Semi-Infinite Programming for Trajectory Optimization with Nonconvex Obstacles

Kris Hauser

Duke University, Durham NC 27708, USA,
kris.hauser@duke.edu,
WWW home page: http://people.duke.edu/~kh269

Abstract. This paper presents a novel optimization method that handles collision constraints with complex, non-convex 3D geometries. The optimization problem is cast as a semi-infinite program in which each collision constraint is implicitly treated as an infinite number of numeric constraints. The approach progressively generates some of these constraints for inclusion in a finite nonlinear program. Constraint generation uses an oracle to detect points of deepest penetration, and this oracle is implemented efficiently via signed distance field (SDF) versus point cloud collision detection. This approach is applied to pose optimization and trajectory optimization for both free-flying rigid bodies and articulated robots. Experiments demonstrate performance improvements compared to optimizers that handle only convex polyhedra, and demonstrate efficient collision avoidance between nonconvex CAD models and point clouds.

Keywords: optimization, semi-infinite programming, collision avoidance

1 Introduction

Optimization in robotics has long been complicated by the difficulty of encoding collision constraints between complex geometries, even though complex geometries are routinely handled by planners that produce feasible (non-optimal) paths. Sampling-based motion planners [6] have been successful because they leverage the extensive body of work fast collision detection between non-convex bodies [5], which allows collision checks to be performed rapidly. However, these checks are binary computations that cannot be easily incorporated into numerical optimizers, which usually require differentiable constraints. A naïve numerical encoding for objects composed of M and N geometric primitives (e.g., triangles) would require $O(MN)$ pairwise constraints. Hence, past optimization approaches require robot links and environments to be represented using simple convex geometries, such as ellipsoids [11], capsules [8], polyhedra [10, 12], and superquadrics [2] to speed up computation. This introduces severe limitations on allowable shapes of objects and robot geometries. The goal of this paper is to overcome these challenges by introducing a novel constraint encoding and optimization method that can handle non-convex objects in a computationally-efficient manner.

The approach taken here is to represent each collision constraint as an *infinite set* of simpler constraints in a semi-infinite programming (SIP) framework. Although it is not possible to optimize an infinite number of constraints directly, a finite subset of constraints, suitably chosen, is sufficient to define an optimum. An *exchange method* is used that instantiates a series of finite optimization problems, each of which progressively adds some number of constraints. Using a judicious constraint selection procedure, called an *oracle*, the series of problems converges toward one that contains a true optimum.

For collision avoidance between a pair of objects, we establish constraints that every point on the surface of one object is required to be outside of or on the surface of the other object. We call the first object the privileged object. The oracle detects the point in the privileged geometry that penetrates most deeply into the other. To make this procedure fast, the privileged geometry is represented as a point cloud, and the other is represented as a signed distance field (SDF) which supports $O(1)$ depth lookup and $O(1)$ gradient estimation at a point. Bounding volume hierarchies speed up the oracle query. For trajectories, dynamic collision detection techniques are able to find the deepest point in both space and time. A benefit of this technique is that raw point clouds can be directly used in optimization, while precomputed SDFs can be employed for robot links. This avoids the expense of computing geometric data structures on-the-fly as environmental point clouds are acquired from sensors.

The technique is implemented for free-flying rigid bodies as well as articulated robots, both to optimize static poses and trajectories. The technique converges quickly to local optima even with highly complex geometries. Poses can be optimized in tens of milliseconds and trajectories in a few seconds on standard PC hardware. Surprisingly, performance is comparable to and sometimes better than optimizers specialized for convex polyhedral geometries.

2 Semi-infinite programming with collision constraints

2.1 Semi-infinite programming

A semi-infinite programming (SIP) problem over the state variable $x \in \mathbb{R}^n$ is defined as follows

$$\begin{aligned} &\min_x f(x) \\ &\text{s.t.} \\ &g_1(x, y_1) \geq 0 \quad \forall\, y_1 \in Y_1 \\ &\vdots \\ &g_M(x, y_M) \geq 0 \quad \forall\, y_M \in Y_M. \end{aligned} \tag{1}$$

where $g_i(x, y) \in \mathbb{R}^{m_i}$ are the constraint functions, y_i denotes the i'th *index parameter*, and $Y_i \subseteq \mathbb{R}^{p_i}$ is its domain. Each of the functions f is assumed to be twice differentiable, $g_1, \ldots, g_M$ are assumed to be differentiable, and inequalities are interpreted element-wise.

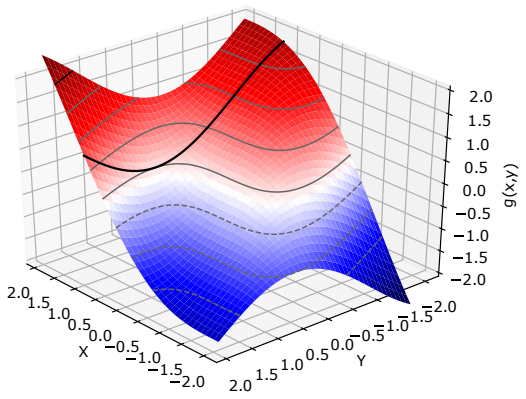


Fig. 1. An example of a simple semi-infinite optimization problem with $f(x) = x$, one constraint $g_1(x, y) = x - \sin(x + y) \geq 0$, and domain $y \in [-2, 2]$. The minimum is at $x = 1$, with the supporting value of $y = \pi/2 - 1$. The dark curve plots the constraint value while x is fixed at the optimum.

The constraints in (1) define an infinite set of parameters for which the constraint must be satisfied. An example of such a problem is shown in Fig. 1. SIP has also been applied to robot trajectory optimization problems under state and control constraints, in which the 1D time variable of the spline is the index parameter [15]. It has also been applied to robust optimization problems in which the disturbance is the index parameter [16].

Of course, existing optimization solvers cannot directly consider a continuous infinity of constraints, which has motivated a rich set of approaches in the optimization literature. One approach [9] is to eliminate the y terms by replacing constraints with inner minimizations of the form:

$$\tilde{g}_i(x) \equiv \min_{y_i \in Y_i} g_i(x, y_i) \geq 0. \tag{2}$$

For example, the constraint in Fig. 1 could be replaced with

$$\tilde{g}(x) = \begin{cases} x - 1 & \text{if } \pi/2 \in [x - 2, x + 2] \\ x - \max(\sin(x - 2), \sin(x + 2)) & \text{otherwise} \end{cases} \tag{3}$$

This approach allows (1) to be encoded as a standard nonlinear program, but a nondifferentiable one. For collision constraints, a common approach is employ the distance function [1, 3], which only depends on the configuration x. But in practice, convergence difficulties arise due to nondifferentiability, which occurs when the minimizing parameter jumps discontinuously. For example, consider a rectangle slightly angled so that vertex a penetrates a plane and defines the penetration depth. On the next iteration, another vertex, say vertex b, may penetrate and define the active constraint. This process may then oscillate between a and b because both vertices cannot be considered simultaneously active.

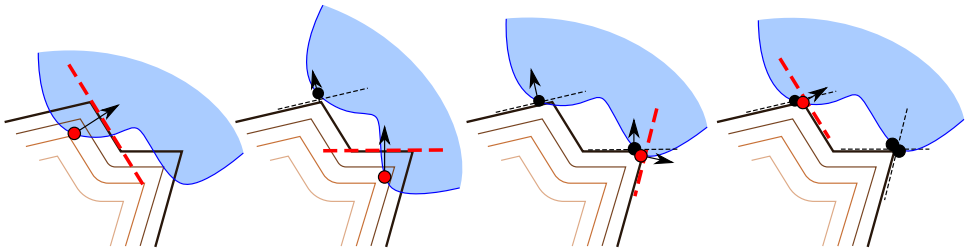


Fig. 2. Illustrating the SIP exchange method to enforce separation of two objects. A contour plot of the fixed object's signed distance field is illustrated. After each iteration, the oracle adds the deepest penetrating point (red circle) to the list of constraint points (black circles).

2.2 Optimization with instantiated constraints

Suppose we are given a finite number N of *instantiated* constraint indices $i^{(1)}, \ldots, i^{(N)}$ and corresponding index parameters $y^{(1)}, \ldots, y^{(N)}$. Here, $i^{(j)} \in \{1, \ldots, M\}$ and $y^{(j)} \in Y_{i^{(j)}}$, for $j = 1, \ldots, N$. We can then define an instantiation of the structured problem corresponding to these parameters as follows:

$$\begin{aligned} &\min_x f(x) \\ &\text{s.t.} \\ &g_{i^{(1)}}\left(x, y^{(1)}\right) \geq 0 \\ &\vdots \\ &g_{i^{(N)}}\left(x, y^{(N)}\right) \geq 0. \end{aligned} \tag{4}$$

This is a finite-dimensional nonlinear program (NLP) with $O(N \max(m_1, \ldots, m_M))$ constraints, and can be solved (locally) using standard methods like sequential quadratic programming (SQP) or interior point methods.

The idea of the *exchange method* is to progressively instantiate constraints and parameters $(i^{(1)}, y^{(1)}), (i^{(2)}, y^{(2)}), \ldots$ giving rise to a sequence of instantiated NLPs whose solutions converge toward the true optimum [9]. Specifically, define P_k as the instantiation corresponding to the first k elements of the constraint sequence, and let $x_k^\star$ its solution. A naïve approach would sample points incrementally from each domain (e.g., randomly or on a grid), and with a sufficiently dense set of points the iterates $x_1^\star, x_2^\star, \ldots$ will eventually approach an optimum. But this approach is inefficient as most samples will not affect the iterated solutions. It is also possible to delete constraints from the constraint set when they are not deemed necessary (the "exchange"), which saves time in later NLP solve steps. For example, one simple strategy is to delete all instantiated constraints whose value at the current iterate exceeds a threshold γ.

Constraint generation oracle The key question for constraint generation is *which* new constraint $g_{i^{(k)}}(x, y^{(k)}) \geq 0$ to add to yield fast convergence. We rely on an *oracle*, a subroutine that performs this selection process. As an example, a *maximum-violation oracle* identifies a parameter value that has a large effect on the next iterated solution. Specifically, on iteration k this strategy calculates the most violating parameter of each constraint, keeping x fixed at x_{k-1}:

$$y_i^{min} \leftarrow \arg\min_{y \in Y_i} \min g_i(x_{k-1}, y). \tag{5}$$

in which the second minimization finds the smallest element in the g_i vector. Then, the constraint with minimum value is computed as

$$\begin{aligned} i^{(k)} &\leftarrow \arg\min_{i=1}^{M} \min g_i\left(x_{k-1}, y_i^{(k)}\right) \\ y^{(k)} &\leftarrow y_{i^{(k)}}^{min}. \end{aligned} \tag{6}$$

The constraint generation process using this oracle is illustrated in Fig. 2.

This approach does, however, beg the question about how to perform the minimization (5) over each Y_i efficiently. Best results are obtained by obtaining a global minimum, and to do this quickly it is often necessary to resort to implementation-specific procedures such as branch-and-bound. In the following discussion we shall assume that such a minimizer is available, and postpone discussion of its implementation until Section 2.3.

Pseudocode and performance considerations The basic meta-algorithm is listed in Algorithm 1. It takes as input the problem, an initial guess x_0, and an iteration count N. It also uses the *Oracle* subroutine. Return status may include *infeasible*, which indicates an infeasible local minimum, *converged*, which indicates a feasible local or global minimum, and *not converged*, which means the iteration count is exhausted.

Besides the choice of N and the oracle, there are a number of performance characteristics to tune.

Constraint instantiation strategy. The strategy used in Steps 4 and 5 is an important component of performance, with most-violating constraint at one end of a spectrum. There is a tradeoff when choosing how many constraints to instantiate, since adding more constraints helps the method converge in fewer iterations, but increases the cost of solving the optimization problem at each iteration. We have experimented with the approach of instantiating M constraints in a single iteration, one for each index i and most-violating parameter y_i^{min}. Another approach, if a global optimization technique is used, might instantiate constraints corresponding to all *local* minima of (5).

It is important to detect and ignore duplicated or near-duplicated index parameters in Step 5, because the most-violating parameter may stay unchanged between subsequent steps. This avoids adding multiple identical constraints that may cause numerical difficulties during NLP solving.

Algorithm 1 Basic SIP solver pseudocode

```
1: procedure SIP(f, g_1, ..., g_M, Y_1, ..., Y_M, x_0, N, S, γ)      ▷ x_0 is an initial guess
2:     I ← {}                                                          ▷ Instantiated constraints
3:     for k = 1, 2, ..., N do
4:         Generate i_k, y_k via Oracle(x_{k-1})
5:         Add (i_k, y_k) to I
6:         Remove from I any (i, y) where g_i(x_{k-1}, y) ≥ γ
7:         Let P_k be (4) instantiated with I.
8:         Run S steps of an NLP solver on P_k, starting from x_{k-1}
9:         If P_k is infeasible, return "infeasible"
10:        Otherwise, set x_k to its solution
11:        if x_k is unchanged, return x_k, "converged"
12:    end for
13:    return x_N, "not converged"
14: end procedure
```

Inner optimization strategy. The method employed to solve for the new optimization variable in Step 8 is also important. Coherence between problems P_{k-1} and P_k may suggest the use of warm-starting to speed up solve times. The number of steps S may be set low to avoid spending too much time on problems whose constraint sets are not sufficiently populated. There is a tradeoff between effort expended on inner optimization and convergence rate, because effort may be wasted on early problems that omit crucial constraints. On the other hand, solve times are slower on later outer iterations because more constraints are instantiated. The extreme $S = 1$ version of this approach is to take a single QP step, which is most favorable when the oracle is fast and f is approximated well by its quadratic Taylor expansion.

Each quadratic programming (QP) call finds a step Δx s.t.

$$\begin{gathered}\min_{\Delta x} \frac{1}{2}\Delta x^T \nabla^2 f(x_k)\Delta x + \nabla f(x_k)\Delta x \\ \text{s.t.} \\ \nabla_x g_{i^{(1)}}\left(x_k, y^{(1)}\right)\Delta x + g_{i^{(1)}}\left(x_k, y^{(1)}\right) \geq 0 \\ \vdots \\ \nabla_x g_{i^{(N)}}\left(x_k, y^{(N)}\right)\Delta x + g_{i^{(N)}}\left(x_k, y^{(N)}\right) \geq 0\end{gathered} \tag{7}$$

and then takes a step $x_{k+1} \leftarrow x_k + \alpha_k \Delta x$. Here α_k is a parameter in $[0, 1]$ determined via line search or a trust region method. One problem that occurs is that after the move to x_k, we may discover that it deeply violates constraints that were not previously instantiated. This may lead to oscillatory behavior or large jumps into basins of attraction of infeasible local minima. To avoid this problem, we implement a trust region approach that limits the step size to a box $-h_k \leq \Delta x \leq h_k$. If a new deeper point is discovered after a candidate step, the step is not taken ($\alpha_k = 0$) and the trust region is shrunk. The step is also

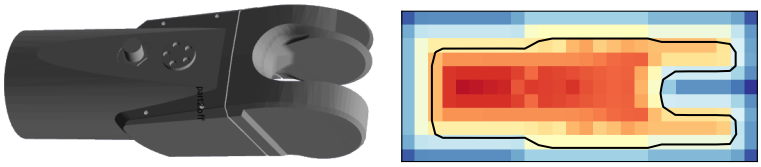


Fig. 3. Illustrating a slice of a signed distance field (SDF) for a non-convex link of an industrial robot.

rejected if a merit function is increased. Otherwise, the full step is taken and the trust region is grown. We implement trust region shrinking with $h_{k+1} \leftarrow h_k \cdot 0.5$ and growing with $h_{k+1} \leftarrow h_k \cdot 2.5$

Another issue to be addressed is that the QP (7) may not be feasible. In this case, we formulate a relaxed QP that introduces slack variables into the constraints. We then minimize a weighted sum of the standard objective function and the sum of squares of slack variables to determine the step.

It should be noted that significant speed gains can be obtained by implementing *branch-and-bound techniques* in the most-violating constraint oracle. Here, an upper bound $\bar{g}$ on the most-violating constraint value is maintained, initialized with the minimum value of g_i over all previous instantiated constraints. Then, during each minimization of (5), $\bar{g}$ is used to discard subsets of the domain that have no chance of yielding a smaller value. $\bar{g}$ is then updated as smaller constraint values are found.

2.3 Collision-free constraint formulation

Let q denote a configuration of the system and A and B two objects. A collision constraint dictates that the workspace occupied by the geometries of the objects $G_A \subset \mathbb{R}^3$ and $G_B \subset \mathbb{R}^3$ do not intersect. Let us assume that G_A and G_B are open sets, so the constraint requires that $G_A(q) \cap G_B(q) = \emptyset$.

We assume each object is a rigid body, and hence the configuration only affects the object's rigid transform $T_A(q)$ relative to the world frame. Hence, we can express $G_A(q) = T_A(q)G_A^0$, where G_A^0 is the object's geometry in its reference frame. Let us assume that each reference geometry has a surface representation ∂G_A^0 and a *signed distance function* (SDF) $D_A(y)$. The SDF is defined so that $|D_A(y)|$ gives the distance from $y \in \mathbb{R}^3$ to ∂G_A^0, and $sign(D_A(y)) = -1$ if $y \in G_A^0$, $sign(D_A(y)) = 0$ if $y \in \partial G_A^0$, and $sign(D_A(y)) = 1$ if y is strictly outside.

Barring the possibility of B being completely enclosed by A, an estimate of distance between A and B is given by

$$d_{AB}(q) \equiv \min_{y \in \partial G_A^0} D_B\left(T_A^{-1}(q)T_B(q)y\right). \tag{8}$$

This value is exact when A and B are disjoint, and is an approximation of the negated penetration depth when A and B intersect (provided that $B \not\subseteq A$.)

The benefit of this approach is that penetration depth lookup for a single point is performed in $O(1)$ time. SDF gradient calculation is also $O(1)$ using finite differencing. Given a closed polygonal mesh, the SDF is calculated using the Fast Marching Method (FMM) applied on a voxel grid. Values off of the grid vertices are approximated via trilinear interpolation. If y is outside of the grid entirely, the distance is approximated by determining the closest point y' in the grid, and assigning $D_A(y) = \|y - y'\| + D_A(y')$.

To represent the distance between two objects, we define

$$g_{AB}(q, y) \equiv D_B \left(T_A^{-1}(q) T_B(q) y\right) \tag{9}$$

to establish a semi-infinite constraint over the domain $y \in G_A^0$.

It is a straightforward matter to provide constraint Jacobian information, which is used by most optimization algorithms like SQP use to determine linearized approximations of constraints at each inner iteration. The Jacobian of the constraint function is

$$\nabla_q g_{AB}(q, y) = \nabla D_B \left(T_B^{-1}(q) T_A(q) y\right) \cdot J_A^B(q, y) \tag{10}$$

where ∇D_B is the distance gradient, and $J_A^B(q, y)$ is $\nabla_q (T_B^{-1}(q) T_A(q) y)$. Specifically this is the Jacobian of the coordinates of point y relative to B's coordinate frame, where y is given in A's local coordinates. This matrix is calculated via forward kinematics depending on whether q encodes an articulated robot or object pose.

2.4 Performance notes

It is important to note that the constraint is non-symmetric, as object A is represented as a surface model while B is the SDF. We call object A the *privileged object*, and the choice of the privileged object in a pair can affect performance in two ways. First, the more complex B is, the more likely optimization will fall into local minima in its SDF. Second, the amount of precomputation needed to build an SDF is non-negligible, and SDF construction from a mesh may suffer from artifacts if the mesh is non-watertight. As a result, it is typically better to represent robot links as SDFs because they do not change over time, and this gives developers a chance to manually inspect the SDF for a high-quality representation. Note that to enforce self-collision constraints, robot links will also need to store a surface representation. Environment geometries are better suited for surface representations, since these can be constructed dynamically from sensor data with minimal amounts of precomputation.

It should be noted that SDFs may contain some non-differentiable points along the medial axis of the object, which may slow convergence of optimization. But each optimization step drives points away from poorly behaved areas, and furthermore the smoothing effect of finite differencing mitigates the potential performance degradation.

2.5 Most-violating parameter calculation

When ∂G_A^0 is approximated via a point cloud of p points, the most-violating parameter of (9), e.g., the closest / deepest penetrating point, can be determined in $O(p)$ time using brute force computation. Brute force computation is trivially parallelizable and suitable for implementation on a GPU. However, for large values of p a bounding volume hierarchy (BVH) approach may be significantly faster.

A BVH is a hierarchical geometric data structure of progressively smaller bounding geometries, where leaf nodes contain one or more primitive points [5]. The significance is that proximity queries between two BVHs can be answered quickly using branch-and-bound techniques. Although BVH proximity queries are more complex and BVH construction incurs some precomputation cost, the speed gains across multiple penetration depth may ultimately make it worthwhile. The BVH approach also allows the branch-and-bound techniques of Sec. 2.2 to be applied across multiple constraints, which speeds up most-violating parameter determination.

Our approach builds a sphere-based BVH of the point cloud. Specifically, an octree data structure is built containing the point cloud, subdividing until each cell contains no more than 10 points. For each leaf octree node, an axis-aligned bounding box is fit to the points it contains, and for each non-leaf node, a bounding box is fit to the bounding boxes of its children.

To query for the closest / deepest penetrating point against an SDF geometry d_B, we use a branch and bound method that relies on a fast lower bound query between a node N in the BVH and the SDF. Given the lower and upper values p_{min} and p_{max} of N's bounding box, we build a circumscribing sphere with center $c = (p_{min} + p_{max})/2$ with and radius $r = \|p_{max} - p_{min}\|/2$. The minimum value of the SDF within this sphere is lower bounded by $l_N = d_B(T_B^{-1} T_A c) - r$, which is also a lower bound on the SDF value at the points contained with N.

The nodes of the BVH are traversed, starting from the root in order of non-decreasing l_N. A lowest encountered distance value d_{min} is maintained, which is initialized to an arbitrary point in ∂G_A^0. If traversal encounters any node N with $l_N \geq d_{min}$, it is pruned. Otherwise, if N is a leaf node then all of its points are checked for being the deepest point. Otherwise, its children $C_1, \ldots, C_k$ are added to the traversal queue with priority values $l_{C_1}, \ldots l_{C_k}$.

2.6 Trajectory collision constraints

We also take a semi-infinite approach to formulate collision constraints along an entire trajectory. This is in contrast to a classical collocation approach, which instantiates static constraints at a discrete set of points in the time domain. The two disadvantages of collocation are that collisions may be missed between collocation points, and that a large number of collocation points leads to a large number of constraints and hence slow optimization times.

Instead, we consider time to be a parameter in a semi-infinite constraint that enforces collision constraints across the entire continuous trajectory. Let the

optimization variable x define the configuration trajectory $q(t)$, e.g., for splines x is a stacked set of control points. To make the dependence of the trajectory on x clear, we shall say $q(t;x)$.

Now, we modify (9) to include time as an additional variable as follows:

$$h_{AB}(x,p,t) \equiv g_{AB}(q(t;x),p) = D_B(T_A^{-1}(q(t;x))T_B(q(t;x))p) \tag{11}$$

which is treated as a semi-infinite constraint over the index parameter $y = (p,t)$ with domain $y \in G_A^0 \times [0,T]$. The Jacobian with respect to x is computed via the chain rule, and for spline representations the Jacobian is sparse.

The most-violating constraint of (11) can be determined efficiently via a branch-and-bound technique. Determine a Lipschitz constraint K_k bounding the magnitude of (9) with respect to the kth entry of q. Then, for any q', the change of (9) from its value at a different configuration q is bounded in magnitude by

$$|g_{AB}(q',y) - g_{AB}(q,y)| \leq K(q-q') \tag{12}$$

where

$$K(\Delta q) \equiv \sum_{k=1}^{n} K_k |\Delta q_k|. \tag{13}$$

Over any time interval $[t^a,t^b]$, from the spline representation we can determine a Lipschitz bound K_Δ on the trajectory derivative. This establishes a parallelipiped in state-time space that is guaranteed to contain the trajectory segment:

$$\begin{aligned} |q(t) - q(t^a)| &\leq |t-t^a| \cdot K_\Delta \\ |q(t) - q(t^b)| &\leq |t-t^b| \cdot K_\Delta \end{aligned} \tag{14}$$

We may then use this in conjunction with (12) to bound the value of $h_{AB}(x,p,t)$ over all p and $t \in [t^a,t^b]$. If we have calculated $g^a = g_{AB}(q(t^a),p^a)$ and $g^b = g_{AB}(q(t^b),p^b)$ along with most-violating points p^a and p^b at the endpoints, then we can obtain the bounds

$$\begin{aligned} |h_{AB}(x,p,t) - g^a| &\leq |t-t^a| \cdot K(K_\Delta) \quad \forall\, p \in \partial G_A^0 \text{ and } t \in [t^a,t^b]. \\ |h_{AB}(x,p,t) - g^b| &\leq |t-t^b| \cdot K(K_\Delta) \quad \forall\, p \in \partial G_A^0 \text{ and } t \in [t^a,t^b]. \end{aligned} \tag{15}$$

Hence, a lower bound on the possible values of $h_{AB}(x,p,t)$ over this domain is obtained at the value of t such that $g^a - (t-t^a)K(K_\Delta) = g^b - (t^b-t)K(K_\Delta)$. This value is $t = \frac{1}{2K(K_\Delta)}(g^a - g^b) + \frac{t^b+t^a}{2}$ giving

$$h_{AB}(x,p,t) \geq \frac{1}{2}(g^a+g^b) - \frac{1}{2}(t_b - t_a)K(K_\Delta). \tag{16}$$

If this lower bound is greater than the least currently established upper bound on the distance $\bar{h}$, then the interval $[t^a,t^b]$ can be pruned from consideration.

Overall the approach uses recursive subdivision to find the most-violating continuous time and point on A, up to a given resolution ϵ (which is far finer than would be reasonable for collocation methods, e.g. 10^{-5}). First, we evaluate

statically the penetration depth at states $q(t_0), \ldots, q(t_S)$ where $t_0, \ldots, t_S$ are the spline knot points. An upper bound $\bar{h}$ on the most violating point and time is initialized to the minimizer of g_{AB} across knot points. For each of the intermediate ranges (t_k, t_{k+1}) we recursively subdivide while pruning any segment for which the r.h.s. of (16) exceeds $\bar{h}$. At the midpoint $(t_k + t_{k+1})/2$, the static penetration depth is evaluated, and the process recurses. Furthermore, to lower $\bar{h}$ as quickly as possible, candidate segments are stored in a priority queue sorted by increasing lower bound. The process quickly narrows down to a small range of possible most-violating times, and empirically we have observed approximately $O(\log \epsilon)$ static penetration depth computations.

As mentioned in Section 2.2 it is useful to perform branch-and-bound over many constraints at once. This is especially true when considering a large number of potential collision pairs. To do so we unify the segment priority queues and maintain a common upper bound $\bar{h}$. This quickly eliminates the need for dynamic collision checking for pairs that have no chance of defining the most violating constraint.

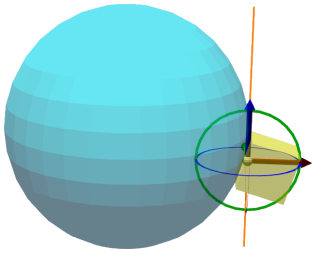
3 Experiments

The SIP algorithm is implemented with front end in the Python programming language, with Python bindings to the OSQP QP solver [14], which is implemented in C, and custom C++ collision detection software. All experiments were run on a single core of a 2.6 GHz Intel i7 processor. Parameters include $N = 50$ maximum iterations, $S = 1$, constraint deletion threshold $\gamma = 0.1$, and a convergence tolerance of $\|\Delta x\| \leq 10^{-4}\sqrt{n}$.

3.1 Performance characteristics

This set of experiments characterize the performance of the algorithm in static pose optimization. First, we consider the cube-sphere collision scenario of Fig. 4. The cube has dimension 0.5 m on each axis and the sphere has radius 1 m. The cube is represented as a signed distance field with 2 cm resolution and the sphere is discretized into a point set of 10,298 points with 5 cm resolution. For each run, 50 targets along the given trajectory are sampled on a uniform grid and the cube is initialized with its center at that position. At all positions, the cube penetrates the sphere. The optimization is used to minimize the distance between the object's center and the target while avoiding collision.

We compare SIP to a standard NLP formulation with max-penetration constraints (2) (MP). Since these objects are convex, we can also formulate them as convex polytopes. We use the exact penetration depth computation of the GJK algorithm [4] to implement the MP constraint (using the libccd library, written in C). We also compare an NLP that instantiates no-penetration constraints between the SDF and 1,000 points sampled from the sphere (NLP-1k). Results show that MP has the lowest computation time, but SIP is not far behind. MP and SIP obtain similar objective function values. NLP-1k is 40x slower due to



Method	Time (ms)	Obj (m)	Pen (mm)	% Pen
SIP	35.7	0.102	**0**	**0**
MP	**30.5**	**0.0998**	4.0	18
NLP-1k	1,281	0.119	0.52	4
SIP-100k	65.8	0.112	**0**	**0**

Fig. 4. The sphere-cube test scenario. Targets are chosen along a straight line trajectory, and for each run the optimization minimizes the distance between the cube center and the target. Test results include average computation time (Time), optimized objective function value (Obj), penetration depth (Pen), and fraction penetrating (% Pen).

the large number of instantiated constraints. Certainly, if all 10,000+ points were included, a standard NLP would be even more expensive.

The most severe weakness of MP is that it often fails to satisfy constraints. In fact, 18% of its runs terminated with some penetration, with an average of 4.0 mm penetration. This is due to oscillation between penetrating and non-penetrating conditions, since MP only considers the effect of one support point at each iteration. NLP-1k does somewhat better, but also fails to detect some collisions due to the limited number of instantiated points.

We also tested a version of SIP where the sphere was discretized even more finely to obtain 108,200 points (SIP-100k). Despite the number of points increasing by an order of magnitude, the added resolution only adds about 40% more computation time. This is because collision detection costs scale sub-linearly, and the number of constraints instantiated still remains small, with SIP-100k instantiating 7.3 constraints and SIP instantiating 2.3.

To illustrate the capabilities of our algorithm in handling complex geometries, Fig. 5 shows the same cube in a point cloud scan of an office environment from the Cornell RGB-D Scene Understanding dataset [7]. The 640×480 RGB-D image contains 307,200 points, 228,352 of which are valid. Precomputation of the point cloud into an octree took 198 ms, although this could be sped up further if the structured nature of the point cloud were taken into account. A similar test to the test above was run, with the target location moving horizontally across the bookshelf, with some penetration at each location. On average, SIP instantiates 7.8 constraints and terminates in 77.5 ms.

3.2 Trajectory optimization

Experiments illustrated in Figs. 6 and 7 test the ability of SIP to handle non-convex geometries in robot trajectory optimization. In each case, the start and end of the trajectory were fixed while 10 intermediate milestones were optimized. The robot model here is the 6DOF Staübli TX90L industrial manipulator, with a pipe attached to its terminal link. In both cases, objects from the Princeton

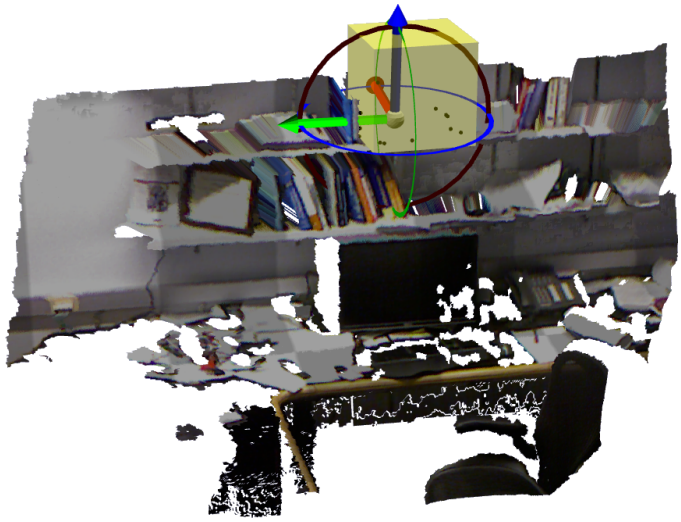


Fig. 5. Optimizing a geometry to be as close as possible to the given widget while avoiding contact with the point cloud. The 640×480 point cloud is obtained from the Cornell RGB-D scene dataset and contains 228,352 valid points.

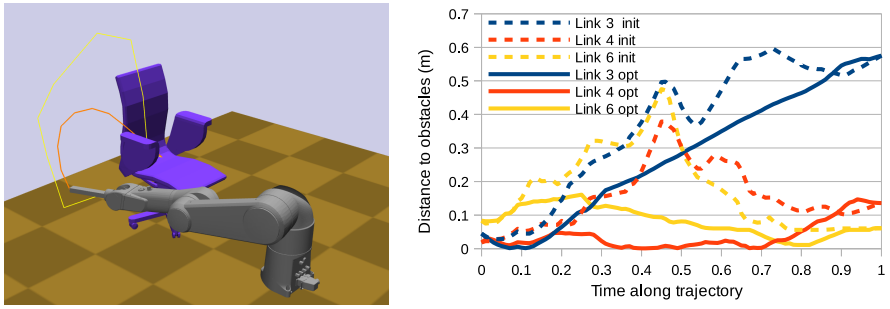


Fig. 6. Optimizing a robot trajectory in close proximity to an office chair obstacle. Left: The yellow curve is the initial end effector trajectory, and the orange one is the optimized trajectory. Right: distances to the environment for 3 links supporting the optimized trajectory.

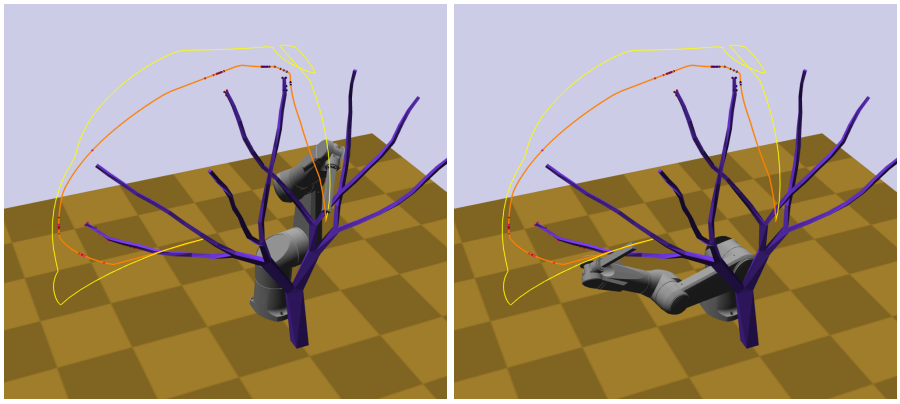


Fig. 7. Optimizing a robot trajectory in close proximity to a tree-shaped obstacle. The yellow curve is the initial end effector trajectory, and the orange one is the optimized trajectory. Dots along the trajectory and on the obstacle indicate the instantiated constraint points. Left: start configuration. Right: goal configuration.

Shape Benchmark [13] are instantiated near the robot, and the optimizer minimizes the sum of squared distances between milestones. The robot link geometries are converted to SDFs with 2 cm resolution, and the obstacle is represented as a point cloud with 2 cm resolution. This supersampling is performed by adaptively subdividing the longest edge of the obstacle mesh until the maximum edge length is less than 2 cm. A suboptimal collision-free trajectory is given as the initial seed.

Fig. 6 shows the optimized path for the chair obstacle, in which the end effector rises from under the seat and slides between the chair back and the armrest. The plot shows robot-obstacle distances along the original and optimized path for each of the limiting links. This demonstrates that the trajectory is first limited by link 3 (the "elbow") which passes very close to the underside of the seat. Next, link 4 (the "forearm") passes around the armrest. Finally, link 6 (the end effector) slides along the seat back and chair. This path was computed in 7.26 s, and it should be noted that running time is heavily dependent on the desired level of convergence. This example exhausts the maximum of 50 iterations, with the bulk of iterations performing "fine-tuning" with step magnitude < 0.01. Terminating at 20 iterations would have terminated in 2.23 s but sacrifices only 10% of optimality.

Fig. 7 shows the optimized path for the tree obstacle. The robot must extract itself between two branches, and then pass underneath the lower-left branch. Here, collision with the end effector is the primary limiting constraint. Total computation time is 5.77 s, and the objective function value is still somewhat improving after 50 iterations. Fig. 8 plots computation time against the objective function extending to 100 iterations, showing convergence at around 7 s. Another issue to examine is the number of milestones used in the path representation. As the number of milestones increases, the path becomes slightly more optimized.

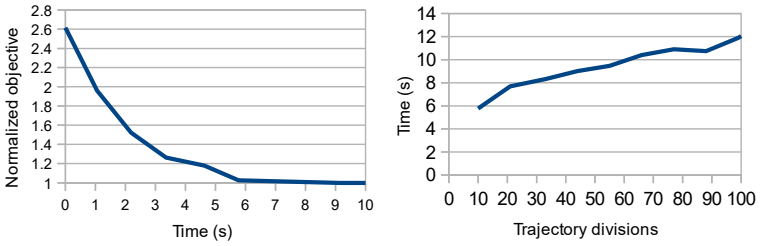


Fig. 8. Left: convergence of trajectory optimization over time on the example of Fig. 7. Right: running times vs the number of milestones in the trajectory representation.

Running times are shown in Fig. 8, showing a roughly linear and relatively shallow relationship between milestone count and running time.

It should be noted that our method, like other optimization methods, performs much better when initialized with a collision-free trajectory, rather than having to extricate the robot from a deeply-penetrating pose. This is because the separation direction is poorly approximated when the objects penetrate deeply, and furthermore the interaction between geometric penetration and robot kinematics is highly nonlinear and complex. For example, if an initial configuration has the robot's hand behind a two-sided wall, the back-face of the wall would suggest that the forearm should move forward through the wall in order to resolve the collision. As a result, optimization performs better when used as a postprocessor for a feasible motion planner, e.g., PRM or RRT, rather than a replacement.

4 Conclusion

This paper presents a novel optimization method to handle nonpenetration constraints with highly non-convex and geometric complex objects. A semi-infinite programming approach combined with an efficient deepest-penetration oracle allows it to be applied to robot pose and trajectory optimization with models composed of hundreds of thousands of primitives. Experiments demonstrate that the approach rapidly converges in challenging scenarios. Code for the algorithms can be found at `https://github.com/krishauser/SemiInfiniteOptimization`.

There are still many issues to be addressed regarding local minima. Especially in deeply-penetrating cases, performance is likely to be improved with better estimation techniques for penetration depth and direction. We are also interested in integrating this approach with feasible motion planners to produce an optimal motion planner with fast convergence rates.

Acknowledgment

The author thanks Tracy Lu for assistance with proofreading this manuscript. This work is partially supported by NSF grants #1253553 and #1527826.

References

1. J. E. Bobrow. Optimal robot plant planning using the minimum-time criterion. *IEEE Journal on Robotics and Automation*, 4(4):443–450, 1988.
2. N. Chakraborty, J. Peng, S. Akella, and J. E. Mitchell. Proximity queries between convex objects: An interior point approach for implicit surfaces. *IEEE Transactions on Robotics*, 24(1):211–220, 2008.
3. S. Dubowsky, M. Norris, and Z. Shiller. Time optimal trajectory planning for robotic manipulators with obstacle avoidance: A cad approach. In *IEEE Int. Conf. on Robotics and Automation*, volume 3, pages 1906–1912. IEEE, 1986.
4. E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE J. Robotics and Automation*, 4(2):193–203, 1988.
5. S. Gottschalk, M. C. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *Proc. Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 171–180. ACM, 1996.
6. S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.
7. H. S. Koppula, A. Anand, T. Joachims, and A. Saxena. Semantic labeling of 3d point clouds for indoor scenes. In *Neural Information Processing Systems*, 2011.
8. N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In *IEEE Int. Conf. Robotics and Automation*, pages 489–494. IEEE, 2009.
9. R. Reemtsen and S. Görner. Numerical methods for semi-infinite programming: a survey. In *Semi-infinite programming*, pages 195–275. Springer, 1998.
10. A. Richards, T. Schouwenaars, J. P. How, and E. Feron. Spacecraft trajectory planning with avoidance constraints using mixed-integer linear programming. *Journal of Guidance, Control, and Dynamics*, 25(4):755–764, 2002.
11. S. F. Saramago and V. S. Junior. Optimal trajectory planning of robot manipulators in the presence of moving obstacles. *Mechanism and Machine Theory*, 35(8):1079–1094, 2000.
12. J. Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, and P. Abbeel. Motion planning with sequential convex optimization and convex collision checking. *The International Journal of Robotics Research*, 33(9):1251–1270, 2014.
13. P. Shilane, P. Min, M. Kazhdan, and T. Funkhouser. The princeton shape benchmark. In *Shape Modeling International*, Genova, Italy, June 2004.
14. B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. OSQP: An operator splitting solver for quadratic programs. *ArXiv e-prints*, Nov. 2017.
15. A. I. F. Vaz, E. M. Fernandes, and M. P. S. Gomes. Robot trajectory planning with semi-infinite programming. *European Journal of Operational Research*, 153(3):607–617, 2004.
16. B. Zeng and L. Zhao. Solving two-stage robust optimization problems using a column-and-constraint generation method. *Operations Research Letters*, 41(5):457–461, 2013.