Safe, Optimal, Real-Time Trajectory Planning With a Parallel Constrained Bernstein Algorithm

Shreyas Kousik, *Member, IEEE*, Bohao Zhang, *Graduate Student Member, IEEE*, Pengcheng Zhao, *Member, IEEE*, and Ram Vasudevan, *Member, IEEE*

***Abstract*—To move while using new sensor information, mobile robots use receding-horizon planning, executing a short plan while computing a new one. A plan should have dynamic feasibility (obeying a robot's dynamics and avoiding obstacles), liveness (planning frequently enough to complete tasks), and optimality (minimizing, e.g., distance to a goal). Reachability-based trajectory design (RTD) is a method to generate provably dynamically feasible plans in real time by solving a polynomial optimization program (POP) in each planning iteration. However, RTD uses a derivative-based solver, which may converge to local minima that impact liveness and optimality. This article proposes a parallel constrained Bernstein algorithm (PCBA) branch-and-bound method to optimally solve RTD's POP at runtime; the resulting optimal planner is called RTD*. The specific contributions of this article are the PCBA implementation, proofs of PCBA's bounded time and memory usage, a comparison of PCBA with state-of-the-art solvers, and a demonstration of PCBA/RTD* on hardware. RTD* shows better optimality and liveness than RTD in dozens of environments with random obstacles.**

***Index Terms*—Motion planning, robot control, trajectory optimization.**

I. Introduction

FOR mobile robots to operate successfully in unforeseen environments, they must plan their motion as new sensor information becomes available. This *receding-horizon* strategy requires iteratively generating a plan while simultaneously executing a previous plan. Typically, this requires solving an optimization program in each planning iteration (see, e.g., [1]).

This work considers a mobile ground robot tasked with reaching a global goal location in an arbitrary static environment. To assess receding-horizon planning performance, we consider the following characteristics of plans. First, a plan should be *dynamically feasible*, meaning that it obeys the dynamic description of the robot and obeys constraints such as actuator limits and obstacle avoidance. Second, a plan should maintain *liveness*, by which we mean it keeps the robot moving without stopping frequently to replan; frequent stops can prevent a robot from achieving a task in a user-specified amount of time. Third, a plan should be *optimal* with respect to a user-specified cost function, such as reaching a goal quickly.

Ensuring that plans have these characteristics is challenging for several reasons. First, robots typically have nonlinear dynamics; this means that creating a dynamically feasible plan often requires solving a nonlinear program (NLP) at runtime.

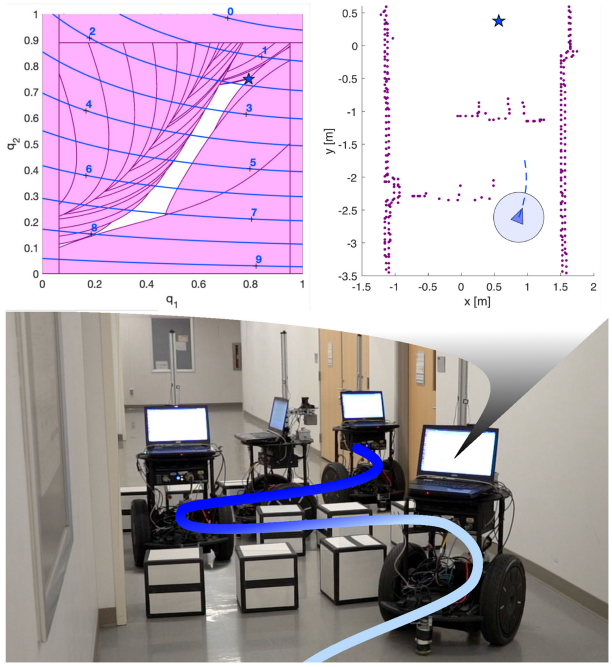

Fig. 1. Segway RMP mobile robot using the proposed PCBA/RTD* method to autonomously navigate a tight obstacle blockade. The executed trajectory is shown fading from light to dark blue as time passes, and the robot is shown at four different time instances. The top right plot shows the Segway's (blue circle with triangle indicating heading) view of the world at one planning iteration, with obstacles detected by a planar lidar (purple points). The top left plot shows the optimization program solved at the same planning iteration; the decision variable is (q_1, q_2), which parameterizes the velocity and yaw rate of a trajectory plan; the pink regions are infeasible with respect to constraints generated by the obstacle points in the right plot; and the blue contours with number labels depict the cost function, which is constructed to encourage the Segway to reach a waypoint (the star in the top right plot). The optimal solution found by PCBA is shown as a star on the left plot, in the nonconvex feasible area (white). This optimal solution generates a provably safe trajectory for the Segway to track, shown as a blue dashed line in the right plot. A video is available at roahmlab.com/PCBA_demo.

Manuscript received August 27, 2020; accepted October 14, 2020. This work was supported by the Ford Motor Company via the Ford-UM Alliance under Award N022977, and in part the Office of Naval Research under Award N00014-18-1-2575, and in part by National Science Foundation Career Award #1751093. This article was recommended for publication by Associate Editor R. Hatton and Editor W. Burgard upon evaluation of the reviewers' comments. *(Shreyas Kousik, Bohao Zhang, and Pengcheng Zhao contributed equally to this work.) (Corresponding author: Shreyas Kousik.)*

The authors are with the Department of Mechanical Engineering, University of Michigan, Ann Arbor, MI 48109 USA (e-mail: skousik@stanford.edu; jimzhang@umich.edu; pczhao@umich.edu; ramv@umich.edu).

Color versions of one or more of the figures in this article are available online at https://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TRO.2020.3036617

However, it is difficult to certify that an NLP can be solved in a finite amount of time, meaning that the robot may have to sacrifice liveness. Furthermore, even if a robot has linear dynamics, the cost function may be nonlinear; in this case, it is challenging to certify optimality due to local minima.

This article extends prior work on reachability-based trajectory design (RTD). RTD is able to provably generate *dynamically feasible* trajectory plans in real time, but cannot guarantee optimality or liveness (the robot will generate plans in real time, but may brake to a stop often). We address this gap by proposing the parallel constrained Bernstein algorithm (PCBA), which provably finds globally optimal solutions to polynomial optimization problems (POPs), a special type of NLP. We apply the PCBA to RTD to produce an optimal version of RTD, which we call RTD* in the spirit of the well-known RRT* algorithm [2]. We show on hardware that RTD* improves a robot's *liveness* (in comparison to RTD) for trajectory optimization.

A. Related Work

1) Receding-Horizon Planning: A variety of methods exist that attempt receding-horizon planning while maintaining dynamic feasibility, liveness, and optimality. These methods can be broadly classified by whether they perform sampling or solve an optimization program at each planning iteration. Sampling-based methods typically either attempt to satisfy liveness and dynamic feasibility by choosing samples offline [3], [4] or attempt to satisfy optimality at the potential expense of liveness and dynamic feasibility [2]. Optimization-based methods attempt to find a single optimal trajectory. These methods typically have to sacrifice dynamic feasibility (e.g., by linearizing the dynamics) to ensure liveness [5] or sacrifice liveness to attempt to satisfy dynamic feasibility [6], [7] (also see [8, Sec. 9] and [9]).

2) Reachability-Based Trajectory Design: RTD is an optimization-based receding-horizon planner that requires solving a POP at each planning iteration [8], [10]–[14]. RTD specifies plans as parameterized trajectories. Since these trajectories cannot necessarily be perfectly tracked by the robot, RTD begins with an offline computation of a forward reachable set (FRS). The FRS contains every parameterized plan, plus the tracking error that results from the robot not tracking any plan perfectly. At runtime, in each planning iteration, the FRS is intersected with sensed obstacles to identify all parameterized plans that could cause a collision (i.e., be dynamically infeasible). This set of unsafe plans is represented as a (finite) list of polynomial constraints, and the user is allowed to specify an arbitrary polynomial cost function, resulting in a POP. At each planning iteration, either the robot successfully solves the POP to find a new plan or it continues executing its previously found plan. While the decision variable is typically only two- or three-dimensional, each POP often has hundreds of constraints, making it challenging to find a feasible solution in real time [8]. Each plan includes a braking maneuver, ensuring that the robot can always come safely to a stop if the POP cannot be solved quickly enough in any planning iteration.

Note that, for RTD, *optimality* means finding the optimal solution to a POP at each planning iteration. The cost function in RTD's POPs typically encodes behavior such as reaching a waypoint between the robot's current position and the global goal (e.g., [8, Sec. 9.2.1]; RTD attempts to find the best dynamically feasible trajectory to the waypoint. RTD does not attempt to find the *best waypoints* themselves (best, e.g., with respect to finding the shortest path to the global goal). Such waypoints can be generated quickly by algorithms such as A* or RRT* by ignoring dynamic feasibility [2], [8].

3) Polynomial Optimization Problems: POPs require minimizing (or maximizing) a polynomial objective function, subject to polynomial equality or inequality constraints. As a fundamental class of problems in nonconvex optimization, POPs arise in various applications, including signal processing [15]–[17], quantum mechanics [18], [19], control theory [8], [20], [21], and robotics [22], [23]. This article presents a novel PCBA for solving POPs.

The difficulty of solving a POP increases with the dimension of the cost and constraints, with the number of constraints, and with the number of optima [24]. Existing methods attempt to solve POPs while minimizing time and memory usage (i.e., complexity). Doing so typically requires placing limitations on one of these axes of difficulty to make solving a POP tractable. These methods broadly fall into the following categories: derivative-based, convex relaxation, and branch-and-bound.

Derivative-based methods use derivatives (and sometimes Hessians) of the cost and constraint functions, along with first- or second-order optimality conditions [24, Sec. 12.3, 12.5], to attempt to find optimal, feasible solutions to nonlinear problems such as POPs. These methods can find local minima of POPs rapidly despite high dimension, a large number of constraints, and high degree cost and constraints [24, Ch. 19.8]. However, these methods do not typically converge to global optima without requiring assumptions on the problem and constraint structure (see, e.g., [25]).

Convex relaxation methods attempt to find global optima by approximating the original problem with a hierarchy of convex optimization problems. These methods can be scaled to high-dimensional problems (up to ten dimensions), at the expense of limits on the degree and sparse structure of the cost function; furthermore, they typically struggle to handle large numbers of constraints (e.g., the hundreds of constraints that arise in RTD's POPs), unless the problem has low-rank or sparse structure [22]. Well-known examples include the lift-and-project linear program (LP) procedure [26], reformulation-linearization technique [27], and semidefinite program (SDP) relaxations [22], [28]. By assuming structure such as homogeneity of the cost function or convexity of the domain and constraints, one can approximate solutions to a POP in polynomial time, with convergence to global optima in the limit [29]–[33]. Convergence within a finite number of convex hierarchy relaxations is possible under certain assumptions (e.g., a limited number of equality constraints [34], [35]).

Branch-and-bound methods perform an exhaustive search over the feasible region. These methods are typically limited

to up to four dimensions, but can handle large numbers of constraints and high degree cost and constraints. Examples include interval analysis techniques [36]–[39] and the Bernstein algorithm (BA) [40]–[42]. Interval analysis requires cost and constraint function evaluations in each iteration and, therefore, can be computationally slow. The BA, on the other hand, does not evaluate the cost and constraint functions; instead, the BA represents the coefficients of the cost and constraints in the Bernstein basis, as opposed to the monomial basis. The coefficients in the Bernstein basis provide lower and upper bounds on the polynomial cost and constraints over box-shaped subsets of Euclidean space by using a subdivision procedure [43], [44]. In other words, the unique properties of the Bernstein basis are suitable for polynomial optimization. Note that one can also use the Bernstein basis to transform a POP into an LP on each subdivided portion of the problem domain, which allows one to find tighter solution bounds given by the Bernstein coefficients alone [42]. Since subdivision can be parallelized [45], the time required to solve a POP can be greatly reduced by implementing BA on a graphics processing unit (GPU). In terms of developing BA, our work picks up where these previous results left off.

B. Contributions and Article Organization

This article makes four contributions. First, we extend the parallelized BA from [45] to include inequality and equality constraints, resulting in our proposed *PCBA*. Second, we prove bounds on the rate of convergence and memory usage of PCBA (and therefore of parallel BA), which are not shown in [45]. Third, we benchmark PCBA on a suite of well-known POPs, on which it outperforms the bounded sum of squares (BSOS) [35] solver, a generic nonlinear solver (MATLAB's fmincon), and the DIRECT branch-and-bound solver [39]. Fourth, we apply the PCBA to RTD to make *RTD**, a provably safe, optimal, and real-time trajectory planning algorithm for mobile robots, thereby demonstrating *dynamic feasibility*, *liveness*, and *optimality*. We have also made our code open source: github.com/ramvasudevan/GlobOptBernstein.

1) Novelty: Our work is most similar to [8], [41], and [45]. However, [45], which presents a parallelized BA, does not include constraints or prove bounds on the rate of convergence or memory usage. Similarly, [41] presents a constrained BA without parallelization, equality constraints, or proofs of bounds. On the other hand, [8], which proposes RTD, does not incorporate any notion of provable optimality, which limits a robot's liveness during real-time trajectory planning. We directly address each of these limitations. Critically, the present work bridges the gap between the BA literature and the robotics literature. To the best of our knowledge, this is the first practical application of BA to mobile robotics.

2) Article Organization: This article is organized as follows. Section II introduces POPs and Bernstein form. Section III introduces RTD. Section IV presents our proposed PCBA. Section V proves bounds on PCBA's convergence and memory usage. Section VI benchmarks PCBA against BSOS [35], fmincon [46], and DIRECT [39]. Section VII demonstrates PCBA/RTD* on hardware. Section VIII concludes this article.

II. POPs and Bernstein Form

This section introduces notation, POPs, the Bernstein form, and subdivision. These form the basis for our proposed PCBA method in Section IV.

A. Notation

1) Polynomial Notation: We follow the notation in [41]. Let $x := (x_1, x_2, \ldots, x_l) \in \mathbb{R}^l$ be real variables of dimension $l \in \mathbb{N}$. A multi-index J is defined as $J := (j_1, j_2, \ldots, j_l) \in \mathbb{N}^l$, and the corresponding multipower x^J is defined as $x^J := (x_1^{j_1}, x_2^{j_2}, \ldots, x_l^{j_l}) \in \mathbb{R}^l$. Given another multi-index $N := (n_1, n_2, \ldots, n_l) \in \mathbb{N}^l$ of the same dimension, an inequality $J \leq N$ should be understood componentwise. An l-variate polynomial p in canonical (monomial) form can be written as

$$p(x) = \sum_{J \leq N} a_J x^J, \quad x \in \mathbb{R}^l \tag{1}$$

with coefficients $a_J \in \mathbb{R}$ and some multi-index $N \in \mathbb{N}^l$. The space of polynomials of degree $d \in \mathbb{N}$, with variable $x \in \mathbb{R}^l$, is $\mathbb{R}_d[x]$.

Definition 1: We call $N \in \mathbb{N}^l$ the *multidegree* of a polynomial p; each ith element of N is the maximum degree of the variable x_i out of all of the monomials of p. We call $d \in \mathbb{N}$ the *degree* of p; d is the maximum sum, over all monomials of p, of the powers of the variable x. That is, $d = ||N||_1$, where $||\cdot||_1$ is the sum of the elements of a multi-index.

2) Point and Set Notation: Let $\mathbf{x} := [\underline{x}_1, \overline{x}_1] \times \cdots \times [\underline{x}_l, \overline{x}_l] \subset \mathbb{R}^l$ denote a general l-dimensional box in $\mathbb{R}^n$, with $-\infty < \underline{x}_\mu < \overline{x}_\mu < +\infty$ for each $\mu = 1, \ldots, l$. Let $\mathbf{u} := [0,1]^l \subset \mathbb{R}^l$ be the l-dimensional unit box. Denote by $|\mathbf{x}|$ the maximum width of a box $\mathbf{x}$, i.e., $|\mathbf{x}| = \max\{\overline{x}_\mu - \underline{x}_\mu : \ \mu = 1, \ldots, l\}$. For any point $y \in \mathbb{R}^l$, denote by $\|y\|$ the Euclidean norm of y, and denote by $\mathcal{B}_R(y)$ the closed Euclidean ball centered at y with radius $R > 0$.

B. Polynomial Optimization Problems

We denote a POP as follows:

$$\begin{array}{ll} \min\limits_{x \in D \subset \mathbb{R}^l} & p(x) \\ \text{s.t} & g_i(x) \leq 0 \ \ i = 1, \ldots, \alpha \\ & h_j(x) = 0 \ \ j = 1, \ldots, \beta. \end{array} \tag{P}$$

The decision variable is $x \in D \subset \mathbb{R}^n$, where D is a compact box-shaped domain and $l \in \mathbb{N}$ is the *dimension* of the program. The cost function is $p \in \mathbb{R}_d[x]$, and the constraints are $g_i, h_j \in \mathbb{R}_d[x]$ $(\alpha, \beta \in \mathbb{N})$. We assume for convenience that d is the greatest degree among the cost and constraint polynomials; we call d the *degree of the problem*.

C. Bernstein Form

To solve POPs, we take advantage of several properties of the *Bernstein form*. A polynomial p in monomial form (1) can be expanded into Bernstein form over an arbitrary l-dimensional

box $\mathbf{x}$ as

$$p(x) = \sum_{J \leq N} B_J^{(N)}(\mathbf{x})\, b_J^{(N)}(\mathbf{x}, x) \tag{2}$$

where $b_J^{(N)}(\mathbf{x}, \cdot)$ is the Jth multivariate *Bernstein polynomial* of multidegree N over $\mathbf{x}$, and $B_J^{(N)}(\mathbf{x})$ are the corresponding *Bernstein coefficients* of p over $\mathbf{x}$. A detailed definition of Bernstein form is available in [47]. Note that the Bernstein form of a polynomial can be determined quickly [48], by using a matrix multiplication on a polynomial's monomial coefficients, with the matrix determined by the polynomial degree and dimension. This matrix can be precomputed, and the conversion from monomial to Bernstein basis only needs to happen once for the proposed method (see Algorithm 1).

For convenience, we collect all such Bernstein coefficients in a multidimensional array $B(\mathbf{x}) := (B_J^{(N)}(\mathbf{x}))_{J \leq N}$, which is called a *patch*. We denote by $\min B(\mathbf{x})$ (respectively, $\max B(\mathbf{x})$) the minimum (respectively, maximum) element in the patch $B(\mathbf{x})$. The range of polynomial p over $\mathbf{x}$ is contained within the interval spanned by the extrema of $B(\mathbf{x})$, formally stated as the following theorem.

Theorem 2 (see [41, Lemma 2.2]): Let p be a polynomial defined as in (2) over a box $\mathbf{x}$. Then, the following property holds for a patch $B(\mathbf{x})$ of Bernstein coefficients:

$$\min B(\mathbf{x}) \leq p(x) \leq \max B(\mathbf{x}) \quad \forall x \in \mathbf{x}. \tag{3}$$

This theorem provides a means to obtain enclosure bounds of a multivariate polynomial over a box by transforming the polynomial to Bernstein form. Note that when the min/max values of p on $\mathbf{x}$ occur at the boundary of $\mathbf{x}$, the bounds are "sharp," meaning the max/min Bernstein coefficients are equal to the max/min of p [49]. This range enclosure can be further improved either by degree elevation or by subdivision. This work uses subdivision, discussed next, to refine the bounds.

D. Subdivision Procedure

Consider an arbitrary box $\mathbf{x} \subset \mathbb{R}^l$. The range enclosure in Theorem 2 is improved by subdividing $\mathbf{x}$ into subboxes and computing the Bernstein patches over these subboxes. A *subdivision* in the rth direction ($1 \leq r \leq l$) is a bisection of $\mathbf{x}$ perpendicular to this direction. That is, let

$$\mathbf{x} := [\underline{x}_1, \overline{x}_1] \times \cdots \times [\underline{x}_r, \overline{x}_r] \times \cdots \times [\underline{x}_l, \overline{x}_l] \tag{4}$$

be an arbitrary box over which the Bernstein patch $B(\mathbf{x})$ is already computed. By subdividing $\mathbf{x}$ in the rth direction, we obtain two subboxes $\mathbf{x}_L$ and $\mathbf{x}_R$, defined as

$$\begin{aligned} \mathbf{x}_L &= [\underline{x}_1, \overline{x}_1] \times \cdots \times [\underline{x}_r, (\underline{x}_r + \overline{x}_r)/2] \times \cdots \times [\underline{x}_l, \overline{x}_l] \\ \mathbf{x}_R &= [\underline{x}_1, \overline{x}_1] \times \cdots \times [(\underline{x}_r + \overline{x}_r)/2, \overline{x}_r] \times \cdots \times [\underline{x}_l, \overline{x}_l]. \end{aligned} \tag{5}$$

Note that we have subdivided $\mathbf{x}$ by halving its width in the rth direction; we choose $1/2$ as the subdivision parameter in this work, but one can choose a different value in [0,1] (see [41, eq. (10)]).

The new Bernstein patches, $B(\mathbf{x}_L)$ and $B(\mathbf{x}_R)$, can be computed by a finite number of linear transformations [41, Sec. 2.2] as

$$\begin{aligned} B(\mathbf{x}_L) &= M_{r,L} B(\mathbf{x}) \\ B(\mathbf{x}_R) &= M_{r,R} B(\mathbf{x}). \end{aligned} \tag{6}$$

where $M_{r,L}$ and $M_{r,R}$ are constant matrices, which can be precomputed, for each r (notice that [41, eq. (10)] obtains $B(\mathbf{x}_L)$ with linear operations on $B(\mathbf{x})$). The patches and one iteration of the subdivision procedure are shown in Fig. 2.

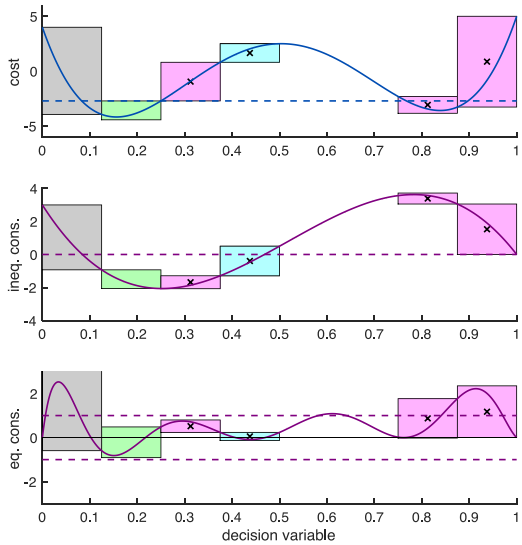

Fig. 2. Third iteration of PCBA (see Algorithm 1) on a one-dimensional polynomial cost (top) with one inequality constraint (middle) and one equality constraint (bottom). The rectangles represent Bernstein patches (as in Section II-D), where the horizontal extent of each patch corresponds to an interval of the decision variable, over which Bernstein coefficients are computed. The top and bottom of each patch represent the maximum and minimum Bernstein coefficients, which bound the cost and constraint polynomials on the corresponding interval. As per Definition 9, the green patch is feasible, the pink patches are infeasible, and the gray patches are undecided; the purple dashed lines show the inequality constraint cutoff (zero) and the equality constraint tolerance $\epsilon_{\text{eq}} = 1$ (note that ϵ_{eq} is chosen to be this large only for illustration purposes). Per Definition 11, the light blue patch is suboptimal; the blue dashed line in the top plot is the current solution estimate (see Definition 10). The infeasible and suboptimal patches are each marked with $\times$ for elimination (see Algorithm 5), since they cannot contain the global optimum (see Theorem 12); the feasible and undecided patches are kept for the next iteration. The leftmost pink patch is both suboptimal and infeasible, but is checked for infeasibility before suboptimality in Algorithm 4.

Remark 3: To reduce wordiness, we say that we *subdivide a patch* to mean the subdivision of a single subbox into two subboxes *and* the computation of the corresponding Bernstein patches for the POP cost and constraints.

By repeatedly applying the subdivision procedure and Theorem 2, the bounds on the range of polynomial in a subbox can be improved. In fact, such bounds can be exact in the limiting sense if the subdivision is applied evenly in all directions.

Theorem 4 (see [50, Th. 2]): Let $\mathbf{x}^{(n)}$ be a box of maximum width 2^{-n} ($n \in \mathbb{N}$) and let $B(\mathbf{x}^{(n)})$ be the corresponding Bernstein patch of a given polynomial p; then

$$\min B(\mathbf{x}^{(n)}) \leq \min_{x \in \mathbf{x}^{(n)}} p(x) \leq \min B(\mathbf{x}^{(n)}) + \zeta \cdot 2^{-2n} \tag{7}$$

where ζ is a nonnegative constant that can be given explicitly independent of n.

Notice from the proof of Theorem 4 that changing the sign of p does not change the value of ζ. By substituting p with $-p$ into Theorem 4, one can easily show that a similar result holds for the maximum of p over $\mathbf{x}^{(n)}$.

Corollary 5: Let $\mathbf{x}^{(n)}$ and $B(\mathbf{x}^{(n)})$ be as in Theorem 4. Then, we have

$$\max B(\mathbf{x}^{(n)}) - \zeta \cdot 2^{-2n} \leq \max_{x \in \mathbf{x}^{(n)}} p(x) \leq \max B(\mathbf{x}^{(n)}) \quad (8)$$

where ζ is the same nonnegative constant as in Theorem 4.

Theorem 4 and Corollary 5 provide shrinking bounds for values of a polynomial over subboxes as the subdivision process continues. By comparing the bounds over all subboxes, one can argue that the minimizers of a polynomial may appear in only a subset of the subboxes. This idea underlies the *BA* for solving POPs [41], [44], discussed in Section IV.

Next, we provide an overview of RTD to motivate our proposed method by introducing a specific POP that arises in safe real-time planning for mobile robots.

III. Reachability-Based Trajectory Design

RTD is a method for provably safe, real-time, receding-horizon trajectory planning, but comes with no optimality guarantees [8], [10]–[14]. In this section, we apply RTD to the differential-drive Segway robot in Fig. 1. RTD performs trajectory optimization by solving a POP at each receding-horizon planning iteration. In [8], this POP is solved with a generic nonlinear solver, which often converges to local minima or infeasible solutions, causing the robot to safely brake as opposed to moving through its environment. In this work, by applying PCBA to the RTD POP, we make RTD*, which provably finds the best feasible trajectory plan (if one exists) in each receding-horizon iteration. See [8] and [51, Ch. 3] for a more detailed and general overview of RTD.

To proceed, we review RTD's offline robot modeling, trajectory parameterization, tracking error, and FRS computation. We then pose RTD's online receding-horizon trajectory optimization POP.

A. Offline Modeling and Reachability Analysis

1) High-Fidelity Model: RTD requires that the robot is described by a *high-fidelity state-space model*, which accurately represents the robot's motion through the world. For the Segway robot in this work, we consider the state $(x_1, x_2, \theta, \omega, v)$ in the space $X_{\text{hi}} \subseteq \text{SE}(2) \times \mathbb{R}^2$. The state consists of position (x_1, x_2), heading θ, yaw rate ω, and speed v. Its control inputs are $(u_1, u_2) \in U \subseteq \mathbb{R}^2$, where u_1 is the yaw acceleration and u_2 is the longitudinal acceleration. Note that U is compact (meaning the inputs saturate). The high-fidelity model is $f_{\text{hi}} : X_{\text{hi}} \times U \to \mathbb{R}^5$, for which

$$f_{\text{hi}}(x_{\text{hi}}(t), u(t)) = \begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \dot{\theta}(t) \\ \dot{\omega}(t) \\ \dot{v}(t) \end{bmatrix} = \begin{bmatrix} v(t)\cos\theta(t) \\ v(t)\sin\theta(t) \\ \omega(t) \\ u_1(t) \\ u_2(t) \end{bmatrix} \quad (9)$$

where $t \in [0, \infty)$ is time. We further require that ω and v are bounded as $\omega(t) \in [-1, 1]$ rad/s and $v(t) \in [0, 1.2]$ m/s.

2) Trajectory Parameterization: Generating safe trajectory plans with (9) in real time is challenging due to nonlinearity and saturation. We, therefore, use a simpler *planning model* $f : T \times X \times Q \to \mathbb{R}^3$, specified offline, to generate plans at runtime. Plans (i.e., trajectories of the planning model) are defined over a short time horizon $T = [0, t_{\text{f}}]$. The model has state space $X = \mathbb{R}^2$ and *trajectory parameters* $q \in Q \subset \mathbb{R}^2$, where Q is compact. Plans obey the differential equation

$$f(t, x(t), q) = \begin{bmatrix} \dot{x}_1(t; q) \\ \dot{x}_2(t; q) \end{bmatrix} = s(t) \begin{bmatrix} q_2 - q_1 x_2(t; q) \\ q_1 x_1(t; q) \end{bmatrix} \quad (10)$$

and $\dot{q} = 0$. The notation $(t; q)$ indicates that a plan is parameterized by $q = (q_1, q_2)$, which are yaw rate q_1 and speed q_2 [8, Example 9]. Note that f implicitly parameterizes the planned heading as $\theta(t; q) = q_1 t$. Also, the position states are shared between the high-fidelity and planning models. The function $s : T \to [0, 1]$ is given by

$$s(t) = \min \left\{ 1, \left(\tfrac{t_{\text{f}} - t}{t_{\text{f}} - t_{\text{plan}}} \right)^4 \right\} \quad (11)$$

which causes the trajectory plan to brake to a stop from the time t_{plan} onwards (t_{plan} is discussed more in the following). The duration t_{f} is chosen to be large enough for the robot to brake to a stop from its maximum speed, subject to its maximum acceleration (recall that U is compact) [8, Remark 37].

3) Receding-Horizon Formulation: The duration $t_{\text{plan}} > 0$ is called the *planning time*, which we set to 0.5 s in this work. In each receding-horizon planning iteration, the robot has t_{plan} seconds within which to find a new trajectory plan (i.e., a new $q \in Q$). Otherwise, it must continue its plan from the previous planning iteration. So, s ensures that every plan ends with a braking maneuver. Therefore, assuming the robot has an initial collision-free plan, then the robot always has a fail-safe maneuver available even if it cannot find a collision-free plan in a given planning iteration [8, Sec. 5.3].

For real-time planning, the robot generates a new plan while executing its previous plan. This requires it to estimate its future initial condition for each plan as follows. First, every plan begins from $t = 0 \in T$; this is without loss of generality (WLOG) since the high-fidelity model is not time dependent. To find where each plan should begin in space, we forward-integrate the high-fidelity model, which uses a controller (discussed below) to track the previously found plan, for the duration t_{plan}. Since the high-fidelity model is not position or heading dependent, we then transform this initial condition to $(0, 0, 0, \omega_0, v_0)$ WLOG. We similarly treat every plan as beginning from the point $0 \in X$.

4) Tracking Controller and Tracking Error: Given a plan $q \in Q$, the robot uses a tracking controller $u_q : T \times X_{\text{hi}} \times X \to U$ to track it. The Segway uses a proportional–derivative controller [8, Example 10]. As mentioned earlier, the robot cannot necessarily perfectly track any plan, resulting in *tracking error* that must be compensated for to ensure collision avoidance. However, it is reasonable to assume that this tracking error is bounded, since 1) plans are drawn from a compact set Q;

2) the high-fidelity model is Lipschitz continuous in t, x_{hi}, and u; and 3) the duration of each plan T is compact. In particular, we assume that there exist functions $f_{\text{err},1}, f_{\text{err},2} : T \to \mathbb{R}$ that bound the tracking error in each coordinate of X [8, Assumption 13]. That is, we require $\max_{x_{\text{hi}} \in X_{\text{hi},0}} |x_{1,\text{hi}}(t; q) - x_1(t; q)| \leq \int_0^{t_{\text{f}}} f_{\text{err},1}(t)dt$, where $x_{1,\text{hi}}$ (respectively, x_1) is the trajectory of the high-fidelity model (respectively, planning model) in the first position coordinate; we require $f_{\text{err},2}$ similarly for the second position coordinate. The set $X_{\text{hi},0} \subset X_{\text{hi}}$ is the set of all initial conditions $x_{\text{hi}}(0; k) = (0, 0, 0, \omega_0, v_0)$; since the yaw rate and speed are bounded, the max is taken over a compact set. Also note that [51, Ch. 7] provides a general approach to estimating these tracking error functions.

Let $f_{\text{err}} = (f_{\text{err},1}, f_{\text{err},2}) : T \to \mathbb{R}^3$. Let $L_d = L^1(T, [-1, 1]^2)$ be the space of absolutely integrable functions from T to $[-1, 1]^2$. Consider the model $f + f_{\text{err}} \cdot d$, where $d \in L_d$ and the product is taken elementwise. By choosing d, this model allows us to add tracking error to a plan evolving in X (with dynamics f) and, therefore, replicate the motion of the high-fidelity model in the lower dimensional subspace X [8, Lemma 16]. In other words, we now have a low-dimensional representation of the high-fidelity model, which we use for reachability analysis next.

5) Forward Reachable Set: The FRS, denoted F, contains the points reached by the high-fidelity model when tracking any plan:

$$F = \{(x, q) \in X \times Q \mid \exists\, t \in T,\ d \in L_d \text{ s.t.} \\ \dot{y} = f + f_{\text{err}} \cdot d,\ y(0) = 0, \text{ and } y(t) = x\} \tag{12}$$

where $f_{\text{err}} \cdot d$ is again taken elementwise.

We compute the FRS by posing a sum-of-squares (SOS) program [8, Sec. 3, Program (D)] (also see [51, Ch. 4]). Per [8, Lemma 19], this program produces $g_{\text{FRS}} : X \times Q \to \mathbb{R}$, for which

$$(x, q) \in F \Rightarrow g_{\text{FRS}}(x, q) \geq 1. \tag{13}$$

In practice, we represent the SOS program using polynomials of finite degree, producing a conservative overapproximation of F [8, Remark 22]. In particular, we find g_{FRS} as a polynomial in $\mathbb{R}_{12}[x, q]$. Next, we discuss how we use g_{FRS} for runtime trajectory optimization.

B. Online Planning With a POP

RTD uses the FRS to generate collision avoidance constraints for trajectory optimization at runtime. To explain this procedure, we first discuss obstacles and then represent trajectory optimization as a POP using the polynomial g_{FRS}.

1) Obstacles: For this work, we assume that obstacles are static; note that RTD can also handle dynamic obstacles [11], [12]. We further assume that, in each planning iteration, obstacles are represented as a union of compact polygons, denoted $O \subset X$. We apply [8, Algorithm 1] to convert O into a finite *discretized obstacle* set $\{x_{\text{obs},i}\}_{i=1}^{N_{\text{obs}}} \subset X$, for which, if the robot avoids colliding with each $x_{\text{obs},i}$, then the robot avoids colliding with O [8, Th. 68]. Note that, per [8, Th. 39], we can also specify a minimum sensor distance, within which obstacles must be sensed to guarantee safety. The Segway senses obstacles, using a planar lidar, well within the 2-m minimum sensor distance required (see [8, Sec. 9.6]). Also note that we can treat occlusions as obstacles.

2) Trajectory Optimization: We perform trajectory optimization as follows. First, offline, we use Euler integration to approximate the planning model's position at time t_{f}, for any plan $q \in Q$, as a degree 10 polynomial $x_{\text{f}} : Q \to X$. Second, at runtime, we create waypoints used to construct an objective function. In particular, we generate $N_{\text{wp}} \in \mathbb{N}$ waypoints (i.e., desired locations), denoted $\{w_n\}_{n=1}^{N_{\text{wp}}} \subset \mathbb{R}^2$. We discuss how we generate these waypoints in Section VII for each hardware demonstration; in general, such waypoints can be created using, e.g., A* or RRT. Third, also at runtime, given the discretized obstacle $\{x_{\text{obs},i}\}_{i=1}^{N_{\text{obs}}}$, we create the following POP:

$$\underset{q \in Q}{\operatorname{argmin}} \prod_{n=1}^{N_{\text{wp}}} \left(\|x_{\text{f}}(q) - w_n\|_2^2 \right) \\ \text{s.t } g_{\text{FRS}}(x_i, q) + \epsilon_q \leq 1 \,\forall\, i = 1, \ldots, N_{\text{obs}}. \tag{14}$$

The objective function is degree $2 \times 10 \times N_{\text{wp}}$, and the constraints are each degree 12. Notice that the objective function has as many global minima as there are waypoints. For the constraints, the tolerance $\epsilon_{\text{obs}} = 10^{-4}$ is used to ensure optimization over a closed set (by (13), $g_{\text{FRS}}(x_i, q) < 1$ implies that the plan q avoids collision with the point x_i).

The takeaway of this section is that, if we can solve (14) every t_{plan} seconds at runtime, a robot can always find safe plans that move it through arbitrary environments in real time. This motivates the next two sections, wherein we propose PCBA and prove its rate of convergence. Then, we use PCBA to find global optima of (14) at runtime.

IV. Parallel Constrained Bernstein Algorithm

This section proposes the PCBA (see Algorithm 1) for solving a general POP. We extend the approach in [41]. This approach utilizes Bernstein form to obtain upper and lower bounds of both objective and constraint polynomials (see Theorem 2), iteratively improves such bounds using subdivision (see Theorem 4 and Corollary 5) and removes patches that are cannot contain a solution (see Theorem 12). We discuss the algorithm, the list used to store patches, tolerances and stopping criteria, subdivision, a cutoff test for eliminating patches, and the advantages and disadvantages of PCBA. Section V proves that PCBA finds globally optimal solutions to POPs up to user-specified tolerances.

Before proceeding, we make an assumption for notational convenience, and to make the initial computation of Bernstein patches easier.

Assumption 6: WLOG, the domain of the decision variable is the unit box (i.e., $D = \mathbf{u}$), since any nonempty box in $\mathbb{R}^l$ can be mapped affinely onto $\mathbf{u}$ [48].

A. Algorithm Summary

We now summarize PCBA, implemented in Algorithm 1. The algorithm is initialized by computing the Bernstein patches of

the cost and constraints on the domain $\mathbf{u}$ (Line 1). Subsequently, the PCBA subdivides each patch as in Remark 8 (Line 5 and Algorithm 2). Then, the PCBA finds the upper and lower bounds of each new patch (Line 6 and Algorithm 3). These bounds are used to determine which patches are feasible, infeasible, and undecided as in Definition 9 (Line 7; see Algorithm 4 and Theorem 12). Algorithm 4 also determines the current solution estimate (the smallest upper bound over all feasible patches) and marks any patches that are suboptimal as in Definition 11. If every patch is infeasible (Line 8), the PCBA returns that the problem is infeasible (Line 13); otherwise, the PCBA checks if the current solution estimate meets user-specified tolerances (Line 9). If the tolerances are met, the PCBA returns the solution estimate (Line 12). Otherwise, the PCBA eliminates all infeasible and suboptimal patches (Line 10 and Algorithm 5) and then moves to the next iteration (Line 11). Note that Algorithms 2, 3, and 5 are parallelized.

B. Items and the List

Denote an *item* as the tuple $\ell = (\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x})), B_{hj}(\mathbf{x}))$, where $B_{gi}(\mathbf{x})$ (respectively, $B_{hj}(\mathbf{x})$) is shorthand for the set of patches $\{B_{g_i}(\mathbf{x})\}_{i=1}^{\alpha}$ (respectively, $\{B_{h_j}(\mathbf{x})\}_{j=1}^{\beta}$). We use the following notation for items. If $\ell = (\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x}), B_{hj}(\mathbf{x}))$, then $\ell_1 = \mathbf{x}$, $\ell_2 = B_p(\mathbf{x})$, $\ell_3 = B_{gi}(\mathbf{x})$, and $\ell_4 = B_{hj}(\mathbf{x})$.

We denote the *list* $\mathcal{L} = \{\ell_\mu : \mu = 1, \ldots, N_\mathcal{L}\}$, $N_\mathcal{L} \in \mathbb{N}$, indexed by $\mu \in \mathbb{N}$. The PCBA adds and removes items from $\mathcal{L}$ by assessing the feasibility and optimality of each item.

C. Tolerances and Stopping Criteria

Recall that, by Theorem 2, Bernstein patches provide upper and lower bounds for polynomials over a box. From Theorem 4 and Corollary 5, as we subdivide $\mathbf{u}$ into smaller subboxes, the bounds of the Bernstein patches on each subbox more closely approximate the actual bounds of the polynomial. However, to ensure that the algorithm terminates, we must set tolerances on optimality and equality constraint satisfaction (the equality constraints $h_j(x) = 0$ may not be satisfied for *all* points in certain subboxes). During optimization, one also typically wants to find the optimal up to some resolution. In our case, this resolution corresponds to the maximum allowable subbox width, which we refer to as the *step tolerance*.

Definition 7: We denote the *optimality tolerance* as $\epsilon > 0$, the *equality constraint tolerance* as $\epsilon_{\text{eq}} > 0$, and the *step tolerance* as $\delta > 0$. We terminate Algorithm 1 either when $\mathcal{L}$ is empty (the problem is infeasible) or when there exists an item $(\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x})), B_{hj}(\mathbf{x})) \in \mathcal{L}$ that satisfies all of the following conditions.

1) $|\mathbf{x}| \leq \delta$.
2) $\max B_{gi}(\mathbf{x}) \leq 0$ for all $i = 1, \ldots, \alpha$.
3) $-\epsilon_{\text{eq}} \leq \min B_{hj}(\mathbf{x}) \leq 0 \leq \max B_{hj}(\mathbf{x}) \leq \epsilon_{\text{eq}}$ for all $j = 1, \ldots, \beta$.
4) $\max B_p(\mathbf{x}) - \min B_p(\mathbf{y}) \leq \epsilon$ for all $\mathbf{y} \in \mathcal{L}$.

We discuss feasibility in more detail in Section IV-E Note that, to implement the step tolerance δ, since we subdivide by halving the width of each subbox, we need only ensure that sufficiently many iterations have passed.

Note that we do not set a tolerance on inequality constraints, since these are "one-sided" constraints; for any inequality constraint g_i and subbox $\mathbf{x}$, we satisfy the constraint if $\max B_{gi}(\mathbf{x}) \leq 0$ (see Definition 9 and Theorem 16).

D. Subdivision

Recall that subdivision is presented in Section II-D. We implement subdivision with Algorithm 2. Since the subdivision of one Bernstein patch is computationally independent of another, each subdivision task is assigned to an individual GPU thread, making Algorithm 2 parallel.

Note that the subdivision of Bernstein patches can be done in any direction, leading to the question of how to select the direction in practice. Example rules are available in the literature, such as maximum width [52, Sec. 3], derivative-based [53, Sec. 3], or a combination of the two [52, Sec. 3]. In the context of constrained optimization, the maximum width rule is usually favored over derivative-based rules for two reasons: first, computing the partial derivatives of all constraint polynomials can introduce significant computational burden, especially when the number of constraints is large (see Section VI-C); second, the precision of Bernstein patches as bounds to the polynomials depends on the *maximum* width of each subbox (see Theorem 4 and Corollary 5), so it is beneficial to subdivide along the direction of maximum width for better convergence results.

In each nth iteration of PCBA, we subdivide in each direction r, in the order $1, 2, \ldots, l$. We halve the width of each subbox each time we subdivide, leading to the following remark.

Remark 8: In the nth iteration, the maximum width of any subbox in $\mathcal{L}$ is 2^{-n}.

E. Cutoff Test

Subdivision would normally occur for every patch in every iteration, leading to exponential memory usage (2^n patches in iteration n). However, by using a cutoff test, some patches can be deleted, reducing both the time and memory usage of PCBA (see Section V for complexity analysis). To decide which patches are to be eliminated, we require the following definitions.

Definition 9: An item $(\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x}), B_{hj}(\mathbf{x})) \in \mathcal{L}$ is *feasible* if both of the following hold.

1) $\max B_{gi}(\mathbf{x}) \leq 0$ for all $i = 1, \ldots, \alpha$.
2) $-\epsilon_{\text{eq}} \leq \min B_{hj}(\mathbf{x}) \leq 0 \leq \max B_{hj}(\mathbf{x}) \leq \epsilon_{\text{eq}}$ for all $j = 1, \ldots, \beta$.

An item is *infeasible* if any of the following hold.

3) $\min B_{gi}(\mathbf{x}) > 0$ for at least one $i = 1, \ldots, \alpha$.
4) $\min B_{hj}(\mathbf{x}) > 0$ for at least one $j = 1, \ldots, \beta$.
5) $\max B_{hj}(\mathbf{x}) < 0$ for at least one $j = 1, \ldots, \beta$.

An item is *undecided* if it is neither feasible nor infeasible.

Notice, in particular, that a feasible item must not be infeasible.

Definition 10: The *solution estimate* p^*_{up} is the smallest upper bound of the cost over all feasible items in $\mathcal{L}$:

$$p^*_{\text{up}} = \min \{\max\{\ell_2 \,|\, \ell \in \mathcal{L},\, \ell \text{ feasible}\}\} \tag{15}$$

Algorithm 1: Parallel Constrained Bernstein Algorithm.

Inputs: Polynomials p, $\{g_i\}_{i=1}^{\alpha}$, $\{h_j\}_{j=1}^{\beta}$ as in (P), of dimension l; optimality tolerance $\epsilon > 0$, step tolerance $\delta > 0$, and equality constraint tolerance $\epsilon_{\text{eq}} > 0$; and maximum number of patches $M \in \mathbb{N}$ and of iterations $N \in \mathbb{N}$.

Outputs: Estimate $p^* \in \mathbb{R}$ of optimal solution, and subbox $\mathbf{x}^* \subset \mathbf{u}$ containing optimal solution.

Algorithm:

1: Initialize patches of p, g_i, and h_j over l-dimensional initial domain box $\mathbf{u}$ as in [48]
$[B_p(\mathbf{u}), B_{gi}(\mathbf{u}), B_{hj}(\mathbf{u})] \leftarrow \text{InitPatches}(p, g_i, h_j)$
2: Initialize lists of undecided patches and patch extrema on the GPU
$\mathcal{L} \leftarrow \{(\mathbf{u}, B_p(\mathbf{u}), B_{gi}(\mathbf{u}), B_{hj}(\mathbf{u}))\}$
$\mathcal{L}_{\text{bounds}} \leftarrow \{\}$
3: Initialize iteration count and subdivision direction
$n \leftarrow 1, r \leftarrow 1$
4: Test for sufficient memory (iteration begins here)
if $2 \times \text{length}(\mathcal{L}) > M$ **then** go to Line 12
else continue
end if
5: (Parallel) Subdivide each patch in $\mathcal{L}$ in the rth direction to create two new patches using Algorithm 2
$\mathcal{L} \leftarrow \text{Subdivide}(\mathcal{L}, r)$
6: (Parallel) Find bounds of p, g_i, and h_j on each new patch using Algorithm 3
$\mathcal{L}_{\text{bounds}} \leftarrow \text{FindBounds}(\mathcal{L})$
7: Estimate upper bound p^*_{up} of the global optimum as the least upper bound of all feasible patches, and determine which patches to eliminate using Algorithm 4
$[p^*_{\text{up}}, p^*_{\text{lo}}, \mathcal{L}_{\text{save}}, \mathcal{L}_{\text{elim}}] \leftarrow \text{CutOffTest}(\mathcal{L}_{\text{bounds}})$;
8: Test if problem is feasible
if $\text{length}(\mathcal{L}_{\text{save}}) = 0$ **then** go to Line 13
9: Test stopping criteria for all $(\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x}), B_{hj}(\mathbf{x})) \in \mathcal{L}$
if $p^*_{\text{up}} - p^*_{\text{lo}} \leq \epsilon$
and $|\mathbf{x}| \leq \delta$
and $-\epsilon_{\text{eq}} \leq \min B_{hj}(\mathbf{x}) \leq \max B_{hj}(\mathbf{x}) \leq \epsilon_{\text{eq}}$
then go to Line 12
end if
10: (Parallel) Eliminate infeasible, suboptimal patches using Algorithm 5
$\mathcal{L} \leftarrow \text{Eliminate}(\mathcal{L}, \mathcal{L}_{\text{save}}, \mathcal{L}_{\text{elim}})$;
11: Prepare for next iteration
$r \leftarrow (\text{mod}(r+1, l)) + 1$
if $r = 1$ **then** $n \leftarrow n + 1$
end if
if $n = N$ **then** go to Line 12
else go to Line 4
end if
12: Return current best approximate solution
$p^* \leftarrow p^*_{\text{up}}$
$\mathbf{x}^* \leftarrow \mathbf{x}$ for which $\max B_p(\mathbf{x}) = p^*_{\text{up}}$
13: **return** $p^*, \mathbf{x}^*$
14: No solution found (problem infeasible)

Algorithm 2: $\mathcal{L} = \text{Subdivision}(\mathcal{L}, r)$ (Parallel).

1: $K \leftarrow \text{length}(\mathcal{L})$
2: **parfor** $k \in \{1, \ldots, K\}$ **do**
3: $(\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x}), B_{hj}(\mathbf{x})) \leftarrow \mathcal{L}[k]$;
4: Subdivide $\mathbf{x}$ along the rth direction into $\mathbf{x}_L$ and $\mathbf{x}_R$
5: Compute patches $B_p(\mathbf{x}_L)$ and $B_p(\mathbf{x}_R)$
6: Compute patches $B_{gi}(\mathbf{x}_L)$ and $B_{gi}(\mathbf{x}_R)$
7: Compute patches $B_{hj}(\mathbf{x}_L)$ and $B_{hj}(\mathbf{x}_R)$
8: $\mathcal{L}[k] \leftarrow (\mathbf{x}_L, B_p(\mathbf{x}_L), B_{gi}(\mathbf{x}_L), B_{hj}(\mathbf{x}_L))$
9: $\mathcal{L}[k+K] \leftarrow (\mathbf{x}_R, B_p(\mathbf{x}_R), B_{gi}(\mathbf{x}_R), B_{hj}(\mathbf{x}_R))$
10: **end parfor**
11: **return** $\mathcal{L}$

where $\ell_2 = B_p(\mathbf{x})$ if $\ell = (\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x}), B_{hj}(\mathbf{x}))$.

Definition 11: An item $(\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x}), B_{hj}(\mathbf{x})) \in \mathcal{L}$ is *suboptimal* if

$$\min B_p(\mathbf{x}) > p^*_{\text{up}}. \tag{16}$$

Note that Definitions 10 and 11 are dependent on $\mathcal{L}$, that is, for the purposes of PCBA, optimality is defined in terms of the elements of $\mathcal{L}$. We show in Corollary 13 how this notion of optimality coincides with optimality of the POP itself.

Feasible, infeasible, undecided, and suboptimal patches are illustrated in Fig. 2. Any item that is infeasible or suboptimal can be eliminated from $\mathcal{L}$, because the corresponding subboxes cannot contain the solution to the POP (formalized in the following theorem). We call checking for infeasible and suboptimal items the *cutoff test*.

Theorem 12 (Cutoff test): Let $(\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x}), B_{hj}(\mathbf{x})) \in \mathcal{L}$ be an item. If the item is infeasible (as in Definition 9) or suboptimal (as in Definition 11), then $\mathbf{x}$ does not contain a global minimizer of (P). Such an item can be removed from the list $\mathcal{L}$.

Proof: Let $(\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x}), B_{hj}(\mathbf{x}))$ be an item in $\mathcal{L}$. We only need to show the following.

a) If $(\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x}), B_{hj}(\mathbf{x}))$ is feasible, then all points in $\mathbf{x}$ are feasible (up to the tolerance ϵ_{eq}).
b) If $(\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x}), B_{hj}(\mathbf{x}))$ is infeasible, then all points in $\mathbf{x}$ are infeasible (up to the tolerance ϵ_{eq}).
c) If $(\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x}), B_{hj}(\mathbf{x}))$ is suboptimal, then all points in $\mathbf{x}$ are not optimal.

Note that (a) and (b) follows directly from Theorem 2. To prove (c), let $\mathbf{y} \subset \mathbf{u}$ be a subbox on which the solution estimate p^*_{up} is achieved, that is, $(\mathbf{y}, B_p(\mathbf{y}), B_{gi}(\mathbf{y}), B_{hj}(\mathbf{y}))$ is feasible and

$$\max B_p(\mathbf{y}) = p^*_{\text{up}}. \tag{17}$$

Let $y \in \mathbf{y}$ be arbitrary; then, it follows from Theorem 2 and the definition of suboptimality that

$$p(x) \geq \min B_p(\mathbf{x}) > \max B_p(\mathbf{y}) \geq p(y) \tag{18}$$

for all $x \in \mathbf{x}$. Since such point y is necessarily feasible [per condition (b)], x cannot be global minimum to the POP.

Corollary 13: Suppose that there exists a (feasible) global minimizer x^* of the POP (P). Then, while executing Algorithm 1, there always exists an item $(\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x}), B_{hj}(\mathbf{x})) \in \mathcal{L}$ such that $x^* \in \mathbf{x}$.

Proof: This result is the contrapositive of Theorem 12. ■

We implement the cutoff tests as follows. Algorithm 3 (FindBounds) computes the maximum and minimum element of each Bernstein patch, Algorithm 4 (CutOffTest) implements the cutoff tests and marks all subboxes to be eliminated with a list $\mathcal{L}_{\text{elim}}$, and Algorithm 5 (Eliminate) eliminates the marked subboxes from the list $\mathcal{L}$. Algorithms 3 and 5 are parallelizable, whereas Algorithm 4 must be computed serially.

F. Advantages and Disadvantages of PCBA

The PCBA has several advantages. First, it always finds a global optimum (if one exists), subject to tolerances. The PCBA does not require an initial guess and does not converge to

Algorithm 3: $\mathcal{L}_{\text{bounds}} = \text{FindBounds}(\mathcal{L})$ (Parallel).

$K \leftarrow \text{length}(\mathcal{L})$
parfor $k \in \{1, \ldots, K\}$ **do**
  $(\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x}), B_{hj}(\mathbf{x})) \leftarrow \mathcal{L}[k]$
  Find $\min B_p(\mathbf{x})$ and $\max B_p(\mathbf{x})$ by parallel reduction
  Find $\min B_{gi}(\mathbf{x})$ and $\max B_{gi}(\mathbf{x})$ similarly
  Find $\min B_{hj}(\mathbf{x})$ and $\max B_{hj}(\mathbf{x})$ similarly
  $\mathcal{L}_{\text{bounds}}[k] \leftarrow (\mathbf{x}, \{\min B_p(\mathbf{x}), \max B_p(\mathbf{x})\}, \{\min B_{gi}(\mathbf{x}), \max B_{gi}(\mathbf{x})\}, \{\min B_{hj}(\mathbf{x}), \max B_{hj}(\mathbf{x})\})$
end parfor
return $\mathcal{L}_{\text{bounds}}$

Algorithm 4: $[p^*_{\text{up}}, p^*_{\text{lo}}, \mathcal{L}_{\text{save}}, \mathcal{L}_{\text{elim}}] = \text{CutOffTest}(\mathcal{L}_{\text{bounds}})$.

$p^*_{\text{up}} \leftarrow +\infty,\ p^*_{\text{lo}} \leftarrow +\infty$
$K \leftarrow \text{length}(\mathcal{L}_{\text{bounds}})$
for $k \in \{1, \ldots, K\}$ **do**
  $(\mathbf{x}, \{\min B_p(\mathbf{x}), \max B_p(\mathbf{x})\}, \{\min B_{gi}(\mathbf{x}), \max B_{gi}(\mathbf{x})\}, \{\min B_{hj}(\mathbf{x}), \max B_{hj}(\mathbf{x})\}) \leftarrow \mathcal{L}_{\text{bounds}}[k]$
  if $-\epsilon_{\text{eq}} \le \min B_{hj}(\mathbf{x}) \le 0 \le \max B_{hj}(\mathbf{x}) \le \epsilon_{\text{eq}}$ **then**
    if $\max B_{gi}(\mathbf{x}) \le 0$ **then**
      $p^*_{\text{up}} \leftarrow \min(p^*_{\text{up}}, \max B_p(\mathbf{x}))$
    end if
    if $\min B_{gi}(\mathbf{x}) \le 0$ **then**
      $p^*_{\text{lo}} \leftarrow \min(p^*_{\text{lo}}, \min B_p(\mathbf{x}))$
    end if
  end if
end for
Initialize lists for indices of patches to save or eliminate $\mathcal{L}_{\text{save}} \leftarrow \{\}$, $\mathcal{L}_{\text{elim}} \leftarrow \{\}$
for $k \in \{1, \ldots, K\}$ **do**
  if $\min B_{hj}(\mathbf{x}) \le 0 \le \max B_{hj}(\mathbf{x})$ **then**
    and $\min B_{gi}(\mathbf{x}) \le 0$
    and $\min B_p(\mathbf{x}) \le p^*_{\text{up}}$
    Append k to $\mathcal{L}_{\text{save}}$
  else
    Append k to $\mathcal{L}_{\text{elim}}$
  end if
end for
return $p^*_{\text{up}}, p^*_{\text{lo}}, \mathcal{L}_{\text{save}}, \mathcal{L}_{\text{elim}}$

local minima, unlike generic nonlinear solvers (e.g., fmincon [46]). It also does not require tuning hyperparameters. As we show in Section V, the PCBA has bounded time and memory complexity under certain assumptions. Finally, due to parallelization, the PCBA is fast enough to enable RTD* for real-time, safe, optimal trajectory planning, which we demonstrate in Section VII.

However, the PCBA also has several limitations in comparison to traditional approaches to solving POPs. First, to prove the bounds on time and memory usage, at any global minimum, we require that active constraints are linearly independent, and that the Hessian of the cost function is positive definite (see Theorems 14 and 16). Furthermore, due to the number of Bernstein patches growing exponentially with the decision variable dimension, we have not yet applied PCBA to problems larger than four-dimensional.

Algorithm 5: $\mathcal{L} = \text{Eliminate}(\mathcal{L}, \mathcal{L}_{\text{save}}, \mathcal{L}_{\text{elim}})$ (Parallel).

$K_{\text{save}} \leftarrow \text{length}(\mathcal{L}_{\text{save}})$
$K_{\text{elim}} \leftarrow \text{length}(\mathcal{L}_{\text{elim}})$
$K_{\text{replace}} \leftarrow K_{\text{elim}} - 1$
if $K_{\text{elim}} = 0$ **or** $\mathcal{L}_{\text{elim}}[1] > K_{\text{elim}}$ **then**
  return$\mathcal{L}$
end if
for $k \in \{1, \ldots, K_{\text{elim}}\}$ **do**
  if $\mathcal{L}_{\text{elim}}[k] \ge K_{\text{save}}$ **then**
    $K_{\text{replace}} \leftarrow k - 1$
    break
  end if
end for
parfor $k \in \{1, \ldots, K_{\text{replace}}\}$ **do**
$\mathcal{L}[\mathcal{L}_{\text{elim}}[k]] \leftarrow \mathcal{L}[\mathcal{L}_{\text{save}}[K_{\text{save}} + 1 - k]]$
end parfor
return $\mathcal{L}$

V. Complexity Analysis

In this section, we prove that Algorithm 1 terminates by bounding the number of iterations of PCBA for both unconstrained and constrained POPs. We also prove the number of Bernstein patches (i.e., the length of the list $\mathcal{L}$ in Algorithm 1) is bounded after sufficiently many iterations, under certain assumptions. For convenience, in the remainder of this section, we use $\mathbf{x} \in \mathcal{L}$ as a shorthand notation for $\mathbf{x} = \ell_1$, where $\ell = (\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x}), B_{hj}(\mathbf{x})) \in \mathcal{L}$. All proofs are provided in the supplementary material available at[1].

A. Unconstrained Case

We first consider unconstrained POPs, whose optimal solutions are not on the boundary of $\mathbf{u}$. Note that we can treat optimal solutions on the boundary of $\mathbf{u}$ as having active linear constraints; see Section V-B for the corresponding complexity analysis. In the unconstrained case, all points in $\mathbf{u}$ are feasible, and we are interested in solving

$$\min_{x \in \mathbf{u} \subset \mathbb{R}} \quad p(x) \tag{19}$$

where p is an l-dimensional multivariate polynomial. Given an optimality tolerance ϵ and step tolerance δ, we bound the number of iterations to solve (19) with PCBA as follows.

Theorem 14: Let p in (19) be a multivariate polynomial of dimension l with Lipschitz constant L_p. Then, the maximum number of iterations needed to solve (19) up to accuracy ϵ and δ is

$$N = \left\lceil \max\left\{ -\log_2 \delta, -\frac{1}{2}\log_2\left(\frac{\epsilon}{4\zeta_p}\right), -\log_2\left(\frac{\epsilon}{2L_p\sqrt{l}}\right) \right\} \right\rceil \tag{20}$$

where ζ_p is the constant in Theorem 4 corresponding to polynomial p, and $\lceil \cdot \rceil$ rounds up to the nearest integer.

[1]https://www.roahmlab.com/s/PCBA_supplement.pdf

According to (20), the rate of convergence with respect to the decision variables is quadratic (first term); the rate of convergence with respect to the objective function is either quadratic (second term) or linear (third term), depending on which term dominates. However, a tighter bound exists if one of the global minimizers satisfies the second-order sufficient condition for optimality [24, Th. 2.4], which we prove in the supplementary material available at[2].

We now discuss the number of patches remaining after sufficiently many iterations, which gives an estimate of memory usage when Algorithm 1 is applied to solve (19).

Theorem 15: Suppose that there are $m < \infty$ global minimizers $x_1^*, \ldots, x_m^*$ of (19), and none of them are on the boundary of the unit box $\mathbf{u}$. Let the Hessian $\nabla^2 p$ be positive definite at these minimizers. Then, after sufficiently many iterations of Algorithm 1, the number of Bernstein patches remaining (i.e., length of the list $\mathcal{L}$ in Algorithm 1) is bounded by a constant.

The constant bound in Theorem 15 scales exponentially with the problem dimension and is a function of the condition number of the cost function's Hessian at the minimizers.

B. Constrained Case

Theorem 16: Suppose that the linear independence constraint qualification [24, Def. 12.4] is satisfied at all global minimizers $x_1^*, \ldots, x_m^*$ of the constrained POP (P), and at least one constraint is active (i.e., the active set $\mathcal{A}(x^*)$ [24, Def. 12.1] is nonempty) at some minimizer $x^* \in \{x_1^*, \ldots, x_m^*\}$. Then, the maximum number of iterations needed to solve (P) up to accuracy ϵ, δ, and equality constraint tolerance ϵ_{eq} is

$$N := \left\lceil \max \left\{ C_7,\ -\log_2 \delta,\ -\log_2 \epsilon_{\text{eq}} + C_8,\ -\log_2 \epsilon + C_9 \right\} \right\rceil \tag{21}$$

where C_7, C_8, and C_9 are constants.

Theorem 16 gives a bound on the number of PCBA iterations needed to solve a POP up to specified tolerances. In particular, (21) shows that the rate of convergence is *linear* in step tolerance (second term), equality constraint tolerance (third term), and objective function (fourth term), once the number of iterations is larger than a constant (first term). We next prove a bound on the number of items in the list $\mathcal{L}$ after sufficiently many iterations.

Theorem 17: Suppose that there are m ($m < \infty$) global minimizers $x_1^*, \ldots, x_m^*$ of the constrained problem (P), and none of them are on the boundary of the unit box $\mathbf{u}$. Let the critical cone (see [24, eq. (12.53)]) be nonempty for (P_n) as in the proof of Theorem 16 see the supplementary material available at[3]. Then, after sufficiently many iterations of Algorithm 1, the number of Bernstein patches remaining (i.e., length of the list $\mathcal{L}$) is bounded by a constant.

The constant proved in Theorem 17 scales exponentially with respect to the dimension of the problem.

[2]https://www.roahmlab.com/s/PCBA_supplement.pdf
[3]https://www.roahmlab.com/s/PCBA_supplement.pdf

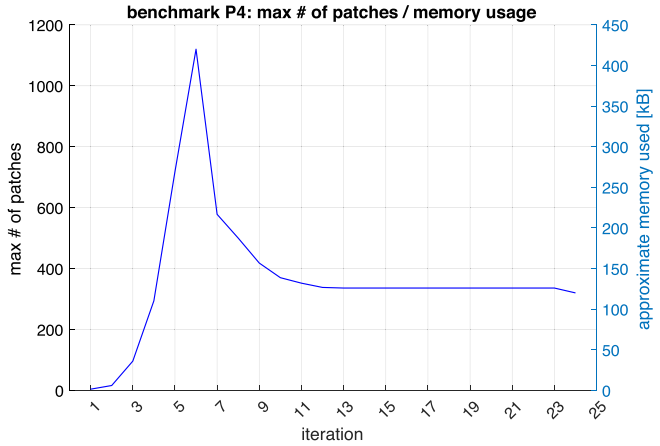

Fig. 3. Maximum number of patches (left axis) and corresponding GPU memory used (right axis) at each iteration of PCBA, for P4 of the benchmark problems (see Section VI). This problem took 24 iterations to solve. Notice that the number of patches peaks in iteration 5 and then stays under 400 patches at every iteration from iteration 9 onwards; this visualizes Theorem 17.

C. Memory Usage Implementation

We now state the amount of GPU memory required to store a single item $(\mathbf{x}, B_p(\mathbf{x}), B_{gi}(\mathbf{x}), B_{hj}(\mathbf{x})) \in \mathcal{L}$, given the degree and dimension of the cost and constraint polynomials. Note that, for our implementation, all numbers in an item are represented using 4B of space, as either floats or unsigned integers.

For a multi-index $J = (j_1, \ldots, j_l) \in \mathbb{N}^l$, let $\Pi J = j_1 \times \cdots \times j_l$, and let $J + n = (j_1 + n, \ldots, j_l + n)$ for $n \in \mathbb{N}$. Let P be the multidegree of the cost p. Let G be a multidegree large enough for all inequality constraints g_i, and H a multidegree large enough for all equality constraints h_j. By "large enough," we mean that, if g_i is the multidegree of any g_i, then $g_i \leq G$ (and similarly for H). Then, as per [48, Sec. 4.1], an item can be stored in memory as an array with the following number of entries:

$$2l + (\Pi(P+1)) + (\alpha \cdot \Pi(G+1)) + (\beta \cdot \Pi(H+1)) \tag{22}$$

where the first $2l$ entries store the upper and lower bounds (in each dimension) of the subbox $\mathbf{x}$.

D. Summary

We have shown that PCBA will find a solution to (P), if one exists, in bounded time. We have also shown that the memory usage of PCBA is bounded after a finite number of iterations, which implies that the memory usage is bounded, and we have provided a way to compute how much memory is required to store the list $\mathcal{L}$.

In terms of RTD, this section provides a critical result: the runtime and memory usage of PCBA are bounded by a constant that is known *before* the robot runs. This is possible because the degree and dimension of the cost function are fixed *a priori* in (14), and, in practice, the number of constraints is bounded (see Fig. 7). This means that we can choose the planning time t_{plan} (as in Section III-A3) to be as small as possible offline. A smaller planning time limit means a less conservative FRS and an increased probability that the robot can find a new plan in each

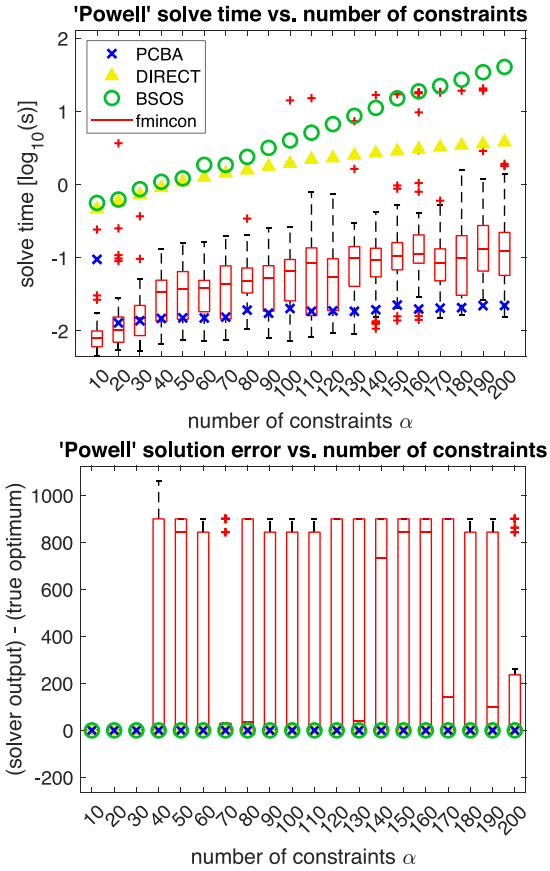

Fig. 4. Results for an increasing number of constraints on the Powell objective function (see the Appendix, available at https://www.roahmlab.com/s/PCBA_supplement.pdf) for PCBA, BSOS, fmincon, and DIRECT. The top plot shows the time required to solve the problem as the number of constraints increases. The bottom plot shows the error between each solver's solution and the true global optimum. For both time and error, fmincon is shown as a box plot over 50 trials with random initial guesses; the central red line indicates the median, the top and bottom of the red box indicate the 25th and 75th percentiles, the black whiskers are the most extreme values not considered outliers, and the outliers are red plus signs. The PCBA solves the fastest in general; fmincon typically solves slightly slower than PCBA for more than 40 constraints; BSOS and DIRECT are the slowest solvers. PCBA, BSOS, and DIRECT always find the global optimum, as does fmincon with few constraints, because the Powell objective function is convex. Above 30 constraints, fmincon frequently has large error due to convergence to local minima.

receding-horizon iteration. In other words, Theorem 16 enables us to increase a robot's liveness when planning with RTD, as we show in Section VII.

Next, before applying PCBA to RTD, we benchmark PCBA and compare it to two other solvers.

VI. PCBA Evaluation

In this section, we compare the PCBA against a convex relaxation solver (Lasserre's BSOS [35]), a derivative-based solver (MATLAB's fmincon [46]), and a branch-and-bound solver (DIRECT [39]). First, we test all four solvers on eight "Benchmark Evaluation" problems. Second, we compare the solvers on several "Increasing Number of Constraints" problems, to assess how each solver scales on a variety of difficult objective functions [54].

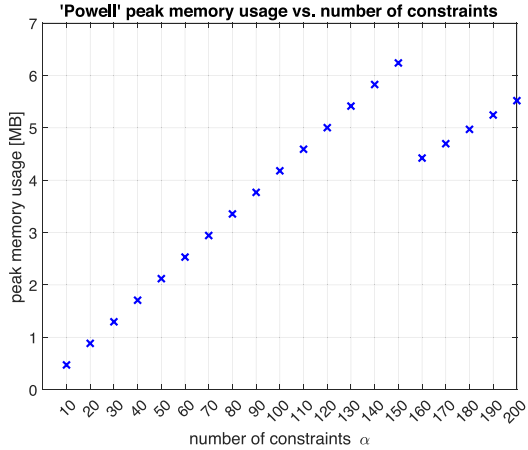

Fig. 5. Approximate peak GPU memory used by the PCBA for the Powell problem, as a function of the number of constraints. Since the amount of memory required per item in the list $\mathcal{L}$ grows linearly with the number of constraints, the overall memory usage also grows linearly. However, at 160 constraints, we see a drop in the memory usage; this is because the additional constraints render more parts of the problem domain infeasible, resulting in more items being eliminated per PCBA iteration. Note that the maximum memory usage is well under the several GB available on a typical GPU.

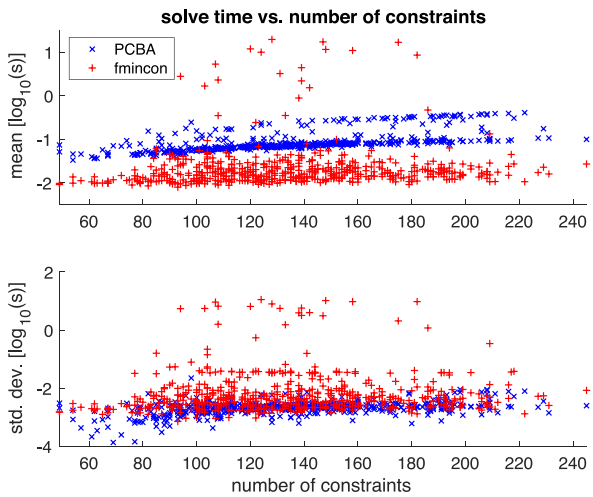

Fig. 6. Solve times of PCBA and fmincon on 528 POPs generated by the Segway robot navigating random scenarios in Demo 1. Each POP was solved 25 times by each solver. While fmincon can often find a solution an order of magnitude faster than PCBA, it also has a much higher standard deviation, meaning that it is less consistent at obeying the real-time limit required by mobile robot trajectory planning.

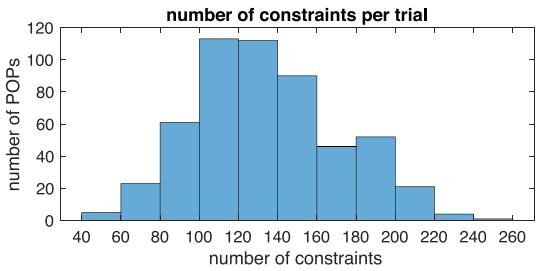

Fig. 7. Number of POPs from Demo 1, out of 528, that fall into the given bins of number of constraints; we see that most of the POPs had 100–140 constraints. This number of constraints can make it challenging to solve a POP while constrained by a real-time planning limit.

All of the solvers/problems in this section are run on a computer with a 3.7-GHz processor, 16 GB of RAM, and an Nvidia GTX 1080 Ti GPU. The PCBA is implemented with MATLAB R2017b executables and CUDA 10.0. Our code is available at[4].

A. Parameter Selection

To set up a fair comparison, we scale each problem to the $\mathbf{u} = [0, 1]^l$ box, where l is the problem dimension. For the PCBA, we use the stopping criteria in Section IV. To choose ϵ, we first compute the patch $B(\mathbf{u})$ and then set

$$\epsilon = (10^{-7}) \cdot (\max B(\mathbf{u}) - \min B(\mathbf{u})) . \tag{23}$$

We set the maximum number of PCBA iterations to $N = 28$. We do not set δ, which determines the minimum number of iterations; δ is only needed to prove the complexity bounds in Section V.

BSOS [35, Sec. 4] requires the user to specify the size of the semidefinite matrix associated with the convex relaxation of the POP. This is done by selecting a pair of parameters, d and k (note these are different from our use of d and k). Though one has to increase d and k gradually to ensure convergence, larger values of d and k correspond to larger SDPs, which can be difficult to solve. We chose d and k separately for each benchmark evaluation problem see the code at[5]. We used $d = k = 2$ for the Increasing Number of Constraints.

For fmincon [46], we set the `OptimalityTolerance` option to ϵ in (23). We set `MaxFunctionEvaluations` $= 10^5$ and `MaxIterations` $= 10^4$. We also provide `fmincon` with the analytic gradients of the cost and constraints.

For DIRECT, we set the optimality tolerance to ϵ as in (23) and the maximum number of function evaluations to 5×10^5. For the benchmark evaluation, we set the maximum number of iterations large enough that there was no change in the cost estimate after this maximum number for at least five iterations. As per [39], we tuned the constraint penalty for each benchmark problem by hand until DIRECT found a solution close to the global optimum. For the Increasing Number of Constraints, we used a constraint penalty of 1.

We note that BSOS, fmincon, and DIRECT are not parallelized. While one could attempt to parallelize these solvers, to the best of our knowledge, no such fully developed implementations are available. Furthermore, since each algorithm operates on different principles, simply applying parallelization does not guarantee a fair comparison. However, these comparisons help place PCBA in the context of readily available state-of-the-art solvers.

B. Benchmark Evaluation

1) Setup: We tested PCBA, BSOS, fmincon, and DIRECT on eight benchmark POPs [41], listed as P1–P8. The problems are reported in the supplementary material available at[6]. We ran each solver 50 times on each problem; we report the median solution error and time required to find a solution. Since fmincon may or may not converge to the global optimum depending on its initial guess, we used random initial guesses for each of the 50 attempts.

2) Results: The results are summarized in Table I. Additional results are available online at https://github.com/ramvasudevan/GlobOptBernstein.

In terms of solution quality, the PCBA always found the solution to within the desired optimality tolerance ϵ, except on P1, where the PCBA stopped at the maximum allowed number of iterations (28); the PCBA always used 22–28 iterations. BSOS always found a lower bound to the solution, as expected. While fmincon converged to the global optimum at least once on every problem, it often converged to local minima, hence the large error values on some problems. DIRECT rarely found a solution within the optimality tolerance, which is likely due to using a constraint penalty as opposed to enforcing hard constraints. Unfortunately, DIRECT struggles to converge to global optima in the constrained case; tuning the penalty leads to a tradeoff between feasibility and optimality [39, p. 31]. This is especially true for P2, which has large coefficients (on the order of 10^6) in the cost and constraints; these cause DIRECT to produce a large range of possible rate-of-change constants, which are used to estimate bounds on the cost function [38, Def. 3.1]. Consequently, DIRECT produces a poor cost estimate.

In terms of solve time, fmincon solves the fastest (in 10–20 ms). The PCBA is about twice as slow as fmincon. BSOS and DIRECT are one to two orders of magnitude slower than the PCBA.

For the PCBA, the memory usage [computed with (22)] increases roughly by one order of magnitude for each additional dimension of the decision variable increases (Table I reports the peak GPU memory used by the PCBA on each benchmark problem). Notice that the PCBA never uses more than several MB of GPU memory, which is much less than the 11 GB available on the Nvidia GTX 1080 Ti GPU. Fig. 3 shows the number of patches and the amount of GPU memory used on P4. We see that the memory usage peaks and then stays below a constant, as predicted by Theorem 17.

C. Increasing Constraint Problems

Next, we tested each solver on problems with an increasing number of constraints, to assess each solver for use with RTD; we find in practice that RTD's POP (14) contains 30–300 constraints at each planning iteration (see Fig. 7).

1) Setup: We first choose an objective function with either many local minima or a nearly flat gradient near the global optimum (the global optimizer is known for each function). In particular, we tested on the ElAttar–Vidyasagar–Dutta, Powell, Wood, Dixon–Price (with $l = 2, 3, 4$), Beale, Bukin02, and Deckkers–Aarts problems (see the supplementary material and [54]).

For each objective function, we generate 200 random constraints in total, while ensuring that at least one global optimizer stays feasible (if there are multiple global optimizers, we choose one at random that will be feasible for all constraints). To generate a single constraint $g : \mathbf{u} \to \mathbb{R}$, we first create a

[4]https://github.com/ramvasudevan/GlobOptBernstein
[5] https://github.com/ramvasudevan/GlobOptBernstein
[6]https://www.roahmlab.com/s/PCBA_supplement.pdf

TABLE I
RESULTS FOR PCBA, BSOS, fmincon, AND DIRECT ON EIGHT BENCHMARK PROBLEMS

		PCBA					BSOS [35]		fmincon [46]		DIRECT [39]	
	l	ϵ	error	time [s]	iterations	memory	error	time [s]	error	time [s]	error	time [s]
P1	2	7.0000e-07	7.9002e-07	0.0141	28	23 kB	-0.0068	1.2264	1.0880	0.0083	1.8925e-04	0.1003
P2	2	0.1369	0.0731	0.0131	26	60 kB	-3.4994e-04	0.7671	-0.4447	0.0180	348.8644	0.1302
P3	2	2.0000e-06	1.9879e-06	0.0128	26	57 kB	-3.2747	0.5065	-3.0000	0.0220	0.0148	1.3850
P4	3	1.7000e-06	6.5565e-07	0.0204	24	416 kB	-0.9455	6.6546	3.2000e-05	0.0130	0.1113	0.2913
P5	3	1.0000e-06	0	0.0294	28	3 MB	-4.4858e-07	0.8462	7.9985e-06	0.0062	0.2174	0.7942
P6	4	4.2677e-04	1.8685e-04	0.0315	23	2 MB	-36.6179	6.2434	0.0040	0.0065	0.4781	0.1981
P7	4	5.0000e-07	2.5280e-07	0.0374	26	282 kB	-1.0899	0.1839	2.4002e-07	0.0139	-1.0864	7.8994
P8	4	1.3445e-04	6.7803e-05	0.0420	22	6 MB	-3.2521	1.8295	9.9989e-04	0.0169	0.3238	0.1751

The column l indicates the problem dimension. The error columns report each solver's result minus the true global minimum. For all four solvers, the reported error and time to find a solution are the median over 50 trials (with random initial guesses for fmincon). For PCBA, we also report the optimality tolerance ϵ (as in (23)), number of iterations to convergence, and peak GPU memory used. Note that, on P1 and P5, PCBA stopped at the maximum number of iterations (28). See the appendix, available at https://www.roahmlab.com/s/pcbasupplement.pdf, for more details.

polynomial g_{temp} as a sum of the monomials of the decision variable with maximum degree 2, with random coefficients in the range $[-5, 5]$. To ensure that x^* is feasible, we evaluate g_{temp} on x^* and then subtract the resulting value from g_{temp} to produce g (i.e., $g \leftarrow g_{\text{temp}} - g_{\text{temp}}(x^*)$).

We ran PCBA, BSOS, fmincon, and DIRECT on each objective function for 20 trials, with ten random constraints in the first trial, plus ten constraints in each subsequent trial. As before, we ran fmincon 50 times for each trial with random initial guesses.

2) Results: To illustrate the results, data for the Powell objective function are shown in Fig. 4. The data (and plots) for the other objective functions are available online at[8].

In terms of solution quality, all four algorithms converge to the global optimum often when the number of constraints is low, but fmincon converges to suboptimal solutions more frequently as the number of constraints increases. PCBA, BSOS, and DIRECT are always able to find the optimal solution. The PCBA is always able to find the global optimum regardless of the number of constraints, unlike BSOS (which runs out of memory) or fmincon (which converges to local minima). Interestingly, DIRECT is less sensitive to the constraint penalty for these problems than for the benchmark problems in Section VI-B.

All four solvers require an increasing amount of solve time as the number of constraints increases. The PCBA is comparable in speed to fmincon on 2-D problems, but is typically slower on higher dimensional problems. Regardless of the number of constraints, BSOS takes three to four orders of magnitude more time to solve than PCBA or fmincon. DIRECT takes two orders of magnitude more time than PCBA, but could potentially be parallelized.

More details on the PCBA are presented in Table II. PCBA's time to find a solution increases roughly by an order of magnitude when the decision variable dimension increases by 1; however, the PCBA solves all of the increasing constraint POPs within 0.5 s. The memory usage increases by one to three orders of magnitude with each additional dimension; however, the PCBA never uses more than 650 MB of GPU memory, well below the 11 GB available. Therefore, it may be possible to introduce heuristics, such as additional subdivisions per iteration, to improve PCBA's performance. Fig. 5 shows PCBA's GPU memory usage versus the number of constraints for the Powell objective function.

TABLE II
RESULTS FOR THE INCREASING CONSTRAINTS PCBA EVALUATION

	l	max time [s]	max items	max memory
E-V-D	2	0.0360	66	171 kB
Powell	2	0.0447	1200	6.24 MB
Wood	2	0.0532	54	220 kB
D-P 2-D	2	0.0259	90	433 kB
D-P 3-D	3	0.0675	356	3.47 MB
D-P 4-D	4	0.402	4994	193 MB
Beale	2	0.0302	106	259 kB
Bukin02	4	0.393	10550	110 MB
D-A	4	0.389	51886	647 MB

Abbreviated problem names (as in the Appendix, available at https://www.roahmlab.com/s/PCBA_supplement.pdf) are on the left, along with each problem's decision variable dimension l. Over all 20 trials (with 10–200 constraints), we report the maximum time spent find a solution, the maximum number of items in the list $\mathcal{L}$, and the maximum amount of GPU memory used. Note that the problems all solved under 0.5 s regardless of the number of constraints, and no problem requested more than 650 MB of memory.

D. Summary

As expected from the complexity bounds in Section V-B, our results indicate that PCBA can quickly solve 2-D POPs with hundreds of constraints. We leverage this next by applying PCBA to solve RTD's POP (14) for real-time receding-horizon planning.

VII. HARDWARE DEMONSTRATIONS

Recall from Section III that RTD enables real-time provably collision-free trajectory planning via solving a POP every planning iteration. RTD is provably collision-free regardless of the POP solver used, meaning that we are able to test PCBA safely on hardware; when the PCBA is applied to RTD, we call the resulting trajectory planning algorithm *RTD**. See Fig. 1 and the video available at[9].

In this section, we apply RTD* to a Segway robot navigating a hallway with static obstacles. Recall that RTD always produces *dynamically feasible* trajectory plans [8]. As proven in Corollary 13 and demonstrated in Section VI, the PCBA always finds *optimal* solutions. This section shows that PCBA/RTD* improves

[8]https://github.com/ramvasudevan/GlobOptBernstein

[9]https://youtu.be/YcH4WAzqPFY

the *liveness* of a robot by successfully navigating a variety of scenarios and outperforming fmincon/RTD.

A. Overview

1) Demonstrations: We ran two demonstrations in a 20×3 m^2 hallway. First, we filled the hallway with random static obstacles and ran RTD*. Second, we constructed two difficult scenarios and ran PCBA/RTD* and fmincon/RTD on each.

In both demonstrations, the robot must find a new plan [i.e., solve (14)] every $t_{\text{plan}} = 0.5$ s, or else it begins executing the braking maneuver associated with its previously computed plan [8, Remark 70]. In other words, we require PCBA to return a feasible solution or that the problem is infeasible. The PCBA is given a time limit of 0.4 s to find a solution, because the robot requires 0.1 s for other onboard processes.

2) Hardware: We use a Segway differential-drive robot with a planar Hokuyo UTM-30LX LIDAR for mapping and obstacle detection (see Fig. 1). Mapping, localization, and trajectory optimization run onboard on a 4.0-GHz laptop with an Nvidia GeForce GTX 1080 GPU.

B. Demo 1

The first demo shows the ability of the RTD* to plan safe trajectories in randomly generated scenarios in real time, demonstrating *dynamic feasibility*, *optimality*, and *liveness*.

1) Setup: The robot was required to move autonomously back and forth ten times between two global goals spaced 12 m apart, while 30-cm^3 box-shaped obstacles were randomly placed in its path. At each planning iteration, we generated $N_{\text{wp}} = 2$ waypoints, w_{L} and $w_{\text{R}} \in \mathbb{R}^2$, both 1.5 m ahead of the robot in the direction of the global goal; w_{L} is on the left side of the hallway relative to the robot, and w_{R} is on the right.

After running the robot with RTD*, we ran fmincon on the 528 saved POPs generated during these ten trials (we do not run BSOS or DIRECT due to their slow solve times). Each POP has 49–245 constraints (see Fig. 7). For each POP, we initialized fmincon with 25 random initial guesses and did not require fmincon to solve within 0.4 s (i.e., we did not enforce the real-time planning constraint). To understand the timing of PCBA and for fair comparison with fmincon, we reran the PCBA 25 times on each trial and did not require it to solve in real time.

2) Results: The robot running RTD* successfully completed every trial (meaning that it reached the global desired goal location without collisions, and without human assistance).

When rerunning on the saved POPs, fmincon performs nearly as well as PCBA in terms of finding solutions. Out of all 25×528 attempts in which fmincon converged to a feasible solution, fmincon converged to a greater cost than PCBA 93.9% of the time (recall that PCBA provably upper bounds the optimal solution); however, the fmincon solution was only 0.77% greater in cost than the PCBA solution on average, indicating that fmincon was often able to find a global optimum when given enough attempts. In terms of feasibility, the PCBA and fmincon also show similar results. The PCBA reports that 7.01% of the POPs are infeasible, whereas fmincon converged to an infeasible result on 8.08% of the 25×528 total attempts. Note that, on 14.2% of the 528 POPs, fmincon converged to an infeasible result least once out of 25 attempts.

Where fmincon suffers with respect to PCBA is in its consistency of finding an answer within the time limit (see Fig. 6). While fmincon is often able to solve in 10^{-2} s (an order of magnitude faster than PCBA), it has a standard deviation of up to 10 s. On the other hand, the PCBA always finds a solution or returns infeasible within 0.4 s and has a standard deviation of 2.4 ms on average over all 25×528 POPs. To summarize, as we expect from the theory in Section V, PCBA's solve time in practice appears constant *on real trajectory optimization problems*.

C. Demo 2

The second demo shows that RTD* can navigate difficult scenarios because the PCBA is able to rapidly solve POPs with hundreds of constraints. Recall that, in any planning iteration, RTD and RTD* commands the robot to begin braking if they cannot find a new trajectory plan (i.e., solve (P)). By *difficult* scenarios, we mean that the obstacles are arranged to cause the robot to have to brake often. Therefore, by RTD*'s successful navigation of these scenarios, we demonstrate *liveness*.

1) Setup: The robot was required to navigate two difficult scenarios autonomously. In the first scenario, static obstacles were arranged to force the robot to turn frequently, and to decide to go left or right around each obstacle. In the second scenario, the robot was required to navigate a tight obstacle blockade. For each scenario, we ran PCBA/RTD* and fmincon/RTD once each. At each planning iteration, we generate $N_{\text{wp}} = 1$ waypoint positioned 1.5 m away from the robot along a straight line to the global goal; this produces a convex cost function for (14), but the constraints make the problem nonconvex.

2) Results: In the first scenario, both RTD* and RTD successfully reach the goal. Recall that the robot begins emergency braking when it does not find a feasible trajectory in a planning iteration. RTD* brakes six times, whereas RTD brakes 13 times; furthermore, RTD* only takes 27 s to navigate to the goal, whereas RTD takes 43 s. In other words, PCBA/RTD* is half as conservative as fmincon/RTD.

The results of the second scenario confirm that RTD* is less conservative than RTD. In this scenario, RTD* is able to navigate the entire scenario autonomously without human assistance, whereas RTD causes the robot to become stuck (see Fig. 8 and the video available at[10], and a human operator must drive the robot for a short time to enable to it to continue moving autonomously.

D. Discussion

We have demonstrated that RTD* is capable of *dynamic feasibility*, *optimality*, and *liveness* for online trajectory optimization on robot hardware. RTD* is able to find an optimal solution, if it exists, at every receding-horizon planning iteration, leading to it consistently navigating random scenarios without collisions. Furthermore, RTD* outperforms RTD at the same tasks. This is

[10]https://youtu.be/YcH4WAzqPFY

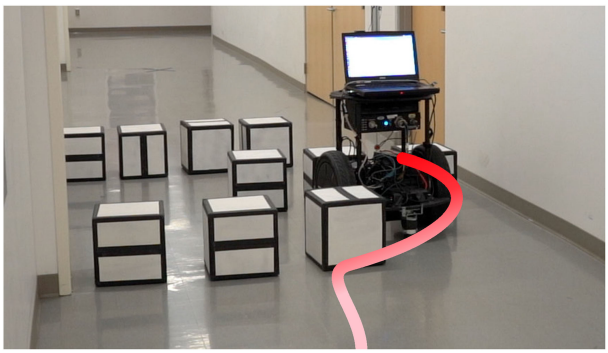

Fig. 8. Robot becomes stuck when planning with RTD/fmincon in the second scene of the second hardware demo, because fmincon cannot find an optimal solution quickly enough given the high number of constraints produced by the surrounding obstacles. The robot requires human assistance to proceed, whereas it is able to navigate the entire scene autonomously when planning with RTD*/PCBA (see Fig. 1). See the video available at https://youtu.be/YcH4WAzqPFY.

due to PCBA's ability to find solutions more quickly than fmincon on problems with hundreds of constraints. To the best of our knowledge, this is the first time any BA has been demonstrated as practical for a real-time mobile robotics application.

VIII. Conclusion

Mobile robots typically use receding-horizon planning to move through the world. Plans should be dynamically feasible and optimal, but also need to be generated quickly; otherwise, a robot may stop frequently and never complete its task. The existing RTD method creates plans by solving a POP, but uses a derivative-based nonlinear solver that cannot guarantee solving speed or optimality. We proposed and implemented a PCBA to rapidly and optimally solve this POP, resulting in the RTD* planning algorithm. RTD* outperforms RTD on a variety of hardware demonstrations. To the best of our knowledge, this is the first time a BA has been used for real-time mobile robotics. Furthermore, the PCBA outperforms the BSOS, fmincon, and DIRECT solvers on a variety of benchmark POPs. For future work, we plan to explore nonpolynomial optimization problems and to improve the proposed time and space complexity bounds of PCBA; our goal is to apply RTD* and PCBA to more types of robots.

References

[1] Y. Kuwata, "Trajectory planning for unmanned vehicles using robust receding horizon control," Ph.D. dissertation, Dept. Aeronaut. Astronaut., Massachusetts Inst. Technol., Cambridge, MA, USA, Feb. 2007.

[2] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *Int. J. Robot. Res.*, vol. 30, no. 7, pp. 846–894, 2011.

[3] M. Pivtoraiko, R. A. Knepper, and A. Kelly, "Differentially constrained mobile robot motion planning in state lattices," *J. Field Robot.*, vol. 26, no. 3, pp. 308–333, 2009.

[4] A. Majumdar and R. Tedrake, "Funn el libraries for real-time robust feedback motion planning," *Int. J. Robot. Res.*, vol. 36, no. 8, pp. 947–982, 2017.

[5] K. R. Muske and J. B. Rawlings, "Model predictive control with linear models," *AIChE J.*, vol. 39, no. 2, pp. 262–287, 1993.

[6] B. Yi, S. Gottschling, J. Ferdinand, N. Simm, F. Bonarens, and C. Stiller, "Real time integrated vehicle dynamics control and trajectory planning with MPC for critical maneuvers," in *Proc. IEEE Intell. Veh. Symp.*, 2016, pp. 584–589.

[7] D. Fridovich-Keil, J. F. Fisac, and C. J. Tomlin, "Safely probabilistically complete real-time planning and exploration in unknown environments," in *Proc. Int. Conf. Robot. Autom*, 2019, pp. 7470–7476.

[8] S. Kousik, S. Vaskov, F. Bu, M. Johnson-Roberson, and R. Vasudevan, "Bridging the gap between safety and real-time performance in receding-horizon trajectory design for mobile robots," *Int. J. Robot. Res.*. [Online]. Available: https://journals.sagepub.com/doi/abs/10.1177/0278364920943266?journalCode=ijra

[9] M. A. Patterson and A. V. Rao, "GPOPS-II: A MATLAB Software for solving multiple-phase optimal control problems using hp-adaptive Gaussian quadrature collocation methods and sparse nonlinear programming," *ACM Trans. Math. Softw.*, vol. 41, no. 1, Oct. 2014, Art. no. 1.

[10] S. Kousik, P. Holmes, and R. Vasudevan, "Safe, aggressive quadrotor flight via reachability-based trajectory design," in *Proc. Dyn. Syst. Control Conf.*, 2019, Art. no. DSCC2019-9214.

[11] S. Vaskov *et al.*, "Towards provably not-at-fault control of autonomous robots in arbitrary dynamic environments," in *Proc. Robot.: Sci. Syst. Conf.*, Freiburg-im-Breisgau, Germany, 2019. [Online]. Available: http://www.roboticsproceedings.org/rss15/p51.html

[12] S. Vaskov, H. Larson, S. Kousik, M. J.-Roberson, and R. Vasudevan, "Not-at-fault driving in traffic: A reachability-based approach," in *Proc. IEEE Intell. Transp. Syst. Conf.*, 2019, pp. 2785–2790.

[13] S. Vaskov, U. Sharma, S. Kousik, M. Johnson-Roberson, and R. Vasudevan, "Guaranteed safe reachability-based trajectory design for a high-fidelity model of an autonomous passenger vehicle," in *Proc. Amer. Control Conf.*, 2019, pp. 705–710.

[14] P. Holmes *et al.*, "Reachable sets for safe, real-time manipulator trajectory design," in *Proc. Robot.: Sci. Syst. Conf.*, 2020. [Online]. Available: http://www.roboticsproceedings.org/rss16/p100.html

[15] L. Qi and K. L. Teo, "Multivariate polynomial minimization and its application in signal processing," *J. Global Optim.*, vol. 26, no. 4, pp. 419–433, 2003.

[16] I. Thng, A. Cantoni, and Y. H. Leung, "Derivative constrained optimum broad-band antenna arrays," *IEEE Trans. Signal Process.*, vol. 41, no. 7, pp. 2376–2388, Jul. 1993.

[17] B. Mariere, Z.-Q. Luo, and T. N. Davidson, "Blind constant modulus equalization via convex optimization," *IEEE Trans. Signal Process.*, vol. 51, no. 3, pp. 805–818, Mar. 2003.

[18] G. Dahl, J. M. Leinaas, J. Myrheim, and E. Ovrum, "A tensor product matrix approximation problem in quantum physics," *Linear Algebra Appl.*, vol. 420, nos. 2/3, pp. 711–725, 2007.

[19] L. Gurvits, "Classical deterministic complexity of edmonds' problem and quantum entanglement," in *Proc. 35th Ann. ACM Symp. Theory Comput.*, 2003, pp. 10–19.

[20] R. Kamyar and M. Peet, "Polynomial optimization with applications to stability analysis and control-alternatives to sum of squares," 2014. [Online]. Available: http://www.aimsciences.org/article/doi/10.3934/dcdsb.2015.20.2383

[21] M. A. B. Sassi and S. Sankaranarayanan, "Stability and stabilization of polynomial dynamical systems using Bernstein polynomials," in *Proc. 18th Int. Conf. Hybrid Syst.: Comput. Control*, 2015, pp. 291–292.

[22] D. M. Rosen, L. Carlone, A. S. Bandeira, and J. J. Leonard, "SE-S ync: A certifiably correct algorithm for synchronization over the special Euclidean group," *Int. J. Robot. Res.*, vol. 38, nos. 2/3, pp. 95–125, 2019.

[23] J. G. Mangelson, J. Liu, R. M. Eustice, and R. Vasudevan, "Guaranteed globally optimal planar pose graph and landmark SLAM via sparse-bounded sums-of-squares programming," in *Proc. Int. Conf. Robot. Autom.*, 2019, pp. 9306–9312.

[24] J. Nocedal and S. Wright, *Numerical Optimization*. New York, NY, USA: Springer, 2006.

[25] L. Qi, Z. Wan, and Y.-F. Yang, "Global minimization of normal quartic polynomials based on global descent directions," *SIAM J. Optim.*, vol. 15, no. 1, pp. 275–302, 2004.

[26] E. Balas, S. Ceria, and G. Cornuéjols, "A lift-and-project cutting plane algorithm for mixed 0–1 programs," *Math. Program.*, vol. 58, nos. 1–3, pp. 295–324, 1993.

[27] H. D. Sherali and W. P. Adams, "A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems," *SIAM J. Discrete Math.*, vol. 3, no. 3, pp. 411–430, 1990.

[28] J. B. Lasserre, "Global optimization with polynomials and the problem of moments," *SIAM J. Optim.*, vol. 11, no. 3, pp. 796–817, 2001.

[29] E. De Klerk, M. Laurent, and P. A. Parrilo, "A ptas for the minimization of polynomials of fixed degree over the simplex," *Theor. Comput. Sci.*, vol. 361, nos. 2/3, pp. 210–225, 2006.

[30] C. Ling, J. Nie, L. Qi, and Y. Ye, "Biquadratic optimization over unit spheres and semidefinite programming relaxations," *SIAM J. Optim.*, vol. 20, no. 3, pp. 1286–1310, 2009.
[31] Z.-Q. Luo and S. Zhang, "A semidefinite relaxation scheme for multivariate quartic polynomial optimization with quadratic constraints," *SIAM J. Optim.*, vol. 20, no. 4, pp. 1716–1736, 2010.
[32] A. M.-C. So, "Deterministic approximation algorithms for sphere constrained homogeneous polynomial optimization problems," *Math. Program.*, vol. 129, no. 2, pp. 357–382, 2011.
[33] S. He, Z. Li, and S. Zhang, "Approximation algorithms for homogeneous polynomial optimization with quadratic constraints," *Math. Program.*, vol. 125, no. 2, pp. 353–383, 2010.
[34] J. Nie, "An exact jacobian SDP relaxation for polynomial optimization," *Math. Program.*, vol. 137, nos. 1/2, pp. 225–255, 2013.
[35] J. B. Lasserre, K.-C. Toh, and S. Yang, "A bounded degree SOS hierarchy for polynomial optimization," *EURO J. Comput. Optim.*, vol. 5, nos. 1/2, pp. 87–117, 2017.
[36] E. Hansen and G. W. Walster, *Global Optimization Using Interval Analysis: Revised and Expanded*, vol. 264. Boca Raton, FL, USA: CRC Press, 2003.
[37] R. Vaidyanathan and M. El-Halwagi, "Global optimization of nonconvex nonlinear programs via interval analysis," *ComputChem. Eng.*, vol. 18, no. 10, pp. 889–897, 1994.
[38] D. R. Jones, C. D. Perttunen, and B. E. Stuckman, "Lipschit zian optimization without the Lipschitz constant," *J. Optim. Theory Appl.*, vol. 79, no. 1, pp. 157–181, 1993.
[39] D. Finkel, "Global optimization with the DIRECT algorithm," Ph.D. dissertation, Operations Research, North Carolina State University, Raleigh, NC, USA, 2005.
[40] J. Garloff, "The Bernstein algorithm," *Interval Comput.*, vol. 2, no. 6, pp. 154–168, 1993.
[41] P. S. Nataraj and M. Arounassalame, "Constrained global optimization of multivariate polynomials using Bernstein branch and prune algorithm," *J. Glob. Optim.*, vol. 49, no. 2, pp. 185–212, 2011.
[42] M. A. B. Sassi and S. Sankaranarayanan, "Bernstein polynomial relaxations for polynomial optimization problems," 2015, *arXiv:1509.01156v1*.
[43] J. Garloff, "Convergent bounds for the range of multivariate polynomials," in *Proc. Int. Symp. Interval Math.*, 1985, pp. 37–56.
[44] P. S. Nataraj and M. Arounassalame, "A new subdivision algorithm for the Bernstein polynomial approach to global optimization," *Int. J. Autom. Comput.*, vol. 4, no. 4, pp. 342–352, 2007.
[45] P. Dhabe and P. Nataraj, "A parallel Bernstein algorithm for global optimization based on the implicit Bernstein form," *Int. J. Syst. Assurance Eng. Manage.*, vol. 8, no. 2, pp. 1654–1671, 2017.
[46] *MATLAB Optimization Toolbox*, MathWorks, Inc., Natick, MA, USA, 2019.
[47] T. Hamadneh, "Bounding polynomials and rational functions in the tensorial and simplicial Bernstein forms," Ph.D. dissertation, Dept. Math. Stat., University of Konstanz, Konstanz, Germany, 2018.
[48] J. Titi and J. Garloff, "Fast determination of the tensorial and simplicial Bernstein forms of multivariate polynomials and rational functions," *Reliable Comput.*, vol. 25, p. 25, 2017.
[49] G. Cargo and O. Shisha, "The Bernstein form of a polynomial," *J. Res. Nat. Bur. Standards B*, vol. 70, pp. 79–81, 1966.
[50] S. Malan, M. Milanese, M. Taragna, and J. Garloff, "B^3 algorithm for robust performances analysis in presence of mixed parametric and dynamic perturbations," in *Proc. Proc. 31st IEEE Conf. Decis. Control.*, 1992, pp. 128–133.
[51] S. Kousik, "Reachability-based trajectory design," Ph.D. dissertation, Dept. Mech. Eng., University of Michigan, Ann Arbor, MI, USA, 2020.
[52] D. Ratz and T. Csendes, "On the selection of subdivision directions in interval branch-and-bound methods for global optimization," *J. Glob. Optim.*, vol. 7, no. 2, pp. 183–207, 1995.
[53] M. Zettler and J. Garloff, "Robustness analysis of polynomials with polynomial parameter dependency using Bernstein expansion," *IEEE Trans. Autom. Control*, vol. 43, no. 3, pp. 425–431, Mar. 1998.
[54] A. Gavana, "Global Optimization Benchmarks and AMPGO," 2019.
[55] H. Chang, W. He, and N. Prabhu, "The analytic domain in the implicit function theorem," *JIPAM. J. Inequal. Pure Appl. Math*, vol. 4, pp. 1–5, 2003.
[56] A. V. Fiacco, "Nonlinear programming sensitivity analysis results using strong second order assumptions," George Washington Univ., Washington, DC, USA, Tech. Rep. ADA058633, 1978.

Shreyas Kousik (Member, IEEE) received the B.S. degree from the Georgia Institute of Technology, Atlanta, GA, USA, in 2014, and the M.S. and Ph.D. degrees from the University of Michigan, Ann Arbor, MI, USA, both in 2020, all in mechanical engineering.

He is currently a Postdoctoral Scholar with Stanford University, Stanford, CA, USA. His research interests include formal guarantees on perception, planning, and control for mobile robots.

Bohao Zhang (Graduate Student Member, IEEE) received the B.S.E. degree in computer engineering from the University of Michigan, Ann Arbor, MI, USA, in 2020 and the dual B.S.E. degree in electrical and computer engineering from Shanghai Jiaotong University, Shanghai, China, in 2020. He is currently working toward the Ph.D. degree in robotics with the University of Michigan.

His current research interests include optimal control of bipedal robots.

Pengcheng Zhao (Member, IEEE) received the B.S. degree from Shanghai Jiao Tong University, Shanghai, China, in 2014, and the Ph.D. degree from the University of Michigan, Ann Arbor, MI, USA, in 2020, both in mechanical engineering.

His research interests include optimal control of nonlinear hybrid systems and its application to robots and self-driving vehicles.

Ram Vasudevan (Member, IEEE) received the B.S. degrees in electrical engineering and computer sciences, the M.S. degree in electrical engineering, and the Ph.D. degree in electrical engineering from the University of California at Berkeley, Berkeley, CA, USA, in 2006, 2009, and 2012, respectively.

He is currently an Assistant Professor of Mechanical Engineering with the University of Michigan, Ann Arbor, MI, USA, with an appointment with the University of Michigan's Robotics Program. His research interests include the development and application of optimization and systems theory to quantify and improve human–robot interaction.