Smooth RRT-Connect: An extension of RRT-Connect for practical use in robots

Chelsea Lau and Katie Byl
UCSB Robotics Lab, Dept. of Electrical and Computer Engineering
University of California at Santa Barbara
Santa Barbara, CA 93106
Email: {cslau12, katiebyl}@gmail.com

***Abstract*—We propose a new extend function for Rapidly-Exploring Randomized Tree (RRT) algorithms that expands along a curve, obeying velocity and acceleration limits, rather than using straight-line trajectories. This results in smooth, feasible trajectories that can readily be applied in robotics applications. Our main focus is the implementation of such methods on RoboSimian, a quadruped robot competing in the DARPA Robotics Challenge (DRC). Planning in a high-dimensional space is also a large consideration in the evaluation of the techniques discussed in this paper as motion planning for RoboSimian requires a search over a 16-dimensional space. We show that our approach produces results that are comparable to the standard RRT solutions in a two-dimensional space and significantly outperforms the latter in a higher-dimensional setting both in computation time and in algorithm reliability, in our experiments.**

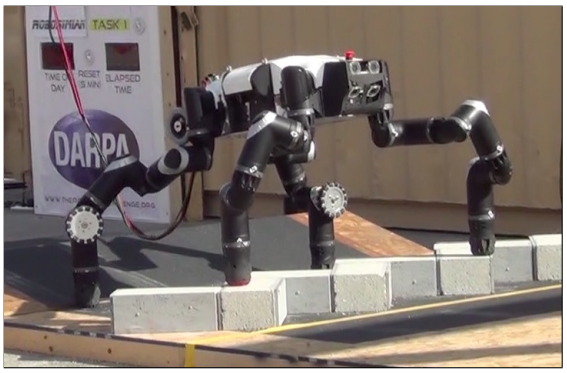


Fig. 1: Robosimian completing the rough terrain task during the DRC trials in December 2013.

I. Introduction

This paper focuses on practical algorithms to find joint trajectories for a high-dimensional quadruped to traverse complex, cluttered environments. In particular, we seek algorithms that are both computationally fast and highly reliable at finding solutions while simultaneously producing trajectories that can be executed quickly, given real-world velocity and acceleration limits of the robot, to allow for fast, autonomous real-time planning and locomotion on rough terrain.

Rapidly-Exploring Randomized Trees (RRTs) are a popular and effective approach to solving path planning problems as they are able to find feasible, albeit non-optimal, solutions quickly even in high-dimensional spaces. In this paper, we consider RRTs for generating joint trajectories for the quadruped robot RoboSimian, shown in Figure 1, as part of the DARPA Robotics Challenge (DRC). Each of RoboSimian's four limbs has seven joints which contribute to its ability to perform complicated manipulation tasks. However, the high degree of freedom design greatly complicates motion planning. By partitioning the full 28 joints into two categories – one planned using RRTs on a kinematic chain including one stance leg, the swing leg, and two passive angles at the ground contact, and the other planned using inverse kinematics tables – the problem can be reduced to a search over a 16-dimensional space, as in [1], [2], [3]. Even with this reduction in the state-space, the dimensionality is still high enough to make most planning algorithms impractical for real-time application. The RRT's ability to accomodate such a high dimensionality makes it one of the few methods available for our purposes. A significant disadvantage of RRTs is the ubiquitous presence of sharp changes in direction in the resulting trajectories. When faced with velocity and acceleration limits, this lack of smoothness can force all joints to slow down to essentially a stop at an inflection point, where velocity reverses for any one joint. With the events in the DRC under strict time constraints, adding unnecessary time in joint execution is highly undesirable. To address this issue, we propose an alternate extend function, following the framework of [4], and present an RRT algorithm that expands the tree along curves rather than straight-line trajectories. These curves obey velocity and acceleration limits, also eliminating the need for post-processing of the resulting trajectories.

Other methods that address the issue of smoothing RRT trajectories include post-process smoothing [4], [5], extending using splines [6], [7], [8], and extending using feedback control methods [9]. The latter two methods are designed for two-dimensional systems and take advantage of two-dimensional coordinate transformations in order to define one of the states as the heading. Limitations are then placed on the variation in the heading which leads to smooth trajectories in the 2D state-space . A value such as heading has no clear analogy in high dimensional spaces, however, which makes it difficult to translate these methods to a 16-dimensional space. Also, we are only interested in smoothness with respect to time, so requiring smoothness in the state-space might lead to unnecessary restrictions on the trajectories. The post-processing procedures of [4] and [5] are more promising for high dimensional problems; however, being post-process procedures, they are limited by the success of the RRT algorithm itself. If the RRT algorithm cannot find a solution in a reasonable amount of time or with the amount of memory available, the post-processing procedures have no value. We

show that expanding along curves reliably finds solutions, and finds them quickly, in cases when the standard RRT algorithm fails to do so, in particular, for higher-dimensional searches.

The rest of this paper is organized as follows. Section II describes RRT-Connect and a post-processing smoothing algorithm, used together to benchmark our results, and III introduces our modified RRT algorithm. Results, conclusions, and future work are presented in Sections IV, V and VI.

II. Background

A. RRT-Connect

The RRT-Connect algorithm [10] is an extension of the original RRT algorithm [11] that builds two trees that grow towards each other rather than just one towards the overall goal location. One of these trees starts from the start location and the other from the goal location. In each iteration, one tree is chosen to be expanded randomly, as per the original algorithm, while the other tree is chosen to extend towards the last node in the first tree. The roles of the trees are then swapped in the next iteration. This process is continued until the connect tree is able to extend within a specified distance of the last node of the extend tree. The extend function expands along a straight line that connects the nearest neighbor node and the "goal" node – the randomly-generated node for the extend tree or the last node in the extend tree for the connect tree. This extend function often results in sharp changes in direction which make execution of the trajectories unnecessarily slow and prone to control errors unless smoothed prior to execution. The extend function we propose expands along curves that obey velocity and acceleration limits rather than straight-line trajectories so that the final trajectory has a relatively short execution time and requires no post-processing.

B. Post-Process Smoothing

To post-process for smoothness, we use an algorithm presented in [4]. In [4], the first step in the smoothing process is finding time-optimal straight-line trajectories between each waypoint in the trajectory. These straight-line trajectories start and end at rest. In an iteration of the algorithm, two points are then randomly sampled from the concatenation of these straight-line trajectories. All segment(s) between these two points are then replaced by the time-optimal trajectory. This time-optimal trajectory is calculated using the methods in Section III-A, and is tested for collision-free feasibility. The number of iterations is specified by the user.

III. RRT Algorithm Modifications

A. Time-Optimal Extend Function

Our new Extend function also follows a general framework similar to [4], constructing time-optimal trajectories along which to extend the tree. However, unlike the post-processing in [4], our Extend trajectories have exact velocity constraints only at the initial node and are designed to minimize the time of arrival at the newly-created goal node of the tree. As in [4], we define a set of motion primitives: P^+ and P^-, which correspond to parabolic joint trajectories over time with positive and negative second derivatives, respectively; and L^+ and L^-, which correspond to straight lines with positive and negative derivatives, respectively. Intuitively, the fastest a joint can move from one point to another is at its maximum speed, $v_{\max}$. However, unless it is already moving at maximum speed, under acceleration constraints, the joint will have to spend time accelerating (or decelerating) to that point. The time spent accelerating (or decelerating) is minimized by doing so at the acceleration limit, $a_{\max}$. Therefore, the possible motion primitive combinations for time-optimal trajectories are: P^+, P^-, P^+L^+, and P^-L^- with the magnitudes of the appropriate second derivatives set as $\pm a_{\max}$ and the magnitudes of the straight-line derivatives set as $\pm v_{\max}$.

To generate the trajectories between waypoints, we first identify which joint requires the longest time, which sets a time constraint on other joints. We then find the appropriate motion primitive combination for each remaining joint in accordance with the aforementioned trajectory time.

1) Minimum Possible Time: The total time to complete a trajectory is determined by the "slowest" joint - i.e. the joint that takes the longest to get from its start to end location. To compute this value, the time-optimal trajectory for each joint is calculated, and the total time, denoted as $T_{\max}$, is set as the maximum execution time of the computed trajectories, denoted as T.

In the case of P^+, the trajectory time is given by the solution to the equation

$$\frac{1}{2}a_{\max}t^2 + v_0 t + q_0 - q_f = 0. \tag{1}$$

For the P^+ case to be valid, velocity must not exceed its limit, so

$$a_{\max}t + v_0 \leq v_{\max}.$$

For P^-, $a_{\max}$ is replaced by $-a_{\max}$ in Equation 1, and the velocity condition is $a_{\max}t - v_0 \leq v_{\max}$.

For P^+L^+, we calculate the switch time which occurs when the fastest joint trajectory reaches its the maximum velocity limit :

$$t_s = \frac{v_{\max} - v_0}{a_{\max}}. \tag{2}$$

and then compute the trajectory time as

$$T = \frac{\frac{1}{2}(v_{\max} - v_0)^2 + a_{\max}(q_f - q_0)}{a_{\max}v_{\max}}\ . \tag{3}$$

Similarly, the time for a P^-L^- trajectory is calculated by replacing $v_{\max}$ with $-v_{\max}$ and $a_{\max}$ with $-a_{\max}$ in Equations 2 and 3.

2) Motion Primitive Combination and Acceleration: Once we have the total trajectory time, we can determine the appropriate motion primitive combination for each joint. The first step in this process is calculating the accelerations for each possible combination. The combination corresponding to the minimum of these values defines the trajectory shape for the respective joint.

For P^+ trajectories, the acceleration is calculated as

$$a = \frac{2}{T_{\max}^2}(q_f - q_0 - v_0 T_{\max}). \tag{4}$$

This trajectory is valid if $a > 0$ and $aT_{\max} + v_0 \leq v_{\max}$. The acceleration for a P^- trajectory is also given by equation 4, but is now valid if $a < 0$ and $aT_{\max} + v_0 \geq -v_{\max}$.

For P^+L^+ trajectories, the acceleration is calculated as

$$a = \frac{\frac{1}{2}(v_{\max} - v_0)^2}{v_{\max}T_{\max} - (q_f - q_0)} \tag{5}$$

and the switch time is $t_s = \frac{1}{a}(v_{\max} - v_0)$. This trajectory is valid if $a > 0$ and $t_s < T_{\max}$. Note that if $v_{\max}T_{\max} =$

$(q_f - q_0)$, $t_s = 0$ since this corresponds to the case where $v_{\max} = v_0$. Equation 5 also gives the acceleration for P^-L^- trajectories, with the same switch time, but the trajectory is now valid if $a < 0$ and $t_s < T_{\max}$.

3) Connecting Two Trees: Unlike the original RRT-Connect algorithm, we require a separate Extend function when connecting nodes to join two trees because of the added velocity constraint from the "goal" node. In this case, the formulation from [4] is used. The calculations are as above except that the four resulting combinations of motion primitives are defined somewhat differently. The corresponding equations are summarized here for completeness.

For P^+P^- and P^-P^+ trajectories, the trajectory execution time is calculated as

$$T = 2t_p + \frac{v_1 - v_2}{sa_{\max}}$$

where t_p is the solution to

$$sa_{\max}t^2 + 2v_1 t + \frac{1}{2sa_{\max}}(v_1^2 - v_2^2) + x_1 - x_2 = 0$$

and $s = 1$ for P^+P^- trajectories and $s = -1$ for P^-P^+ trajectories. This execution time is valid if $t_p \leq \frac{1}{sa_{\max}}(sv_{\max} - v_1)$. The acceleration based on the computed $T_{\max}$ is calculated as the solution to the equation

$$T_{\max}^2 a^2 + (2T_{\max}(v_1 + v_2) + 4(x_1 - x_2))sa - (v_1 - v_2)^2 = 0.$$

The validity condition is $at_s - sv_1 \leq v_{\max}$, where $t_s = \frac{1}{2}\left(T_{\max} - \frac{1}{sa}(v_1 - sv_2)\right)$.

For $P^+L^+P^-$ and $P^-L^-P^+$ trajectories, the trajectory execution time is calculated as

$$T = t_1 + t_L + t_2 \ ,$$

where $t_1 = \frac{1}{sa_{\max}}(sv_{\max} - v_1)$, $t_2 = \frac{1}{sa_{\max}}(sv_{\max} - v_2)$, $t_L = \frac{1}{2a_{\max}v_{\max}}(v_1^2 + v_2^2 - 2v_{\max}^2) + \frac{1}{sv_{\max}}(x_2 - x_1)$, $s = 1$ for $P^+L^+P^-$ trajectories, and $s = -1$ for $P^-L^-P^+$ trajectories. This execution time is valid if t_1, t_L, and t_2 are all nonnegative. The acceleration based on the computed $T_{\max}$ is calculated as

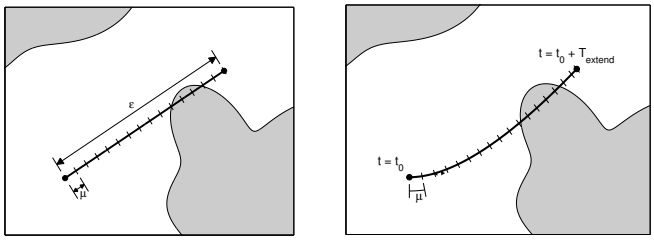
$$a = \frac{v_{\max}^2 - sv_{\max}(v_1 + v_2) + \frac{1}{2}(v_1^2 + v_2^2)}{T_{\max}v_{\max} - s(x_2 - x_1)} \ .$$

The solution is valid if the segment times – $t_1 = \frac{1}{sa}(sv_{\max} - v_1)$, $t_L = \frac{1}{2av_{\max}}(v_1^2 + v_2^2 - 2v_{\max}^2) + \frac{1}{sv_{\max}}(x_2 - x_1)$, and $t_2 = \frac{1}{sa}(sv_{\max} - v_2)$ – are all nonnegative.

4) Extension: In the standard RRT algorithms, the tree is expanded along a straight-line trajectory by a distance ε, as illustrated in Figure 2 . Even though we have closed-form solutions for the trajectories, calculating the distance along the curve would require numerical integration which would signficantly increase computation effort. As an alternative, we instead expand along the curve for a time T_{extend}.

B. Other modifications

Increasing ε allows for quicker expansion of the tree; however, in the traditional RRT (and RRT-Connect) algorithm, this results in lower resolution of feasibility checks which are only done on the new node. To circumvent this issue, we adjusted both algorithms to check for feasibility at a spacing specified by parameter μ, see Figure 2. This allows the use of larger values ε without the reduced accuracy in feasibility. This adjustment is particularly important for expansion along a curve since we are expanding according to a specified time rather than a distance.



(a) (b)

Fig. 2: A larger (a) ε or (b) T_{extend} value can decrease computation time, however, it also leads to imprecision in terms of feasibility. Checking for feasibility at intervals between the nodes reduces the likelihood of incorrectly labeling an expansion as feasible. Note that we still always check the new node location regardless of the value of μ with respect to ε.

RRT algorithms select nodes to extend based on a "nearness" metric to randomly generated points in space. Choice of an appropriate distance metric is already a topic of interest for RRT modifications [12]. Figure 3 illustrates the disparity between the Euclidean distance between two nodes and the time it takes to get from one to the other, given velocity and acceleration constraints. To account for this issue, we adjusted the distance metric when searching for a "nearest" neighbor when connecting trees so that, instead of finding the nearest node based on Euclidean distance alone, we also test for points that are close in velocity. We calculate "distance" as:

$$\Delta = \gamma_d ||q_{\text{rand}} - q_{\text{near}}|| + \gamma_v \max_j |v_{\text{rand}}(j) - v_{\text{near}}(j)|$$

where $v_{(.)}(j)$ corresponds to the velocity of the jth joint. This reduces unnecessary detours in the trajectory that are required to transition to the velocity of the end point.

C. Quantifying "Smoothness"

To quantitatively compare the methods discussed in this paper, we take the ratio of the execution time for resulting joint trajectories calculated according to both velocity and acceleration constraints to the execution time with only velocity limited. In this definition, a "smoother" trajectory, i.e. one with a ratio closer to unity, is one that does not require significant

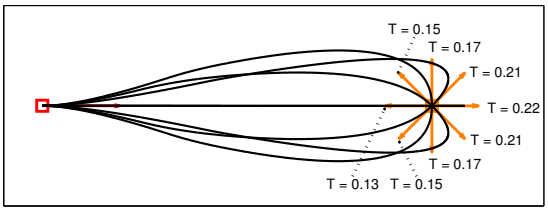


Fig. 3: When taking velocity and acceleration into consideration, a "nearest" neighbor is not necessarily the node that is closest in terms of Euclidean distance only. In the example illustrated in this figure, the direction of the velocity vector affects the time-to-go by as much as 70%.

	% Success	Total FC	Sm. FC	Num. Iter.	Num. Iter. Sm.	T_{elapsed}	T_{execute}	R
2D Maze								
RRTC only	100	5776.9 ± 1371.5	—	1094.1 ± 264.4	—	0.496 ± 0.122	7.849 ± 0.803	3.103 ± 0.286
RRTC+post (R)	100	7694.8 ± 1675.8	1917.9 ± 871.8	1094.1 ± 264.4	97.4 ± 41.5	0.756 ± 0.177	4.917 ± 0.491	2.193 ± 0.158
RRTC+post (n_{iter})	100	9198.2 ± 1583.5	3421.3 ± 624.4	1094.1 ± 264.4	150.0 ± 0.0	0.806 ± 0.134	4.095 ± 0.973	1.883 ± 0.390
Smooth RRTC	100	13369.4 ± 12418.3	—	2656.7 ± 2399.8	—	2.644 ± 2.710	5.829 ± 0.889	2.130 ± 0.222
2D Random								
RRTC only	100	1300.3 ± 418.0	—	247.2 ± 80.9	—	0.175 ± 0.050	4.027 ± 0.574	3.027 ± 0.357
RRTC+post (R)	100	2823.8 ± 1034.4	1523.4 ± 893.2	247.2 ± 80.9	60.7 ± 36.3	0.451 ± 0.171	2.245 ± 0.245	1.883 ± 0.173
RRTC+post (n_{iter})	100	5540.5 ± 744.3	4240.2 ± 531.7	247.2 ± 80.9	150.0 ± 0.0	0.844 ± 0.179	1.696 ± 0.234	1.499 ± 0.199
Smooth RRTC	100	2144.7 ± 1733.8	—	315.4 ± 382.7	—	0.504 ± 0.457	2.598 ± 0.580	1.806 ± 0.278
16 Dimensions								
RRTC only	81	10375.1 ± 2631.1	—	1287.0 ± 307.4	—	0.971 ± 0.289	15.778 ± 2.002	5.735 ± 2.819
RRTC+post (R)	81	21447.8 ± 6986.7	11072.8 ± 6320.9	1287.0 ± 307.4	125.6 ± 63.4	1.992 ± 0.642	1.654 ± 0.102	1.378 ± 0.081
RRTC+post (n_{iter})	81	29966.8 ± 3206.5	19591.8 ± 1659.1	1287.0 ± 307.4	200.0 ± 0.0	3.194 ± 0.383	1.569 ± 0.103	1.327 ± 0.084
Smooth RRTC	100	884.7 ± 354.1	—	30.9 ± 29.8	—	0.102 ± 0.052	2.347 ± 0.394	1.343 ± 0.064

TABLE I: Summary of experimental results. For each environment, the means and standard deviations of the number of feasibility checks while building the tree and when smoothing (when applicable), the number of iterations while building the tree and when smoothing (when applicable), the elapsed time, the execution time, and the value of R over 100 trials are shown for RRT-Connect (shortened as RRTC for space) alone, RRT-Connect with post-process smoothing terminated after reaching a specified R value, RRT-Connect with post-processing smoothing terminated after a specified number of iterations, and Smooth RRT-Connect.

acceleration or deceleration and spends most of the trajectory with at least one joint moving at its maximum velocity. We denote this measurement with the letter R, and compute it as

$$R = \frac{T_{\text{execute}}}{\sum_{i \in \{0,1,\ldots,N\}} \frac{\max_j |q_i(j) - q_{(i-1)}(j)|}{v_{\max}}}$$

where T_{execute} is the total execution time of the trajectory, N is the number of waypoints in the trajectory, and $q_i(j)$ corresponds to the position of joint j at the ith waypoint in the trajectory.

IV. Results

To test our framework, we constructed various environments in which to run the standard RRT-Connect with post-process smoothing as in [4] and the Smooth RRT-Connect with the time-optimal extend function. Two of the environments are in two-dimensions for a more intuitive interpretation of the results, and the third is in 16 dimensions, which is the dimension of the search used for RoboSimian [1], [2], [3].

In all experiments, we used an ε value of 0.03 (rad) for the RRT-Connect algorithm, and an extend value T of 0.2 (sec) for the Smooth RRT-Connect algorithm. These values were chosen based on experimental results indicating that they consistently provided the quickest results for their respective algorithms. Feasibility checks were done every 0.01 unit of distance for both the standard RRT-Connect and Smooth RRT-Connect. In the connect phase of the Smooth RRT-Connect algorithm, the difference in velocity was weighted by $\gamma_v = 5$ and the Euclidean distance by $\gamma_d = 1$. The maximum velocity and acceleration for each joint were set to 1.2 rad/sec and 1.5π rad/sec^2, respectively. These values correspond to the velocity and acceleration constraints for RoboSimian [1]. Joint limits were set as 0 rad to 1 rad, and the start and end nodes were set as the extremes of the joint ranges (i.e. all zeroes and all ones, respectively). The trees were allowed to expand for a maximum of 50,000 extend iterations. Trials that didn't return a solution within this allowed number of iterations were considered as "failed" trials. For each environment, 100 trials were run for each algorithm.

The termination criteria for post-process smoothing was done in two ways: 1) by running the algorithm until a desired R, as defined in Section III-C, was met, and 2) by running the algorithm for a set number of iterations (150 for the 2D cases; 200 for the 16D searches). The value for R for the former case was set to one standard deviation above the mean value for the corresponding set of Smooth RRT-Connect trials.

As seen in Figure 4 and Table I, in the two-dimensional cases, the performances of the RRT-Connect algorithm with post-process smoothing and the Smooth RRT-Connect algorithm are comparable. Neither one is vastly superior to the other; although, the Smooth RRT-Connect does remove the extra layer of uncertainty involved with applying a random procedure to a randomly-generated solution and the ambiguity of selecting the termination parameter of said procedure.

However, when faced with even a simple obstacle such as a block in the center of the 16-dimensional state-space, RRT-Connect takes much longer than our Smooth RRT-Connect algorithm, even without the additional computation involved with post-processing. Results from our experiments also show that the RRT-Connect failed 19% of the time, see Table I, and even when successful, the algorithm took roughly 20 times longer, on average, to obtain a solution and smooth it. Also, note that the number of iterations required for the 81 successful RRTC solutions were distributed fairly evenly, falling strictly between 500 and 2,000 iterations. Thus, it is practical to call the 19 trials not successful after 50,000 iterations "failures".

Figure 5 shows plots for all 16 joints over time for representative runs with RRTs with post-process smoothing (left) and for our Smooth RRTC (S-RRTC). For this simple case, with a single hypercube obstacle from 0.3 to 0.7 in each dimension, the theoretical optimal time can be computed and is $t_{opt} = \frac{7}{6} + \frac{4}{5\pi} \approx 1.4213$ sec, and RRTC with post-processing came within 3% of this value for the fastest of the 81 successful trials. One can visualize the requirement here is that not all joints fall between 0.3 and 0.7 at the same moment in time; i.e., at least one trajectory must have reached $q = 0.7$ (on the left subplot, look to the topmost, black line) before the last of the 16 joints (i.e., the red line at bottom) passes $q = 0.3$.

We step through the detail of the previous paragraph to make two important points. First, it is clear our S-RRTC results can be further improved by post-processing, if desired, just as the plain RRTC solutions were; however, our focus in this work is to quantifiably compare smoothing during tree

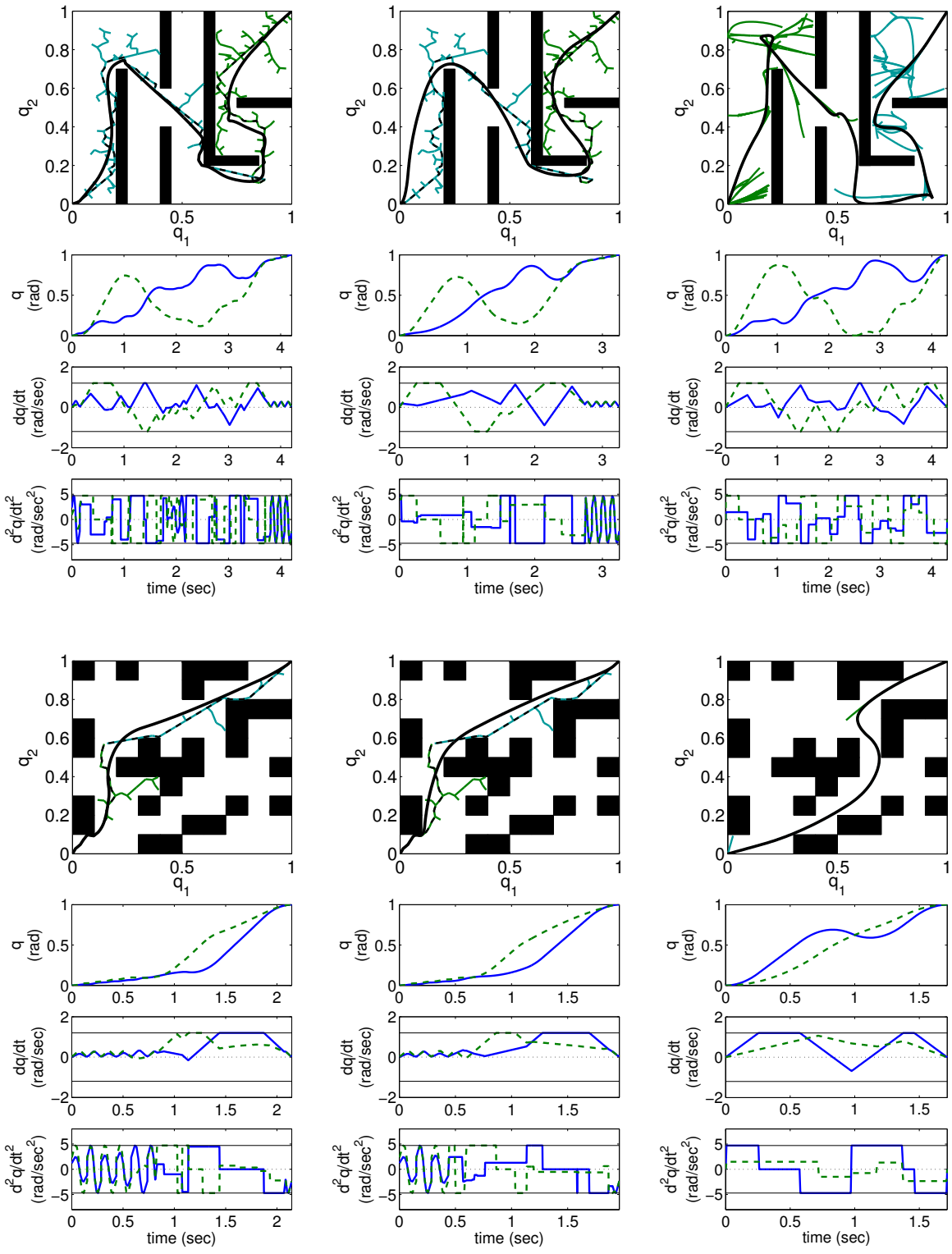


Fig. 4: Top row (left to right): trajectory solutions for a 2D maze-like environment for (a) RRT-Connect with post-process smoothing terminating with a specified R value, (b) RRT-Connect with post-processing, terminating after a specified n_{iter}, and (c) S-RRTC. Bottom row: trajectory solutions for an environment with randomly-generated obstacles from (d) RRT-C with post-processing, terminating at a specified R, (e) RRT-C with post-processing, terminating after a specified n_{iter}, and (f) our S-RRTC algorithm. In the trajectory vs. time plots, the solid line shows q_1, and the dashed line is q_2. In the state-space plots, the solid, colored lines correspond to the trees, the dotted line to the unsmoothed trajectories (when applicable), and the solid black lines to the smoothed paths.

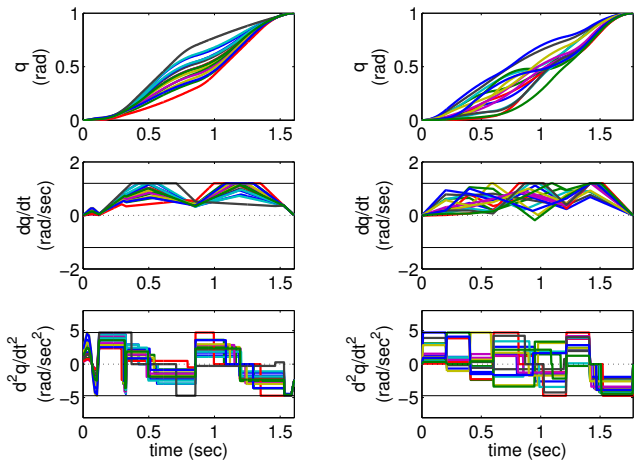


Fig. 5: Trajectory solutions for a 16-D system with a single obstacle spanning from 0.3 rad to 0.7 rad in each dimension with limits of 0 rad to 1 rad on each state. (a) RRT-Connect with post-process smoothing and (b) Smooth RRT-Connect.

extension versus via post-processing. Second, we emphasize that our 16D problem was relatively simple, yet this is enough to demonstrate the value of the S-RRTC as an alternative to the RRTC in certain environments. In the 2-dimensional examples presented in Figure 4, we can see that the RRTC trajectories tend to "hug" obstacles, even with post-process smoothing. The S-RRTC trajectories, on the other hand, frequently venture away from the obstacles which allows the trajectories to move around large obstacles more effectively. This property holds when the problem is extended to a 16-DOF scenario, evident in the ability of the S-RRTC to move around the 16-dimensional block obstacle with relative ease as oppose to the standard RRTC which spent many, if not all, of its iterations scaling the boundary of the obstacle. In addition to the single block, we tested a variety of other obstacle scenarios for the 16-DOF case, not presented here due to space constraints, which support this observation. These obstacle scenarios included environments with 2000 to 4000 randomly generated polytopes and environments in which feasible paths were constrained to "tunnels" in the state-space. In this latter case, the standard RRTC was, in fact, able to find solutions more quickly and consistently than the S-RRTC.

V. Conclusion

Our main goal is to generate joint trajectories for RoboSimian to traverse complex terrain. This requires planning in high-dimensions, for which most optimal planners require too much computational effort for real-time implementation, especially in high-dimensional cases. RRTs provide quick solutions in these high-dimensional spaces; however, these trajectories are not smooth and require significantly more time to execute than a smoothed trajectory. It is then worth the extra computation to smooth the paths since the amount of time saved in execution far outweighs the amount of time to smooth. We considered two methods for smoothing: 1) post-processing the original RRT-Connect trajectories, and 2) extending along curves rather than straight lines within the RRT search itself. While the extension along curves does not provide too much of an advantage in the two-dimensional cases, in the 16-dimensional environment, it greatly outperforms the standard RRT-Connect in both consistency in obtaining solutions and computation time in our testing. This is particularly persuasive for us since we will be planning for a 16-dimensional space. Another important aspect of the Smooth RRT-Connect algorithm is that there is no need for any post-processing – trajectories are already returned with timestamps that obey velocity and acceleration constraints.

VI. Discussion and Future Work

For real-world applications, one practical approach is to run multiple versions of the RRT-Connect algorithm in parallel and to pick a solution based on some criteria such as the fastest trajectory from the first N returned solutions. Indeed, we anticipate there may be situations in which RRT's with post-processing outperform our Smooth RRTC algorithm, in which case running parallel trees on multiple cores as proposed in [1] would increase the chances of finding a quick, viable solution, taking advantage of the strengths of both algorithms. Of course, this comes at the cost of requiring the capability and capacity to run multiple instances at once. However, this is a nonissue in the case of the DRC since we are provided with enough cores to run parallel trials for on-the-fly planning, and parallel planning will only become more practical in robotics over time, as computing capabilities grow.

As a point of discussion, mean values for t_{exec} in Table I are roughly 50% longer in the S-RRTC cases than for RRTC with a fixed number of post-processing iterations. Two potential factors contributing to this are worth mentioning. First, post-processing obviously allows "short cuts" to be made, bypassing original waypoints. Second, the unidirectional nature of tree extension in S-RRTC means that paths grow in a causal way: i.e., they cannot anticipate where upcoming waypoints will be. Figure 6 highlights this.

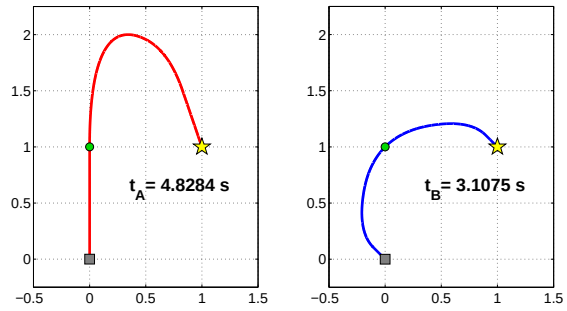


Fig. 6: Causal smoothing (left) versus post-processing for non-causal smoothing (right). Starting at $(0, 0)$, solutions show optimal trajectories through the waypoint $(0, 1)$ and arriving with zero velocity at the yellow star goal $(1, 1)$. R (smoothing factor) values are proportional to t_{exec} for each.

Shown are causal versus non-causal time-optimal trajectories to traverse the waypoints (0,0), (0,1), and (1,1) in order, with zero velocity at the first and last waypoints, and subject only to an acceleration limit of $1\ unit/s^2$. At left (red path), a causal solution mimicking our S-RRTC approach is shown. There is a constant acceleration directly toward (0,1), resulting

in a large "overshoot" and requiring roughly 50% more time ($t_A = 2 + 2\sqrt{2}$, vs $t_B = 2\frac{2}{2\sqrt{2}-2}$ sec) to get to the goal (yellow star), compared with a non-causal solution (at right) analogous to post-processing, in which the full trajectory is truly optimized, since all waypoints are known at once.

In practice, S-RRTC is not quite as simplistic as illustrated above, since the choice of which node to extend is biased by the velocity vector at each tree node, as illustrated earlier in Figure 3. The R values at far right in Table I are similar for RRTC-post (n_{iter}) and S-RRTC, which (encouragingly) seems to indicate the causal nature of our Extend algorithm is not the root cause of slower t_{exec} values for the latter. For future work, as suggested in Section IV, post-processing of S-RRTC therefore seems likely to yield faster joint execution times, comparable to those of RRTC with post-processing. Referring again to Figure 5, one can visualize the $q(t)$ trajectories in the upper right subplot can be further "smoothed out" to more closely approach those in the upper left. Other future work includes improving distance metrics for the Extend portion of the algorithm, and seeking optimal solutions, e.g., via RRT*.

We also plan to extend the formulation to the RoboSimian framework to run experiments with the actual robot. Currently, the planning process for swing trajectories uses RRT-Connect trajectories that are pruned and assigned timestamps based on maximizing the acceleration or velocity at each step in the trajectory. This process does not reduce all unnecessary changes in direction inherent of RRT-Connect trajectories that result in slow and unnatural trajectories. One of the key considerations in generating trajectories for RoboSimian is the number of feasibility checks since these require solving inverse and forward kinematics and perhaps collision checks, which all require significant computation time. As the results from our experiments show, the post-process smoothing adds a significant number of feasibility checks, sometimes more than 50% of the total amount. We were curious whether building a tree using Smooth RRTC might require more feasibility checks than post-processing a standard RRT, but in fact, the exact opposite is (happily) the case, by roughly a factor of 20x. We anticipate future testing with the same-dimensionality (16-DOF) search on RoboSimian will only highlight the benefits of smoothing while building the tree versus after.

ACKNOWLEDGEMENT

This work is supported by JPL NASA subcontract #1471138, for the DARPA Robotics Challenge (DRC).

REFERENCES

[1] B. W. Satzinger, C. Lau, M. Byl, and K. Byl, "Tractable locomotion planning for robosimian." Submitted to IJRR 2015.

[2] B. Satzinger, J. I. Reid, M. Bajracharya, P. Hebert, and K. Byl, "More solutions means more problems: Resolving kinematic redundancy in robot locomotion on complex terrain," in Proc. Int. Conf. on Intelligent Robots and Systems (IROS), 2014.

[3] B. Satzinger, C. Lau, M. Byl, and K. Byl, "Experimental results for dexterous quadruped locomotion planning with RoboSimian," in Proc. Int. Symp. on Experimental Robotics (ISER), 2014.

[4] K. Hauser and V. Ng-Thow-Hing, "Fast smoothing of manipulator trajectories using optimal bounded-acceleration shortcuts," in Proc. Int. Conf. on Robotics and Automation (ICRA), pp. 2493–2498, 2010.

[5] J. Pan, L. Zhang, and D. Manocha, "Collision-free and smooth trajectory computation in cluttered environments," Int. J. of Robotics Research, vol. 31, no. 10, pp. 1155–1175, 2012.

[6] K. Yang and S. Sukkarieh, "An analytical continuous-curvature path-smoothing algorithm," IEEE Transactions on Robotics, vol. 26, no. 3, pp. 561–568, 2010.

[7] E. Shan, B. Dai, J. Song, and Z. Sun, "A dynamic rrt path planning algorithm based on b-spline," in Proc. 2nd Int. Symp. on Computational Intelligence and Design (ISCID), vol. 2, pp. 25–29, 2009.

[8] R. Kala and K. Warwick, "Planning of multiple autonomous vehicles using rrt," in Proc. Int. Conf. on Cybernetic Intelligent Systems (CIS), pp. 20–25, 2011.

[9] L. Palmieri and K. O. Arras, "A novel RRT extend function for efficient and smooth mobile robot motion planning," 2014.

[10] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in Proc. Int. Conf. on Robotics and Automation (ICRA), vol. 2, pp. 995–1001, 2000.

[11] S. M. LaValle, "Rapidly-exploring random trees a new tool for path planning," 1998.

[12] P. Cheng and S. M. LaValle, "Reducing metric sensitivity in randomized trajectory design," in Proc. Intelligent Robots and Systems (IROS), vol. 1, pp. 43–48, 2001.