ?loc) $\Rightarrow$?rp-?loc = ?g). If the robot is holding an object (i.e., inManip is true for some object and grasp) then it is treated as part of the robot in all signed distance checks.

Dynamics. The dynamics of this problem are simple. The robot has a maximum distance it can move during any timestep. The objects remain at their previous location. The inManip fluent ensures that held objects are always in the same relative position to the robot.

High-level actions U**.** We have four high-level actions in our domain: MOVE, MOVEWITHOBJ, PICK, and PLACE.

The MOVE action moves the robot from one location to another, assuming it holds no object. We use ?rp$_t$ to represent the robot pose at time t within the move action's trajectory.

MOVE(?rp1-pose ?rp2-pose)
pre robotAt(?rp1)
$\wedge$ ($\forall$?obj-can, ?g-grasp $\neg$ inManip(?obj ?g)
mid ($\forall$?c-can, ?l-loc $\neg$ obstructs(?c, ?l ?rp$_t$))
post robotAt(?rp2)

The MOVEWITHOBJ action is similar to the move action. The primary difference is that the preconditions require that the robot be holding an object and that said object remain rigidly attached to the robot.

MOVEWITHOBJ(?rp1-pose ?rp2-pose ?obj-can ?g-grasp)
pre robotAt(?rp1) $\wedge$ inManip(?obj ?g)
mid ($\forall$?c-can, ?l-loc $\neg$ obstructs(?c, ?l ?rp$_t$))
$\wedge$ inManip(?obj ?g)
post robotAt(?rp2)

The final two actions pickup objects from locations and put them down. They only consist of a single timestep, so they have no mid-conditions. In order to pick up an object, the robot must be holding nothing and be next to the object. To put an object down it must be currently held and the robot has to be in the appropriate relative location.

PICK(?obj-can ?l-loc ?rp-pose ?g-grasp)
pre robotAt(?rp) $\wedge$ objAt(?obj ?l)
$\wedge$ ($\forall$?c-can, ?g-grasp $\neg$ inManip(?c ?g)
$\wedge$ isGraspPose(?obj ?rp ?l)
mid $\emptyset$
post inManip(?obj ?g)

PLACE(?obj-can ?l-loc ?rp-pose ?g-grasp)
pre robotAt(?rp) $\wedge$ inManip(?obj ?g)
$\wedge$ isGraspPose(?obj ?rp ?l)
mid $\emptyset$
post $\neg$ inMaip(?obj ?g) $\wedge$ objAt(?obj ?l)

V. TASK PLAN OPTIMIZATION

A common operation in task and motion planning is *plan refinement*. This is the process of converted a partially specified abstract plan into a fully specified trajectory. We focus on a special case of plan refinement where all discrete variables are fixed by the task plan. This is a common type of abstract plan that is used in, e.g., [3],[5], [1], and [6].

First, we describe how our formulation of task and motion planning encodes a joint trajectory optimization over intermediate states and plan parameters. Then, we discuss our trajectory initialization and reuse schemes. These are important in light of the size and non-convexity of the trajectory optimization problems we consider. We show how the movement primitives of [4] can be used to leverage previous solutions to guide initialization. Finally, we give an algorithm for early detection of infeasibility. This is crucial for task and motion planning, because it is important to fail fast if no motion planning solution exists.

Abstract Plans Encode Trajectory Optimizations

We adopt the view taken in [3] that abstract plans encode trajectory optimizations. In our formulation, we maintain a precise connection between pre-conditions and effects of actions and the trajectory optimizations those actions encode. Before describing the optimization formulation in general, we go through an example from the CL-2D-NAMO domain.

Example: Trajectory Optimization for a Pick-Place: Consider an abstract task plan for the CL-2D-NAMO domain.

- MOVE(rp$_{init}$ gp$_1$)
- PICK(can$_1$ c1$_{init}$ gp$_1$ g$_1$)
- MOVEWITHOBJ(gp$_1$ pdp$_1$ can$_1$ g$_1$)
- PLACE(can$_1$ c1$_{goal}$ pdp$_1$ g$_1$)

This plan moves to a grasping pose for can$_1$, picks up can$_1$, moves to a goal location, and then places the object at the goal. The parameters plan refinement determines are the continuous action parameters: the grasping pose, gp$_1$; the grasp to use, g$_1$; and the putdown pose, pdp$_1$.

Setting the values for these parameters defines the intermediate states in the plan, so these variables are directly constrained by the pre-conditions and post-conditions of actions in the plan.

Next, we need to find trajectories through the state space that connect these intermediate states. The variables in the trajectory optimization will be a sequence of world states. We fully determine the world state by setting a value for each primitive predicate, so we optimize over the continuous parameters for a sequence of primitive predicates, subject to the mid-conditions from the high-level action and dynamics constraints. This results in the following trajectory optimization:

$$\min_{gp_1, g_1, pdp_1, \tau^0, \tau^2} \quad \sum ||\tau_t^0 - \tau_{t+1}^0||^2 + \sum ||\tau_t^2 - \tau_{t+1}^2||^2.$$

subject to

$$\begin{aligned}
\tau_0^0 &= rp_{init}, \tau_T^0 = gp_1 \\
\tau_0^2 &= gp_1, \tau_T^2 = pdp_1 \\
|\tau_t^0 - \tau_{t+1}^0| &\leq \delta \\
|\tau_t^2 - \tau_{t+1}^2| &\leq \delta \\
\forall o \in O \ SD(\tau_t^0, o) &\geq d_{safe} \\
\forall o \in O \ SD(\tau_t^2, o) &\geq d_{safe} \\
isGraspPose(can_1, c1_{init}, gp_1) & \\
isGraspPose(can_1, c1_{goal}, pdp_1) & \\
inManip(can_1, g_1) &
\end{aligned}$$

The constraints on the start and end of the trajectories come from the robotAt preconditions. The final inManip constraint holds for every state in τ^2. Each constraint defined above is either linear or a signed distance constraint. This means that the problem is suitable for the sequential quadratic programming approach described in Section III.

Converting a General Abstract Plan to a Trajectory Optimization: To translate a general high-level action A(p_1, p_2, ...) we apply the following sequence of steps. First, determine the parameters in the high-level action that are not set. Second, determine the variables for a trajectory for this action. In our formulation, these are defined by the set of primitive predicates. In the CL-2D-NAMO domain, this adds variables for robot poses and object locations.

Now that we have a set of variables, we can add in constraints. We iterate through A's pre-conditions. We add them as constraints on the parameters of the action and the first state in the trajectory. We repeat that process with the post-conditions and the last state in the trajectory. Finally, we add A's mid-conditions as constraints on each intermediate step of the trajectory. Algorithm 2 shows pseudocode to set up and refine this trajectory optimization.

The sequential quadratic programming approach that we use is a local improvement algorithm, so good initialization leads to faster convergence. Bad initializations often fail to converge, even when a solution exists. This is a difficult challenge in regular trajectory optimization and trajectories considered here are substantially longer than those considered in typical motion planning.

To deal with this challenge, we use the structure of our formulation to help guide search. We define a distribution over continuous values for each parameter type, called a generator [14]. Our first step in initialization uses these generators to obtain initial values for each parameter. After, we need to initialize trajectories and make sure the the parameters are self-consistent. We do this with an optimization that considers the trajectory costs but only includes constraints at end states. Finally, we add in all constraints and optimize the full problem.

Trajectory Reuse

Often, the first attempt at refinement fails to converge. Figure 3 (a) shows an example of one such trajectory. The initial grasp pose was sampled on the wrong side of the object, so it is unreachable. At this point, we want to use a randomized restart to try to find a solution. However, completely starting over from scratch as in Figure 3 (b) is undesirable because we through away a lot of information. In particular, the previous trajectory has figured out that it should go around the corner, not through it. The optimization can figure this out again, but it will require a lot of collision checks and will increase the total time. This problem gets much worse with very long plans (e.g., 20 different move actions). If a single action has no feasible trajectory, we do not want to throw away the rest of the solution.

We would like to re-initialize only the variables in violated constraints. This often fails because the rest of the plan has too much 'inertia:' it has already settled into a local optimum and so the first step of the optimization moves the re-initialized variables back to their previous (infeasible) values.

Algorithm 2 Refining an Abstract Task Plan

Define: PLANOPT(π)
Input: partially specified abstract plan π.
/* iterate through high-level actions in the plan */
for $a \in \pi.ops$ **do**
  params = GetVariables(a)
  for $p \in a.preconditions$ **do**
    p.AddConstraint(params, τ_1^a)
  end for
  for $p \in a.postconditions$ **do**
    p.AddConstraint(params, τ_T^a)
  end for
  for $p \in a.midconditions$ **do**
    for $t = 2, \ldots, T-1$ **do**
      p.AddConstraint(params, τ_t^a)
    end for
  end for
end for
/* call SQP to optimize all the τ^a */

Instead, we run an optimization that keeps re-initialized variables at their (new) values and propagates the changes to the rest of the trajectory. This is done by minimizing the norm of the changes in the trajectories. The choice of trajectory norm is important. Figure 3 (c) shows what happens if this projection is performed under an ℓ_2-norm. Although some of the trajectory moves to account for the new parameters, enough of it is stuck behind the object that the optimization is still stuck in the same basin of attraction.

[4] formulates movement primitives as projections under different norms in a Hilbert space of trajectories. We adopt their approach and use a *minimum velocity* norm to project old trajectories onto new initializations. This is shown in Figure 3 (d). We can see that the new trajectory maintains the

qualitative structure of the previous solution (and so avoids collisions) and naturally moves to the new pick pose.

Early Detection of Unsatisfiability

With long task plans, it is important that the optimization fail fast. Very often an optimization quickly determines that a constraint is infeasible and converges for that constraint. However, the rest of the plan may still be very far from a local optimum. Thus, a vanilla implementation of the convergence check may spend a large number of extra QP minimizations and collision checks optimizing a plan that we know to already be infeasible.

In SQP, one convergence test checks that approximate improvement in the objective value is above a threshold. This is the improvement we make during a QP solve, but measured with respect to the convex approximation. If the approximate improvement is small it is likely that we are at a local optimum of the real objective.

Our approach is to check this convergence constraint independently for each constraint. We terminate the optimization early if the following conditions are met: 1) there is a constraint that is unsatisfied; 2) the approximate improvement on the constraint's infeasibility is below a threshold; 3) constraints that share variables with this constraint are satisfied or have low approximate improvement. The first two conditions extend the standard convergence criterion to a per-constraint criterion. The final condition catches situations where the optimization allocates its effort to satisfying a different, coupled, constraint.

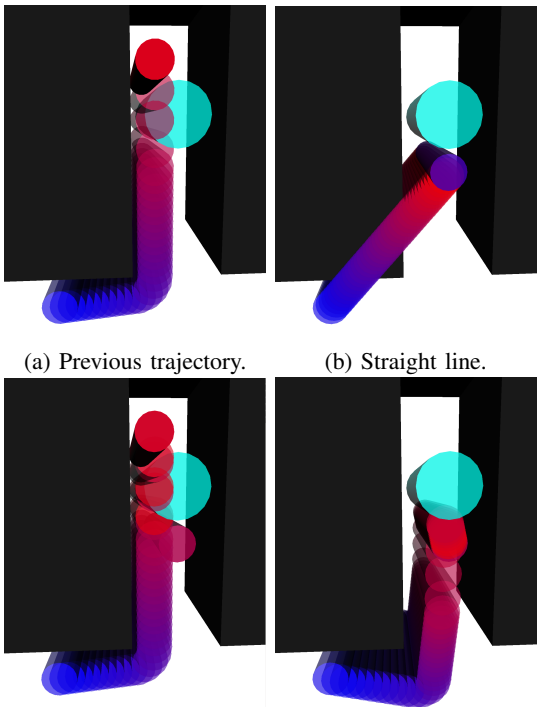
VI. EXPERIMENTS

Methodology

We evaluate our approach in the NAMO domain with two distinct experimental setups: the *swap* task and the *putaway* task. In the swap task, there are two objects inside the closet. The robot must reverse the positions of both objects. This requires reasoning about obstructions and proper plan ordering. In the putaway task, two target objects are located among several obstructions in the room. The robot must retrieve the two objects and place them both anywhere inside the closet. An important aspect of this task is that once one object is placed inside the closet, the robot cannot navigate behind it to the place the other. We run experiments for this task with 0, 3, and 5 obstructing objects.

We compare performance with the backtracking baseline established in [1], which performs exhaustive backtracking search over plan parameters. We implement the motion planning by applying SQP to each action independently.

Manipulated Variables. We perform two experiments. Experiment 1 compares the performance of four systems: the backtracking baseline (B), standard SQP (S), SQP with our early convergence criteria (E), and standard SQP initialized using the solution found by backtracking (T). There are two manipulated variables in this experiment: which of these systems is run, and which experimental scenario we test on (swap or putaway with 0, 3, or 5 obstructions).



(a) Previous trajectory. (b) Straight line.

(c) ℓ_2-norm. (d) Minimum velocity.

Fig. 3: Trajectory (a) has collisions, so the robot end pose is resampled. (b) initializes with a straight line trajectory, and needs to rediscover the path around the wall. (c) uses an l2-norm to find a similar trajectory with the new end point. This doesnt change the trajectory enough to get to a new basin of attraction. The minimum-velocity trajectory (d) adapts to the new endpoint but reuses information from the previous trajectory.

Experiment 2 considers the effects of different types of trajectory reuse on each of our novel systems, S and E. There are two manipulated variables in this experiment: which system is run (S or E), and which trajectory reuse strategy we use. We consider three such strategies: 1) straight-line initialization (i.e., ignore previous trajectories), 2) ℓ_2-norm minimization (i.e., stay as close as possible to previous trajectories), and 3) minimum-velocity ℓ_2-norm minimization (i.e., stay close to a linear transformation of the trajectory). Our experiments reveal that minimum-velocity ℓ_2-norm minimization worked best, so Experiment 1 uses this technique.

Dependent Measures. We measure success rate, the sum of squared velocities on the trajectories, planning time, and number of new task plans generated.

Problem Distributions. Experiment 1 is evaluated on fixed test sets of 50 randomly generated environments. Environments for the putaway task are generated by randomly spawning N objects within the room and designating two as the targets. Experiment 2 is evaluated on a smaller test set of 30 environments for the swap task.

Our experiments are conducted in Python 2.7 using the OpenRAVE simulator [15]. Our task planner is Fast-Forward [16]. Experiments were carried out in series on an Intel Core i7-4770K machine with 16GB RAM. The time limit was set to 1200 seconds for the swap task and 600

Condition	% Solved	Traj Cost	Time (s)	# Replans
Swap, B	100	42.4	37.7	5.0
Swap, S	100	10.2	267.2	6.0
Swap, E	100	10.4	217.8	14.4
Swap, T	100	10.9	115.1	5.0
P(0), B	100	12.2	16.8	3.2
P(0), S	100	7.7	21.1	1.8
P(0), E	100	7.7	23.9	2.3
P(0), T	100	7.8	20.7	3.2
P(3), B	98	16.9	58.8	4.9
P(3), S	96	8.9	109.4	3.9
P(3), E	98	9.1	101.1	4.3
P(3), T	98	9.1	76.5	5.1
P(5), B	86	21.3	91.4	8.4
P(5), S	83	9.7	154.4	5.4
P(5), E	88	9.7	160.3	6.8
P(5), T	94	11.0	135.0	7.3

TABLE I: Success rate, average trajectory cost, average total time, and average number of calls to task planner for each system in each experimental scenario. P indicates a putaway task. The number in parentheses is the number of obstructions. B: backtracking baseline. S: standard SQP. E: SQP with early convergence criteria. T: SQP with initialization from B. Results are obtained based on performance on fixed test sets of 50 randomly generated environments. All failures were due to timeout: we gave 1200 seconds for each swap task problem and 600 seconds for each putaway task problem.

Condition	% Solved	Traj Cost	Time (s)	# Replans
SL, S	100	10.7	338.6	9.4
SL, E	100	10.8	217.6	9.4
ℓ_2-norm, S	63	10.4	336.1	9.5
ℓ_2-norm, E	67	11.0	181.0	13.6
Min-V, S	100	10.0	247.3	5.7
Min-V, E	100	10.4	200.7	13.1

TABLE II: Success rate, average trajectory cost, average total time, and average number of calls to task planner for several systems. SL indicates straight-line initialization; ℓ_2-norm and Min-V use an ℓ_2 or minimum velocity projection to initialize; S denotes standard SQ; E denotes SQP with early convergence criteria. Results are obtained based on performance on fixed test sets of 30 environments. All failures were due to timing out the 1200 second limit.

seconds for the putaway task. Tables I and II summarize results for Experiments 1 and 2.

Discussion

Experiment 2 shows that trajectory reuse with minimum-velocity projection outperforms standard ℓ_2 projection and straight-line initialization. ℓ_2 projection performs poorly because it gets trapped in bad local optima. Experiment 1 shows that full joint optimization (systems S and E) over plan parameters leads to significant improvements in overall trajectory cost. This comes at the expense of increased running time. Using our algorithm as a trajectory smoother (System T) merges the benefits of both approaches.

We attribute backtracking's speed advantage to two factors. First, backtracking is able to rule out plans faster than the joint optimization. Early-convergence helps, but leaves room for improvement. Second, the joint optimization ends up making more collision check calls. This is because the trust region in the optimization is shared across the whole plan. So the algorithm will take small steps when one part of the plan is poorly approximated. In future work, we intend to optimize our implementation (the current implementation is somewhat optimized Python) and experiment on more realistic robots (e.g., the PR2).

ACKNOWLEDGMENTS

This research was funded in part by the NSF NRI program under award 1227536, and by the Intel Science and Technology Center (ISTC) on Embedded Systems. Dylan was supported by a Berkeley Fellowship and a NSF Fellowship.

REFERENCES

[1] R. Chitnis, D. Hadfield-Menell, A. Gupta, S. Srivastava, E. Groshev, C. Lin, and P. Abbeel, "Guided search for task and motions plans using learning heuristics," in *IEEE Conference on Robotics and Automation (ICRA)*, 2016.

[2] C. R. Garrett, T. Lozano-Perez, and L. P. Kaelbling, "Backward-forward search for manipulation planning," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2015, pp. 6366–6373.

[3] M. Toussaint, "Logic-geometric programming: An optimization-based approach to combined task and motion planning," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.

[4] A. D. Dragan, K. Muelling, J. A. Bagnell, and S. S. Srinivasa, "Movement primitives via optimization," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 2339–2346.

[5] T. Lozano-Pérez and L. P. Kaelbling, "A constraint-based method for solving sequential manipulation planning problems," in *International Conference on Intelligent Robots and Systems (IROS)*, 2014.

[6] F. Lagriffoul, D. Dimitrov, J. Bidot, A. Saffiotti, and L. Karlsson, "Efficiently combining task and motion planning using geometric constraints," in *International Conference on Robotics and Automation (ICRA)*, 2014.

[7] J. D. Schulman, J. Ho, A. Lee, I. Awwal, H. Bradlow, and P. Abbeel, "Finding locally optimal, collision-free trajectories with sequential convex optimization," in *Proceedings of Robotics: Science and Systems (RSS)*, 2013.

[8] B. J. Cohen, S. Chitta, and M. Likhachev, "Search-based planning for manipulation with motion primitives," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2010, pp. 2902–2908.

[9] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," Stanford, CA, USA, Tech. Rep., 1994.

[10] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," Tech. Rep., 1998.

[11] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "Chomp: Gradient optimization techniques for efficient motion planning," in *International Conference on Robotics and Automation (ICRA)*, 2009.

[12] J. Nocedal and S. Wright, *Numerical optimization*. Springer Science & Business Media, 2006, ch. 18.

[13] M. Stilman and J. Kuffner, "Planning among movable obstacles with artificial constraints," *The International Journal of Robotics Research*, vol. 27, no. 11-12, pp. 1295–1307, 2008.

[14] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *International Conference on Robotics and Automation (ICRA)*, 2011.

[15] R. Diankov and J. Kuffner, "Openrave: A planning architecture for autonomous robotics," Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34, July 2008.

[16] J. Hoffmann, "FF: The fast-forward planning system," *AI Magazine*, vol. 22, pp. 57–62, 2001.