Senior Design Fall Research Activities with Trajectory Optimization based on Signal Temporal Logic Specifications

Yann Gilpin
Department of Electrical Engineering
University of Notre Dame
Notre Dame, IN, USA
ygilpin@nd.edu

***Abstract*—This work is a report detailing efforts to improve optimal trajectory synthesis based on signal temporal logic constraints. Signal Temporal Logic offers a convenient and rigorous way of describing high level tasks, such as motion planning, for automatic solution synthesis. While there are known methods to solve this problem, the computations are NP-Hard and the complexity is exponential. Many practical systems have high dimensionality making this otherwise promising approach unusable. The approach detailed here is to use a variety of meta-heuristic global optimization techniques and pointwise robustness to achieve a lower computational complexity while possibly sacrificing optimally. Pointwise robustness is a custom formulation that evaluates the robustness of a particular point with respect to a signal temporal logic specification. In working in this direction, I ran into some theoretical limitations and discovered a similarity between trajectory optimization literature (based on additive cost functions and dynamic constraints) and pointwise robustness that offers the potential for lower computational complexity in future work.**

I. Introduction

The control of robots with many degrees of freedom (e.g. state variables) in order to complete high level tasks is difficult. By hand, one must determine the desired motion of all free variables. This multi-dimensional motion often has several constraints, such as avoiding self contact or collision with obstacles, in addition to dynamic constraints (i.e. acceleration). While hand calculations may be acceptable for simple systems performing repetitive tasks in controlled environments, it is increasingly inconvenient for systems with non-invertible dynamics or changing tasks. For example a pick-and-place robot must generate a new path plan if the source parts are not all in the same starting location each time or if the destination is out of alignment. If a quick automated approach existed, it would save lots of programming time and increase the range of activities that can be automated in areas such as in search and rescue, domestic aids and national defense.

II. Previous Approaches and Related Work

One promising approach is to define the problem in terms of Signal Temporal Logic (STL) specifications. The core idea is to gather the variables of interest into a signal space. Then constraints upon the signal space can be written in terms of Boolean operators, *AND, OR, NOT* and temporal operators *ALWAYS, EVENTUALLY*. This allows many practical tasks to be encoded rigorously. For example, if a robot was tasked to water the plants once while the owner was on a short vacation, one could write a simple specification: *by time one week, water the plants beside the window while ALWAYS avoiding the furniture*.

While it is easy to evaluate if a potential solution satisfies all STL predicates, a binary satisfied or unsatisfied result does not yield much information on how well a solution satisfies or how badly it violates the constraints. Hence the robust semantics of STL were developed. They evaluate the quality of a potential trajectory by finding and evaluating how close the proposed solution is from violating a specification, in the case that the solution satisfies all constraints. If one or more predicate is violated, the robustness finds the worst part of the trajectory and evaluates the amount of violation. Formally finding the worst point corresponds to taking the minimum of the set of robustness values for each predicate, at each point of the proposed solution. For more information about the semantics and formal notation used with traditional robust STL see [1].

The traditional robust STL semantics can and have been used for trajectory optimization (i.e. finding the most robustness solution). Unfortunately, these approaches struggle due to the non-smooth (due to the use of min and max), and non-convex (due to the shape of the constraints) nature of this problem formulation. Currently the fastest approach is Mixed Integer Linear Programming (MILP) [1]. As the name implies the formulation consists of a continuous portion and an integer portion. The integer part is formed by a set of binary variables representing the violation or satisfaction of a predicate at a particular point in time. Hence to have a complete understanding of a proposed solution, there is a binary variable for each predicate at each point in time. The continuous part is the robustness of the overall trajectory. Lastly the word linear refers to how the predicates can only limit linear combinations of signals in the signal space in a linear way which yields a linear robustness/cost function. For information about MILP can also be found in [2]. After formulating the problem as MILP, off-the-shelf MILP solvers

are used. They usually perform a binary search through the sea of binary variables to find a combination that satisfies all predicates. Next using the linear constraints it can find the union of these regions and then optimize the robustness within this solution space. This approach suffers from lengthy searches through the large number of binary variables when used on systems with a many degrees of freedom.

Recent work approaches this problem from a different angle. The use of min and max operators in the definition of traditional robustness resulted in a non-smooth function excluding gradient descent and similar calculus based optimization techniques. Hence smooth approximations to min or max operators have been proposed. For example [3] uses a log exponential sum:

$$max(\alpha, \beta, \gamma...) \approx \frac{1}{k} \log(e^{k\alpha} + e^{k\beta} + e^{k\gamma} + ...) \quad (1)$$

Where α, β, γ could be robustness of a particular predicate at different points in time and k is a free parameter that affects the accuracy of the approximation. Specifically the error of the approximation is limited by:

$$\left| max(\alpha, \beta, ...) - \frac{log(e^{k\alpha} + e^{k\beta} + ...)}{k} \right| \leq \frac{logN}{k} \quad (2)$$

Where N is the number of arguments While this approximation opens the door to the use of gradient descent based methods, it presents numerical stability problems and may not preserve the sign of the arguments. Often the sign of the argument is how satisfaction or violation of a predicate is encoded, but this is lost in this approximation.

Similarly, the authors of [4] use a combination of arithmetic and geometric means to approximate the min/max operators. This has the advantage of avoiding the numerical stability concerns associated with (1). It also preserves the sign of the arguments so satisfaction and violation are easy to see. Unfortunately due to problems associated with differences in orders of magnitude, this approach requires normalization of the range over which a signal is allowed to vary. This additional information is theoretically unnecessary and places an extra burden on the user.

III. My Summer Proposal

During the summer of 2019, I proposed an original method, based on pointwise robustness for solving this planning based optimization problem. Specifically, pointwise robustness would allow one to compute the robustness at each point, optimize them individually, and combine them at the end. Such a parallel method would be a way of limiting the computational complexity and enable use of available hardware for computational acceleration (e.g. GPU or FPGA). Additionally it opens the door to custom heuristics to find quick though potentially sub-optimal solutions.

My original schedule to complete this research is shown in Table I. In short the plan was to fully understand the current approaches to trajectory optimization and theory of optimization in the Fall semester so that I would be ready to implement my best ideas in the Spring semester.

TABLE I
Proposed Schedule

Month	Activity
August	Familiarize myself with the intricacies of STL, its robust semantics and its translation into Mixed Integer Programming
September	Explore the current optimization methods
October	Investigate metaheuristic algorithms
November-December	Decide upon best metaheuristics to take advantage of the common situations in robotics such as obstacle avoidance.
January-May	Implement the current approach and some metaheuristic algorithms. Compare robustness and computation speed.

IV. Fall Semester Activities

It turns out that I did not estimate the time it would take to complete the tasks in Table I very accurately. It took only a couple weeks for me to feel comfortable with the current approaches and the general ideas behind optimization. To write effective heuristics to take advantage of pointwise robustness, I had to establish them first, hence that became the first major topic of my work this semester.

A. Pointwise Robustness

Building from the standard definitions (see [1]), I developed a pointwise robustness definition for the standard signal temporal logic operators *AND, OR, NOT, ALWAYS* and *EVENTUALLY*. Since these definitions can be applied recursively only the base case for each operator is discussed.

1) Boolean Operators: The Boolean operators are largely the same as the traditional ones. If one wants the opposite of a particular predicate they can apply the *NOT* operator. Formally the robustness follows the negation operation:

$$\neg\alpha = -\alpha \quad (3)$$

Where α is the pointwise robustness of particular predicate. When two predicates are both active for the same point at the same time, for instance, this is expressed formally using the *AND* (conjunction) operator. The robustness with *AND* is:

$$\alpha \cap \beta = min(\alpha, \beta) \quad (4)$$

Where α and β represent then robustness of a point with respect to two distinct predicates. The use of minimum reflects the worst case thinking and penalizes being close to any constraint, such as an obstacle. Similarly the *OR* (disjunction) operator's robustness is implemented with the maximum function:

$$\alpha \cup \beta = max(\alpha, \beta) \quad (5)$$

This reflects the idea that when given a choice the robustness measures the better option.

In other words, applying Boolean operations to predicates combines their robustnesses using maximum, minimum or negation. This is why optimization over robustness is non-smooth. If smoothness is required, they can be approximated with an equation such as (1).

2) Temporal Operators: For the pointwise *ALWAYS* operator's robustness, I evaluated the distance between a particular point of interest and a constraint. To simplify this calculation to matrix multiplication, I restricted my implementation to linear constraints. Mathematically this can written as:

$$\alpha = b - Ax_c \tag{6}$$

Where A is a row vector and b is a scalar such that $Ax \leq b$ forms a linear constraint and x_c is the (potentially multidimensional) position at a particular point in time along a proposed trajectory. First, notice how if the predicate is violated then then: $\alpha < 0$ and vice-versa for satisfaction. This makes it easy to see if a predicate is violated and at which points. Secondly through the use of negative b or negative entries in A there is no loss of generality. Visually the *ALWAYS* operator's effects can be seen though a plot of the pointwise robustness function over the signal space (in the 2D case). See Fig. 1 for such an example.

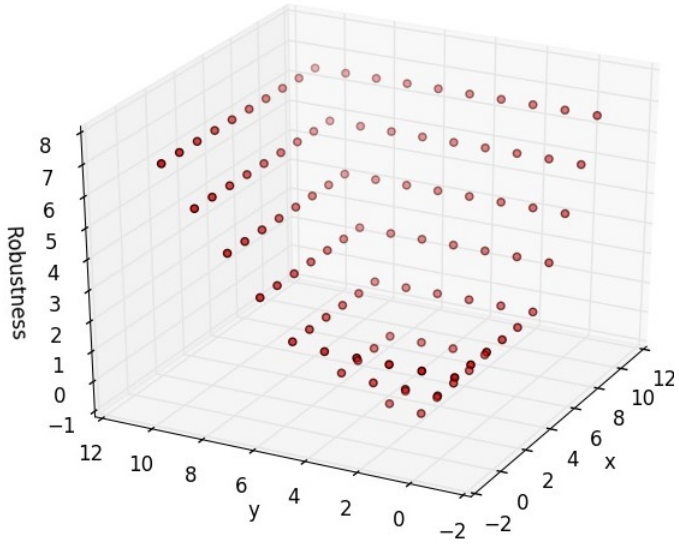

Fig. 1. Example of the *ALWAYS* operator's effect on robustness over a portion of a 2D signal space with a rectangle to be avoided at $(1 \leq x \leq 3) \cap (1 \leq y \leq 3)$.

Adapting the *EVENTUALLY* operator, was not as simple as the others. In thinking that most robots, like cars, have a particular maximum speed, I defined the robustness around the concept of constant speed. It also matters whether or not the trajectory later (or previously) satisfied the constraints hence there are two slightly different expressions. If the trajectory satisfies the constraint at that point or later in time the robustness is computed as follows.

$$\alpha = \frac{|1 + t_c - t_2|}{|Ax_c - b| + \epsilon} \tag{7}$$

Where x_c and t_c are the current point and time of interest respectively. The last time at which the eventually constraint may be satisfied is t_2 hence the 1 in the numerator, which prevents zero robustness when the predicate is satisfied at t_2. There is also a correction factor, ϵ, that helps with numerical stability. I found through experimentation that the value 0.4 worked well. Notice how this function rewards being close to the goal as quickly as possible. Similarly in the case where none of the points along a trajectory satisfy the predicate:

$$\alpha = \frac{-|Ax_c - b|}{|1 + t_c - t_2| + \epsilon} \tag{8}$$

The only differences between (7) and (8) are the negative and the swapping of the position and temporal terms. I used the negative so at each point it is obvious whether or not the predicate was satisfied. The positional and temporal terms are flipped to penalize being far from the goal and to reward having more time to satisfy the predicate. The last case is after the *EVENTUALLY* is satisfied. When this occurs, its robustness is no longer computed. I reasoned that, after the the eventually is satisfied it should have no impact on the robustness of later points. This behavior is also reflected in the traditional case where only the best point along the entire trajectory is computed and it may be masked by another robustness value of another predicate etc. As with the *ALWAYS* case, *EVENTUALLY* operator's effect can be interpreted shaping the robustness value over a 2D signal space. See Fig. 2 for such a plot of the satisfied case and Fig. 3 for the dissatisfied case.

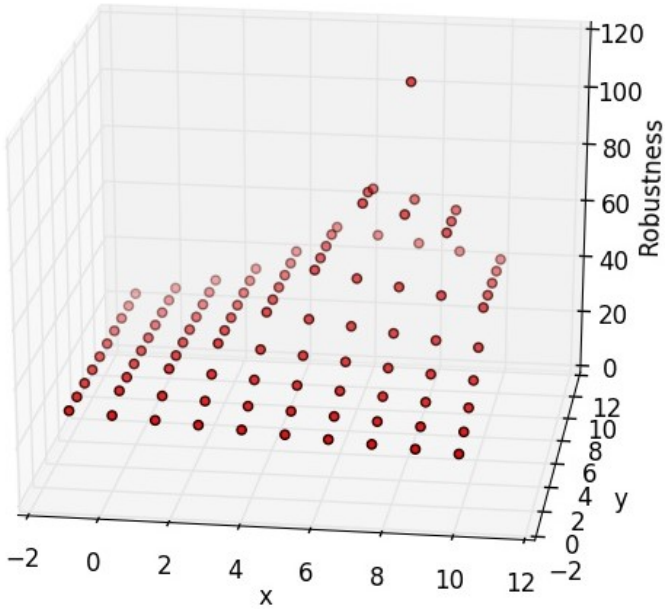

Fig. 2. Example of the robustness over a portion of satisfactory 2D signal space with a rectangle to be *EVENTUALLY* reached at $(6 \leq x \leq 9) \cap (6 \leq y \leq 9)$

In summary, pointwise robustness is defined here for all of the common STL operators in the unary or binary base cases. Once the predicates are combined, the robustness is easily calculated recursively. The only major differences from the traditional form is the *EVENTUALLY* operator which now rewards points for making quick progress towards the goal, rather than measuring the distance from the best point. It also encourages early departure times by no longer computing

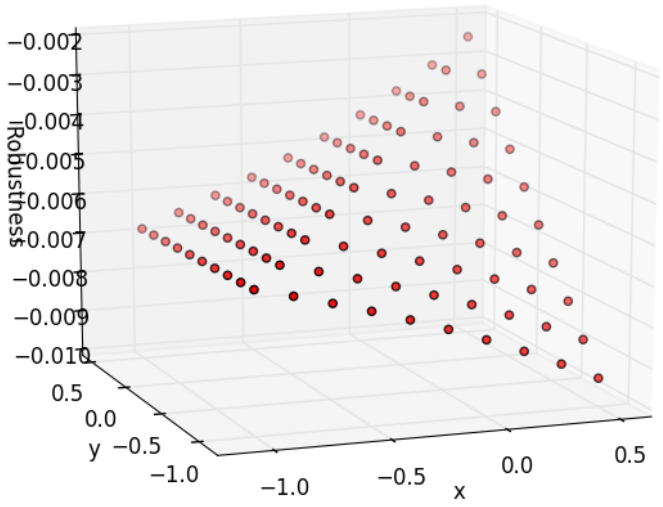

Fig. 3. Example of the robustness over an unsatisfactory portion of a 2D signal space with a rectangle to reached *EVENTUALLY* at $(1 \leq x \leq 2) \cap (1 \leq y \leq 2)$

the robustness after arrival in the same way as the traditional definition.

B. Metaheuristic Optimization

With a definition of pointwise robustness ready, I next looked into previous work in global metaheurisitic optimization techniques. I found that they were all formulated as minimization of a generic cost function. While it is easy to use the pointwise robustness to evaluate the entire trajectory (to a single robustness value), this would have defeated the purpose of developing it. Additionally it would have never been any faster than the MILP approach which makes use of the details provided by the predicates. Thinking that the optimal trajectory would be equivalent to having the highest robustness for each point, I planned on running particle swarm, simulated annealing, and differential evolution in parallel. That is to say optimizing each point individually and then combining the results.

I did also come up with two fundamentally pointwise algorithms. The simpler one is worst-point-first. In this approach the best candidate of a pool of randomly generated candidates is chosen. Then its robustness trajectory is computed. The algorithm then finds the point with the lowest robustness (except for the first point which is fixed-given). It then nudges that point in a random direction. By comparing the robustness at that point to the original, a better point can be found. As the algorithm progresses the worst point will change until the trajectory is at a local optimum of robustness.

The more complicated one is based on Genetic Evolution. The first step in adapting this algorithm to trajectory optimization based on STL constraints was to determine the "genes" and "alleles". I chose each point as gene with an infinite number of alleles (positions). The next step was to determine the mutation and cross over functions. Specifically, I made the mutation rate a function of the robustness, such that worst points have a wider variance in the children trajectories. Conversely good points are less likely to be adjusted. Similarly, the cross-over function presents the opportunity to pass off partial trajectories that are satisfactory to the children; hence, the points with the best robustness are more likely to be passed on to the children. Lastly, parents are picked based on their overall robustness such that only the best trajectories in a population reproduce. In this way the ideas behind the algorithm originally designed for blackbox (single valued) cost function optimization can take advantage of the information provided by the pointwise robustness.

C. Preliminary Implementation

With the theoretical groundwork in place, I wanted to make sure that everything would work before trying to accelerate performance on a practical system. So I implemented the semantics of pointwise STL in a custom python class. For easier use later on, I decided to make each predicate an individual object. Then I linked them together in a binary tree by overwriting the * and + operators, where I mapped conjunction to * and disjunction to +. To allow for experiments with smooth min/max, I encoded the log-sum (1) as well.

V. Preliminary Results

I first wanted to make sure that I could roughly replicate the general results of previous work with pointwise robustness, by finding the optimal trajectory using the overall robustness (after combining each point using the conjunction [min] operator). I tried using a generic gradient descent based minimization function from the scipy Python library. I found that it worked reasonably well. The cost function that I used penalized the distance between points to mimic speed constraints and rewarded larger overall robustness. Unfortunately the search space is highly non-convex meaning the optimizer often converged well before a satisfactory solution was found i.e. it violated one or more predicate. To help make sure that the optimizer produced solutions with satisfactory, positive robustness, I changed my cost function to a reward function whose value is simply the overall robustness. This change improved the robustness of the solutions, but did not seem to help the non-convexity of the problem much. Hence the quality of the solution depended heavily on the initial guess. I tried using an all zero initial guess which happened to be a local optimum in my test scenario, and never converged to anything useful. A completely random trajectory worked, but had inconsistent results and took a long time to converge. I found that a random walk with a random step size worked better yielding more consistent results and faster convergence. I credit this to how a random walk more closely mimics a trajectory than linking randomly chosen points.

Confident that optimization using the overall pointwise robustness would yield similar global optima to the traditional robustness, and with a decent method for generating an initial guess, I investigated optimizing each point individually using a differential evolution based optimizer. This worked when I considered a scenario of only memoryless constraints, that is

to say predicates whose robustness does not depend on the position of previous points. Examples of memoryless predicates include generic *ALWAYS* and *EVENTUALLY* restrictions on position. When I added a speed constraint improving each point led to a diverging overall solution that violated many predicates. I found that is a fundamentally theoretical flaw that is easy to show using the smooth min/max approximation (1). If the goal is to maximize:

$$\frac{1}{k}log(e^{k\alpha} + e^{k\beta})$$

It would be logical to try to increase α and β as much as possible. This is why the memoryless optimization problem worked. When dynamic constraints, such as speed limits are applied it adds a negative relationship between α and β. Now increasing one decreases the other and vice-versa. Hence purely pointwise optimization is not possible for situations with dynamic constraints. I tried to salvage this approach by only accepting improvements to a particular point if they also improved the overall robustness. While this stopped the divergence, convergence was very slow. Additionally it breaks the parallelism of the problem. As dynamic constraints are present in every practical system, I stopped pursing purely parallel based heuristics.

To see if my other ideas would work, I tried implementing the worst-point-first heuristic. I found that it often converges quickly to local optima. I credit the quick convergence to the simplicity of each iteration which is only a slight adjustment of one point. This suggests that a random restart upon mediocre convergence would be effective. In any case an example of a successful optimization run is shown in Fig. 4 below.

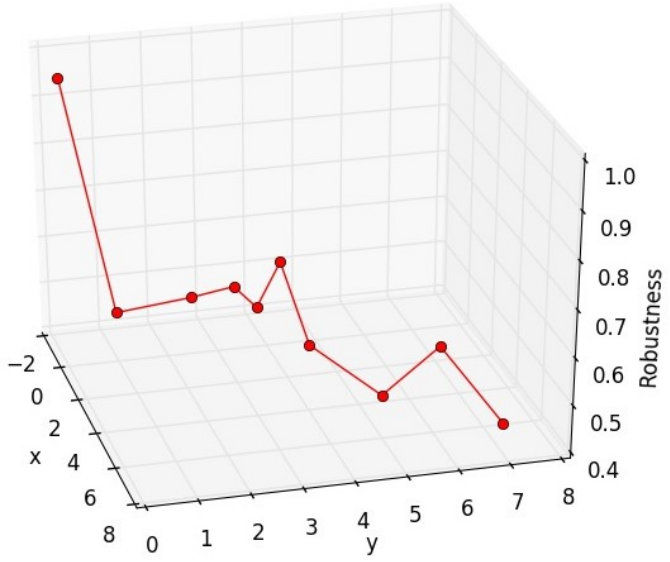

Fig. 4. Example Worst Point First Optimized Result for the test scenario: an obstacle to always avoided at $(1 \leq x \leq 3) \cap (1 \leq y \leq 3)$ and a goal to be reached at $(6 \leq x \leq 9) \cap (6 \leq y \leq 9)$ while never going above a speed of 2 units/sample. The pointwise robustness for this trajectory is: 0.435.

With the success of worst-point-first, I am confident in my genetic evolution heurisitic. Though it does use some of the ideas behind optimizing each point, it picks parents based on the overall trajectory robustness which has already been shown to work. Additionally the magnitude of the changes introduced by the mutation function can be small to similarly converge towards the optimal solution.

In looking for new approaches, I was introduced the work on trajectory optimization in optimal control such as the approaches explained in [5]. These approaches in discrete-time generally minimize an additive cost function of the form:

$$J = \sum_{i=1}^{N} L(x_i, u_i) \tag{9}$$

Where N is the number of time steps in the finite horizon and x_i and u_i are the current position and control inputs respectively. This has not been applied to STL based trajectory optimization, because optimizing over the traditional robustness is not additive.

With pointwise robustness however, these approaches appear viable. Thinking back to the smooth approximation of the min/max function (1), and noticing that $log(x)$ is convex, I realized that:

$$argmax(log(e^{k\alpha} + e^{k\beta}...)) = argmax(e^{k\alpha} + e^{k\beta}...) \tag{10}$$

Which led me to try the following additive, cost function:

$$J = \sum_{i=1}^{N} e^{-kp_n} \tag{11}$$

Where k an adjustable parameter as in (1) and p_n is the robustness of a particular point in time n. The negative in the exponent is so that this can be used for minimization. There is no control input dependence for simplicity. Equivalently, this independence can be viewed as an assumption that the control input is taken care of by an appropriate controller, and the optimizer is only generating the reference input. For simplicity of this viability test, I tried single shooting, which is to use a generic optimizer of choice on the cost function, (11) in my case. I used my black-box differential evolution optimizer. Though slow, it converged nonlinearly. An example of the result is shown in Fig. 5. Clearly this method produces smoother trajectories than the worst-point-first approach, but it takes significantly longer. Perhaps using the worst-point-first algorithm to improve upon initial guesses before starting single-shooting would be helpful.

The results of this semester are summarized in Table II on the next page.

VI. Plan for Spring Semester

Given the progress from this semester, for next semester I plan on continue where this semester is leading, that is say continuing to develop new algorithms for this trajectory optimization based on STL constraints. Specifically, I would like to write an efficient worst-point-first algorithm with a random restart upon convergence to non-robust solution. Similarly I want to implement and verify the performance of the custom genetic evolution algorithm. Next, given the

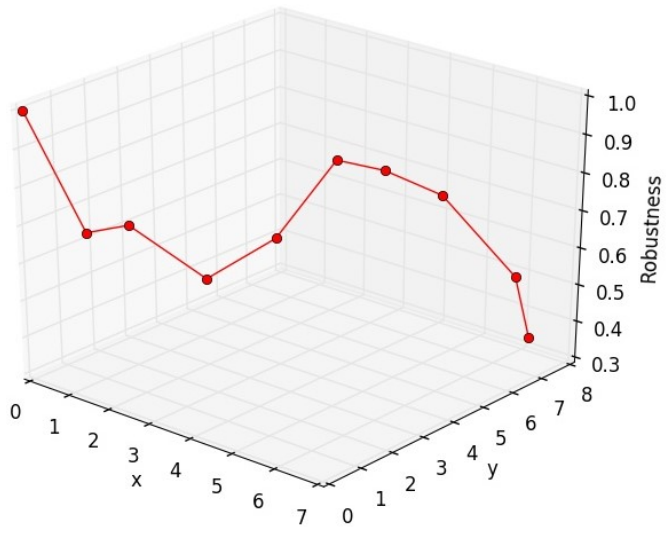

Fig. 5. Example of Single Shooting Optimization of the additive, non-negative cost function (11) after 3000 iterations for the test scenario: an obstacle to always avoided at $(1 \leq x \leq 3) \cap (1 \leq y \leq 3)$ and a goal to be reached at $(6 \leq x \leq 9) \cap (6 \leq y \leq 9)$ while never going faster than 2 units/sample

TABLE II
FALL SEMESTER EFFORTS AND RESULTS

Effort	**Result**
Develop Pointwise Robustness Definition	Successful and appears to behave similarly to the traditional robustness
Replicate optimal solutions using the pointwise robustness of the overall trajectory	Successful though this robustness function is non-smooth and highly non-convex.
Optimize each point individually and combine results	Successful for memoryless case; theoretically impossible for cases with dynamic constraints.
Custom Heuristics	Worst-Point-First: successful though it gets stuck in local optima due to complex cost function non-convexity. Genetic Evolution: Not yet implemented but appears promising and does not possess the same flaw as the purely parallel attempt.
Application of optimal control trajectory optimization approaches	Successful for the single shooting method, though convergence is non-monotonic.

success with single shooting, I want to try direct collocation using a 3rd party solver, as they include the ability to handle constraints and include many speed enhancements beyond the scope of the time I have left on this project. Once I have these approaches working, I would like to compare them to the existing approaches, MILP, exp/log sum and arithmetic geometric mean. By the end of the semester I will implement the most promising approach on a Turtlebot.

To accomplish this in the spring semester I propose the schedule in Table III.

TABLE III
SPRING SEMESTER PROPOSED SCHEDULE

Month	**Activity**
January	Implement and verify convergence of genetic evolution algorithm.
February	Implement and verify direct collocation approach
March	Implement and Compare MILP, Log/Exp, and Arithmetic-Geometric approaches
April-May	Run most promising algorithm on a Turtlebot and demonstrate results

ACKNOWLEDGMENT

Thank you Dr. Lin and Vince Kurtz for their guidance in selecting this project and helpful knowledge of related research.

REFERENCES

[1] C. Belta, S. Sadraddini, "Formal Methods for Control Synthesis: An Optimization Perspective", In Annual Review of Control, Robotics, and Autonomous Systems, pp115-40, 2019.

[2] V. Raman, M. Maasoumy, A. Donzé, R. M. Murray, A. Sangiovanni-Vincentelli, and S. A. Seshia. Model predictive control with Signal Temporal Logic Specifications. In Proc. of the IEEE Conf. on Decision and Control, 2014.

[3] Y. V. Pant, H. Abbas, R. Mangharam, "Smooth operator: Control using the smooth robustness of temporal logic", IEEE Conference on Control Technology and Applications, 2017.

[4] N. Mehdipour, C Vasile, and C. Bela, "Arithmetic-Geometric Mean Robustness for Control from Signal Temporal Logic Specifications", In Proc. of American Control Conf. (ACC), 2019.

[5] M. Diehl, H. G. Bock, H. Diedam, P.-B. Wieber, "Fast Direct Multiple Shooting Algorithms for Optimal Robot Control". Fast Motions in Biomechanics and Robotics, 2005, Heidelberg, Germany.