TECHNICAL UNIVERSITY OF MUNICH

SCHOOL OF ENGINEERING AND DESIGN

Semester Thesis in Mechatronics and Robotics

Follow the Rules: Efficient Optimization-based Motion Planning for Autonomous Vehicles

Mo Chen

TECHNICAL UNIVERSITY OF MUNICH

SCHOOL OF ENGINEERING AND DESIGN

Semester Thesis in Mechatronics and Robotics

Follow the Rules: Efficient Optimization-based Motion Planning for Autonomous Vehicles

Folge den Regeln: Effiziente Optimierungsbasierte Bewegungsplanung für Autonome Fahrzeuge

Author:	Mo Chen
Supervisor:	Prof. Dr.-Ing. habil. Knoll Alois Christian
Advisor:	Yuanfei Lin, M.Sc. and Patrick Halder, M.Sc.
Submission Date:	June 20, 2023

I confirm that this semester thesis is my own work and I have documented all sources and material used.

Ich versichere, dass ich diese Semester Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Munich, June 20, 2023 Mo Chen

Abstract

Real-time capability in autonomous driving systems poses a significant challenge, particularly when facing complex traffic rules. Traffic rules pose constraints for autonomous driving systems over time and STL is well suited to formalizing these constraints. At present, one of the state-of-the-art methods for trajectory planning involving signal temporal logic hinges on Mixed-Integer Programming, an NP-hard problem in the worst case. Consequently, the discovery of a new solving methodology becomes a necessity to ensure the real-time capacity of autonomous driving systems. This thesis uses the new method of successive convexification, designed to reduce the solve time for trajectory planning. The successive convexification algorithm linearizes the original trajectory planning problem into several convex subproblems, promising a polynomial solve time. Although successive convexification has been applied to optimal control problems with simple signal temporal logic constraints, it hasn't been employed for trajectory planning in autonomous driving involving complex logic constraints. Therefore, we propose a hybrid encoding method for signal temporal logic constraints and use successive convexification on trajectory planning for autonomous vehicles. The performance of this solving method is evaluated based on scenarios from the stlpy and CommonRoad benchmark suits. Furthermore, we examine the time complexity of our successive convexification-based approach and compare it to the Mixed-Integer Programming solving method.

Contents

1. Introduction

This introductory chapter furnishes a comprehensive overview of this thesis, including the motivations, a review of related work, and our contributions.

1.1. Motivation

Autonomous driving, with its immense commercial potential, continues to be a captivating field of research [1]. Its realization hinges heavily on the system's real-time capabilities [2]. This intriguing area encompasses four critical processes: perception, decision-making, trajectory planning, and control [3].

Significant progress in deep learning, particularly in computer vision, has enhanced the ability of autonomous vehicles to comprehend their environment [4]. Consequently, numerous real-time deep learning architectures have emerged, bolstering the prospects for autonomous driving [5, 6]. Moreover, with advancements in feedback control systems, autonomous vehicles can now accurately track predefined trajectories and control their longitudinal motion [7]. However, the processes of decision-making and trajectory planning still pose significant challenges to the system's real-time capability.

In the realm of autonomous driving, trajectory planning plays a critical role. It necessitates creating a future route and schedule, accounting for obstacles, traffic regulations, and short-term goals [8]. To capture all constraints of obstacles and traffic rules, temporal logic can be used, which is a formal language developed to specify the logic constraints over time and temporal relationships [9]. In the context of traffic rules, these temporal relationships are frequently employed. Nonetheless, our concern extends beyond mere satisfaction of traffic rules to the extent of their fulfillment. Signal Temporal Logic (STL) employs a robustness function to gauge how close a signal is to fulfilling a temporal logic specification [10].

In essence, trajectory planning in autonomous driving can be represented by a cost function and logical constraints. Consequently, we apply STL on logical constraints, and the problem can be transformed into an optimization problem with constraints governed by the robustness function of STL. Current solutions for STL-embedded optimization problems rely on Mixed-Integer Programming (MIP) [11]. However, MIP problems, characterized by exponential time complexity in relation to the number of integer variables, challenge the real-time capabilities required for autonomous driving. Therefore, real-time solutions for STL-based optimization problems must be explored in the field of autonomous driving.

1.2. Related work

This section provides an overview of STL, solving methods for trajectory planning problems based on STL, successive convexification for trajectory planning, and trajectory planning benchmarks.

1.2.1. Signal Temporal Logic

Temporal logic has been instrumental in representing logical relations with time such as 'before', 'after', and 'until' [9]. Within the context of traffic rules, temporal relations are widely applied. Temporal logics like Linear Temporal Logic (LTL) [12, 13] and Metric Temporal Logic (MTL) [14] provide Boolean values to indicate whether traffic rules are satisfied or violated. However, our concern extends beyond the mere satisfaction of traffic rules; we also aim to quantify the degree of fulfillment. STL aptly meets this need by employing a robustness function to evaluate the proximity of a signal to satisfying a temporal logic specification [10].

1.2.2. Solving Methods for Trajectory Planning Problem Based on STL

There are several methods to solve trajectory planning problems based on STL. The most commonly employed method is based on MIP [15]. However, MIP brings the exponential time complexity associated with the number of binary variables, leading to increased computational overheads.

Kurtz et al. [16] have made strides in this area by developing a new mixed-integer encoding method for STL specifications, which is called SOS1 encoding. This method substantially reduces the number of binary variables and constraints required to represent an STL specification, even reducing it to a logarithmic number in certain cases. Although this work signifies a more efficient approach to trajectory planning in autonomous driving, the SOS1 encoding method still needs the binary variables, which means the optimization problem with the SOS1 encoding method is still an NP-hard problem.

In addition, control barrier functions can be introduced by the STL specifications [17]. The framework based on control barrier functions can provide a control law within polynomial time. However, it requires a strong assumption on the convexity of constraints, which is hard to be guaranteed in autonomous driving scenarios [18].

Another method encodes STL specifications into computational graphs [19], facilitating the use of gradient-based methods. However, the gradient-based methods are applied only to objective functions and require constraints to be in the convex form [20]. Consequently, this approach may not adequately handle non-convex constraints.

In summary, the currently existing methods are either NP-hard or lack the capabilities to deal with non-convex constraints.

1.2.3. Successive Convexification for Trajectory Planning

Trajectory planning is commonly characterized as an optimal control problem [21], and nonlinear programming is the prevalent method for addressing such problems [22]. However, given the non-convex nature of most nonlinear constraints, this method can demand lots of computational resources. As such, the linearization and convexification of the optimal control problem have gained considerable attention in recent years [23].

Successive convexification (SCvx) is a class of iterative algorithms designed to tackle non-convex constrained optimal control problems featuring nonlinear dynamics [24]. SCvx operates by iteratively resolving a series of convex subproblems approximating the original issue until a solution is identified. This algorithm family has certain advantages over other methods, such as assurances of global convergence and superlinear convergence rates. Furthermore, SCvx can manage nonlinear dynamics and non-convex constraints [25], elements frequently encountered in autonomous driving. SCvx has seen successful application in rocket trajectory planning [26].

As introduced before, we use STL specifications to represent complex traffic rules. To utilize SCvx for trajectory planning, the STL specifications need to be encoded first. There is a method to convert STL into additional sub-dynamics and then linearize them [27]. However, this work only focuses on an optimal control problem with a simple STL specification.

1.2.4. Trajectory Planning Benchmarks

Numerous benchmarks are available for trajectory planning. One example is stlpy [16], which offers a broad range of trajectory planning benchmarks. In these scenarios, the agent is modeled as a point on a two-dimensional plane, simplifying the complexity of real-world conditions. Despite their simplification, these scenarios serve as effective tools for assessing algorithm performance.

Another benchmark is CommonRoad [28], which encompasses a blend of recorded and constructed scenarios covering highways, rural roads, and urban settings. The scenarios from CommonRoad closely resemble real-world conditions, making it a more practical benchmark. In CommonRoad, convex optimization was implemented for set-based reachability analysis [29], and this method was further generalized to allow integration with any existing optimization-based algorithms [30].

1.3. Contributions

The current state-of-the-art method for resolving trajectory planning problems based on STL specifications relies on MIP. However, due to its exponential computation time with the increasing of the problem's scale, MIP is unsuitable for real-time autonomous driving systems. Therefore, this thesis aims to identify an efficient algorithm capable of solving trajectory planning problems based on STL specifications. Previous work

has demonstrated the potential of SCvx in addressing simple optimal control problems under STL specifications. In this thesis, we:

1. introduce a novel STL encoding method that enhances the compatibility of the trajectory planning problem, based on STL specifications, with the SCvx algorithm;
2. apply the SCvx algorithm to trajectory planning problems with more complex STL specifications; and
3. conduct an analysis of the time complexity between the SCvx method and the MIP method.

2. Preliminaries

In this chapter, we will delve into the foundational concepts that underpin our approach. Initially, we explore the nature of the optimal control problem. Subsequently, we shed light on the STL from the standpoint of constraints and robustness. We then discuss a newly proposed optimization method, SCvx. Lastly, we will explain the dynamic models pertinent to our thesis.

Let us introduce $\square$ as the placeholder for a variable. In the context of a 2-dimensional plane, $\square_x$ and $\square_y$ denote the variable in the x and y direction, respectively. The notations $\square^L$ and $\square_{min}$ represent the lower boundary and the minimum possible value, and $\square^U$ and $\square_{max}$ symbolize the upper boundary and the maximum possible value of the variable. If $\square$ is a real number, the floor function, symbolized as $\lfloor\square\rfloor$, signifies the largest integer not exceeding $\square$. Conversely, the ceiling function, expressed as $\lceil\square\rceil$, corresponds to the smallest integer that is not less than $\square$. In addition, $\mathcal{O}$ signifies the Big-O notation, often referred to as Landau's symbol, which is utilized to characterize algorithmic complexity. Moreover, $\mathbb{R}$ denotes the set of real numbers.

2.1. Optimal Control Problem

In the field of control engineering, we usually need to control a dynamic system:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}), \tag{2.1}$$

where $\boldsymbol{x}$ is the system state, $\boldsymbol{u}$ is the control input and $\boldsymbol{f}$ is the system dynamic equation. In practice, we often do not require an exact solution for the control inputs function $\boldsymbol{u}$. Instead, we can discretize the problem into N time intervals [22], which means a time horizon of N and a total of $N+1$ moments, as Fig. 2.1 shows. Then the discretized dynamic system is:

$$\boldsymbol{x}_{k+1} = \tilde{\boldsymbol{f}}(\boldsymbol{x}_k, \boldsymbol{u}_k),\ k = 0, 1, 2, ..., N-1, \tag{2.2}$$

where k represents the index of the k-th time interval, $\tilde{\boldsymbol{f}}$ is the discretized system dynamic equation, and $\boldsymbol{x}_k, \boldsymbol{u}_k$ correspond to the system state and the control input of start time on the k-th time interval, respectively.

In order to represent boundaries of the system state and the control input in practice, we introduce the acceptable set of the system states X_k and the acceptable set of the control inputs U_k for every moment:

$$\boldsymbol{x_k} \in X_k, \boldsymbol{u_k} \in U_k,\ k = 0, 1, 2, ..., N. \tag{2.3}$$

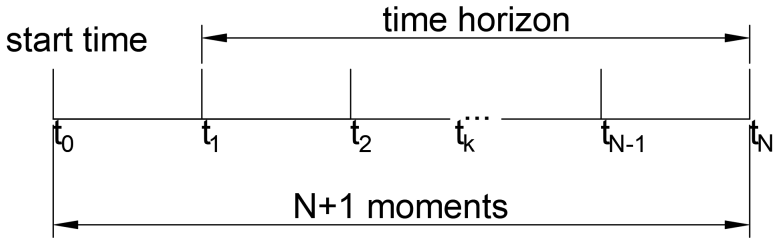


Figure 2.1.: Time horizon and moments.

In addition to the acceptable set, states and control inputs can also be constrained by some inequations. Due to the complexity of the inequations function, it is hard to represent those inequations in the acceptable set of the system state and the control input. We can write those inequations in this form:

$$\boldsymbol{s}(\boldsymbol{x_k}, \boldsymbol{u_k}) \geq 0,\ k = 0, 1, 2, ..., N, \tag{2.4}$$

where $\boldsymbol{s}$ represents those inequations functions.

There may also be equations that impose constraints on the system states and control inputs, such as fixed initial states $\boldsymbol{x}_0$, sliding mode [31], and so on. In general, we can express those constraints as:

$$\boldsymbol{q}(\boldsymbol{x}_k, \boldsymbol{u}_k) = 0,\ k = 0, 1, 2, ..., N, \tag{2.5}$$

where $\boldsymbol{q}$ denotes those equations functions.

When discussing the control of a dynamic system described by (2.1), we need to define what constitutes an optimal control input $\boldsymbol{u}$. This is achieved by defining a cost function $\mathcal{C}(\boldsymbol{x}, \boldsymbol{u})$. Thus, finding the best control inputs can be formulated as follows:

$$\min_{\boldsymbol{u}} \mathcal{C}(\boldsymbol{x}, \boldsymbol{u}). \tag{2.6}$$

If we write (2.1)-(2.6) together, we can get the optimal control problem with discretized time [32]:

$$\begin{aligned}
&\min_{\boldsymbol{u}} \mathcal{C}(\boldsymbol{x}, \boldsymbol{u}) \\
&\text{s.t. } \boldsymbol{x}_{k+1} = \tilde{\boldsymbol{f}}(\boldsymbol{x}_k, \boldsymbol{u}_k),\ k = 0, 1, 2, ..., N-1, \\
&\boldsymbol{x}_k \in X_k, \boldsymbol{u}_k \in U_k,\ k = 0, 1, 2, ..., N, \\
&\boldsymbol{q}(\boldsymbol{x}_k, \boldsymbol{u}_k) = 0,\ k = 0, 1, 2, ..., N, \\
&\boldsymbol{s}(\boldsymbol{x}_k, \boldsymbol{u}_k) \geq 0,\ k = 0, 1, 2, ..., N.
\end{aligned} \tag{2.7}$$

2.1.1. Discretization Method for dynamic systems

We can linearize the dynamic system equation (2.1) by [33]:

$$\dot{\boldsymbol{x}} = A\boldsymbol{x} + B\boldsymbol{u} + \boldsymbol{z}, \tag{2.8a}$$

$$A = \frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})|_{(\hat{\boldsymbol{x}}, \hat{\boldsymbol{u}})}, \tag{2.8b}$$

$$B = \frac{\partial}{\partial \boldsymbol{u}} \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})|_{(\hat{\boldsymbol{x}}, \hat{\boldsymbol{u}})}, \tag{2.8c}$$

$$\boldsymbol{z} = -A\hat{\boldsymbol{x}} - B\hat{\boldsymbol{u}} + \boldsymbol{f}(\hat{\boldsymbol{x}}, \hat{\boldsymbol{u}}), \tag{2.8d}$$

where $\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{u}}$ are the reference state and the control input, respectively.

Then, we can discretize the model. We assume the time interval is $[0, T]$ and divide it into $N + 1$ evenly distributed discretion points. We define the start time t on the k-th time interval as:

$$t_k = \frac{k}{N} T, k = 0, 1, 2, ..., N. \tag{2.9}$$

We assume the control input $\boldsymbol{u}$ changes linear from $\boldsymbol{u}_k$ to $\boldsymbol{u}_{k+1}$ over each time step $[t_k, t_{k+1}]$. Thus, over the interval $[t_k, t_{k+1}]$, we can express $\boldsymbol{u}$ as follows [26]:

$$\boldsymbol{u} = \alpha_k(t)\boldsymbol{u}_k + \beta_k(t)\boldsymbol{u}_{k+1}, t \in [t_k, t_{k+1}], k = 0, 1, 2, ..., N-1, \tag{2.10a}$$

$$\alpha_k(t) = \frac{t_{k+1} - t}{t_{k+1} - t_k}, \tag{2.10b}$$

$$\beta_k(t) = \frac{t - t_k}{t_{k+1} - t_k}. \tag{2.10c}$$

We use $\Phi_A(t_{k+1}, t_k)$ to denote the state transition matrix that describes the change from $\boldsymbol{x}_k$ to $\boldsymbol{x}_{k+1}$ without control inputs. The state transition matrix is governed by the following differential equation [34]:

$$\frac{d}{dt}\Phi_A(t, t_k) = A(t)\Phi_A(t, t_k), \Phi_A(t_k, t_k) = I, k = 0, 1, 2, ..., N-1, \tag{2.11}$$

where I is the identity matrix.

Then, we can express the discrete-time dynamics that relate $\boldsymbol{x}_k$ to $\boldsymbol{x}_{k+1}$ as follows [26]:

$$\boldsymbol{x}_{k+1} = \tilde{A}_k\boldsymbol{x}_k + \tilde{B}_k\boldsymbol{u}_k + \tilde{C}_k\boldsymbol{u}_{k+1} + \tilde{\boldsymbol{z}}_k, k = 0, 1, 2, ..., N-1 \tag{2.12a}$$

$$\tilde{A}_k = \Phi_A(t_{k+1}, t_k) \tag{2.12b}$$

$$\tilde{B}_k = \int_{t_k}^{t_{k+1}} \Phi_A(t_{k+1}, \tau) B(\tau) \alpha_k(\tau) d\tau \tag{2.12c}$$

$$\tilde{C}_k = \int_{t_k}^{t_{k+1}} \Phi_A(t_{k+1}, \tau) B(\tau) \beta_k(\tau) d\tau \tag{2.12d}$$

$$\tilde{\boldsymbol{z}}_k = \int_{t_k}^{t_{k+1}} \Phi_A(t_{k+1}, \tau) \boldsymbol{z}(\tau) d\tau \tag{2.12e}$$

2.2. Signal Temporal Logic

STL formulas are described using the grammar [27]:

$$\varphi ::= \pi^{\mu} | \neg\varphi | \varphi_1 \wedge \varphi_2 | \varphi_1 \vee \varphi_2 | F_{[a,b]}\varphi | G_{[a,b]}\varphi | \varphi_1 U_{[a,b]} \varphi_2,$$

where π^{μ} is a Boolean predicate whose truth value is determined by the sign of a real-valued function μ. We also call π^{μ} atomic predicate [35]. $\varphi, \varphi_1, \varphi_2$ are STL formulas. $\neg$, $\wedge$, and $\vee$ correspond to the logical operators *negation*, *conjunction* and *disjunction*, respectively. We can also call them logic operators. $F_{[a,b]}$, $G_{[a,b]}$, and $U_{[a,b]}$ denote the *eventually*, *always* and *until* modalities, respectively, with a restriction to a given time frame $[a, b]$ with $a \leq b$. Because $F_{[a,b]}$, $G_{[a,b]}$, and $U_{[a,b]}$ are related to the time interval, we call them temporal operators.

An simulation run $\xi(\boldsymbol{x}, \boldsymbol{u})$ satisfies an STL formula φ is denoted by $\xi \models \varphi$. To check whether a simulation run satisfies an STL formula, a robustness function ρ is used such that $\xi \models \varphi$ at time t if and only if $\rho^{\varphi}(\xi, t) \geq 0$. The sign of the robustness score indicates if the specifications are satisfied or violated, whereas the absolute value indicates how strongly this happens.

The robustness function can also be defined recursively as follows [27]:

$$\rho^{\pi^{\mu}}(\xi, t_k) = \mu(\boldsymbol{x}_k, \boldsymbol{u}_k),$$

$$\rho^{\neg\varphi}(\xi, t_k) = -\rho^{\varphi}(\xi, t_k),$$

$$\rho^{\varphi_1 \vee \varphi_2}(\xi, t_k) = \min(\rho^{\varphi_1}(\xi, t_k), \rho^{\varphi_2}(\xi, t_k)),$$

$$\rho^{\varphi_1 \wedge \varphi_2}(\xi, t_k) = \max(\rho^{\varphi_1}(\xi, t_k), \rho^{\varphi_2}(\xi, t_k)),$$

$$\rho^{F_{[a,b]}\varphi}(\xi, t_k) = \max_{t_{k'} \in [t_k + a, t_k + b]} \rho^{\varphi}(\xi, t_{k'}),$$

$$\rho^{\varphi_1 U_{[a,b]} \varphi_2}(\xi, t_k) = \max_{t_{k'} \in [t_k + a, t_k + b]} \min(\rho^{\varphi_2}(\xi, t_{k'}), \min_{t_{k''} \in [t_k + a, t_{k'}]} \rho^{\varphi_1}(\xi, t_{k''})),$$

where t_k maintains its prior definition as the time at index k.

When considering an STL formula in optimal control problem (2.7), you can add the following constraints to (2.7):

$$\rho^{\varphi}(\xi(\boldsymbol{x}, \boldsymbol{u}), t_k) \geq 0,\ k = 0, 1, 2, ..., N. \tag{2.13}$$

2.2.1. Tree representations for STL

The STL tree is a hierarchical representation of an STL formula as a tree structure. It organizes the subformulas of the STL formula according to their logical relationships, making it easier to manipulate and encode them based on specific requirements. Moreover, the STL tree enables efficient storage of the corresponding STL formula within a computer system.

In an STL tree, there are following components:

- Nodes: Each node represents a part of the STL formula, such as a predicate, a conjunction, or a disjunction. The node may also represent a temporal operator (e.g., $F_{[a,b]}$, $G_{[a,b]}$, and $U_{[a,b]}$) applied to a subformula.
- Edges: Edges serve as the link between a parent node and its child nodes. These child nodes are also referred to as subformulas.
- Root: The root of the tree represents the overall STL formula.
- Leaf Nodes: The leaf nodes represent the atomic predicates of the STL formula.

For example, let's consider an STL formula: $\varphi = (\phi_1 \vee \phi_2) \vee \phi_3$. This formula can be represented as an STL tree, as Fig. 2.2(a) shows. The STL tree allows for easy manipulation of the subformulas based on encoding requirements. Because the logical operators inside and outside the bracket are the same, we can remove the bracket. From the perspective of STL tree, we move the child nodes of $(\phi_1 \vee \phi_2)$ upwards and delete the redundant middle node $(\phi_1 \vee \phi_2)$. The result of the manipulation for subformulas is illustrated in Figure 2.2(b).

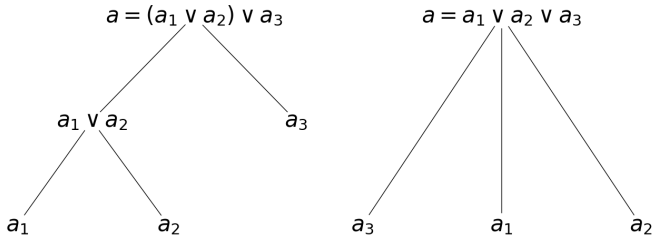


Figure 2.2.: Example of moving node on an STL tree. The logic operators of both parent node and child node are disjunctions. Thus, we can move the child node upwards.

2.2.2. Approximation for max/min Function

We can see that the robustness function includes max/min functions due to logical operators $\wedge$ and $\vee$. In (2.7), the robustness function is a constraint. If we want to

solve (2.7), the gradient-based solvers will require differentiable constraints. Because the max/min function is not differentiable, we need to use appropriate functions to approximate it.

smin Function

When we just have two inputs, smin function is a good approximation for the min function. It uses a smooth interpolator to approximate the min function when two inputs are enough close [27].

$$\text{smin}(a,b,C) = \begin{cases} a, & \text{if } a-b \leq -C, \\ b, & \text{if } a-b \geq C, \\ a(1-h)+hb-Ch(1-h), & \text{if } a-b \in (-C,C), \end{cases} \tag{2.14}$$

where $h = \frac{1}{2} + \frac{a-b}{2C}$ and $C \in \mathbb{R}_{\geq 0}$ represents a parameter that controls the smoothness of the smin function.

smax Function

smax function takes the same method as smin function to approximate the max function

$$\text{smax}(a,b,C) = \begin{cases} a, & \text{if } a-b \geq C, \\ b, & \text{if } a-b \leq -C, \\ a(1-h)+hb+Ch(1-h), & \text{if } a-b \in (-C,C), \end{cases} \tag{2.15}$$

where $h = \frac{1}{2} + \frac{a-b}{2C}$ and $C \in \mathbb{R}_{\geq 0}$. smax function is the oppsite of smin function: $\text{smax}(a,b,C) = -\text{smin}(-a,-b,C)$.

Under-approximation for min Function

If we want to take min for multiple inputs $\boldsymbol{a} = [a_1, a_2, ..., a_m]^\top$, the log-sum-exponential approximation can be used [36]:

$$\widetilde{\min}(\boldsymbol{a}, C) = -\frac{1}{C}\log(\sum_{i=1}^{m} e^{-Ca_i}), \tag{2.16}$$

where $C \in \mathbb{R}_{\geq 0}$ represents a parameter that controls the accuracy of the log-sum-exponential approximation.

An associated error bound is given by:

$$\min(\boldsymbol{a}) - \widetilde{\min}(\boldsymbol{a}, C) \leq \frac{\log(m)}{C}. \tag{2.17}$$

Under-approximation for max Function

If we want to take max for multiple inputs $\boldsymbol{a} = [a_1, a_2, ..., a_m]^\top$, the following approximation function can be used [36]:

$$\widetilde{\max}(\boldsymbol{a}, C) = \frac{\sum_{i=1}^{m} a_i e^{Ca_i}}{\sum_{i=1}^{m} e^{Ca_i}}, \tag{2.18}$$

where $C \in \mathbb{R}_{\geq 0}$ is same as before.

We assume the inputs are sorted from largest to smallest. An associated error bound is given by:

$$\max(\boldsymbol{a}) - \widetilde{\max}(\boldsymbol{a}, C) \leq \frac{a_1 - a_m}{\frac{e^{C(a_1 - a_2)}}{m-1} + 1}. \tag{2.19}$$

2.3. Successive Convexification

Successive Convexification (SCvx) is a technique that linearizes the optimal control problem depicted in form (2.7). It leverages a sequence of solutions from the linearized problem to approximate the original problem's solution. The linearization process might result in the loss of some non-convex feasible sets for states and control inputs, potentially leading to an absence of feasible solutions. To ensure a feasible solution for the linearized problem, the introduction of a penalty function $\mathcal{P}$ is often required [25]. This function is designed to penalize the objective function when constraints are violated, with greater violations incurring larger penalties. The degree of penalty is regulated by a penalty weight $\lambda_i > 0$, which should be sufficiently large. Consequently, the cost function $\mathcal{L}$ of the linearized problem deviates from the original problem's cost function (2.7). The linearized problem sequence is defined as follows [25]:

Convex Optimal Control Subproblem

$$\begin{aligned}
&\min_{\boldsymbol{d},\boldsymbol{w},\boldsymbol{v},\boldsymbol{\iota}} \ \mathcal{L}(\boldsymbol{d}, \boldsymbol{w}, \boldsymbol{v}, \boldsymbol{\iota}) = \mathcal{C}(\boldsymbol{x}^{(i)} + \boldsymbol{d}, \boldsymbol{u}_k^{(i)} + \boldsymbol{w}) + \sum_{k=1}^{N-1} \lambda_i \mathcal{P}(\boldsymbol{v}_k, \boldsymbol{\iota}_k), \\
&\text{s.t. } \boldsymbol{x}_{k+1}^{(i)} + \boldsymbol{d}_{k+1} = \boldsymbol{f}(\boldsymbol{x}_k^{(i)}, \boldsymbol{u}_k^{(i)}) + A_k^{(i)} \boldsymbol{d}_k + B_k^{(i)} \boldsymbol{w}_k + \boldsymbol{v}_k, \ k = 0, 1, 2, ..., N-1, \\
&\boldsymbol{s}(\boldsymbol{x}_k^{(i)}, \boldsymbol{u}_k^{(i)}) + S_k^{(i)} \boldsymbol{d}_k + Q_k^{(i)} \boldsymbol{w}_k \geq \boldsymbol{\iota}_k, \ k = 0, 1, 2, ..., N, \\
&\boldsymbol{q}(\boldsymbol{x}_k^{(i)} + \boldsymbol{d}_k, \boldsymbol{u}_k^{(i)} + \boldsymbol{w}_k) = 0, \ k = 0, 1, 2, ..., N, \\
&\boldsymbol{x}_k^{(i)} + \boldsymbol{d}_k \in X, \boldsymbol{u}_k^{(i)} + \boldsymbol{w}_k \in U, ||\boldsymbol{d}|| + ||\boldsymbol{w}|| \leq r^{(i)}, \ k = 0, 1, 2, ..., N,
\end{aligned} \tag{2.20}$$

where superscript (i) refers to i-th iteration, $\boldsymbol{d} = \boldsymbol{x} - \boldsymbol{x}^{(i)}, \boldsymbol{d}_k = \boldsymbol{x}_k - \boldsymbol{x}_k^{(i)}, \boldsymbol{w} = \boldsymbol{u} - \boldsymbol{u}^{(i)}, \boldsymbol{w}_k = \boldsymbol{u}_k - \boldsymbol{u}_k^{(i)}$ are the differences between the solution and the current iteration. $\boldsymbol{v}_k$ is an unconstrained *virtual control* term which is used to avoid obtaining an infeasible linearized convex subproblem. $\boldsymbol{\iota}_k$ is non-positive and acts as a *relaxation term* that allows

the non-convex constraints to be violated. $A_k^{(i)}, B_k^{(i)}$ and $S_k^{(i)}$ are the respective partial derivatives, at the kth time:

$$A_k^{(i)} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}|_{(\boldsymbol{x}_k^{(i)}, \boldsymbol{u}_k^{(i)})},$$
$$B_k^{(i)} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}|_{(\boldsymbol{x}_k^{(i)}, \boldsymbol{u}_k^{(i)})},$$
$$S_k^{(i)} = \frac{\partial \boldsymbol{s}}{\partial \boldsymbol{x}}|_{(\boldsymbol{x}_k^{(i)}, \boldsymbol{u}_k^{(i)})},$$
$$Q_k^{(i)} = \frac{\partial \boldsymbol{s}}{\partial \boldsymbol{u}}|_{(\boldsymbol{x}_k^{(i)}, \boldsymbol{u}_k^{(i)})}.$$

An important concept in problem (2.20) is the trust region radius $r^{(i)}$. The trust region radius is used to restrict the maximum distance that the optimization algorithm can move away from the current linearization point in each iteration. This is done to ensure that the algorithm does not move too far away from the current linearization point and ensure the accuracy of the solution for every iteration. The trust region radius can be adjusted dynamically during the optimization process.

2.3.1. Successive Convexification Algorithm

After introducing the convex optimal control subproblem, we need to consider the accuracy of subproblem's solution relative to the solution of original problem (2.7). We can see that the *virtual control* $\boldsymbol{v}_k$ and *relaxation term* $\boldsymbol{\iota}_k$ only reveal the degree of violating the linearized constraints. When we discuss the the accuracy of subproblem's solution, we would like to consider the degree of violating the original nonlinear constraints. We can substitute the subproblem's solution $\boldsymbol{x}_k = \boldsymbol{x}_k^{(i)} + \boldsymbol{d}_k$ and $\boldsymbol{u}_k = \boldsymbol{u}_k^{(i)} + \boldsymbol{w}_k$ into the original nonlinear constraints (2.2) and (2.4):

$$\begin{aligned} &\hat{\boldsymbol{v}}_k = \boldsymbol{x}_{k+1} - \boldsymbol{f}(\boldsymbol{x}_k, \boldsymbol{u}_k),\ k = 0, 1, 2, ..., N-1, \\ &\boldsymbol{s}(\boldsymbol{x}_k, \boldsymbol{u}_k) \geq \hat{\boldsymbol{\iota}}_k,\ k = 0, 1, 2, ..., N, \end{aligned} \tag{2.21}$$

where $\hat{\boldsymbol{v}}_k$ and $\hat{\boldsymbol{\iota}}_k$ are *virtual control* and *relaxation term* for the original nonlinear constraints at start time on the k-th time interval, respectively.

We can substitute $\hat{\boldsymbol{v}}_k$ and $\hat{\boldsymbol{\iota}}_k$ into the cost function of subproblem (2.20). Then, we will obtain a different cost function which has no direct relation with the subproblem's solution $\boldsymbol{v}_k$ and $\boldsymbol{\iota}_k$. We call this new cost function nonlinear penalized cost which is:

$$\mathcal{J}(\boldsymbol{x}, \boldsymbol{u}) = \mathcal{C}(\boldsymbol{x}, \boldsymbol{u}) + \sum_{k=1}^{N-1} \lambda_i \mathcal{P}(\boldsymbol{x}_{k+1} - \boldsymbol{f}(\boldsymbol{x}_k, \boldsymbol{u}_k), \boldsymbol{s}(\boldsymbol{x}_k, \boldsymbol{u}_k)). \tag{2.22}$$

The most important part of the SCvx algorithm is to determine the trust region radius. In the SCvx algorithm, the trust region radius is updated by the optimal cost of the convex subproblem in every iteration. By using the nonlinear penalized cost and convex subproblem (2.20), we can construct the SCvx algorithm in Algorithm 1.

The SCvx algorithm begins with an initial trajectory and the convex subproblem as inputs, as indicated in the input of Algorithm 1. This process then finally generates an approximate solution, which can be found in the output of Algorithm 1. The process involves the initialization of parameters such as $r^{(1)} > 0$ and $\lambda > 0$, as indicated in line 1, followed by a sequence of computations and updates carried out until convergence, which can be found from line 2 to line 18. The algorithm dynamically adjusts the trust region radius according to specific criteria in order to ensure the effectiveness of the optimization process, as shown in (2.23). Each iteration includes solving the convex subproblem in line 3, calculating changes in both the nonlinear penalized cost and the cost function of the convex subproblem in line 4 and 5, computing the ratio between these changes in line 9, and updating the solution and trust region radius accordingly from line 11 to line 17. The final output of the SCvx algorithm is an approximate solution to the original problem (2.7), as indicated in line 7.

In the SCvx algorithm, there are some parameters which we need to set:

- **Initial trust region radius** $r^{(1)} > 0$: This parameter determines the size of the optimization space around the current estimate. A larger trust region radius could enable exploring a larger optimization space but may also increase computational cost and time.
- **Penalty weight** $\lambda > 0$: This is the weight for the penalty term in the cost function. A higher value of λ increases the penalty for violating constraints, thus forcing the solution to respect constraints more closely.
- **Minimum trust region radius** r_l: This parameter sets a lower bound on the trust region size. If the trust region size falls below this value during iterations, it is reset to this minimum value. This ensures that the optimization space does not become too narrow during the optimization process.
- **Scaling factors of trust region radius** α, β: These are the factors by which the trust region radius is expanded (β) or contracted (α). They should be greater than 1. A larger value of α or β leads to a more aggressive change in trust region size in response to the performance of the current solution.
- **Threshold factors of cost function** ψ_0, ψ_1, ψ_2: These parameters determine thresholds for the ratio of actual to predicted change in cost. They control when to accept or reject a step, and how to update the trust region radius based on the quality of the current solution.

Algorithm 1 SCvx Algorithm [25]

Input: An initial trajectory $(\boldsymbol{x}^{(1)}, \boldsymbol{u}^{(1)})$, the convex subproblem (2.20).
Output: Approximate solution $(\boldsymbol{x}^{(i+1)}, \boldsymbol{u}^{(i+1)})$.

Initialize trust region radius $r^{(1)} > 0$, penalty weight $\lambda > 0$ and parameters $r_l > 0, 0 \leq \psi_0 \leq \psi_1 \leq \psi_2 < 1, \alpha > 1, \beta > 1$ and $\epsilon > 0$.
while not converged, i.e. cost reduction **do**
    At $(i+1)$th succession, solve the convex subproblem, (2.20), at $(\boldsymbol{x}^{(i)}, \boldsymbol{u}^{(i)}, r^{(i)})$ to get an optimal solution $(\boldsymbol{d}, \boldsymbol{w}, \boldsymbol{\iota})$.
    Compute the "actual" change in the penalized cost: $\Delta\mathcal{J}(\boldsymbol{x}^{(i)}, \boldsymbol{u}^{(i)}) = \Delta\mathcal{J}^{(i)} \leftarrow \mathcal{J}(\boldsymbol{x}^{(i)}, \boldsymbol{u}^{(i)}) - \mathcal{J}(\boldsymbol{x}^{(i)} + \boldsymbol{d}, \boldsymbol{u}^{(i)} + \boldsymbol{w})$.
    Compute the "predicted" change by linear approximation: $\Delta\mathcal{L}(\boldsymbol{d}, \boldsymbol{w}, \boldsymbol{v}, \boldsymbol{\iota}) = \Delta\mathcal{L}^{(i)} \leftarrow \mathcal{J}(\boldsymbol{x}^{(i)}, \boldsymbol{u}^{(i)}) - \mathcal{L}(\boldsymbol{d}, \boldsymbol{w}, \boldsymbol{v}, \boldsymbol{\iota})$.
    if $\Delta\mathcal{L}^{(i)} < \epsilon$ **then**
        return $(\boldsymbol{x}^{(i)}, \boldsymbol{u}^{(i)})$.
    else
        Compute the ratio $\delta^{(i)} \leftarrow \Delta\mathcal{J}^{(i)} / \Delta\mathcal{L}^{(i)}$.
    end if
    if $\delta^{(i)} < \psi_0$ **then**
        Reject this step, contract the trust region radius, i.e. $r^{(i)} \leftarrow r^{(i)}/\alpha$ and **continue**.
    else
        Accept this step, i.e. assign $\boldsymbol{x}^{(i+1)} \leftarrow \boldsymbol{x}^{(i)} + \boldsymbol{d}, \boldsymbol{u}^{(i+1)} \leftarrow \boldsymbol{u}^{(i)} + \boldsymbol{w}$.
        update $r^{(i+1)}$ by:

$$r^{(i+1)} \leftarrow \begin{cases} r^{(i)}/\alpha, & if\ \delta^{(i)} \leq \psi_1, \\ r^{(i)}, & if\ \psi_1 \leq \delta^{(i)} < \psi_2, \\ \beta r^{(i)}, & if\ \delta^{(i)} \geq \psi_2. \end{cases} \tag{2.23}$$

    end if
    $r^{(i+1)} \leftarrow \max(r^{i+1}, r_l), i \leftarrow i+1$.
end while

2.3.2. Convergence Analysis

As we delve into the convergence analysis of the SCvx algorithm, it becomes essential to contrast its outcomes with those of the original problem (2.7). The SCvx algorithm produces a sequence of solutions $(\boldsymbol{x}^{(i)}, \boldsymbol{u}^{(i)})$, and it is crucial to analyze the progression of $(\boldsymbol{x}^{(i)}, \boldsymbol{u}^{(i)})$ with increasing iterations i. In particular, we're interested in the limit point of the sequence $(\boldsymbol{x}^{(i)}, \boldsymbol{u}^{(i)})$ as $i \to \infty$.

In the context of this section, we introduce $\mathcal{I}_{eq1}$ as the set of indices for the dynamic system equations $(\boldsymbol{x}_{k+1})_i = (\tilde{\boldsymbol{f}}(\boldsymbol{x}_k, \boldsymbol{u}_k))_i$, where $(\cdot)_i$ refers to ith component of $(\cdot)$. Furthermore, we define $\mathcal{I}_{ineq}$ and $\mathcal{I}_{eq2}$ as the sets of indices for inequality constraints $\boldsymbol{s}$

and equality constraints $\boldsymbol{q}$, respectively.

Moreover, it is noteworthy that the problem (2.7) can be restated in the form of unconstrained optimization augmented with a penalty function [25]. This alternate representation can be mathematically expressed as:

$$\begin{aligned}\min_{\boldsymbol{x},\boldsymbol{u}} \mathcal{G}(\boldsymbol{x},\boldsymbol{u}) =& \mathcal{C}(\boldsymbol{x},\boldsymbol{u}) + \sum_{k,i\in\mathcal{I}_{eq1}} \lambda_{k,i}(\tilde{\boldsymbol{f}}(\boldsymbol{x}_k,\boldsymbol{u}_k) - \boldsymbol{x}_{k+1})_i + \\ & \sum_{i\in\mathcal{I}_{ineq}} \lambda_i \max(0, -s_i(\boldsymbol{x},\boldsymbol{u})) + \sum_{j\in\mathcal{I}_{eq2}} \lambda_j ||q_j(\boldsymbol{x},\boldsymbol{u})||,\end{aligned} \tag{2.24}$$

where $\lambda_{k,i}$, λ_i ,and λ_j are the penalty coefficients.

Assumption 1 *Define the following sets of indices corresponding to the active inequality constraints:*

$$\mathcal{I}_{ac}(\overline{\boldsymbol{x}},\overline{\boldsymbol{u}}) := \{i | s_i(\overline{\boldsymbol{x}},\overline{\boldsymbol{u}}) = 0, i \in \mathcal{I}_{ineq}\} \subseteq \mathcal{I}_{ineq}. \tag{2.25}$$

Theorem 1 (Karush–Kuhn–Tucker (KKT)) *[37, Theorem 3.2] Suppose Assumption 1 holds and $(\overline{\boldsymbol{x}},\overline{\boldsymbol{u}})$ is a local optimum of the original problem, Problem (2.7), then there exist Lagrange multipliers $\overline{\mu}_k$ for all $\mathcal{I}_{eq1}$, $\overline{\mu}_i \geq 0$ for all $\mathcal{I}_{ac}(\overline{\boldsymbol{x}},\overline{\boldsymbol{u}})$, and $\overline{\mu}_j$ for all $\mathcal{I}_{eq2}$ such that:*

$$\nabla\mathcal{C}(\overline{\boldsymbol{x}},\overline{\boldsymbol{u}}) + \sum_{k,i\in\mathcal{I}_{eq1}} \overline{\mu}_{k,i}\nabla(\tilde{\boldsymbol{f}}(\overline{\boldsymbol{x}}_k,\overline{\boldsymbol{u}}_k) - \overline{\boldsymbol{x}}_{k+1})_i + \sum_{i\in\mathcal{I}_{ac}(\overline{\boldsymbol{x}},\overline{\boldsymbol{u}})} \overline{\mu}_i(-s_i(\overline{\boldsymbol{x}},\overline{\boldsymbol{u}})) + \sum_{j\in\mathcal{I}_{eq2}} \overline{\mu}_j q_j(\overline{\boldsymbol{x}},\overline{\boldsymbol{u}}) = 0. \tag{2.26}$$

We refer to a point that satisfies the above conditions as a KKT point.

Theorem 2 *Global Weak Convergence [25, Theorem 3.14]: Regardless of initial conditions, the SCvx algorithm (Algorithm 1) always has limit points, and any limit point $(\overline{\boldsymbol{x}},\overline{\boldsymbol{u}})$ is a stationary point of the non-convex penalty (2.24). Furthermore, if $(\overline{\boldsymbol{x}},\overline{\boldsymbol{u}})$ is feasible for the original non-convex problem (2.7), then it is a KKT point of (2.7).*

The weak convergence result can only guarantee that the SCvx algorithm converges to a set of limit points and the final result will oscillate among those limit points.

Assumption 2 *$\boldsymbol{s}_i(\boldsymbol{x},\boldsymbol{u})$ have Lipschitz continuous gradients, that is, $\exists L_i > 0$, s.t.*

$$||\nabla\boldsymbol{s}_i(\boldsymbol{x}^{(2)},\boldsymbol{u}^{(2)}) - \nabla\boldsymbol{s}_i(\boldsymbol{x}^{(1)},\boldsymbol{u}^{(1)})|| \leq L_i||(\boldsymbol{x}^{(2)},\boldsymbol{u}^{(2)}) - (\boldsymbol{x}^{(1)},\boldsymbol{u}^{(1)})||, \tag{2.27}$$

where $||\cdot||$ is L2 norm.

The key assumption used in [25] to establish single point convergence is that the function been optimized satisfies the (nonsmooth) Kurdyka–Lojasiewicz (KL) property [38, Definition 2.4]. The KL property is a deep geometric property of real-valued functions. In the context of optimization, it provides valuable insights about the behavior of the gradient flow and the rate of convergence of certain optimization algorithms near the solution. The detailed definition of KL property see definition 1 in appendix.

Assumption 3 *The penalized cost function $\mathcal{G}$ in (2.24) has Kurdyka–Lojasiewicz property [38, Definition 2.4].*

Theorem 3 (Global Strong Convergence) *[25, Theorem 3.21] Suppose Assumptions 1, 2, and 3 hold, then regardless of initial conditions, the sequence $(\boldsymbol{x}^{(i+1)}, \boldsymbol{u}^{(i+1)})$ generated by the SCvx algorithm (Algorithm 1) always converges to a single limit point, $(\overline{\boldsymbol{x}}, \overline{\boldsymbol{u}})$, and $(\overline{\boldsymbol{x}}, \overline{\boldsymbol{u}})$ is a stationary point of the non-convex penalty problem (2.24). Furthermore, if $(\overline{\boldsymbol{x}}, \overline{\boldsymbol{u}})$ is feasible for the original problem (2.7), then it is a KKT point of (2.7).*

Although the SCvx algorithm provides a good convergence result using mild assumptions [25], we cannot say the final result is feasible or optimal. The algorithm may converge to an infeasible solution or a local minimum solution of (2.7). We will discuss it in the following chapters.

2.4. Dynamic Model

In this thesis, we will use two types of simplified vehicle dynamic models in 2-dimensional space. In addition, the boundary condition of each single variable is defined as:

$$\boldsymbol{x}^L \leq \boldsymbol{x} \leq \boldsymbol{x}^U, \boldsymbol{u}^L \leq \boldsymbol{u} \leq \boldsymbol{u}^U. \tag{2.28}$$

2.4.1. Double Integral Model

The double integral model is also called as point mass model, which assumes the agent as a point mass. The double integral model is widely used for motion planning [16, 39]. And it is also the main dynamic model in stlpy [16]. The state vector and the control input vector of the double integral model are defined as:

$$\boldsymbol{x} = [s_x, s_y, \dot{s}_x, \dot{s}_y]^\top, \boldsymbol{u} = [a_x, a_y]. \tag{2.29}$$

where s_x and s_y are position in x and y axis at global coordinate system, a_x and a_y are corresponding acceleration. The state-space model can be expressed as:

$$\dot{\boldsymbol{x}} = A\boldsymbol{x} + B\boldsymbol{u}, A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{2.30}$$

And we can discretize the model from time t_k to t_{k+1}:

$$\boldsymbol{x}_{k+1} = \tilde{A}\boldsymbol{x}_k + \tilde{B}\boldsymbol{u}_k, \tilde{A} = \begin{bmatrix} 1 & 0 & \Delta t & \frac{1}{2}\Delta t^2 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tilde{B} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix}, \tag{2.31}$$

where $\Delta t = t_{k+1} - t_k$ is the time interval.

The upper and lower boundaries of the input and state variables are fixed values determined by stlpy benchmarks [16] or CommonRoad scenarios [28]. In addition, the double integral model also has constraints on the maximum acceleration a_{max}:

$$\sqrt{a_x^2 + a_y^2} \leq a_{max}. \tag{2.32}$$

2.4.2. Kinematic Single-Track Model

The kinematic single-track (KS) model assumes a vehicle with only two wheels: the front wheel and the rear wheel. Only the front wheel can be rotated. The KS model is illustrated in Fig. 2.3(a), where the reference point of the vehicle is attached to the rear wheel. The state and input vector of the KS model are defined as:

$$\boldsymbol{x} = [s_x, s_y, \delta, v, \Psi]^\top, \boldsymbol{u} = [v_\delta, a_{long}], \tag{2.33}$$

respectively. Here, s_x and s_y are position components in the dynamic model, δ is the angle between the front wheel and the ego vehicle, v is the speed of the ego vehicle, Ψ is the angle between the ego vehicle and x axis, v_δ is the speed of δ, and a_{long} is the acceleration in longitudinal direction. The KS model is then described as:

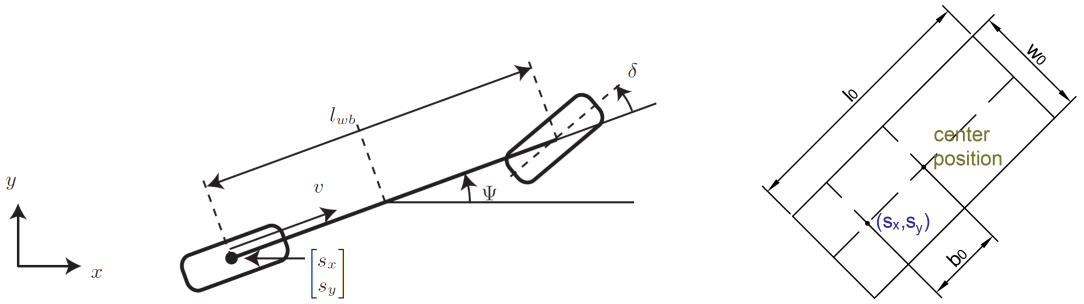
$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}) = [x_4\cos(x_5), x_4\sin(x_5), u_1, u_2, \frac{x_4}{l_{wb}}\tan(x_3)]^\top, \tag{2.34}$$

where l_{wb} is the length from the rear wheel to the front wheel.

The position components s_x and s_y of the ego state $\boldsymbol{x}$ are not located at the geometric center of the ego vehicle, as Fig. 2.3(b) shows. We define b_0 as the distance from (s_x, s_y) to the geometric center of the ego vehicle, and define w_0, l_0 as the width and length of the ego vehicle, respectively.

The upper and lower boundaries of input values in the KS models depend on the current state [28]. Besides, it remains to consider the friction circle:

$$\sqrt{u_2^2 + (x_4\dot{x}_5)^2} \leq a_{max}. \tag{2.35}$$



(a) Kinematic single-track model [28].

(b) Geometric center and position components in the dynamic model (s_x, s_y).

Figure 2.3.: Kinematic single-track model.

3. Methodology

In this chapter, we delve into various ways of encoding STL specifications. Following that, we will apply the SCvx algorithm to the optimization problem, incorporating STL specifications as seen in (2.13). Then, we will engage in a discussion on the time complexity of various optimization techniques. Ultimately, various predicates used in the following chapter are introduced.

3.1. Encoding Methods for STL specifications

The encoding methods are essential in solving optimal control problems with STL specifications. Existing encoding methods for STL constraints in optimization problems are not suitable for the SCvx algorithm. Therefore, we raise a new hybrid encoding method in this section.

3.1.1. Encoding Methods for STL specifications in stlpy

To simplify the encoding of STL formulas, we assume that they are written in negation normal form [15]. We then move the negations into the predicates by changing the sign of the robustness function, resulting in STL formulas without negation. This reduces the effort required for encoding negation and minimizes the number of required encoding variables [40].

In the following sections, we introduce different methods in stlpy for transforming STL specifications into constraints for solvable optimization problems.

MICP encoding

Mixed-Integer Convex Programming (MICP) is a type of Mixed-Integer Programming with convex constraints and convex objective function. stlpy implements MICP by using pydrake [41]. However, not all constraints in optimal control problems with STL specifications are guaranteed to be convex. In order to use MICP to solve these problems, we must restrict ourselves to linear atomic predicates, which have a linear specification function of the form $\rho^{\pi^\mu}(\xi, t_k) = \mu(\boldsymbol{x}_k, \boldsymbol{u}_k) = A\boldsymbol{x}_k + B\boldsymbol{u}_k + b$. Linear functions are always convex [42, p. 67], allowing us to use MICP for solving optimal control problems with STL specifications that contain only linear atomic predicates.

We use binary variables $z_k^{\pi_i} \in \{0, 1\}$ to represent the satisfaction of predicates over time. When predicate π^{μ_i} is satisfied at time t_k, $z_k^{\pi_i}$ takes the value 1, and 0 otherwise.

The following inequalities relate $z_k^{\pi_i}$ to the state $\boldsymbol{x}$, the control input $\boldsymbol{u}$ and the robustness margin ρ_r [15, 40]:

$$\begin{aligned} \mu_i(\boldsymbol{x}_k, \boldsymbol{u}_k) + M(1 - z_k^{\pi_i}) \geq \rho_r, \\ \mu_i(\boldsymbol{x}_k, \boldsymbol{u}_k) - M z_k^{\pi_i} \leq \rho_r, \end{aligned} \tag{3.1}$$

where M is a constant greater than or equal to the maximum output of all atomic predicates at any timestep, $M \geq \max_{i,k} \mu_i(\boldsymbol{x}_k)$.

To handle conjunctions and disjunctions of predicates, we introduce a continuous variable $z \in [0, 1]$ that represents the overall truth value of the formula. For conjunctions, the following constraint holds [15]:

$$\rho^{\varphi} = \rho^{\bigwedge_i \varphi_i} \Rightarrow z \leq z_i. \tag{3.2}$$

Note that z must be 0 if the conjunction is not satisfied because at least one z_i must be 0, and $z \leq z_i = 0$ in this case.

For disjunctions, the following constraint holds [15]:

$$\rho^{\varphi} = \rho^{\bigvee_i \varphi_i} \Rightarrow z \leq \sum_i z_i. \tag{3.3}$$

Again, if the disjunction is not satisfied, z must be 0 because all z_i are 0, and $z \leq \sum_i z_i = 0$ in this case.

To encode constraints on the STL formula, we can use inequations (3.1)-(3.3) on the STL formula. This encoding method declares binary variables only on atomic predicates, and the time complexity of MICP mainly depends on the number of binary variables [43]. Given that the number of atomic predicates in a specific STL formula is fixed, this encoding method does not depend on the formula's exact form. For instance, $\varphi = (\phi_1 \vee \phi_2) \vee \phi_3$ and $\varphi = \phi_1 \vee \phi_2 \vee \phi_3$ possess equivalent time complexity when using the MICP encoding. This independence of time complexity from the specific form of the given STL formula implies that altering the subformulas within the STL tree does not impact the time complexity.

In summary, the MICP encoding creates binary variables for every atomic predicate at each time step, which is inefficient as it increases time complexity exponentially.

SOS1 encoding for MICP

To decrease the number of binary variables, the Special Ordered Sets of type 1 (SOS1) encoding method is used [16].

SOS1 are a set of variables, at most one of which can take a non-zero value, all others being at zero [44]. Employed within the scope of disjunctions, SOS1 ensures that only one z_i in (3.3) equals 1. Although this method seems to dismiss some combinations of binary variables, it is still sound and complete [16].

SOS1 encoding method uses the same constraints with (3.1)-(3.3). However, we declare all $z_k^{\pi_i}$ as continuous variables. $z_k^{\pi_i} = 1$ means that predicate π^{μ_i} is totally satisfied. For conjunctions, $z = 1$ implies $z_i = 1$. This can be encoded without adding binary variables.

For disjunctions, we see the set $[1-z, z_1, z_2, ..., z_i, ...]$ as SOS1 and add SOS1 constraints on it. We assume that the number of z_i is n. We declare binary variables ζ_j with number of $\lceil \log_2(n+1) \rceil$ [16].

$$\begin{aligned} 1 - z + \sum_{i=1}^{n} z_i &= 1, \\ \sum_{m=1}^{2^{\lceil \log_2(n) \rceil - j}} \sum_{i=(m-1)2^j+2^{j-1}}^{m2^j-1} z_i &\leq \zeta_j. \end{aligned} \tag{3.4}$$

The principle rests on the idea that $\zeta_{\lceil \log_2(n+1) \rceil} ... \zeta_j ... \zeta_1$ represents a binary number. Upon conversion to its decimal equivalent, this number corresponds to the index i for which $z_i = 1$. Furthermore, when the binary number reads as 0...0...0, it signifies that $1 - z = 1$, indicating the disjunction is violated.

By using SOS1, we do not need to declare binary variables for every atomic predicate. We only need to declare binary variables for disjunctions, and other constraints will force the corresponding atomic predicates satisfying $z_k^{\pi_i} = 1$. However, this method prefers a flattened STL tree with fewer disjunctions and a lower depth, as the number of binary variables depends linearly on the number of disjunctions and logarithmically on the child nodes. A flattened STL tree can be achieved by flattening the STL tree, a process that aims to minimize the depth of the STL as much as possible. For the process of flattening an STL tree see Algorithm 2.

This algorithm begins with the root of STL tree, as indicated in the Require of Algorithm 2. Then, it recursively detects whether the logical operator of child nodes is the same with the logical operator of the parent node, which can be found in line 10. The recursion ends when the node under detection is a predicate, as depicted in line 2.

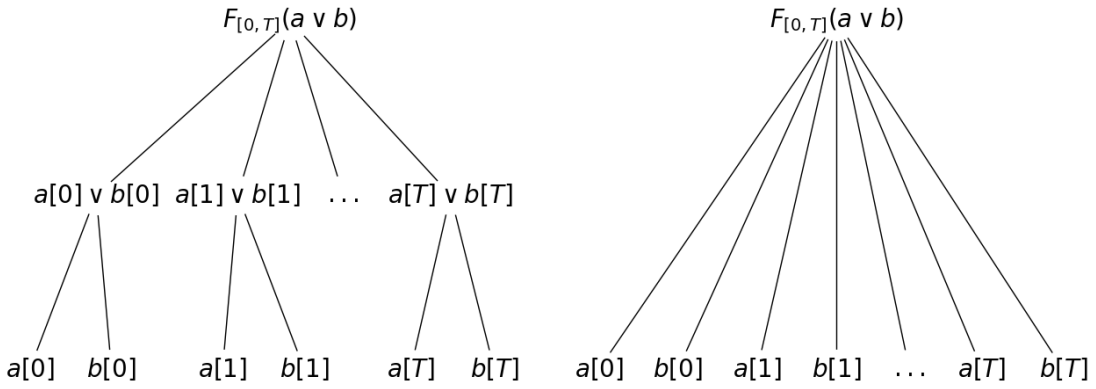
The resulting flattened STL tree is equivalent to the original STL tree in terms of its semantics. However, it has a smaller depth and fewer middle nodes, which makes it computationally more efficient to evaluate.

Algorithm 2 flattenSTLTree

Require: root of STL tree

```
1:  p ← root of STL tree
2:  if isPredicate(p) then
3:      return
4:  end if
5:  for all subformula ← p.children do
6:      if p.combinationType = subformula.combinationType then
7:          p.children.pop(subformula)
8:          p.children.insert(subformula.children)
9:      end if
10:     flattenSTLTree(subformula)          ▷ Recursively flatten the subformula
11: end for
12: return
```

In most cases, we can get a flat STL tree with more child nodes. And in this situation, the SOS1 encoding method can reduce the required number of binary variables and obtain a better encoding result than the MICP encoding method. Fig. 3.1 shows an example of this situation. However, Algorithm 2 does not work for the STL tree which has different logical operators at every layer, which means the STL tree cannot be flattened. Furthermore, if the STL tree is binary, the SOS1 encoding method will not be better than the MICP encoding method. An example is shown in Fig. 3.2.



(a) An STL tree before flattening.

(b) An STL tree after flattening.

Figure 3.1.: Example of the better case of SOS1. a and b are atomic predicates in this figure. Figure (a) shows an STL tree before flattening. Figure (b) shows the result after the flattening algorithm 2. This STL tree has $2(T+1)$ atomic predicates. Therefore, we need $2(T+1)$ binary variables in the MICP encoding method. However, we just need $\lceil \log_2(2(T+1)+1) \rceil$ binary variables in the SOS1 encoding method. In this example, the SOS1 encoding method reduces the binary variables into logarithmic numbers, which is dramatically better than the MICP encoding method.

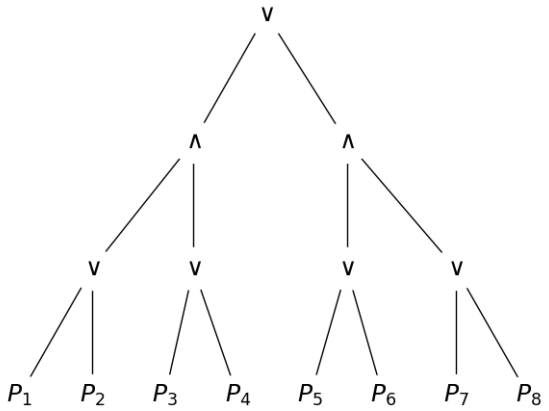


Figure 3.2.: An STL tree with worse encoding in the SOS1 encoding than the MICP encoding method. The STL formula is $((P_1 \vee P_2) \wedge (P_3 \vee P_4)) \vee ((P_5 \vee P_6) \wedge (P_7 \vee P_8))$. The atomic predicates, also considered leaf nodes, are binary distributed on the STL tree. And the STL tree has different logical operators on every layer. In this STL tree, there are 5 disjunctions and 8 atomic predicates. In the SOS1 encoding, $\lceil \log_2(n+1) \rceil = \lceil \log_2(2+1) \rceil = 2$ binary variables must be generated for every disjunction. Conversely, the MICP encoding method declares a binary variable for each atomic predicate. Therefore, the SOS1 encoding needs 10 binary variables, while the MICP encoding method just needs 8. As such, the MICP encoding method is superior to the SOS1 encoding in this case.

To summarize, this encoding method flattens the STL tree and then uses (3.1)-(3.4) to encode the STL formula. This creates a mixed-integer optimization problem that can be solved using a pydrake-based solver [41].

Continuous variables encoding for Ipopt

Above two encoding methods are used for linear atomic predicates. However, in practice, many atomic predicates have a nonlinear robustness function, which requires dealing with non-convex constraints for a more general optimal control problem.

Interior Point Optimizer (Ipopt) [45] is an open-source software package for large-scale nonlinear optimization. It can be used to solve general nonlinear programming problems of the form:

$$\begin{aligned} &\min_{\boldsymbol{x} \in \mathbb{R}^n} f(\boldsymbol{x}), \\ &s.t. \boldsymbol{g}^L \leq \boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{g}^U, \\ &\boldsymbol{x}^L \leq \boldsymbol{x} \leq \boldsymbol{x}^U, \end{aligned} \tag{3.5}$$

where $\boldsymbol{x}$ are general optimization variables, $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function, $\boldsymbol{g}$ are constraint functions, $\boldsymbol{g}^L$ and $\boldsymbol{g}^U$ are the lower and upper boundary of $\boldsymbol{g}$, respectively.

We need to encode the STL formula into constraints with continuous variables in order to use Ipopt. The robustness function of the STL sentence includes a large amount of min/max functions due to conjunctions and disjunctions. stlpy use $\widetilde{\max}/\widetilde{\min}$ functions [36] to approximate all min/max functions. In addition, stlpy adds intermediate variables ρ_i for every approximation min/max function and nonlinear predicates in order to avoid too much complexity on the robustness function. The encoding process is illustrated in Fig. 3.3.

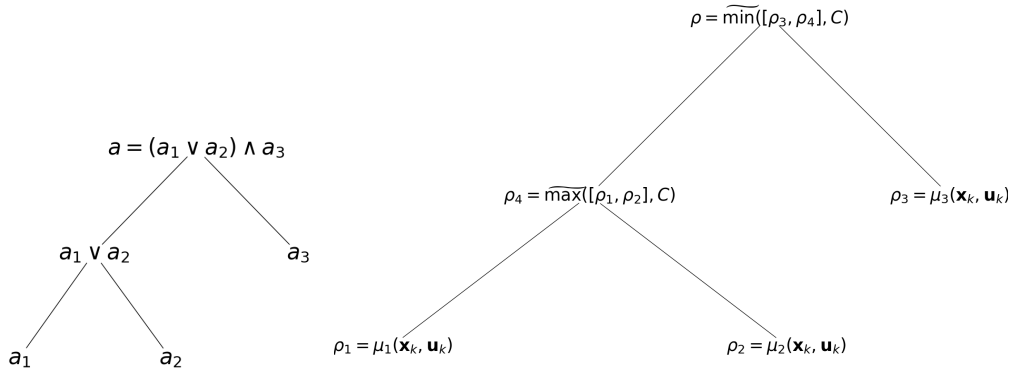


Figure 3.3.: Example of encoding the STL formula into equations of ρ_i.

By encoding the STL formula into equations of ρ_i, we can add a constraint on the final robustness $\rho > 0$ and include ρ in the objective function. For example, a commonly used objective function is:

$$f(\boldsymbol{x}, \boldsymbol{u}, \rho) = \boldsymbol{x}^\top Q \boldsymbol{x} + \boldsymbol{u}^\top R \boldsymbol{u} - \rho. \tag{3.6}$$

It is difficult to determine which STL tree is better for solving the problem using this encoding method. Flattening the STL tree will result in fewer variables ρ_i but more complex constraint expressions, while the opposite is true if the STL tree is not flattened.

Optimization encoding based on Scipy

Although Ipopt can solve all nonlinear programming problems, it can take much time and may fail to find a solution due to the non-convexity of constraints. Therefore, the idea is to dismiss all constraints and add robustness to the objective function. The optimization problem becomes:

$$\min_{\boldsymbol{u}} f(\boldsymbol{x}(\boldsymbol{u}), \boldsymbol{u}, \rho(\boldsymbol{x}(\boldsymbol{u}), \boldsymbol{u})). \tag{3.7}$$

f can take the form 3.6, but the state $\boldsymbol{x}$ and the robustness ρ are seen as functions of the control input $\boldsymbol{u}$. Then, only a control input sequence $\boldsymbol{u}$ needs to be optimized. Using

this method, the actual expression of f will be too complex to write down. Therefore, a numerical method must be used to solve this optimization problem.

The solver implemented in stlpy, based on Scipy [46], aims to minimize the objective function 3.7 without borders. As Scipy does not mandate a continuous gradient function, stlpy solely defines a cost function based on the control input, without requiring a unified continuous form. Each control input sequence corresponds to a defined trajectory with fixed robustness and cost. Consequently, Scipy can utilize the defined cost function to calculate the difference and approximate the cost function's gradient. Therefore, by merely defining a cost function and supplying an initial guess, Scipy can optimize problem 3.7.

One of the advantages of this encoding method is that it is not affected by the form of the STL tree. Instead, the only requirement is to calculate the robustness and cost based on a given control input sequence. This makes the method more flexible and adaptable to various nonlinear predicates.

Summary

In this section, we delve into the exploration of four distinct encoding strategies for STL specifications in stlpy. As we navigate through subsequent sections and chapters, we will adopt abbreviations to denote both the encoding techniques and the corresponding solver involved.

- **MICP:** Represents the traditional encoding technique using MICP, paired with a pydrake-based MICP solver. This approach offers a conventional method of encoding STL specifications.
- **SOS1:** Embodies the SOS1 encoding technique, applied alongside a pydrake-based MICP solver. This technique helps reduce the number of binary variables, offering more efficient time complexity.
- **Ipopt:** Denotes the Continuous variables encoding strategy employed with an Ipopt solver. This approach effectively handles nonlinear predicates and non-convex constraints within STL specifications.
- **Scipy:** Stands for the Optimization encoding strategy implemented with a Scipy solver. This method directly integrates the robustness into the objective function, providing a flexible solution for encoding STL specifications.

These encoding techniques provide varied approaches to handling STL specifications, each bringing its unique advantages to the table depending upon the specific requirements of the problem at hand.

3.1.2. Encoding Methods for STL specifications in SCvx

In this section, we discuss the encoding of STL specifications in SCvx. The original paper [27], which applies the SCvx algorithm on the optimal control problem with STL

specifications, solely relies on smin/smax functions for approximation of min/max functions. However, these smin/smax functions can handle only two inputs at a time. Consequently, when dealing with multiple inputs, we must establish a sequence for applying the smin/smax functions, resulting in the loss of input interchangeability. To solve this problem, we adopt a hybrid encoding method that combines the use of smin/smax functions for temporal operators and $\widetilde{\max}/\widetilde{\min}$ functions for conjunctions and disjunctions with multiple inputs. The following subsection provides a simple example illustrating why the smin/smax function is appropriate for temporal operators but falls short when applied to conjunctions and disjunctions with multiple inputs.

The Preference for Approximation of min/max function

1. **smin/smax function for multiple functions**

   The application of smin/smax functions can present challenges when multiple functions share identical values at a particular point but exhibit different gradients. To illustrate this, consider the following simplified optimization problem:

   $$\max_{x} \min\{f_1(x), f_2(x), f_3(x)\}. \tag{3.8}$$

   Problem (3.8) can be seen as a simplified maximum robustness of conjunction problem:

   $$\max_{\xi} \rho^{\varphi_1 \wedge \varphi_2 \wedge \varphi_3}(\xi, t_k), \tag{3.9}$$

   where $\xi(\boldsymbol{x}_k), \boldsymbol{u}_k$ corresponds to optimization variable x in (3.8), and robustness function of atomic predicates $\varphi_1, \varphi_2, \varphi_3$ corresponds to functions f_1, f_2, f_3. The problem (3.8) is transformed by problem (3.9) in the case of only one optimization variable.

   Assuming that $f_1(x_0) = f_2(x_0) = f_3(x_0)$ and that $f_1(x)$ and $f_2(x)$ are monotonically increasing while $f_3(x)$ is monotonically decreasing (as shown in Figure 3.4). Using smin function on the problem (3.8) would require selecting the first two functions into the smin function and then applying smin again to the result along with the remaining function. The expression would be as follows:

   $$\min\{f_1(x), f_2(x), f_3(x)\} = \text{smin}(\text{smin}(f_1(x), f_2(x), C), f_3(x), C). \tag{3.10}$$

   Obviously, the solution to the problem (3.8) is the equilibrium x_0. If we analyze (3.10) at the equilibrium x_0, we will see:

   $$\text{smin}(\text{smin}(f_1(x_0), f_2(x_0), C), f_3(x_0), C) = \frac{5}{16} f_1(x_0) + \frac{5}{16} f_2(x_0) + \frac{3}{8} f_3(x_0) - \frac{25}{64} C. \tag{3.11}$$

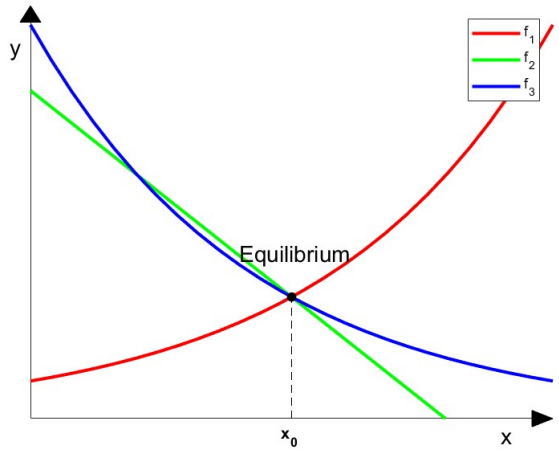


Figure 3.4.: Function plot based on assumptions for problem (3.8).

a) *smooth parameter C large*

In equation (3.11), the coefficients before different functions are different, which means that the transformed problem loses the of the sequence $f_1(x)$, $f_2(x)$, and $f_3(x)$. If we use a gradient-based method to find the solution of (3.10), the gradient of $f_3(x)$ will take more account. Assuming that the gradient of problem (3.11) is not zero:

$$\frac{5}{16}\frac{df_1(x)}{dx}|_{x_0} + \frac{5}{16}\frac{df_2(x)}{dx}|_{x_0} + \frac{3}{8}\frac{df_3(x)}{dx}|_{x_0} \neq 0, \tag{3.12}$$

the solution of problem (3.10) is no longer x_0. Moreover, the solution error between problem (3.10) and problem (3.8) increases with the increasing C, as Fig. 3.5 shows. Therefore, if we set the C enough small, it seems the final solution will be acceptable.

b) *smooth parameter C small*

However, unfortunately, we cannot set C too small. If we want to apply smin/smax functions into the SCvx algorithm, smin/smax functions need to satisfy Assumption 2 in order to get a strong convergence result. Here, we analyze the gradients of $f(x) = \text{smin}(f_1(x), f_2(x), C)$ and assume $f_1(x_1) - f_2(x_1) = -C$, $f_1(x_2) - f_2(x_2) = C$:

$$||\frac{d}{dx}f(x_2) - \frac{d}{dx}f(x_1)|| = ||\frac{d}{dx}f_2(x_2) - \frac{d}{dx}f_1(x_1)|| \leq L||x_2 - x_1||. \tag{3.13}$$

Therefore, Lipschitz constant L must satisfy:

$$L \geq \frac{||\frac{d}{dx}f_2(x_2) - \frac{d}{dx}f_1(x_1)||}{||x_2 - x_1||}, \tag{3.14}$$

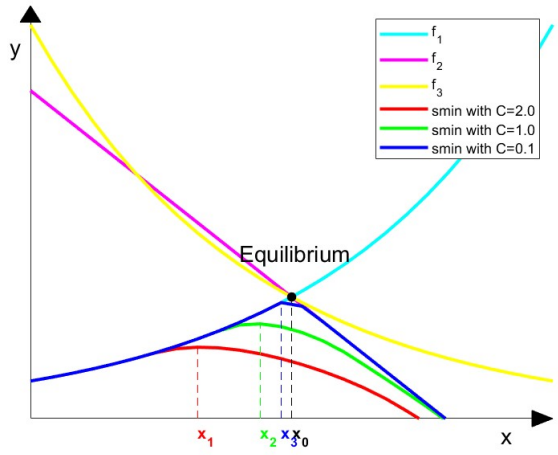


Figure 3.5.: smin optimization problem solutions. x_0 is the solution of problem (3.8). x_1, x_2, x_3 are solutions of problem (3.10) with $C = 2.0, 1.0, 0.1$ respectively. The solution error between problem (3.10) and problem (3.8) increases with the increasing C.

where $||\frac{d}{dx}f_2(x_2) - \frac{d}{dx}f_1(x_1)||$ can be seen as a constant and $||x_2 - x_1||$ is positive correlated to the smin constant C. Therefore, small C means large Lipschitz constant L, which is negatively correlated to the convergence rate in the SCvx [25].

Therefore, both large C and small C are not suitable for the requirements of SCvx. There may have a possible C for solving a specific problem, but it is difficult to apply to general problems.

2. **smin/smax function for the same function with different inputs**

   smin/smax functions are just unsuitable for multiple different functions. If we use smin/smax functions on the same function, there will be no problem. For example, let us consider:

$$\max_{x_1,x_2,x_3} \min\{f(x_1), f(x_2), f(x_3)\}. \tag{3.15}$$

   Even though we still lose the interchangeability by using smin function:

$$\text{smin}(\text{smin}(f(x_1), f(x_2), C), f(x_3), C) = \frac{5}{16}f(x_1) + \frac{5}{16}f(x_2) + \frac{3}{8}f(x_3) - \frac{25}{64}C, \tag{3.16}$$

   the different terms have different optimization variables and do not affect each other. Problem (3.15) can be seen as a simplified maximum robustness of *always*

problem:

$$\max_{\xi} \rho^{G_{[0,2]}\varphi}(\xi, t_k) = \max_{\xi} \min_{t_{k'} \in [t_k, t_k+2]} \rho^{\varphi}(\xi, t_{k'}) = \max_{\xi(\boldsymbol{x}_k, \boldsymbol{u}_k), \xi(\boldsymbol{x}_{k+1}, \boldsymbol{u}_{k+1}), \xi(\boldsymbol{x}_{k+2}, \boldsymbol{u}_{k+2})} \min\{\rho^{\varphi}(\xi, t_k), \rho^{\varphi}(\xi, t_{k+1}), \rho^{\varphi}(\xi, t_{k+2})\}, \tag{3.17}$$

where $\xi(\boldsymbol{x}_k, \boldsymbol{u}_k), \xi(\boldsymbol{x}_{k+1}, \boldsymbol{u}_{k+1}), \xi(\boldsymbol{x}_{k+2}, \boldsymbol{u}_{k+2})$ corresponds to x_1, x_2, x_3, respectively.

This elementary problem demonstrates that the usage of smin/smax functions doesn't create issues when applied to functions with an iterative form. Therefore, it is safe to say that smin/smax functions can be effectively used on temporal operators.

In summary, we can apply smin/smax functions on temporal operators but cannot use smin/smax functions on other parts.

Auxiliary Variable for STL

In this section, we discuss how to transform temporal operators and the robustness function inside temporal operators into an iterative form, in order to apply smin/smax functions on temporal operators.

Solving the constraints in (2.13) is often challenging due to the complexity of the robustness function. These constraints are typically non-convex, which can lead to difficulties when applying numerical optimization methods [15]. To address this issue, an approach is to introduce new dynamic states and establish relationships between these states and the robustness function [47]. By embedding the entire feasible set into a higher dimension, the problem can be convexified, making it easier to solve.

It is worth noting that the temporal operators $F_{[a,b]}$, $G_{[a,b]}$, and $U_{[a,b]}$ are inherently linked to specific time intervals. Since many STL formulas consist of multiple temporal operators, it is common to introduce auxiliary variables for each time step as new dynamic states.

1. **The Eventually Operator**

   The *eventually* operator calculates maximum robustness in a given interval. Notice that with the discretized state and control variables, we can denote the first and last time instance for $t_k \in [t_{k'} + a, t_{k'} + b]$ as $k^{(a)}$ and $k^{(b)}$ respectively. Additionally, we define the set of instances in between $k^{(a)}$ and $k^{(b)}$ as $K := \{k^{(a)}, k^{(a)} + 1, ..., k, ..., k^{(b)}\}$. For instance k, the time is represented as t_k. Then, the robustness of *eventually* is:

   $$\rho^{F_{[a,b]}\varphi}(\xi, t_{k'}) = \max_{k \in K} \rho^{\varphi}(\xi, t_k). \tag{3.18}$$

   In order to satisfy the condition of dynamic states, the new auxiliary variable must be calculated iteratively, similar to the system dynamic equations in discrete

form, as shown in equation (2.7). To introduce a new auxiliary variable as a new dynamic state, we can define an auxiliary variable $\boldsymbol{y} = [y_0, y_1, ..., y_k, ..., y_N]$ for every time step. Here, $y_k = \rho^{F_{[a,a+t_k]}\varphi}(\xi, t_{k'})$. We can then list the iterative equation of the new auxiliary variable $\boldsymbol{y}$ as follows:

$$\begin{cases} y_k = \rho^{\varphi}(\xi, t_{k^{(a)}}), & \forall 0 \leq k \leq k^{(a)}, \\ y_{k+1} = \max(y_k, \rho^{\varphi}(\xi, t_{k+1})), & \forall k^{(a)} < k < k^{(b)}, \\ y_k = y_{k^{(b)}}, & \forall k^{(b)} \leq k \leq N. \end{cases} \tag{3.19}$$

2. **The Always Operator**

The *always* operator calculates minimum robustness in a given interval. We use the same representation with the *eventually* operator:

$$\rho^{G_{[a,b]}\varphi}(\xi, t_{k'}) = \min_{k \in K} \rho^{\varphi}(\xi, t_k). \tag{3.20}$$

By using the same method in the *eventually* operator, we can then introduce an auxiliary variable $\boldsymbol{y} = [y_0, y_1, ..., y_k, ..., y_N]$ as follows:

$$\begin{cases} y_k = \rho^{\varphi}(\xi, t_{k^{(a)}}), & \forall 0 \leq k \leq k^{(a)}, \\ y_{k+1} = \min(y_k, \rho^{\varphi}(\xi, t_{k+1})), & \forall k^{(a)} < k < k^{(b)}, \\ y_k = y_{k^{(b)}}, & \forall k^{(b)} \leq k \leq N. \end{cases} \tag{3.21}$$

3. **The Until Operator**

To calculate the robustness of the *until* operator, we introduce $K' := \{k^{(a)}, k^{(a)} + 1, ..., k, ..., k'\}$, where $k^{(a)}$ has the same definition as before, and $k' \in K$ refers to an optimization variable. Then, we can rewrite the formula for the *until* operator as:

$$\rho^{\varphi_1 U_{[a,b]} \varphi_2}(\xi, t_k) = \max_{k' \in K} \min(\rho^{\varphi_2}(\xi, t_{k'}), \min_{k'' \in K'} \rho^{\varphi_1}(\xi, t_{k''})). \tag{3.22}$$

The *until* operator means that φ_1 holds until φ_2 becomes true within the time interval $[a, b]$. It needs three times min/max operation. Although there are three min/max functions in the *until* operator, we still can use the same idea as before. First, we introduce an auxiliary variable $\boldsymbol{\zeta} = [\zeta_0, \zeta_1, ..., \zeta_k, ..., \zeta_N]$ to replace the inner min function of ρ^{φ_1}:

$$\begin{cases} \zeta_k = \rho^{\varphi_1}(x, t_{k^{(a)}}), & \forall 0 \leq k \leq k^{(a)}, \\ \zeta_{k+1} = \min(\zeta_k, \rho^{\varphi_1}(\xi, t_{k+1})), & \forall k^{(a)} < k < k^{(b)}, \\ \zeta_k = \zeta_{k^{(b)}}, & \forall k^{(b)} \leq k \leq N. \end{cases} \tag{3.23}$$

Next, for the outer max and min function, we introduce another auxiliary variable $\boldsymbol{\eta} = [\eta_1, \eta_2, ..., \eta_k, ..., \eta_N]$ as follows:

$$\begin{cases} \eta_k = \min(\rho^{\varphi_2}(\xi, t_{k^{(a)}}), \zeta_k), & \forall 0 \leq k \leq k^{(a)}, \\ \eta_{k+1} = \max(\min(\zeta_{k+1}, \rho^{\varphi_2}(\xi, t_{k+1})), \eta_k), & \forall k^{(a)} < k < k^{(b)}, \\ \eta_k = \eta_{k^{(b)}}, & \forall k^{(b)} \leq k \leq N. \end{cases} \tag{3.24}$$

4. **Conjunction**

   In STL, the conjunction of two functions is represented by the logic "and", and it calculates the minimum robustness. Mathematically, the function for conjunction can be written as:

$$\rho^{\varphi_1 \wedge \varphi_2}(\xi, t_k) = \min(\rho^{\varphi_1}(\xi, t_k), \rho^{\varphi_2}(\xi, t_k)). \tag{3.25}$$

   A conjunction in an STL formula does not require information from other times; it only relies on the current state information. However, the subformulas inside temporal operators can be composed of logical operators. Since temporal operators require the calculation of the new auxiliary variable at each time step, we also need to compute the robustness of logical operators at each time step. In other words, we need to evaluate the conjunction of the new auxiliary variable with the corresponding formula at each time step in order to calculate the robustness of the temporal operator.

   Then, we can introduce an auxiliary variable $\boldsymbol{y} = [y_0, y_1, ..., y_k, ..., y_N]$ as follows:

$$y_k = \min(\rho^{\varphi_1}(\xi, t_k), \rho^{\varphi_2}(\xi, t_k)), \ \forall 0 < k < N. \tag{3.26}$$

5. **Disjunction**

   The disjunction of two functions has an opposite effect compared with a conjunction. The disjunction calculates the maximum:

$$\rho^{\varphi_1 \vee \varphi_2}(\xi, t_k) = \max(\rho^{\varphi_1}(\xi, t_k), \rho^{\varphi_2}(\boldsymbol{x}, t_k)). \tag{3.27}$$

   Using the same reasoning and methodology as in the case of conjunction, we can introduce an auxiliary variable $\boldsymbol{y} = [y_0, y_1, ..., y_k, ..., y_N]$ for disjunction as follows:

$$y_k = \max(\rho^{\varphi_1}(\xi, t_k), \rho^{\varphi_2}(\xi, t_k)), \ \forall 0 < k < N. \tag{3.28}$$

6. **Summary**

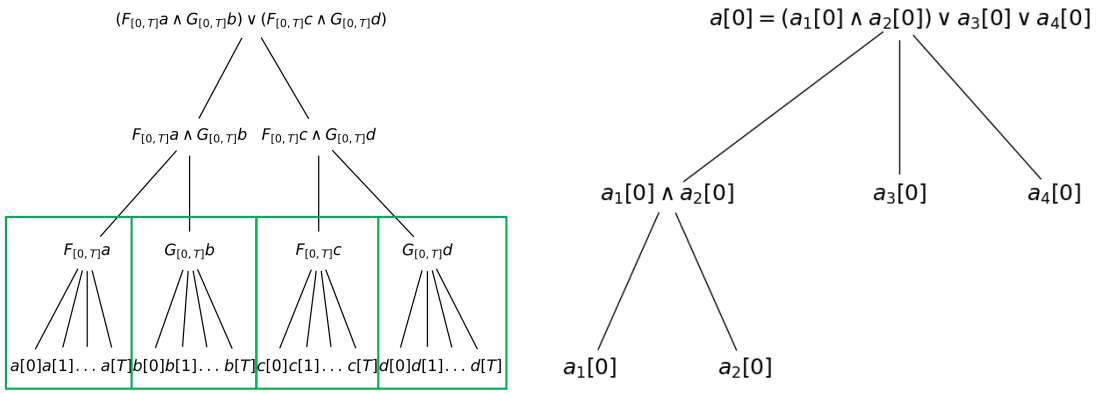
   We introduce auxiliary variables for the computation of robustness in temporal operators. These variables enable the use of an iterative form for calculating robustness. Then, we can use smin/smax functions on the iterative form. The last variable in the auxiliary variable sequence represents the robustness of the temporal operator. We denote this final variable, and thus the robustness of the temporal operator, as $\boldsymbol{y}[\text{end}]$, where $\boldsymbol{y}$ refers to the auxiliary variable.

The STL Tree Preference in SCvx

In this section, we discuss the STL Tree Preference in SCvx based on applying smin/smax functions for temporal operators and $\widetilde{\max}/\widetilde{\min}$ functions for other parts

The temporal operator can be seen as a node in the STL tree. The STL formula may use different conjunctions and disjunctions to combine various temporal operators to obtain the final robustness.

On the other hand, the temporal operators are composed of sub-formulas. Those sub-formulas are composed of different atomic predicates. A sub-formula can also be represented as a tree structure. This is because the sub-formula of the temporal operator consists of conjunctions and disjunctions, as Fig 3.6(a) shows. Furthermore, the sub-formula of the temporal operator for each timestep is the same. Therefore, we only need to consider one tree for each temporal operator, which can be reused for each timestep. By breaking down the temporal operator function into a separate tree, it can be more easily analyzed and evaluated to determine its contribution to the overall robustness of the STL formula.



(a) The whole structure of STL tree.

(b) A example of sub-formula a

Figure 3.6.: Example of STL tree. The tree can be divided into two parts by temporal operators. Temporal operators (inside green squares) are seen as nodes in (a). The root of the tree in (b) is the sub-formula of a temporal operator. The expressions and tree structure are the same for $a[0]$ to $a[T]$, but they are associated with different states in time. Every $a[i]$ can be composed of various predicates, as shown in (b).

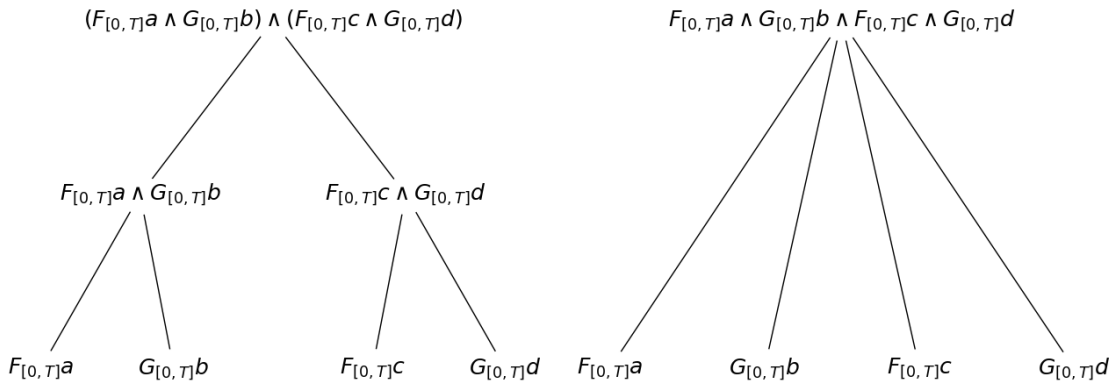
As Fig. 3.6 shows, STL trees can be represented in various forms, but we prefer to use the tree that temporal operators have the smallest depth, in order to reduce the number of min/max operations on temporal operators. To achieve this, we use algorithm 3 to reduce the depth of the temporal operators and flatten the STL tree. The only difference between Algorithm 2 and 3 is that Algorithm 3 flattens the STL tree from root to temporal operators rather than from root to predicates, as indicated in

line 2. Because we want to apply smin/smax function for the child nodes of temporal operators and do not want to change the structure inside the temporal operators.

Algorithm 3 flattenSTLTreeBeforeTemporalOperator

Require: root of STL tree

```
1: p ← root of STL tree
2: if isTemporalOperator(p) then
3:     return
4: end if
5: for all subformula ← p.children do
6:     if p.combinationType = subformula.combinationType then
7:         p.children.pop(subformula)
8:         p.children.insert(subformula.children)
9:     end if
10:    flattenSTLTreeBeforeTemporalOperator(subformula) ▷ Recursively flatten
   the subformula
11: end for
12: return
```



(a) STL tree before flatten.

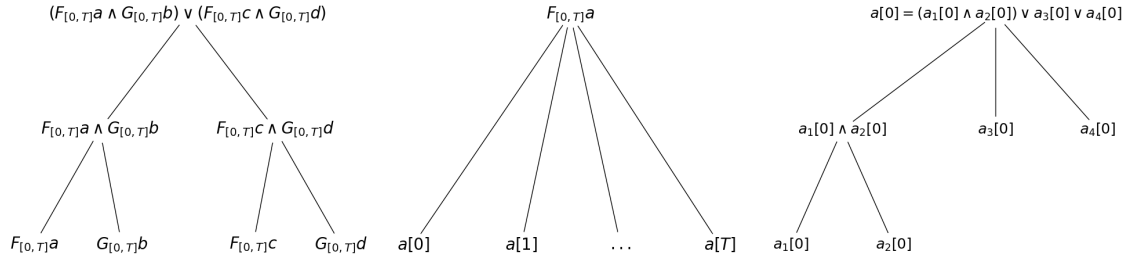
(b) STL tree after flatten.

Figure 3.7.: Effect of flatten Algorithm 3.

As Fig. 3.7 shows, the tree representing final robustness and the tree representing meta-predicates of temporal operators can be related to multiple nodes which have different robustness functions. This makes it impossible to use an iterative form to calculate the robustness of the parent node. Additionally, when multiple nodes are connected to a common parent node, each node should have equal weight when calculating robustness. To achieve this, we can use $\widetilde{\max}/\widetilde{\min}$ function to calculate the robustness of each parent node.

However, it is possible to calculate the robustness of temporal operators iteratively

with two inputs, as shown in equations (3.18) to (3.28). Therefore, it is feasible to use the smin/smax function for temporal operator approximation.



(a) The whole structure. (b) Temporal operator. (c) Sub-formula of temporal operator a.

Figure 3.8.: Using different approximations for min/max. In Fig (a) and (c), the tree can have several child nodes. Therefore, Under-approximation for the min/max Function is used in (a) and (c). In (b), sub-formula a has the same expression and robustness can be iteratively calculated. Therefore, the smin/smax function is used in (b).

Summary

In this section, we explore the implementation of STL specifications in SCvx. This process includes approximation of min/max functions, conversion of temporal operators into iterative form, and prioritization within the STL tree. For brevity, we will henceforth refer to the encoding of STL specifications using the SCvx algorithm as simply 'SCvx'.

3.2. Successive Convexification for STL

We can use auxiliary variables (3.18) to (3.28) to transform the problem (2.13) of optimal control with robustness functions into:

$$
\begin{aligned}
&\min_{\boldsymbol{u}} \; C(\boldsymbol{x}, \boldsymbol{u}) \\
&\text{s.t.}\, \boldsymbol{x}_{k+1} = \tilde{\boldsymbol{f}}(\boldsymbol{x}_k, \boldsymbol{u}_k), k = 0, 1, 2, ..., N-1, \\
&\boldsymbol{z}_{k+1} = \boldsymbol{g}(\boldsymbol{x}_{k+1}, \boldsymbol{z}_k, \boldsymbol{z}_{k+1}), k = 0, 1, 2, ..., N-1, \\
&\boldsymbol{x}_k \in X_k, \boldsymbol{u}_k \in U_k,\; k = 0, 1, 2, ..., N, \\
&\boldsymbol{q}(\boldsymbol{x}_k, \boldsymbol{u}_k) = 0,\; k = 0, 1, 2, ..., N, \\
&\boldsymbol{s}(\boldsymbol{x}_k, \boldsymbol{u}_k) \geq 0,\; k = 0, 1, 2, ..., N, \\
&\boldsymbol{z}_N[\text{end}] \geq 0,
\end{aligned}
\tag{3.29}
$$

where the auxiliary variables $\boldsymbol{Z} = [\boldsymbol{z}_1, \boldsymbol{z}_2, ..., \boldsymbol{z}_k, ..., \boldsymbol{z}_N]$ are referred to as sub-dynamic states. All $\boldsymbol{z}_i$ are introduced according to rules (3.18) to (3.28). Therefore, the sub-dynamic states $\boldsymbol{z}_i$ are vectors with the length of $N+1$, the same length with the dynamic state. The last sub-dynamic state is represented by $\boldsymbol{z}_N[\text{end}]$ which signifies the robustness of the entire STL formula. The function $\boldsymbol{g}$ is generated iteratively, as illustrated from (3.18) to (3.28).

Then, we can introduce successive convexification into problem (3.29). To this end, we conceive $(\boldsymbol{x}^{(1)}, \boldsymbol{z}^{(1)})$ from problem Eq. (3.29) as $\boldsymbol{x}^{(1)}$ in Algorithm 1. Consequently, Algorithm 1 can be applied directly to problem Eq. (3.29).

For a outline of successive convexification implementation for STL, refer to Algorithm 4. This algorithm initially flattens the STL tree before temporal operators. Subsequently, it computes the constraints functions $\tilde{\boldsymbol{f}}, \boldsymbol{g}$ and $\boldsymbol{q}$, along with their respective gradients. Following the initialization of all states, control inputs and the final sub-dynamic state, the SCvx algorithm is utilized to produce an approximate solution.

Algorithm 4 STL-SCvx Algorithm

Require: root of STL tree, start state $\boldsymbol{x}_0$, end state $\boldsymbol{x}_f$, Time Horizon T.

1: p $\leftarrow$ root of STL tree.
2: flattenSTLTreeBeforeTemporalOperator(p) ▷ see Algorithm 3.
3: functionList $\leftarrow$ addConstraintFunctions(p) ▷ calculate functions $\boldsymbol{f}, \boldsymbol{g}, \boldsymbol{q}$.
4: Initial guess state $\boldsymbol{x}^{(1)} \leftarrow$ interpolate($\boldsymbol{x}_0$, $\boldsymbol{x}_f$, T). ▷ give initial trajectory
5: Initial guess input $\boldsymbol{u}^{(1)} \leftarrow \boldsymbol{0}$.
6: Initial guess sub-state $\boldsymbol{z}^{(1)} \leftarrow$ calculateSubState($\boldsymbol{x}^{(1)}$, functionList).
7: $((\boldsymbol{x}^{(i)}, \boldsymbol{z}^{(i)}), \boldsymbol{u}^{(i)}) \leftarrow$ SCvx Algorithm($((\boldsymbol{x}^{(1)}, \boldsymbol{z}^{(1)}), \boldsymbol{u}^{(1)})$, functionList). ▷ see Algorithm 1
8: **return** $((\boldsymbol{x}^{(i)}, \boldsymbol{z}^{(i)}), \boldsymbol{u}^{(i)})$.

3.3. Time Complexity Analysis

In this section, we delve into the time complexity of convex optimization, a critical component of both MICP and SCvx. Following this analysis, we will explore the time complexities of MICP and SCvx in detail.

3.3.1. Convex Optimization

Convex optimization deals with optimization problems where the objective function and the constraints are both convex [42]. This category includes diverse problem types such as linear programming (LP) and convex quadratic programming (QP). The most commonly adopted method to tackle these convex optimization problems is the interior-point method [20, chapter 2.2.2].

Consider the following convex optimization problem (3.30):

$$\begin{aligned} &\min_{\boldsymbol{x}\in R^n} f(\boldsymbol{x}), \\ &s.t. A\boldsymbol{x} = b, \\ &\boldsymbol{g}(\boldsymbol{x}) \leq 0. \end{aligned} \tag{3.30}$$

The interior-point method reformulates this original problem (3.30) to the following problem (3.31):

$$\begin{aligned} &\min_{\boldsymbol{x}(\mu)\in R^n} f(\boldsymbol{x}) + \mu\Phi_g(x), \\ &s.t. A\boldsymbol{x} = b, \end{aligned} \tag{3.31}$$

where $\Phi_g(x)$ is called barrier function. The barrier function $\Phi_g(x)$ ensures that the optimal solution $\boldsymbol{x}^*(\mu)$ of the problem (3.31) satisfies the original inequality constraint $\boldsymbol{g}(\boldsymbol{x}) \leq 0$. The parameter μ in this context is referred to as the barrier parameter. As μ approaches zero, the solution $\boldsymbol{x}^*(\mu)$ converges to the optimal solution of the original problem $\boldsymbol{x}^*$ under mild conditions [42, chapter 11.3]. By solving the problem (3.31) iteratively, we can obtain an approximation solution to the original problem.

Notably, the most efficient known interior-point method algorithms for linear and convex quadratic programming can find an ϵ-accurate solution in $\mathcal{O}(n^{0.5}log(\frac{1}{\epsilon}))$ iterations for an LP or a convex QP problem [48], where n refers to the number of optimization variables. Here, one iteration corresponds to a single solution to the problem (3.31).

Typically, problem (3.31) is solved employing Newton's method [20]. The time complexity of Newton's method is generally $\mathcal{O}(n^3)$ for dense gradient matrices. However, when Newton's method is applied to the problem (3.31) with a sparse constraint matrix A and a sparse gradient matrix for the objective function, a more favourable time complexity can be achieved [49, chapter 8].

To sum up, the interior-point method offers a polynomial-time solution to convex optimization problems with a complexity of $\mathcal{O}(n^\gamma)$, where $\gamma \leq 3.5$ for linear and convex quadratic programming. This highlights its efficiency and the promising prospects it provides in the field of optimization.

3.3.2. MICP

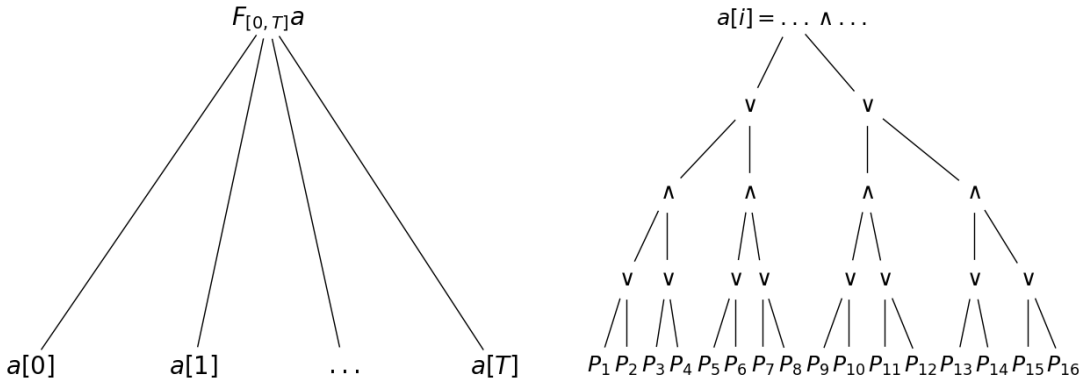
As a subset of Mixed-Integer Programming, MICP is categorized as NP-hard [16], implying that in the worst-case scenario, the time required could be exponential relative to the problem's size. The branch-and-bound algorithm is frequently used to solve MICP [50].

The branch-and-bound algorithm begins by tackling a relaxed version of the problem, replacing the integrality constraints with simpler bounds. Subsequently, it "branches" based on a selected variable: the variable is fixed at various integer settings, each generating a relaxed sub-problem and providing a superior bound on the optimal solution. In the worst case, all combinations of the integer variables need to be evaluated.

Let us assume that the number of binary variables is m and the number of continuous variables is n in a MICP problem. In the worst-case scenario, we would need to solve 2^m convex optimization problems. Hence, the complexity of MICP is $\mathcal{O}(2^m n^\gamma)$, where $\gamma \leq 3.5$.

In the MICP encoding method, we declare a binary variable for each atomic predicate. As per the problem (2.13), assuming $N+1$ moments, the number of binary variables is $(N+1)M_\phi$ [51], where M_ϕ is the number of atomic predicates in a single moment.

For the SOS1 encoding method, binary variables are declared based on the structure of the STL tree. It's challenging to ascertain the number of binary variables. Nonetheless, we can assess the best-case and worst-case scenarios. In an ideal scenario, where all predicates are connected by conjunctions, no binary variables are needed. However, this situation is impractical. Consider a more likely case, where all predicates are connected by disjunctions. Here, the number of binary variables would be $\lceil \log_2((N+1)\phi+1) \rceil$ [16]. The worst-case scenario arises when all predicates are binary distributed on the STL tree and within the same temporal operator *eventually*, and the STL tree cannot be flattened. In this situation, the number of binary variables is $\frac{4}{3}(N+1)(4^{\lceil \frac{\log_2 M_\phi}{2} \rceil} - 1) + \lceil \log_2(N+2) \rceil$, as shown in Fig. 3.9. Generally, the number of binary variables can be represented as $C_1(N+1)\phi + \log_2(C_2(N+1)M_\phi + 1)$, where C_1 and C_2 are constants influenced by the STL tree structure. Although MICP with the SOS1 encoding method remains NP-hard in most cases, it can be solved in polynomial time under specific circumstances.



(a) (b)

Figure 3.9.: Example of the worst case of SOS1. In this situation, there is just a temporal operator *eventually*. All predicates are binary distributed on subformula tree of *eventually*. In this case, SOS1 needs $\frac{4}{3}(N+1)(4^{\lceil \frac{\log_2(16)}{2} \rceil} - 1) + \lceil \log_2(N+2) \rceil = 20(N+1) + \lceil \log_2(N+2) \rceil$ binary variables. The mathematical formula is up limit. If we change the structure of the STL tree, the number of binary variables will decrease. For example, using P_{15} to replace $P_{15} \vee P_{16}$, we just need $18(N+1) + \lceil \log_2(N+2) \rceil$ binary variables.

3.3.3. SCvx

The SCvx method converts the initial problem as outlined in (2.7) into a convex subproblem indicated by (2.20). The time complexity associated with SCvx is contingent upon the iteration count in resolving the convex subproblem and the computational necessities involved in the process of solving the convex subproblem. Assume that the number of iterations is represented as b and the variable count as n. The time complexity of SCvx can thus be represented as $\mathcal{O}(bn^{\gamma})$, where $\gamma \leq 3.5$.

3.4. Predicates

In this section, we will introduce some predicates and their robustness functions.

3.4.1. Atomic Predicates

The atomic predicate π^{μ} refers to the predicate with a continuous function μ. According to the robustness functions of atomic predicates, we can classify atomic predicates into linear predicates and nonlinear predicates.

Linear Predicates

The linear predicate refers to the atomic predicate with a linear robustness function. The robustness function is:

$$\rho(\boldsymbol{x}) = \boldsymbol{a}^\top \boldsymbol{x} + b, \tag{3.32}$$

where ρ is the robustness function, $\boldsymbol{a}$ and b are the factors of the linear function.

Nonlinear Predicates

The nonlinear predicate refers to the atomic predicate with a nonlinear robustness function. The robustness function is:

$$\rho(\boldsymbol{x}) = s(\boldsymbol{x}), \tag{3.33}$$

where s is a nonlinear function.

3.4.2. Predicates in stlpy

stlpy treats the agent as a point in an 2-dimensional plane and utilizes predicates to model the relationship between this point and various shapes. Four commonly employed predicates include: inside a rectangle, outside a rectangle, inside a circle, and outside a circle.

Inside a Rectangle

We introduce a predicate $inside(rectangle)$ to check if the position of the agent is inside the rectangle. The predicate $inside(rectangle)$ is composed by 4 atomic predicates:

$$inside(rectangle) = right \wedge left \wedge bottom \wedge top, \tag{3.34}$$

where $right, left, bottom, top$ refer to 4 atomic predicates. Their robustness is the distance to the respective boundaries of the rectangle:

$$\begin{aligned} \rho^{right} &= \mu(s_x, s_y) = s_x - x_{min}, \\ \rho^{left} &= \mu(s_x, s_y) = x_{max} - s_x, \\ \rho^{bottom} &= \mu(s_x, s_y) = s_y - y_{min}, \\ \rho^{top} &= \mu(s_x, s_y) = y_{max} - s_y, \end{aligned} \tag{3.35}$$

where s_x, s_y are position of the agent and $x_{min}, x_{max}, y_{min}, y_{max}$ are the boundaries of the rectangle, as Fig. 3.10 shows.

The robustness function of $inside(rectangle)$ is given by:

$$\rho^{inside(rectangle)} = \min(s_x - x_{min}, x_{max} - s_x, s_y - y_{min}, y_{max} - s_y). \tag{3.36}$$

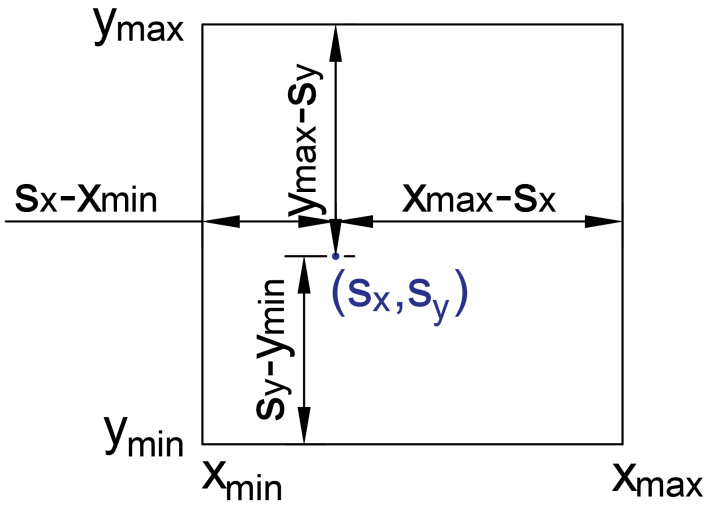


Figure 3.10.: A point inside a rectangle.

Outside a Rectangle

Similarly, we introduce a predicate $outside(rectangle)$ to check if the position of the agent is inside the rectangle. $outside(rectangle)$ is the negation of $inside(rectangle)$:

$$outside(rectangle) = \neg inside(rectangle) = \neg right \vee \neg left \vee \neg bottom \vee \neg top. \tag{3.37}$$

The robustness function of $outside(rectangle)$ is given by:

$$\rho^{outside(rectangle)} = \max(x_{min} - s_x, s_x - x_{max}, y_{min} - s_y, s_y - y_{max}). \tag{3.38}$$

Inside a Circle

The $inside(circle)$ predicate is used to to check if the position of the agent is inside a circle. This nonlinear predicate has the following robustness function:

$$\rho^{inside(circle)} = R^2 - (s_x - x_0)^2 - (s_y - y_0)^2, \tag{3.39}$$

where R is the radius of the circle and (x_0, y_0) is the position of circle center.

Outside a Circle

Similarly, we introduce a predicate $outside(circle)$ to check if the position of the agent is outside a circle. From the definition, we can get $outside(circle) = \neg inside(circle)$. The robustness function of is given by:

$$\rho^{outside(circle)} = (s_x - x_0)^2 + (s_y - y_0)^2 - R^2. \tag{3.40}$$

3.4.3. Predicates in CommonRoad

We treat the agent as a rectangle in CommonRoad. In this section, we use various predicates to model the different relationships between various shapes and objects present on the ego vehicle. Due to the limitations in solely using position components in the dynamic model, it is crucial to include considerations regarding objects on the ego vehicle. For instance, we might consider the scenario where the geometric center of the ego vehicle resides within a certain area.

a Circle Outside a Circle

Different from stlpy, we use $outside(circle_1, circle_2)$ to represent a circle present on the ego vehicle outside another circle. $circle_1$ represents the circle on the ego vehicle and $circle_2$ represents the outside circle. The robustness function is:

$$\rho^{outside(circle_1,circle_2)} = \sqrt{(s_x + x_1 - x_2)^2 + (s_y + y_1 - y_2)^2} - R_1 - R_2, \tag{3.41}$$

where (x_1, y_1) is the relative position between $circle_1$ and the dynamic states position, (x_2, y_2) is position of $circle_2$ in the global coordinate, and R_1, R_2 are radius of $circle_1$ and $circle_2$, respectively, as Fig. 3.11 shows.

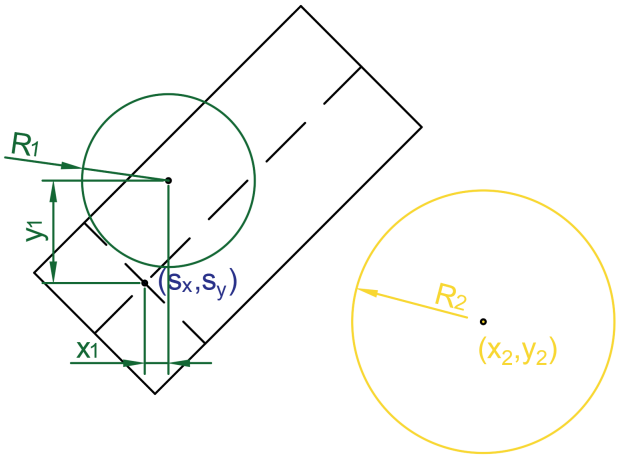


Figure 3.11.: $circle_1$ outside $circle_2$. The green $circle_1$ is rigidly attached to the ego vehicle which is the black rectangle. The yellow $circle_2$ has a different motion pattern with the ego vehicle, such as remaining stationary or moving with the other vehicles. The predicate $outside(circle_1, circle_2)$ is used to check if $circle_1$ remain outside of $circle_2$.

The preference for utilizing the distance to the circle as a measure of robustness arises from unifying the unit of the robustness. In the following introduced predicates, we also use the distance as the robustness.

Avoid Collision between Two Vehicles

Vehicles typically boast an approximate rectangular shape, but can be sufficiently represented by a set of overlapping circular disks. For example, a rectangle of length L and width W can be covered by n circles of radius [52]

$$R = \sqrt{\frac{L^2}{4n^2} + \frac{W^2}{4}} \tag{3.42}$$

placed at distance

$$D = 2\sqrt{R^2 - \frac{W^2}{4}}. \tag{3.43}$$

The n circles are uniformly positioned along the center line of the rectangle's longer side, as illustrated in Fig. 3.12.

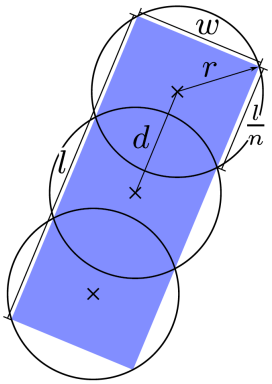


Figure 3.12.: Approximations of vehicle shape with three disks [52].

Then, the predicates $avoidcollision(rectangle)$ can be written as:

$$avoidcollision(rectangle) = \bigwedge_{i,j} outside(circle_i, circle_j),\ i = 1, 2, ..., n,\ j = 1, 2, ..., n. \tag{3.44}$$

Reach Rectangle Goal Area

In CommonRoad, many scenarios present the goal area in the form of a rectangle. Unlike stlpy, this rectangle are not axis aligned, as illustrated in Fig. 3.13. It is noteworthy that reaching the goal area is defined as the center position arriving at the goal area. Thus, we employ the center position of the ego vehicle instead of the dynamic states position. The center position of the ego vehicle is expressed as:

$$(x_c, y_c) = (s_x + b_0 \cos(\Psi_1), s_y + b_0 \sin(\Psi_1)), \tag{3.45}$$

where Ψ_1 refers to the orientation of the vehicle.

We use $reach(rectangle)$ to represent the predicate "Reach Rectangle Goal Area". Similar to $inside(rectangle)$, the predicate $reach(rectangle)$ is composed by 4 atomic predicates:

$$inside(rectangle) = right \wedge left \wedge bottom \wedge top, \tag{3.46}$$

where $right, left, bottom, top$ refer to atomic predicates. Their robustness is the distance to the respective boundaries of the rectangle. We assume that Ψ_2 is the orientation of the rectangle goal area, and that (x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4) are the vertices of the rectangle goal area.

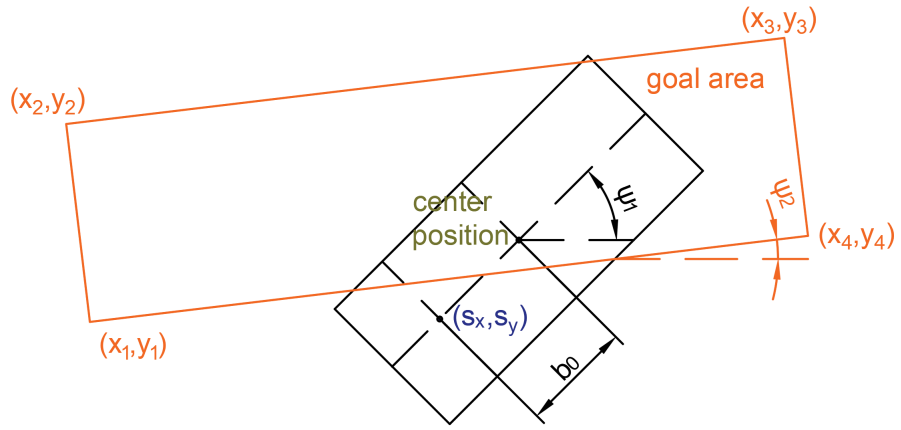


Figure 3.13.: Geometric center of the ego vehicle inside a rectangle. The orange rectangle is the goal area. $reach(rectangle)$ is the predicate used to check if the geometric center position is inside the rectangle goal area.

The robustness function of $reach(rectangle)$ is:

$$\rho^{reach(rectangle)} = \min \begin{pmatrix} x_c\cos(\Psi_2) + y_c\sin(\Psi_2) - x_1\cos(\Psi_2) - y_1\sin(\Psi_2) \\ x_c\sin(\Psi_2) - y_c\cos(\Psi_2) - x_2\sin(\Psi_2) + y_2\cos(\Psi_2) \\ -x_c\cos(\Psi_2) - y_c\sin(\Psi_2) + x_3\cos(\Psi_2) + y_3\sin(\Psi_2) \\ -x_c\sin(\Psi_2) + y_c\cos(\Psi_2) + x_4\sin(\Psi_2) - y_4\cos(\Psi_2)) \end{pmatrix}. \tag{3.47}$$

Inside Boundary

The path of the ego vehicle should be confined within the lane boundaries. To represent these boundaries, we apply two nonlinear predicates, each for the left and right boundary of a lane. The boundaries are fitted using a polynomial function:

$$y = \sum_{i=0}^{n} a_i x^i, \tag{3.48}$$

where n is the regression polynomial degree, a_i are the regression polynomial coefficients, and (x, y) is the point on the boundary.

We use $inside(boundary)$ to represent the "Inside Boundary" predicate. $inside(boundary)$ is composed by nonlinear predicates for the left and right boundary:

$$inside(boundary) = leftboundary \wedge rightboundary. \tag{3.49}$$

The robustness function of $inside(boundary)$ is:

$$\rho^{inside(boundary)} = \min(\sum_{i=0}^{n} a_{left_i} s_x^i - \frac{2}{3} w_0 - s_y, s_y - \sum_{i=0}^{n} a_{right_i} s_x^i - \frac{2}{3} w_0), \tag{3.50}$$

where a_{left_i} and a_{right_i} are the regression polynomial coefficients for left and right boundary respectively, s_x and s_y are position components in the dynamic model (2.33), and w_0 is the width of the ego vehicle. We use $\frac{2}{3} w_0$ instead of $\frac{1}{2} w_0$, because we want to keep enough distance from the boundaries.

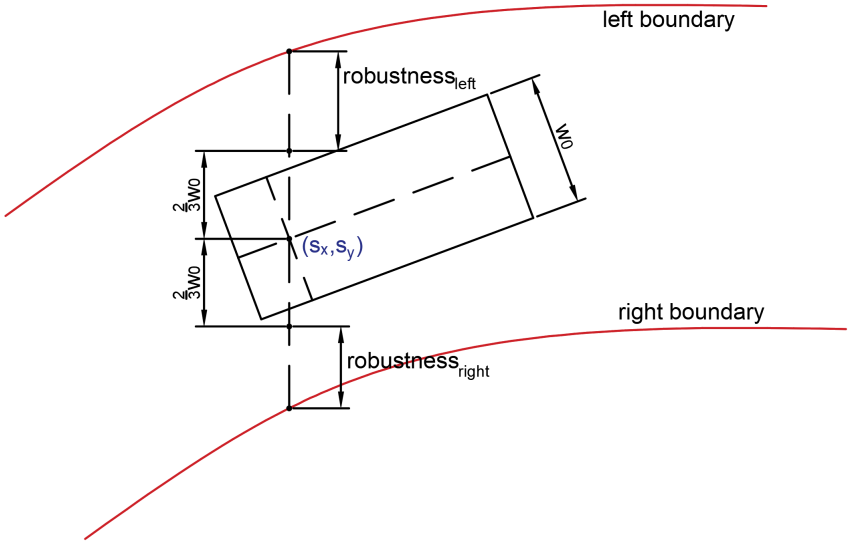


Figure 3.14.: Geometric explanation for robustness of $leftboundary$ and $rightboundary$. The two red lines are left and right boundary. $robustness_{left}$ and $robustness_{right}$ refer to the robustness of predicates $leftboundary$ and $rightboundary$, respectively. The robustness of $inside(boundary)$ is to take a minimum of $robustness_{left}$ and $robustness_{right}$.

Speed Limitation

At times, it becomes essential to incorporate speed limits into the Signal Temporal Logic (STL) formula. We can conveniently define a linear predicate to signify $limit(speed)$. The robustness function for $limit(speed)$ is given by:

$$\rho^{limit(speed)} = v_{max} - v, \tag{3.51}$$

where v_{max} represents the speed limit, and v indicates the speed of the ego vehicle.

4. Experiment

In this chapter, the SCvx algorithm is evaluated with various scenarios, including benchmarks in stlpy [16] and CommonRoad [28]. The algorithms are implemented in *Python* on a computer with an Intel i7 2.70 GHz processor and 16GB of DDR5 4800 MHz memory. We use a linear interpolation from the start configuration to the end configuration for computing an initial trajectory without considering obstacles and then optimize it with our algorithms. The convex programming package CVXPY [53] and the solver ECOS [54] are used to solve convex optimization problems.

4.1. Experiments in stlpy

In this section, the introduced SCvx algorithm is compared with different solving methods provided by stlpy. We use the double integral model for the agent in all experiments.

4.1.1. Comparison of Solvers for Linear Predicates

In this section, the comparison focuses on STL formulas composed of linear predicates. Specifically, the scenarios in this section use squares as obstacles and goal regions. The comparison is conducted among SCvx, MICP and SOS1.

In the following experiments, we set the time interval as $1s$ and set our state and control boundary:

$$0 \leq s_x \leq 10,\ 0 \leq s_y \leq 10,\ -1 \leq \dot{s}_x \leq 1,\ -1 \leq \dot{s}_y \leq 1,$$

$$a^U \leq a_x \leq a^L,\ a^U \leq a_y \leq a^L,$$

where a^L and a^U are the boundaries of the acceleration, and they are defined based on requirements.

The comparison is based on two types of tasks: "Either or" and "Multitask". After introducing these tasks, we present the results with figures and tables.

Either Or

In the "Either or" task, the agent must choose and pass one of two blue squares, avoid the grey square obstacle, and then reach the green goal square, as Fig. 4.1 shows. All positions of squares are fixed in the following experiments.

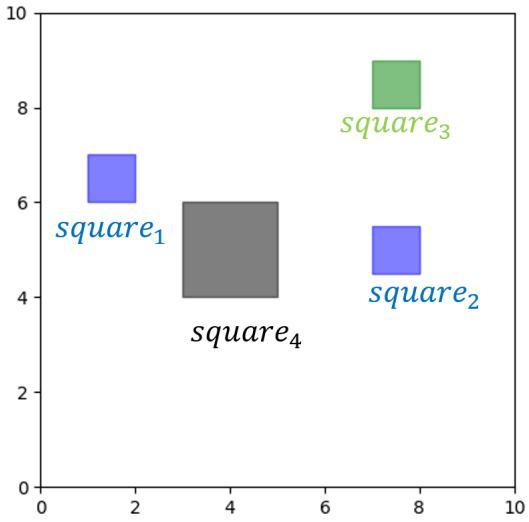


Figure 4.1.: "Either Or" task. The grey square symbolizes an obstacle, while the green square is the goal area. Moreover, the agent must pass through one of the two blue squares.

The STL sentence of the task "Either or" is:

$$F_{[0,T]}(inside(square_1) \vee inside(square_2)) \wedge F_{[0,T]}(inside(square_3)) \wedge G_{[0,T]}(outside(square_4)).$$

In this task, we set the cost function as:

$$\mathcal{C} = ||\boldsymbol{a}|| + 0.1||\boldsymbol{v}|| - \rho_r,$$

where ρ_r is the robustness value.

Multitask

In the "Multitask" task, there are one or several obstacles and multiple target groups, each with its own set of areas. The agent must pass through at least one of the areas in each target group and avoid the square obstacles to complete the task. We assume that there are n_o obstacles, n_t target groups and n_s squares in a target group. Then, The STL sentence of the task "Multitask" is:

$$G_{[0,T]}(\bigwedge_i outside(obstaclesquare_i)) \bigwedge_j F_{[0,T]}(\bigvee_k inside(targetsquare_{n_s j+k})),$$
$$i = 1, 2, ..., n_o, j = 1, 2, ..., n_t, k = 1, 2, ..., n_s.$$

Fig. 4.2 shows a "Multitask" task with $n_o = 1$, $n_t = 5$ and $n_s = 2$.

In this task, we set the cost function as:

$$\mathcal{C} = 0.01||\boldsymbol{a}|| - \rho_r.$$

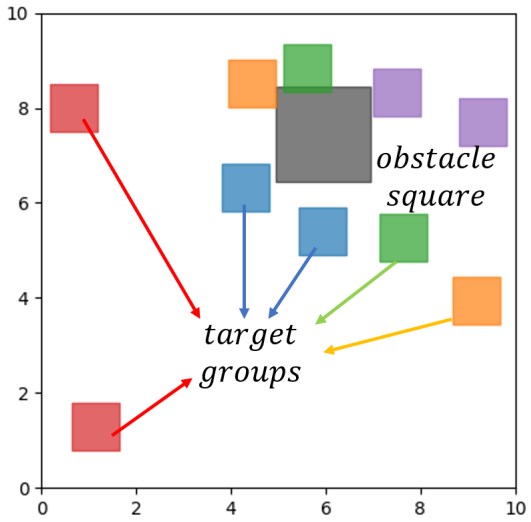


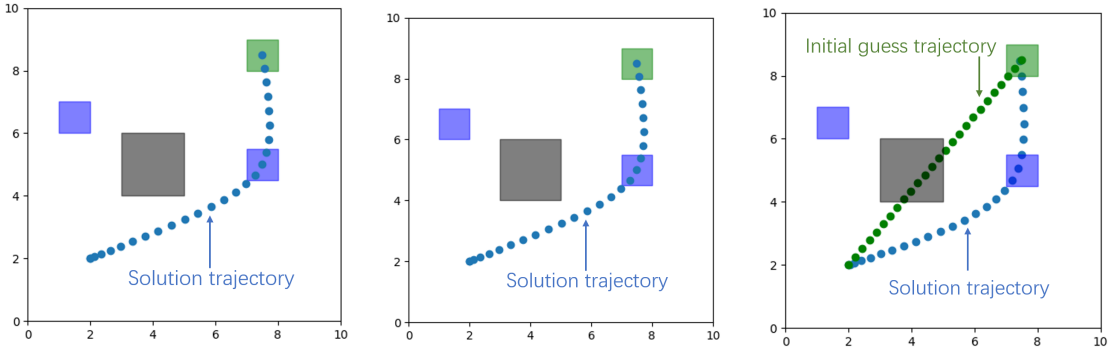
Figure 4.2.: A "Multitask" task. The grey square symbolizes an obstacle, while squares of diverse colors represent target areas. The agent must traverse all color-coded areas. However, for areas with the same color, the agent only needs to pass through one of those.

Result

The specific solution trajectories in two tasks are presented in Fig. 4.3 and 4.4. Other solution trajectories with different parameters can be found in Appendix A.2.2. As a gradient-based method, SCvx requires an initial guess that is close to the solution for faster convergence. Otherwise, a large trust region is needed, which may lead to slower convergence or convergence to an infeasible solution. Fortunately, the linear interpolation from the start point to the goal point is sufficient as an initial guess for SCvx. In contrast, MICP and SOS1 are more tolerant of the initial guess. stlpy uses random numbers as the initial guess for both solvers.

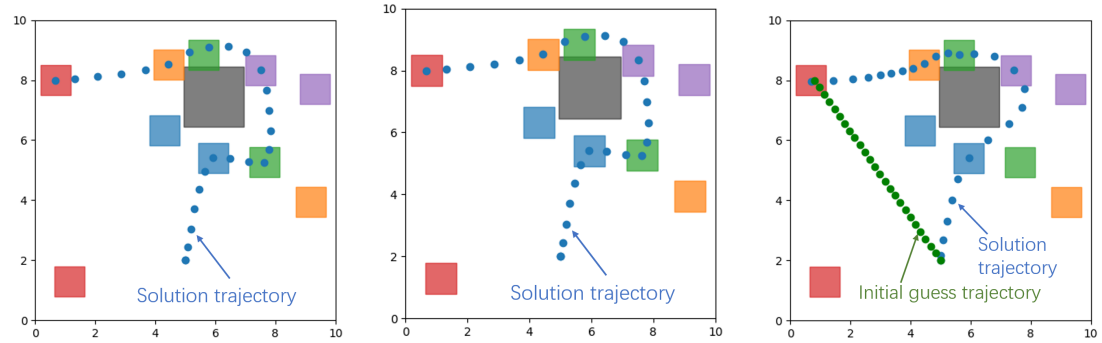
Table 4.1 shows the solve time for three solvers on two tasks with different time horizons. Notably, we use the fixed scenario for the "Multitask" in Fig. 4.2 to obtain the result in table 4.1. For the detailed parameters setting see Appendix A.2.1.

The solve time depicted in table 4.1 underscores that SCvx necessitates substantial compile time, due to the application of CVXPY [53] and ECOS [54] as solvers. These solvers demand much time to confirm the convexity of the problem and subsequently compile it into a solver based on the *C* language. However, considering the convex characteristic of the subproblem (2.20), the need to ascertain its convexity is redundant. Moreover, direct implementation of the algorithm in the *C* or *C++* languages leads to the completion of compilation before the initiation of the entire program. Consequently, within a running program, only the solve time remains necessary. Therefore, the compile time in our implementation can be ignored to some degree.



(a) SOS1 solution. (b) MICP solution. (c) SCvx solution.

Figure 4.3.: Solution of the task "Either or" with the time horizon of 25 and boundary $[-0.5, 0.5]$ by different solvers.



(a) SOS1 solution. (b) MICP solution. (c) SCvx solution.

Figure 4.4.: Solution of task "Multitask" with time horizon of 25 and boundary $[-0.5, 0.5]$ by different solvers.

In the "Either Or" scenario, the data indicate that MICP yields the best results for a time horizon of 25, while SOS1 excels for a time horizon of 50. Yet, when we disregard SCvx's compile time and focus solely on the solving time, SCvx emerges as the overall superior performer.

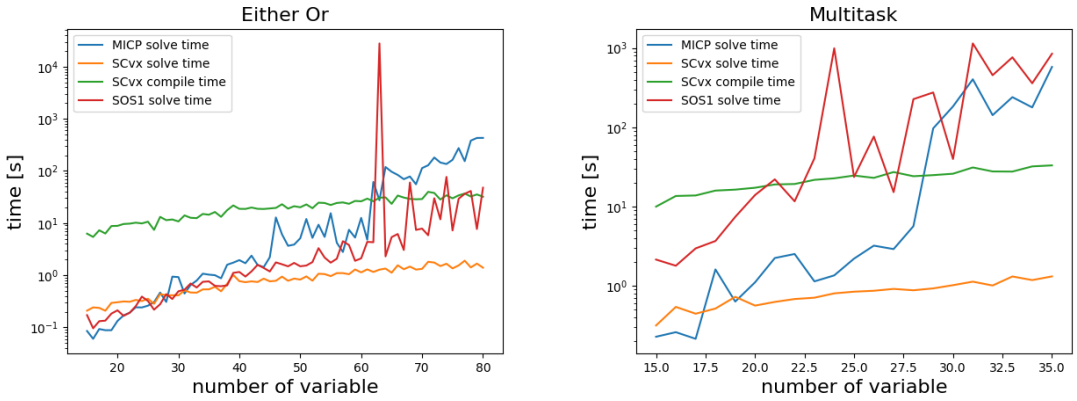
In the "Multitask" scenario, if we ignore the compile time, the solve time of SCvx is smaller than that of MICP and SOS1, especially in situations with more decision variables, including the situation with more time horizons and the situation with more complex STL sentence. This is because MICP and SOS1 need to try out every decision variable in the worst case, while SCvx is a gradient-based method and only optimizes on the gradient direction.

The variation in solve/compile time with an increasing time horizon is illustrated in Fig. 4.5. It is observable that the solve times for MICP and SOS1 exhibit exponential

Table 4.1.: Solve time of solvers for "Either or" and "Multitask".

Tasks	Time horizon	Control boundary	SOS1 solve time	MICP solve time	SCvx solve time + compile time
Either or	25	[-0.5,0.5]	0.27s	**0.26s**	**0.26**+7.72s
Either or	25	[-1,1]	0.27s	**0.22s**	0.32+9.64s
Either or	50	[-0.5,0.5]	**1.37s**	5.30s	**0.81**+20.20s
Either or	50	[-1,1]	**1.73s**	2.97s	**0.85**+20.99s
Multitask	25	[-0.5,0.5]	13.32s	**10.26s**	**0.80**+30.27s
Multitask	25	[-1,1]	21.14s	**16.59s**	**0.84**+30.48s
Multitask	35	[-0.5,0.5]	577.5s	1100s	**1.36+45.24s**
Multitask	35	[-1,1]	254.4s	411.0s	**1.30+43.48s**

growth, whereas SCvx requires only polynomial time.



(a) "Either Or" task.

(b) "Multitask" task.

Figure 4.5.: Number of Variables Vs. Solve/Compile Time for two different tasks.

Table 4.2.: Comparison between solvers for 50 randomly generated scenarios in the "Multitask" task.

Solvers	Time horizon	Success rate	Mean solve time (+compile time)
SOS1	25	**1.0**	1.493s
MICP	25	**1.0**	**1.308s**
SCvx	25	0.82	0.66+23.25s

Table 4.2 presents the results of three solvers for 50 randomly generated scenarios in the "Multitask" task. Different from "Multitask" in table 4.1, we use randomly generated

"Multitask" scenarios with $n_o = 1$, $n_t = 2$ and $n_s = 2$ to obtain the result in table 4.2. The randomly generated "Multitask" scenarios mean that we randomly generate positions of obstacles and target squares. This task involves randomly generating two square obstacles and two target groups with two feasible square areas. The results show that SCvx does not provide feasible solutions for all tasks, often failing to converge in certain instances. However, on average, SCvx's solve time is half that of the other two solvers, despite its longer compile time.

4.1.2. Comparison of Solvers for Nonlinear Predicates

In this section, the comparison focuses on STL formulas composed of nonlinear predicates. Specifically, the scenarios in this section involve circles as obstacles and goal regions. The comparison is between SCvx, Ipopt and Scipy.

In the following experiments, we set the time interval as $0.1s$ and set our state and control boundary:

$$0 \leq s_x \leq 10,\ 0 \leq s_y \leq 10,\ -1 \leq \dot{s}_x \leq 1,\ -1 \leq \dot{s}_y \leq 1,$$

$$-1 \leq a_x \leq 1,\ -1 \leq a_y \leq 1.$$

We set the cost function as:

$$\mathcal{C} = 0.001||\boldsymbol{a}|| - \rho_r.$$

Nonlinear Either Or

In the "Nonlinear Either Or" task, the agent must choose and pass one of two circles, avoid collision with circular obstacles and then reach the circular goal area. We assume that there are n_o obstacles. Then, the STL sentence of the task "Nonlinear Either Or" is:

$$F_{[0,T]}(inside(circle_1) \vee inside(circle_2)) \wedge F_{[0,T]}(inside(circle_3))$$
$$\bigwedge_i G_{[0,T]}(outside(obstaclecircle_i)), i = 1, 2, ..., n_o.$$

Fig. 4.6 shows a "Nonlinear Either Or" scenario with $n_o = 2$.

Two Circle Obstacles

In the "Two Circle Obstacles" task, there are two circular obstacles. Agent must avoid collision with obstacles and finally reach the circular goal area [27]. The STL sentence of the task "Two Circle Obstacles" is:

$$F_{[0,T]}(inside(circle_1)) \wedge G_{[0,T]}(outside(circle_2) \wedge outside(circle_3)).$$

Fig. 4.6 shows the "Two Circle Obstacles" scenario. All positions of obstacles and the goal area are fixed in this scenario.

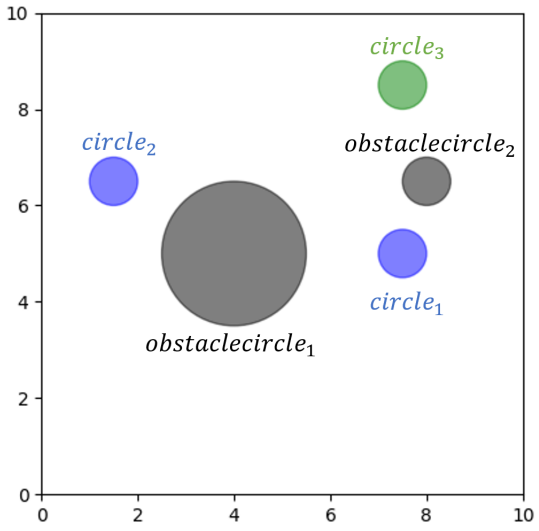


Figure 4.6.: A "Nonlinear Either Or" task. The green circle is the goal area. The grey circles are obstacles. The agent must choose and pass one of two blue circles.

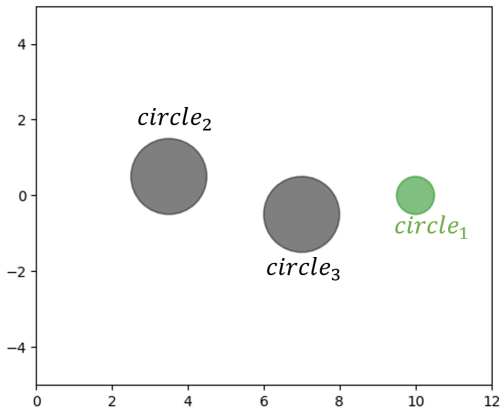


Figure 4.7.: "Two Circle Obstacles" task. The green circle is the goal area. The grey circles are obstacles.

Nonlinear Multitask

In the "Nonlinear Multitask" task, there are a circular obstacle, a circular goal area, and two target groups, each with two circles. The agent must pass through at least one of the areas in each target group and avoid the circular obstacle to reach the goal. The STL sentence of the task "Two Circle Obstacles" is:

$$F_{[0,T]}(inside(circle_1)) \wedge F_{[0,T]}(inside(circle_2) \vee inside(circle_3))$$
$$\wedge F_{[0,T]}(inside(circle_4) \vee inside(circle_5)) \wedge G_{[0,T]}(outside(circle_6) \wedge outside(circle_7)).$$

Fig. 4.8 shows a "Nonlinear Either Or" scenario with $n_o = 2$. In this task, positions of circular obstacles and circular target areas are randomly generated.

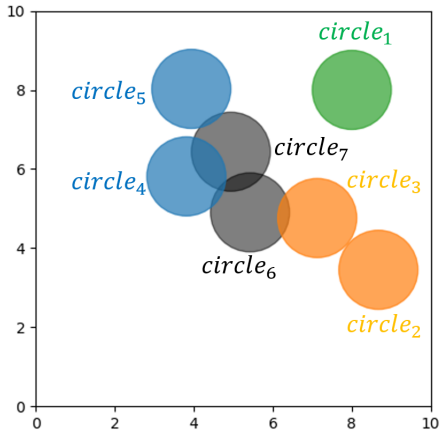


Figure 4.8.: A "Nonlinear Multitask" task. The green circle is fixed as the goal area. The positions of grey circular obstacles and colorful circular target areas are randomly generated. The agent must pass through at least one orange circle and one blue circle and reach the green circle.

Result

The solution trajectories for tasks "Nonlinear Either Or" and "Two Circle Obstacles" can be found in Appendix A.2.2. All three solvers, namely Ipopt, Scipy, and SCvx, successfully accomplish these tasks. Information on the solve time for the "Nonlinear Either Or" and "Two Circle Obstacles" tasks, as got by the three solvers, is provided in table 4.3.

Table 4.3.: Solve time of solvers for "Nonlinear Either Or" and "Two Circle Obstacles".

Tasks	Time horizon	Ipopt solve time	Scipy solve time	SCvx solve time + compile time
Nonlinear Either Or	25	**0.38s**	0.74s	**0.27**+6.05s
Two Circle Obstacles	40	3.38s	2.90s	**0.15 + 2.60s**

Table 4.3 indicates that SCvx outperforms other methods when dealing with the "Two circle obstacles" scenario, though it suffers from a considerably slow compilation speed in the "Nonlinear Either Or" scenario. The Ipopt solver has an unstable solve time. It shines in the "Nonlinear Either Or" task, while it exhibits the poorest performance in the "Two Circle Obstacles" task.

Notably, Scipy tends to get trapped in local minima. Within the framework of stlpy,

the Scipy solver does not impose any boundaries for state and control input. This is evidenced in Fig 4.9, where the agent doesn't halt upon reaching the goal after optimization using Scipy. Moreover, it doesn't incorporate a constraint for robustness greater than 0, causing the solution trajectory to run dangerously close to obstacles.

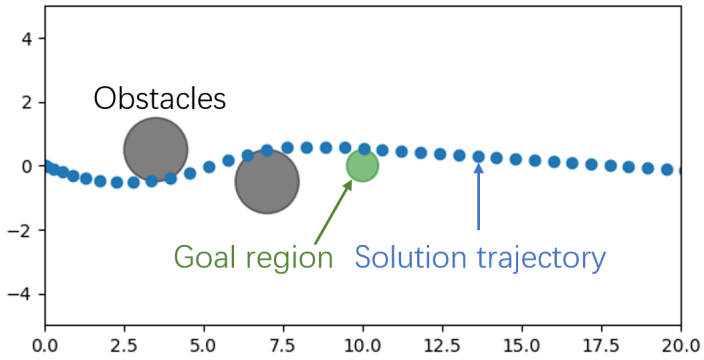
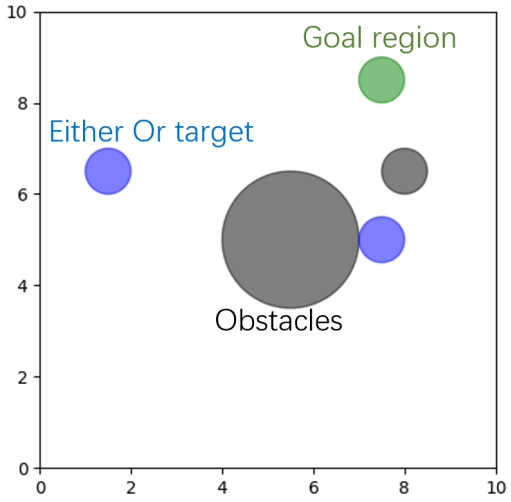
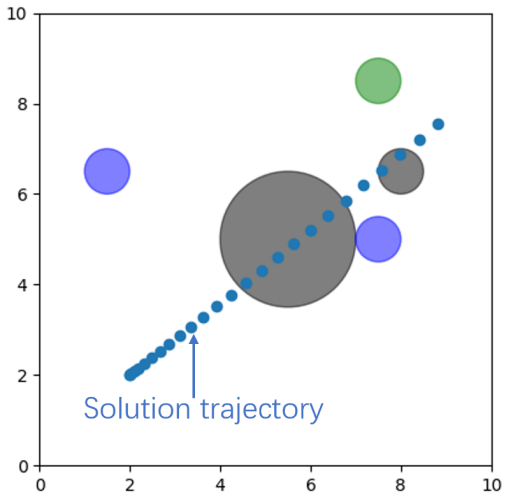


Figure 4.9.: Scipy solution for "Two Circle Obstacles" scenario.

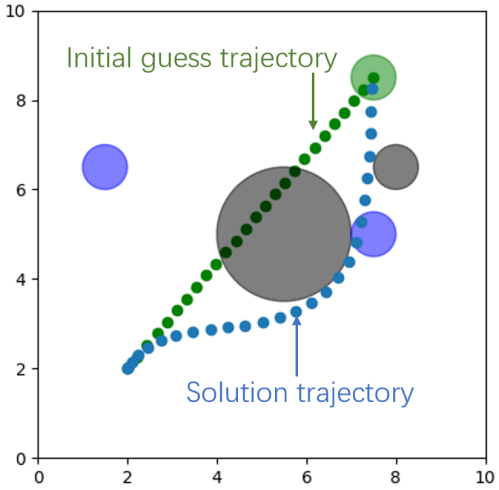
In addition, both Ipopt and Scipy have proven unstable in handling nonlinear problems. Even slight modifications to the "Nonlinear either or" scenario result in a failure to find a solution, as demonstrated in Fig 4.10.



(a) Ipopt fail to provide the solution for the changed "Nonlinear Either Or" scenario.



(b) Scipy solution.



(c) SCvx solution.

Figure 4.10.: A scenario Ipopt cannot solve. Ipopt fails to give the solution and the solution of Scipy is not feasible, while SCvx provides a feasible solution.

Table 4.4 presents the performance of three solvers in 50 randomly generated scenarios for the "Nonlinear Multitask" task. The analysis of the results reveals that SCvx boasts the highest success rate compared to the other solvers. Moreover, the solve time of SCvx is also dramatically lower than that of the other two nonlinear solvers.

Table 4.4.: Comparison between solvers for 50 randomly generated scenarios in the "Nonlinear Multitask" task.

Solvers	Time horizon	Success rate	Mean solve time (+compile time)
Ipopt	25	0.64	5.262s
Scipy	25	0.66	**1.044s**
SCvx	25	**0.96**	**0.40** + 10.88s

4.1.3. Suboptimality of SCvx

As discussed in section 2.3.2, it is important to acknowledge that the convergence of the SCvx algorithm to the global optimum cannot be assured. A pertinent example is when the obstacle in consideration is circular. The introduced function in section 3.4.2 denoting robustness outside of a circle is given by:

$$\rho(\boldsymbol{x}) = (s_x - x_0)^2 + (s_y - y_0)^2 - R^2, \tag{4.1}$$

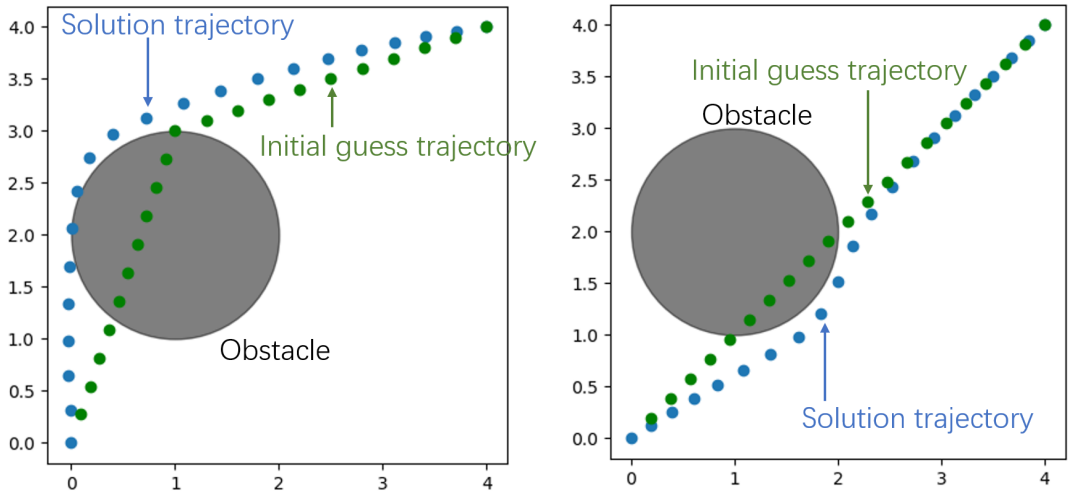
where R is the radius of the circle.

Accordingly, the gradient of the robustness function can be expressed as:

$$\nabla_{s_x,s_y}\rho(\boldsymbol{x}) = [2(s_x - x_0), 2(s_y - y_0)]^\top \tag{4.2}$$

In instances where $s_x > x_0$, the gradient along the direction of s_x remains consistently positive, resulting in a continuous increase in s_x when maximizing $\rho(\boldsymbol{x})$. This holds true even if the optimum lies at $s_x < 0$, as s_x will not fall within this range. Analogously, this is also applicable for scenarios where $s_x < x_0$ and for s_y. The experiment of suboptimality is shown in Fig. 4.11. The solution always converges to the same side of the initial guess trajectory. For instance, if the initial guess trajectory lies to the right of the circular obstacle, the solution trajectory will be positioned on the right side of the obstacle. The same principle holds true for trajectories on the left side.

Therefore, although SCvx can solve the nonconvex problem in a polynomial time, it cannot guarantee to find the global optimum of the nonconvex problem, while MICP can guarantee to find it [15]. This is because SCvx linearizes the nonconvex problem and loses the information about nonconvex constraints. The price paid for solving the nonconvex problem in polynomial time is a loss of optimality. In other words, we do not solve the NP-hard problem in polynomial time but instead make a trade-off between computational efficiency and solution optimality.



(a) The initial guess trajectory at the left side of the circle.

(b) The initial guess trajectory at the right side of the circle.

Figure 4.11.: Different trajectories based on different initial trajectories. An agent sidesteps the circular obstacle to arrive at the predetermined destination. The eventual path taken depends on the initial trajectory selected. While the optimal route involves circumventing the obstacle from the right, should the initial trajectory be designated on the left, the solution will correspondingly lie on the left side.

4.1.4. Influence of Parameters in SCvx

There are several parameters required to be set in Algorithm 1. In this section, we will discuss the influence of those parameters on the solutions and the robustness of the solutions. The specific parameters setting can be found in Appendix A.2.1.

Threshold Factors of Cost Function

As highlighted in section 2.3.1, the disparity between the nonlinear penalized cost $\mathcal{J}$ and the convex subproblem cost $\mathcal{L}$ is indicative of the accuracy achieved through the linearization process. Moreover, the threshold factors of the cost function play a pivotal role in deciding whether a solution is accepted or rejected, and in determining whether the trust region radius expands or contracts.

The rejection threshold factor ψ_0 is usually assigned a value of 0. This is because $\delta^{(i)}$ in Algorithm 1 ranges between 0 and 1 for a successful linearization process [25, Lemma 3.11].

The convergence speed of Algorithm 1 is influenced by the contracting threshold factor ψ_1 and the expanding threshold factor ψ_2. A larger ψ_1 prompts contraction of the trust region radius, even when the solution's accuracy is deemed satisfactory. This

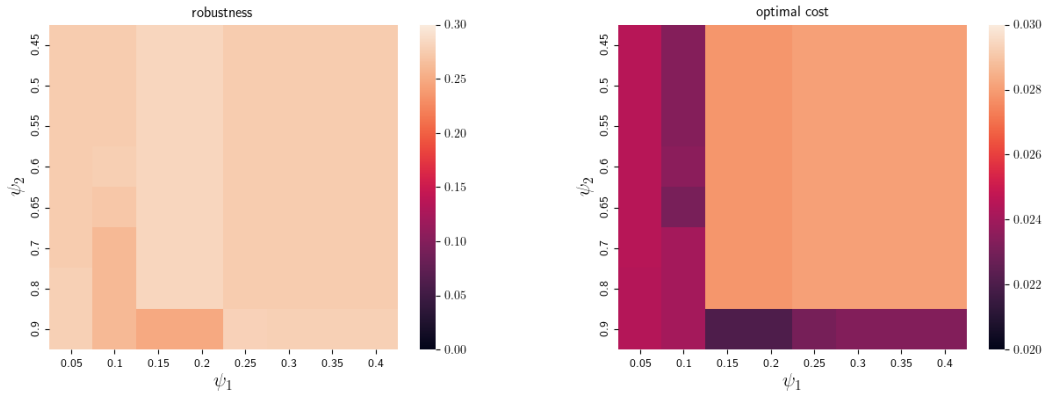
results in a rapid reduction of the trust region radius, leading to a very small radius, even if our algorithm did not yet converge. Under these conditions, numerous iterations of the convex subproblem are required to achieve a converged solution.

Conversely, a smaller ψ_1 implies that the trust region radius doesn't contract, even in the presence of poor solution accuracy, causing the trust region radius to reduce at a slower pace. Consequently, we are left with a significantly large trust region radius, even when our algorithm is nearing the solution. In this situation, we would still need to wait for several iterations to fulfill the convergence condition, i.e., the trust region radius being adequately small.

The expanding threshold factor ψ_2 operates similarly to the contracting threshold factor ψ_1. Therefore, determining a balanced intermediary value for both ψ_1 and ψ_2 is of utmost importance.

Fig. **??** illustrates the heat map of solutions' robustness and optimal cost based on ψ_1 and ψ_2. Since the optimal cost integrates the negative robustness, the heat maps of robustness and optimal cost present contrasting color schemes. The variations in ψ_1 and ψ_2 yield negligible differences in final robustness and optimal cost. There is no combination of ψ_1 and ψ_2 that results in a failed solution.

However, alterations in ψ_1 and ψ_2 significantly affect the time required to find a solution. Fig. 4.13 illustrates the heat map of solve time based on ψ_1 and ψ_2. In Fig. 4.13, we can see that smaller values of ψ_1 and ψ_2 contribute to a longer solve time.



(a) Robustness of solutions. A lighter color signifies a better result.

(b) Optimal cost of solutions. A darker color signifies a better result.

Figure 4.12.: Solutions heat maps based on the parameters ψ_1 and ψ_2 for "Either Or" task.

Scaling Factors of Trust Region Radius

Scaling factors associated with the trust region radius are used to either expand or contract the trust region. When the expansion factor β is too large, the trust region

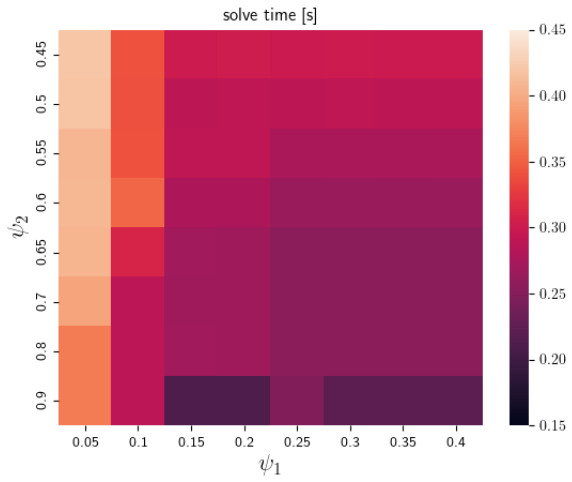


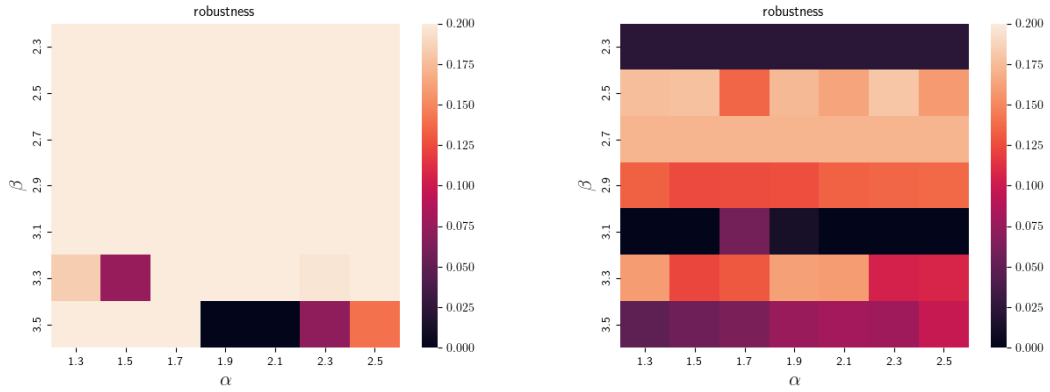
Figure 4.13.: Solve time heat map based on the parameters ψ_1 and ψ_2 for "Either Or" task. A darker color signifies a better result.

radius might expand excessively after an iteration that delivers a relatively accurate solution. This could result in less precise solutions in the ensuing iterations. In certain cases, an infeasible solution might be accepted due to the enlarged trust region radius, and this could hinder optimization in the subsequent convex subproblem iteration with a reduced trust region radius. Conversely, if the expansion factor β is too small, the trust region radius might not sufficiently expand, even when the solutions exhibit high accuracy, resulting in a slower convergence rate.

On the other hand, if the contraction factor α is overly large, the trust region radius might contract excessively following an iteration with a relatively inaccurate solution. This could result in a decreased convergence speed in the subsequent iterations. Inversely, if the contraction factor α is too small, the trust region radius might not shrink efficiently, even when the solutions present poor accuracy. This could lead to a continued rejection of solutions, culminating in a slower convergence rate.

As illustrated in Fig. 4.14, certain combinations of α and β can lead to optimization failure, indicated by an inability to find the solution of the convex subproblem. This situation is signaled by negative robustness values, with the darkest color in Fig. 4.14 signifying this negative robustness.

For tasks with simpler STL formulas such as "Either Or", most combinations of α and β yield similar solutions. However, optimization may fail with combinations of some α values and large β values, such as $\alpha = 1.9, \beta = 3.5$ and $\alpha = 2.1, \beta = 3.5$. In contrast, for tasks with more complex STL formulas like "Multitask", both large and small β values can lead to optimization failure. This underscores the importance of assigning a suitable value to β. Although α does not cause optimization failure with a suitable β value, it can still influence the robustness of the solution. Besides, smaller values of β contribute to a shorter solve time, as Fig. 4.15 shows.



(a) "Either Or" task.

(b) "Multitask" task.

Figure 4.14.: Robustness heat maps based on the parameters α and β.

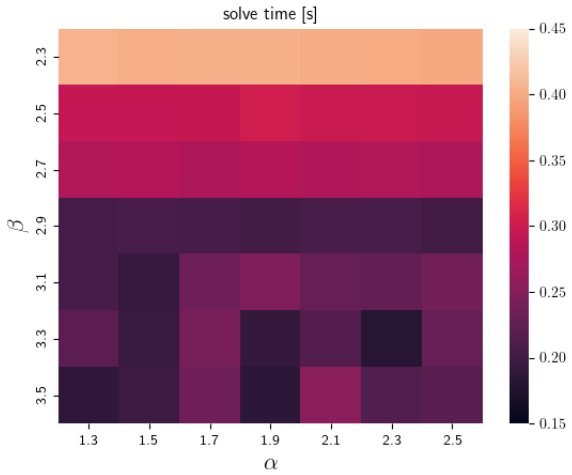


Figure 4.15.: Solve time heat map based on the parameters α and β for "Either Or" task.

Penalty Weight and Initial Trust Region Radius

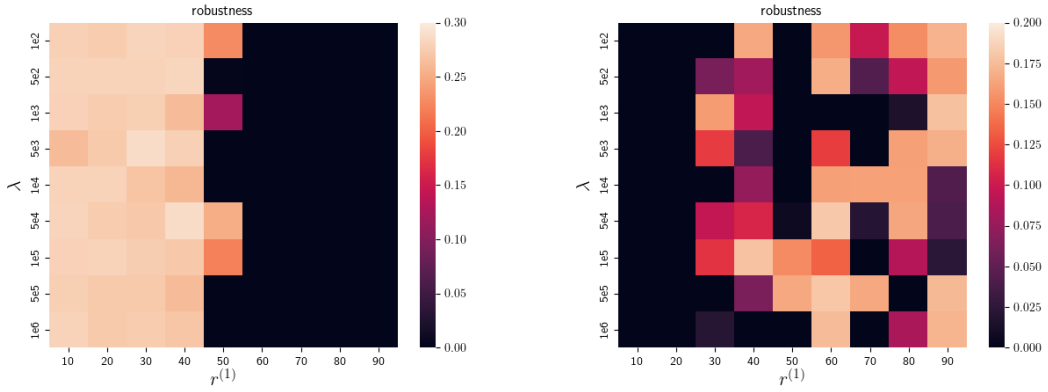
The penalty weight λ serves to restrict the magnitude of the virtual control input to maintain its smallness. Elevating the value of λ can enhance the reliability of the solution, albeit at the risk of causing an optimization failure. Conversely, reducing the penalty weight value could render the solution unattainable.

The initial trust region radius $r^{(1)}$ exerts a significant influence on the ultimate solution. To ensure the existence of a solution for the convex subproblem, the initial trust region must be adequately large. However, care must be taken to avoid overly large values for the initial trust region radius, as this could result in the acceptance of an infeasible

solution. Such an outcome could trigger an optimization failure in the next several steps.

Fig. 4.16 illustrates the heat map of solutions' robustness based on $r^{(1)}$ and λ for the tasks "Either Or" and "Multitask". As depicted in Fig. 4.16(a), in the "Either Or" task, the penalty weight has a small influence on the ultimate solution, whereas an excessively large initial trust region radius can induce optimization failure. The corresponding trajectories are displayed in Fig. 4.17, revealing that trajectories with $r^{(1)} \leq 50$ are markedly similar, while those with $r^{(1)} \geq 60$ are unsuccessful in task completion.

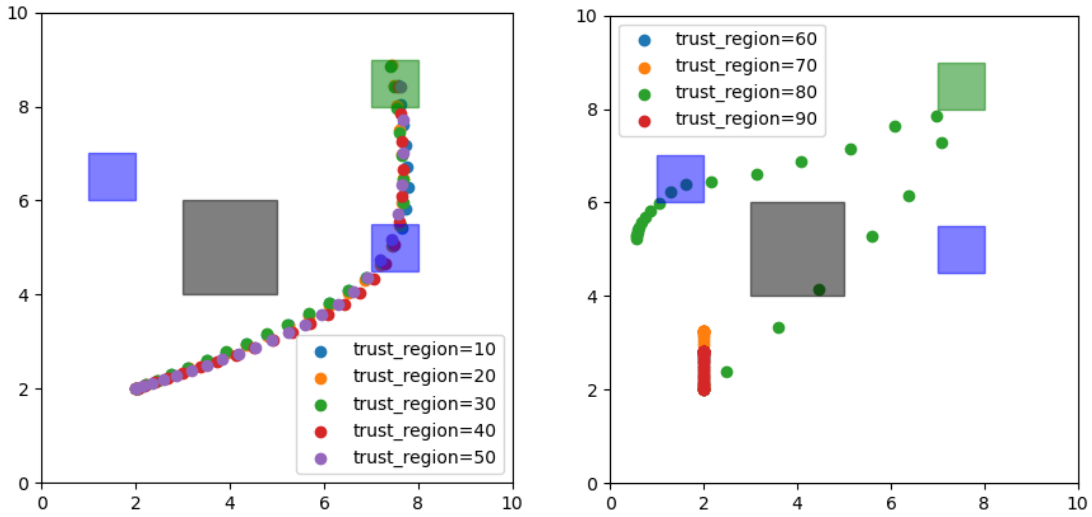
The "Multitask" task presents a slightly more intricate scenario. As demonstrated in Fig. 4.16(b), specific penalty weight values may also trigger optimization failure. The trajectories with $\lambda = 5 \times 10^4$, presented in Fig. 4.17, indicate that trajectories with $r^{(1)} = 10$ and $r^{(1)} = 20$ are unsuccessful, while those with $r^{(1)} = 50$ and $r^{(1)} = 70$, albeit suboptimal, accomplish the tasks. Other trajectories with varying $r^{(1)}$ exhibit similar patterns.



(a) "Either Or" task.

(b) "Multitask" task.

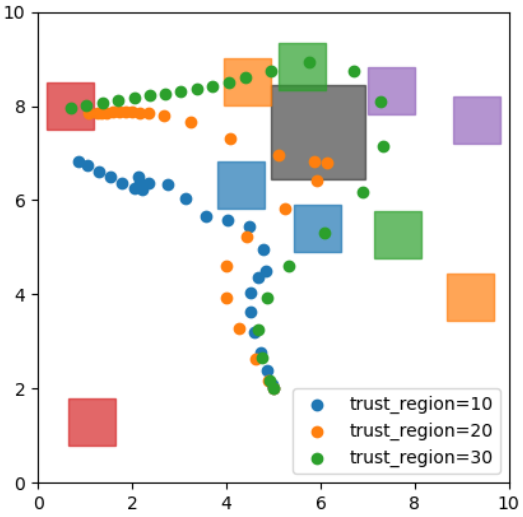
Figure 4.16.: Solutions heat map based on the parameters $r^{(1)}$ and λ.



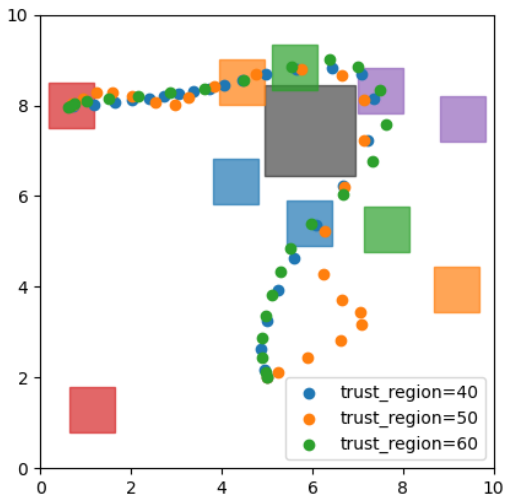
(a) Trajectories from $r^{(1)} = 10$ to $r^{(1)} = 50$.

(b) Trajectories from $r^{(1)} = 60$ to $r^{(1)} = 90$.

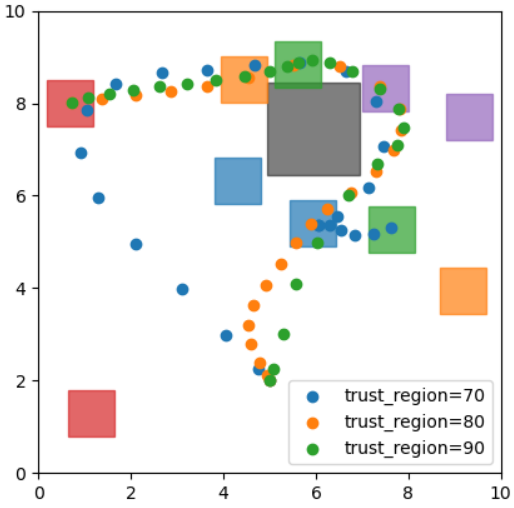
Figure 4.17.: Trajectory based on different $r^{(1)}$ for "Either Or" task.



(a) Trajectories from $r^{(1)} = 10$ to $r^{(1)} = 30$.



(b) Trajectories from $r^{(1)} = 40$ to $r^{(1)} = 60$.



(c) Trajectories from $r^{(1)} = 70$ to $r^{(1)} = 90$.

Figure 4.18.: Trajectory based on different $r^{(1)}$ for "Multitask" task.

4.2. Experiments in CommonRoad

In this section, the introduced SCvx algorithm is evaluated with traffic scenarios from the CommonRoad benchmark suite [28]. The motion assumptions of the ego vehicle and general simulation settings are listed in table 4.5. Moreover, CommonRoad provides various utilities for detecting the feasibility of trajectories. We use those utilities to verify whether our solutions are feasible.

Table 4.5.: Motion assumptions of the ego vehicle.

Parameter	Symbol	Value	Unit
vehicle length	l_0	4.508	m
vehicle width	w_0	1.608	m
wheelbase	l_{wb}	2.578	m
distance from center of gravity to rear axle	b_0	1.422	m
velocity limitation	$[v^L, v^U]$	$[0.0, 50.8]$	m/s
maximum acceleration	a_{max}	11.5	m/s^2
steering angle limitation	$[\delta^L, \delta^U]$	$[-1.066, 1.066]$	rad
steering velocity limitation	$[v_\delta^L, v_\delta^U]$	$[-0.4, 0.4]$	rad/s
initial time	t_0	0.0	s
time step size	Δt	0.1	s

We will use the predicates introduced in section 3.4.3. We employ a third-degree polynomial fit for $inside(boundary)$. Additionally, we approximate a rectangle for $avoidcollision(rectangle)$ using three circular disks. We also define the cost function for the following experiments as:

$$\mathcal{C} = 0.1||\boldsymbol{u}|| - \rho_r,$$

where $\boldsymbol{u}$ denotes the control input for the vehicle dynamic model (2.34), and ρ_r represents the robustness value, as previously defined. The detailed parameters of the following experiments can be found in Appendix A.3.

4.2.1. Scenarios with Static Obstacles

In this section, we evaluate SCvx on scenarios with static obstacles.

Reach Avoid

The first evaluated scenario is a reach avoid task[1]. In this scenario, a static obstacle blocks the ego vehicle's driveway. The obstacle represents, e.g., a construction site, or a parked vehicle. The STL formula in this scenario is:

$$F_{[0,T]}reach(goal) \wedge G_{[0,T]}(inside(boundary)) \wedge G_{[0,T]}(avoidcollision(obstacle)),$$

[1]CommonRoad ID: ZAM_Over-1_1

where the time horizon is $T = 30$.

We mark the occupancy of the ego vehicle in green, the trajectory of geometric center points after optimization in blue, the trajectory of center points before optimization in light green, the fitted boundary in red, the static obstacle in dark red, and the goal region in yellow. Those settings are the same for the following experiments.

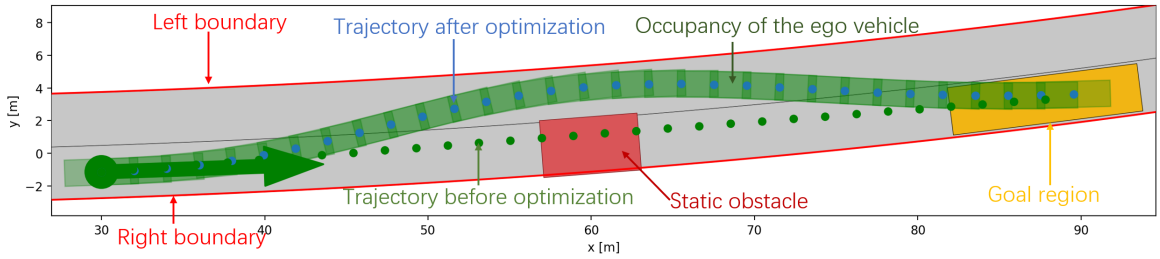


Figure 4.19.: Reach avoid scenario.

Fig. 4.19 presents the solution trajectory for the "Reach Avoid" scenario. Additionally, Fig. 4.20 Shows the motion components $(v, \Psi, v_\delta, a_{long})$ of the solution trajectory over time in this scenario.

Notably, our solution successfully passes the feasibility detection of CommonRoad. As seen in Fig. 4.20, there is no collision between the ego vehicle and the static obstacle, indicating safe navigation. The ego vehicle remains within the boundary and successfully reaches the goal area within the specified time horizon.

Furthermore, our solution optimizes the robustness to a larger value, as evidenced by the considerable distance maintained between the ego vehicle and the static obstacle. This distance signifies successful avoidance and increased safety.

Lastly, control inputs v_δ and a_{long} are kept minimal throughout, contributing to efficient vehicle control. This can be observed in Fig. 4.20, indicating effective management of vehicle dynamics.

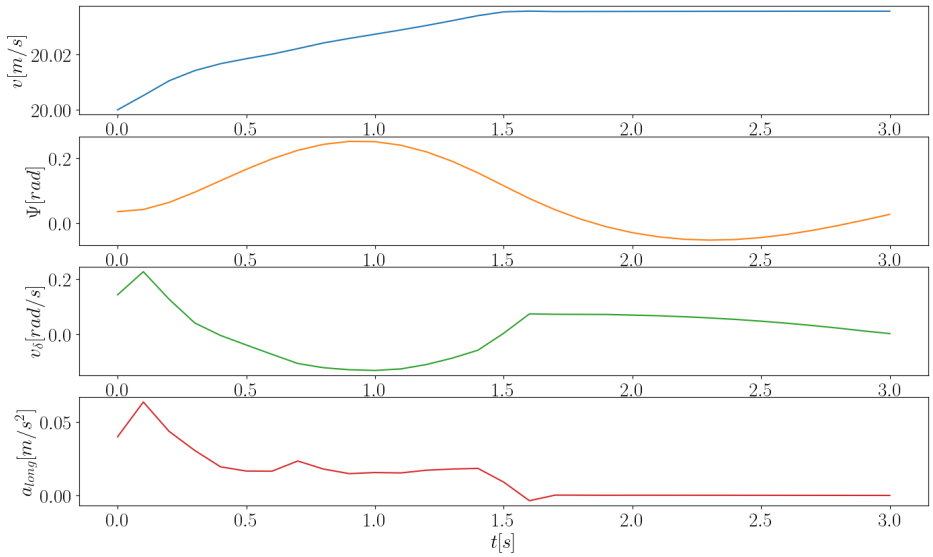


Figure 4.20.: The motion profiles $(v, \Psi, v_\delta, a_{long})$ in scenario "Reach Avoid".

Narrow Passage

The second evaluated scenario is a narrow passage task[2]. Two static obstacles lead to a narrow passage in the middle of the road. The STL formula in this scenario is:

$$F_{[0,T]}reach(goal) \wedge G_{[0,T]}(inside(boundary)) \wedge$$
$$G_{[0,T]}(avoidcollision(obstacle_1) \wedge avoidcollision(obstacle_2)),$$

where the time horizon is $T = 28$.

The resulting trajectory is illustrated in Fig. 4.21. It is noteworthy to point out the failure of the SCvx algorithm in this scenario. Due to the underestimation of the robustness, the final robustness is negative and cannot satisfy the constraint $z_N[\text{end}] \geq 0$ in problem (3.29) with a small trust region radius. This underestimation is attributed to the approximation of a rectangle using three circular disks. As evidenced in Fig. 4.22, while there is no collision between rectangles, a collision between the approximation disks does occur. Nevertheless, despite the failure of SCvx to converge, the solution derived from the final SCvx iteration still successfully passes the feasibility detection of CommonRoad. This outcome is possible because the subproblem remains solvable with a larger trust region radius, and the SCvx algorithm optimizes towards the correct direction during the early several iterations.

[2]CommonRoad ID: ZAM_Urban-3_3_Repair

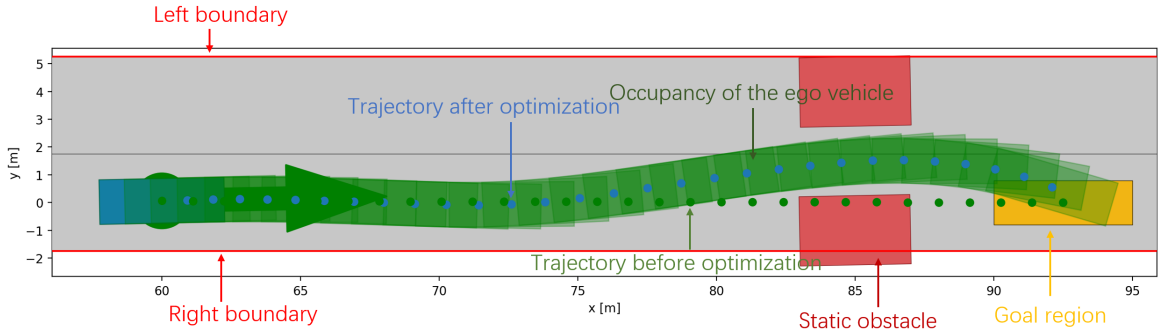


Figure 4.21.: Narrow passage scenario.

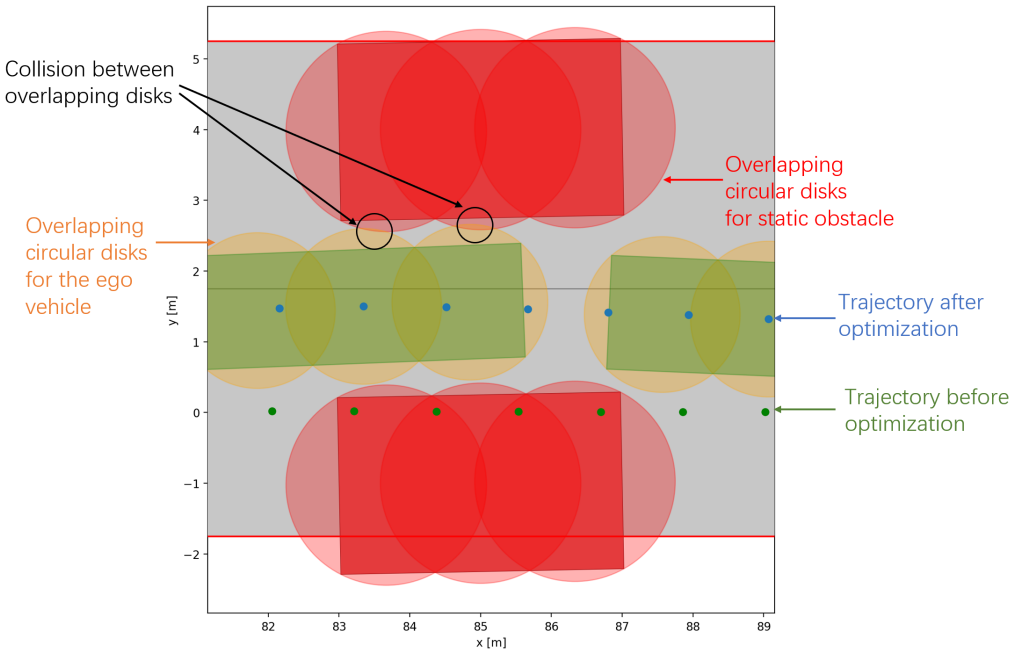


Figure 4.22.: The approximation disks collide in the narrow passage.

4.2.2. Scenarios with Dynamic Obstacles

In this section, we evaluate SCvx on scenarios with dynamic obstacles.

Straight with Dynamic Obstacles

The urban scenario[3], set on a straight path, presents a simple configuration with a static obstacle and two dynamic obstacles. Notably, this scenario lacks a specific goal region.

[3]CommonRoad ID: ZAM_Tutorial-1_2_T-2

Consequently, the ego vehicle's primary task is to maintain its current lane and prevent any collisions. The STL formula in this scenario is:

$$G_{[0,T]}(inside(boundary)) \wedge G_{[0,T]}(avoidcollision(staticobstacle_1)) \wedge$$
$$G_{[0,T]}(avoidcollision(dynamicobstacle_2) \wedge avoidcollision(dynamicobstacle_3)),$$

where the time horizon is $T = 29$.

The solution trajectory is shown in Fig. 4.23. The ego vehicle keeps driving in its current lane because there is no collision risk with other vehicles. This scenario demonstrates the capability of SCvx to solve simple scenarios with dynamic obstacles.

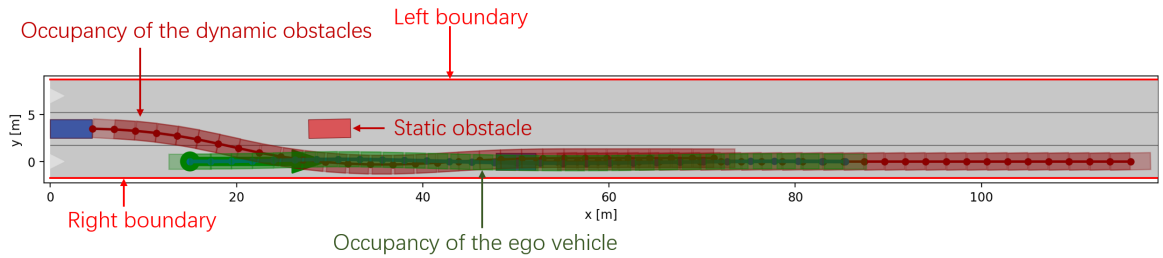


Figure 4.23.: Straight with dynamic obstacles scenario.

T-Intersection

In the T-intersection scenario[4], the ego vehicle needs to avoid collision and reach the goal regions in the T-intersection. The STL formula in this scenario is:

$$F_{[0,T]}reach(goal) \wedge G_{[0,T]}(\bigwedge_i avoidcollision(obstacle_i)), i = 1, 2, 3,$$

where the time horizon is $T = 146$.

In this scenario, the polynomial regression cannot obtain a good function to describe the boundary. However, the goal regions provide enough information about the boundary. Therefore, although we do not add boundary constraints, the ego vehicle does not leave the boundary with the attraction of goal regions. The solution trajectory is shown in Fig. 4.24.

Given the size of the goal area in this scenario, the task completion isn't dependent on the distance the ego vehicle travels. Consequently, the SCvx solution chooses an energy-efficient trajectory. The ego vehicle only needs to reach the forefront of the goal area, thereby minimizing energy expenditure.

Despite this scenario having a longer time horizon compared to other scenarios, SCvx still succeeds in producing a feasible solution, showcasing its ability to adapt and optimize within diverse scenarios.

[4]CommonRoad ID: ZAM_Tjunction-1_288_T-1

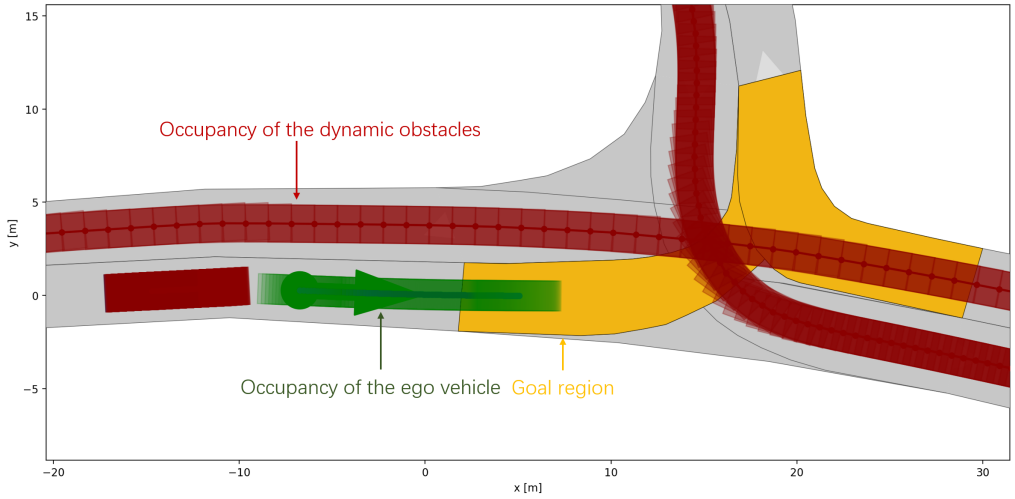


Figure 4.24.: T-Junction scenario.

Keep Safe Distance

For the purpose of safety, the ego vehicle must maintain a safe distance from the vehicle ahead. In a highway scenario[5], the ego vehicle is required to decelerate in order to keep a safe distance from the preceding vehicle, while also steering clear of any other potential collisions. Moreover, due to the presence of speed limitations, the ego vehicle must maintain its speed within a reasonable range. The STL formula used in this scenario is:

$$G_{[0,T]}keepdistance(obstacle_1) \wedge G_{[0,T]}limit(speed)$$
$$\wedge G_{[0,T]}(avoidcollision(obstacle_2) \wedge avoidcollision(obstacle_3)),$$

where the time horizon is $T = 20$ and the limitation of the speed in predicate $limit(speed)$ is $20m/s$. The solution trajectory is shown in Fig. 4.25, and Fig. 4.26 shows the positions of the ego vehicle and other vehicles at the end time $t = 20$.

In this scenario, our solution effectively avoids all potential collisions, maintaining a safe distance from the vehicle in front. Notably, the vehicle drifts to the right at the conclusion of the scenario at time $t = 20$. This could be attributed to two potential factors: the SCvx algorithm may be attempting to maximize the distance from the left boundary, or it might be initiating a lane change to further increase the distance from the leading vehicle. In fact, the changing lane is not what we want, because the vehicle in the right lane is closer to the ego vehicle. Our formula does not include the right front vehicle as keeping the distance target. Changing the keep distance target is required to avoid the undesired lane change. However, it is hard to be described by a continuous function. Therefore, SCvx can still not solve this issue.

[5]CommonRoad ID: DEU_Gar-1_1_T-1

However, this lane change is not desirable, as it brings the ego vehicle closer to the vehicle in the right lane. Our STL formula does not include the vehicle in the right lane as a target to keep distance. To prevent undesired lane changes, it would be necessary to alter the safety distance target, which is difficult to describe with a continuous function. Consequently, SCvx still falls short of addressing this issue.

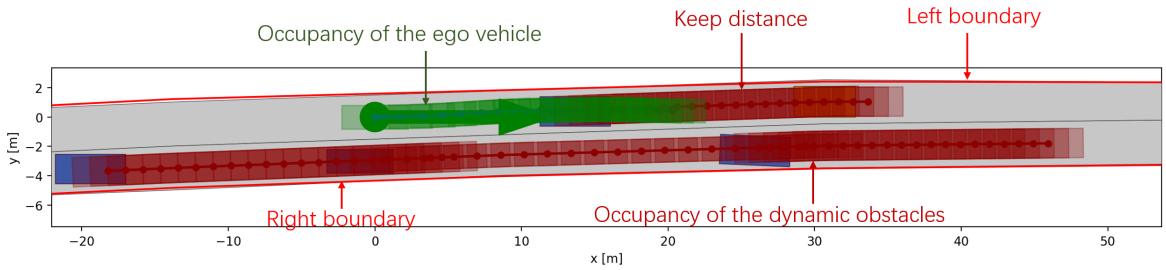


Figure 4.25.: Keep safety distance scenario.

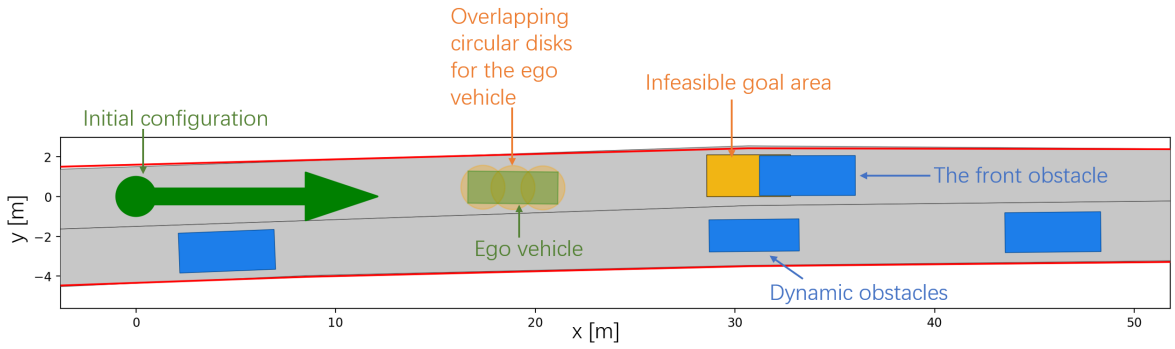


Figure 4.26.: The positions of the ego vehicle and other vehicles at the end time $t = 20$ in the keep safety distance scenario.

4.2.3. Success Rate on Selected Scenarios of CommonRoad

To assess the effectiveness of our algorithm, we conducted a series of tests on 23 distinct scenarios[6] sourced from the CommonRoad. The detailed CommonRoad ID can be found in Appendix A.3.1. These evaluations are designed to measure the success rate of our algorithm across these diverse scenarios, providing a robust evaluation of its performance.

Table 4.6 shows the results of these tests, illustrating our algorithm's success rate, mean compile time and mean solve time over a time horizon of 30.

[6]CommonRoad Data Source: Technische Hochschule Ingolstadt

Table 4.6.: Performance of SCvx on selected scenarios of CommonRoad.

Success rate	Mean solve time	Mean compile time
0.652	0.513s	9.65s

The results demonstrate a strong performance level of our algorithm within the selected CommonRoad scenarios, underscoring its potential for effective use in the field of autonomous driving.

4.3. Summary

The experiments presented in section 4.1 and 4.2 allow us to identify the strengths and limitations of the SCvx algorithm with our encoding method:

1. **Advantages**
   a) The algorithm can converge or fail in polynomial time.
   b) If the algorithm converges, the solution will be the approximation of the local optimum.
   c) Even if the algorithm fails to converge, the solution could still be acceptable.
2. **Disadvantages**
   a) The algorithm cannot guarantee finding the global optimum.
   b) The algorithm is sensitive to the initial guess.
   c) The algorithm tends to overly underestimate robustness.
   d) The algorithm needs the continuous function for each atomic predicate.

If our algorithm successfully converges, it provides a feasible solution within a polynomial timeframe, with feasibility ensured by underestimating the robustness value. However, if the algorithm fails within this timeframe, there remains a sufficient opportunity to resort to alternative solvers. The local optimality is vouched for by the SCvx algorithm. The "Narrow Passage" scenario demonstrates that even in instances where the algorithm does not converge, the solution could still be acceptable, as the solution trajectory passes the feasibility detection.

The limitations, such as the inability to guarantee the discovery of a global optimum and the high sensitivity to the initial guess, are evidenced in Section 4.1.3. The tendency to excessively underestimate robustness is apparent in the "Narrow Passage" scenario. The requirement of a continuous function for each atomic predicate is demonstrated in the keep safe distance scenario.

5. Summary and Outlook

This final chapter evaluates the merits and shortcomings of the method employed in this thesis, while also outlining prospective directions for future research.

5.1. Summary

This thesis applies the SCvx algorithm for trajectory planning problems based on STL specifications on account of its real-time capability. An in-depth analysis of existing encoding methods has been performed, highlighting the gap between these methods and the SCvx algorithm. Consequently, we have developed a novel hybrid encoding method for STL specifications, facilitating the application of the SCvx algorithm to trajectory planning problems characterized by complex STL specifications.

Our encoding approach presents an underestimate of the true robustness, thereby ensuring that upon convergence of the SCvx algorithm, the solution generated will indeed be feasible. Furthermore, we discuss the complexity of the SCvx algorithm in comparison with various MICP solving methods. The polynomial time efficiency of the SCvx algorithm far outweighs the exponential timeframe of MICP-based methods. However, this does not imply a definitive resolution to the NP-hard problem. Instead, our algorithm navigates a trade-off between computational efficiency and solution optimality.

We conducted the final evaluation of our algorithm using scenarios from the stlpy and CommonRoad benchmark suites, which demonstrated its adaptability across a range of tasks. Moreover, we also explored the impact of varying parameters within the SCvx algorithm on the solution based on different tasks.

5.2. Outlook

Given the local optimality of the SCvx algorithm, there is a lack of experiments with well-defined initial guess trajectories. One potential approach to addressing this is through the use of probabilistic roadmap methods for sampling initial trajectories. Evaluating the objective function's optimization in this scenario could help establish our algorithm as a useful tool for smoothing trajectories sampled from probabilistic roadmaps.

Currently, the algorithm is presently implemented in Python, utilizing the convex programming package CVXPY and the solver ECOS. As a result, it suffers from extended compile times, primarily lost on verifying the convexity of constraints. However, given

the linearization process inherent in the SCvx algorithm, the convexity of constraints in subproblems is assured, rendering additional convexity checks unnecessary. Eliminating this component would significantly enhance program efficiency.

Additionally, compilation time includes the period spent on transforming the problem into a C++ form. Thus, direct C++ implementations of all algorithms may result in decreased required times, potentially unlocking further enhancements in performance.

A. Appendix

A.1. Mathematical Definition

Definition 1 (Kurdyka-Lojasiewicz(KL) property) *[38, Definition 2.4] The function $f : \mathbb{R}^n \to \mathbb{R} \cup +\infty$ is said to have the KL property at x^*, if there exist $\eta \in (0, +\infty]$, a neighborhood U of x^* and a continuous concave function $\varphi : [0, \eta) \to \mathbb{R}_+$ such that:*

- $\varphi(0) = 0$,
- *φ is continuously differentiable on $(0, \eta)$,*
- *for all $x \in (0, \eta)$, $\varphi'(x) > 0$,*
- *for all x in $U \cap [f(x^*) < f < f(x^*) + \eta]$, $\varphi'(f(x) - f(x^*))||\nabla f(x)|| \geq 1$.*

A.2. Experiment on stlpy

A.2.1. Parameters Setting

The parameters in section 4.1.1 and 4.1.2, including tasks "Either Or", "Multitask", "Nonlinear Either Or", "Two Circle Obstacles", and "Nonlinear Multitask", are shown in table A.1:

Table A.1.: Parameters in five comparison tasks.

Parameter	Tasks	Symbol	Value
Penalty weight	all tasks	λ	10^5
Minimum trust region radius	all tasks	r_l	10^{-5}
Initial trust region radius	"Either Or"	$r^{(1)}$	30
	"Multitask"		54
	"Nonlinear Either Or"		50
	"Two Circle Obstacles"		15
	"Nonlinear Multitask"		80
Scaling factors of trust region radius	all tasks	α	1.5
		β	2.5
Threshold factors of cost function	all tasks	ψ_0	0
		ψ_1	0.1
		ψ_2	0.8

The parameters plotting heat maps in section 4.1.4 are little different. They are shown in table A.2, A.3 and A.4:

Table A.2.: Parameters in plotting heat maps based on ψ_1 and ψ_2.

Parameter	Tasks	Symbol	Value
Penalty weight	all tasks	λ	5×10^4
Minimum trust region radius	all tasks	r_l	10^{-5}
Initial trust region radius	"Either Or"	$r^{(1)}$	20
	"Multitask"		60
Scaling factors of trust region radius	all tasks	α	1.5
		β	2.5
Threshold factors of cost function	all tasks	ψ_0	0

Table A.3.: Parameters in plotting heat maps based on α and β.

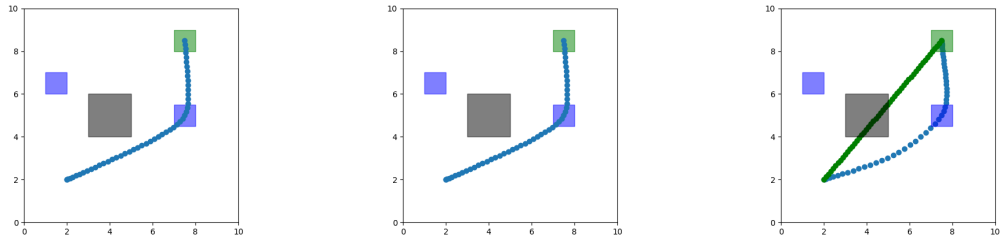
Parameter	Tasks	Symbol	Value
Penalty weight	all tasks	λ	5×10^4
Minimum trust region radius	all tasks	r_l	10^{-5}
Initial trust region radius	"Either Or"	$r^{(1)}$	20
	"Multitask"		60
Threshold factors of cost function	all tasks	ψ_0	0
		ψ_1	0.1
		ψ_2	0.7

Table A.4.: Parameters in plotting heat maps based on λ and $r^{(1)} > 0$.

Parameter	Symbol	Value
Minimum trust region radius	r_l	10^{-5}
Scaling factors of trust region radius	α	1.5
	β	2.5
Threshold factors of cost function	ψ_0	0
	ψ_1	0.1
	ψ_2	0.8

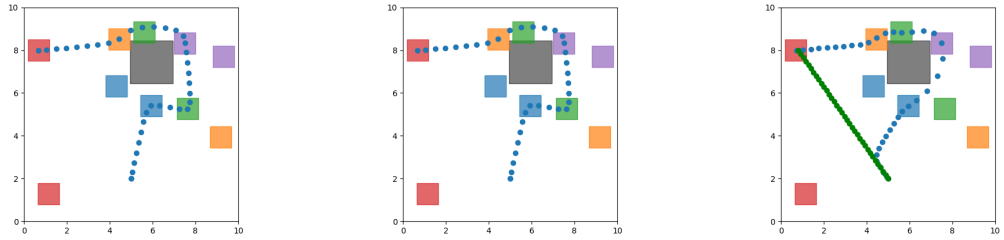
A.2.2. Trajectories

This section shows all trajectories in table 4.1 and 4.3.



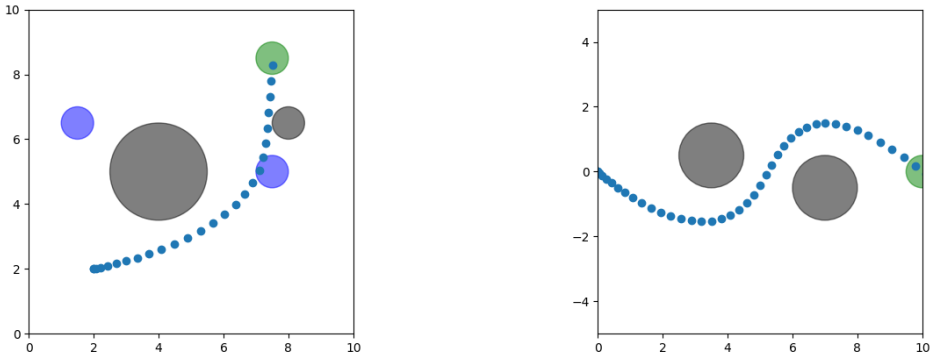
(a) SOS1 solution. (b) MICP solution. (c) SCvx solution.

Figure A.1.: Solution of task "Either or" with the time horizon of 50 and boundary [-0.5,0.5] by different solvers. The solutions of MICP and SOS1 are the same, while the solution trajectory of SCvx is more smooth. In addition, the solution of SCvx has a higher speed, because the distances between points are larger than that of SOS1 and MICP solution.



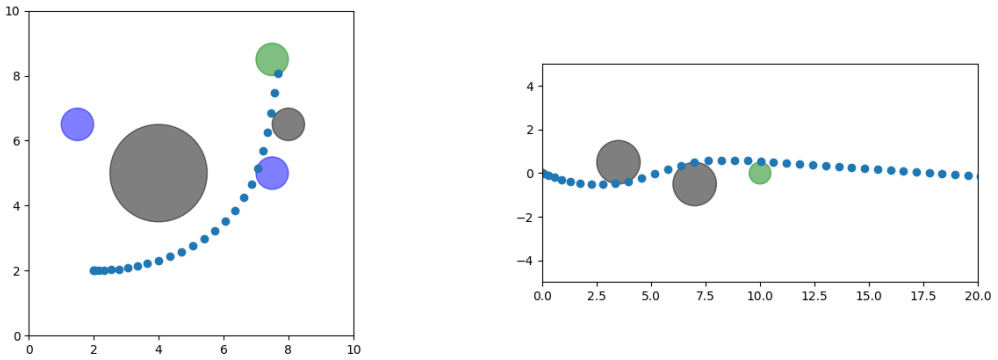
(a) SOS1 solution. (b) MICP solution. (c) SCvx solution.

Figure A.2.: Solution of task "Multitask" with time horizon of 35 and boundary $[-0.5, 0.5]$ by different solvers. The solutions of MICP and SOS1 are the same, while the solution trajectory of SCvx is suboptimal. The first several points in the SCvx solution trajectory are close to the initial guess trajectory.



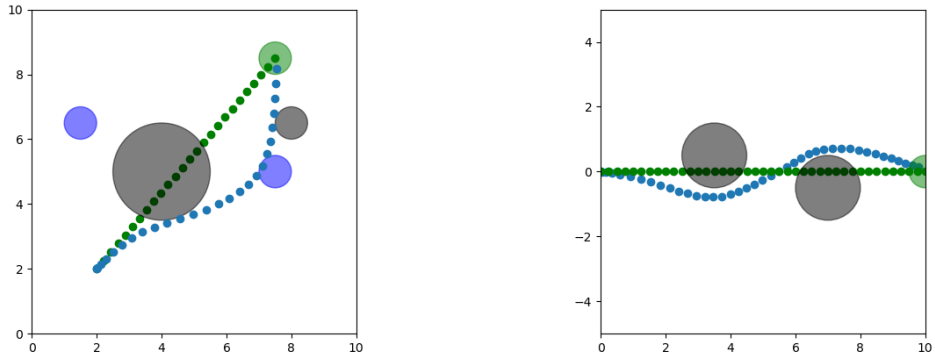
(a) Ipopt solution for task "Nonlinear Either Or". (b) Ipopt solution for task "Two Circle Obstacles".

Figure A.3.: Ipopt solutions for two tasks. The solutions are successful to complete tasks.



(a) Scipy solution for task "Nonlinear Either Or". (b) Scipy solution for task "Two Circle Obstacles".

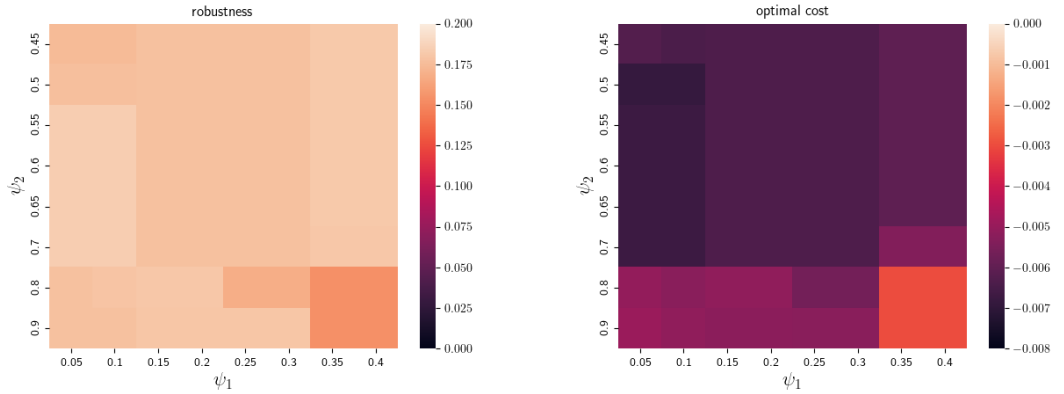
Figure A.4.: Scipy solutions for two tasks. The solution trajectory does not stop after reaching goal region in task "Two Circle Obstacles".



(a) SCvx solution for task "Nonlinear Either Or". (b) SCvx solution for task "Two Circle Obstacles".

Figure A.5.: SCvx solutions for two tasks. The solution stuck in the local optimum. The solution trajectory is very close to the first obstacle. On the one side, the distance to the second obstacle is very small. Therefore, the short distance to the first obstacle does not affect the final robustness value. On the other side, the agent can take more straight path to the blue circle, which can save the control input energy. However, SCvx stuck in the local optimum and does not provide a more energy saving trajectory.

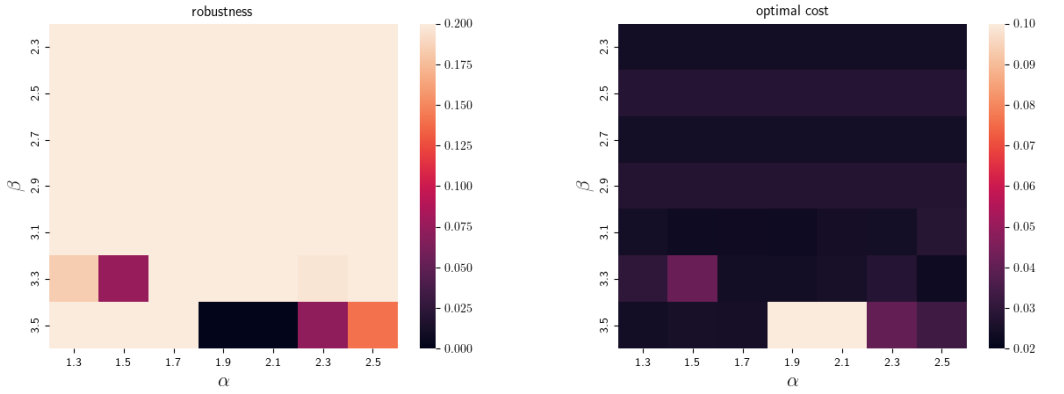
A.2.3. Heat Maps



(a) Robustness of solutions. A lighter color signifies a better result.

(b) Optimal cost of solutions. A darker color signifies a better result.

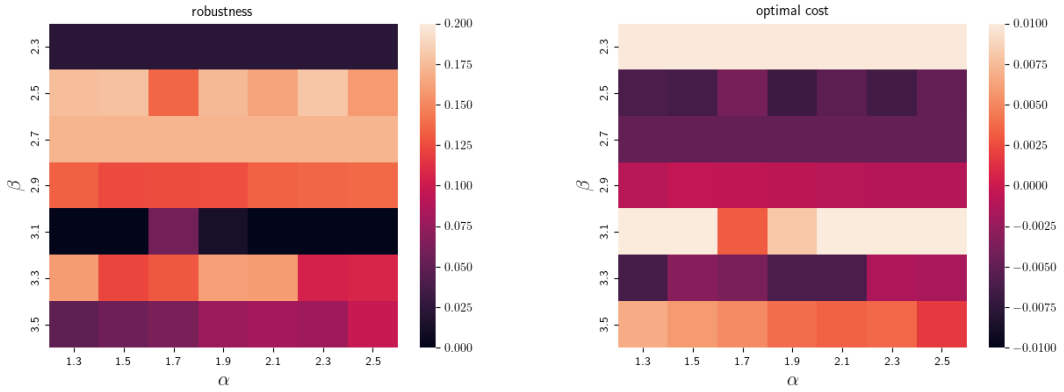
Figure A.6.: The heat map of solutions for the "Multitask" task, based on ψ_1, ψ_2, reveals minimal difference among various solutions. However, a larger value of ψ_1 paired with a smaller ψ_2, observable at the lower right corner of the heat maps, denotes a relatively inferior result.



(a) Robustness of solutions. A lighter color signifies a better result.

(b) Optimal cost of solutions. A darker color signifies a better result.

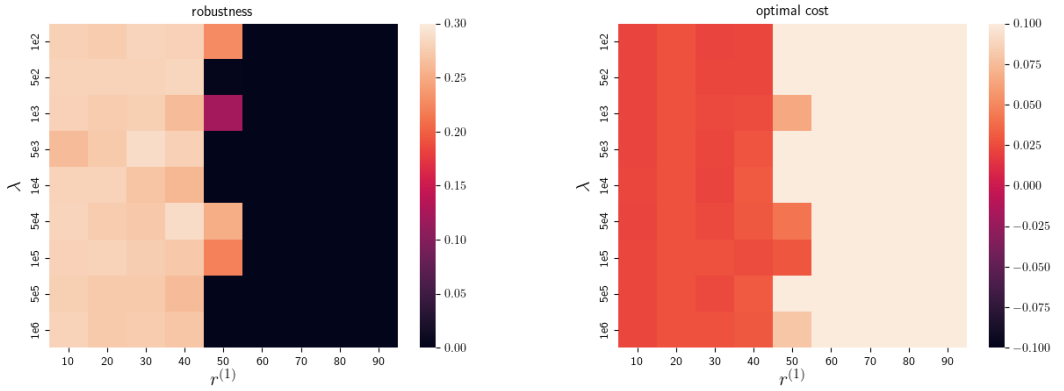
Figure A.7.: Solutions heat map based on α, β for task "Either Or". $\beta = 3.5$ paired with $\alpha = 1.9$ or $\alpha = 2.1$ leads to the failure of SCvx.



(a) Robustness of solutions. A lighter color signifies a better result.

(b) Optimal cost of solutions. A darker color signifies a better result.

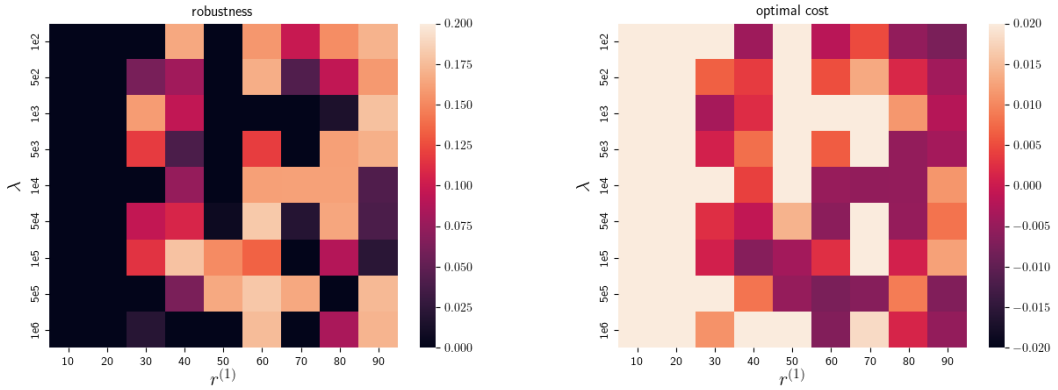
Figure A.8.: Solutions heat map based on α, β for task "Multitask". $\beta = 2.7$ and $\beta = 2.5$ reveal better solutions than other β value.



(a) Robustness of solutions. A lighter color signifies a better result.

(b) Optimal cost of solutions. A darker color signifies a better result.

Figure A.9.: Solutions heat map based on $r^{(1)}, \lambda$ for task "Either Or". $r^{(1)} \leq 40$ reveal similar solutions, while SCvx usually fails with $r^{(1)} \geq 40$.



(a) Robustness of solutions. A lighter color signifies a better result.

(b) Optimal cost of solutions. A darker color signifies a better result.

Figure A.10.: Solutions heat map based on $r^{(1)}, \lambda$ for task "Multitask". $r^{(1)} = 60$ paired with λ has the best solution.

A.3. Experiments in CommonRoad

In the scenarios in the CommonRoad benchmark, we use the SCvx algorithm parameters shown in table A.5.

Table A.5.: Parameters in five comparison tasks.

Parameter	Tasks	Symbol	Value
Penalty weight	all tasks	λ	10^5
Minimum trust region radius	all tasks	r_l	10^{-5}
Initial trust region radius	all tasks	$r^{(1)}$	20
Scaling factors of trust region radius	all tasks	α	1.5
		β	2.5
Threshold factors of cost function	all tasks	ψ_0	0
		ψ_1	0.1
		ψ_2	0.8

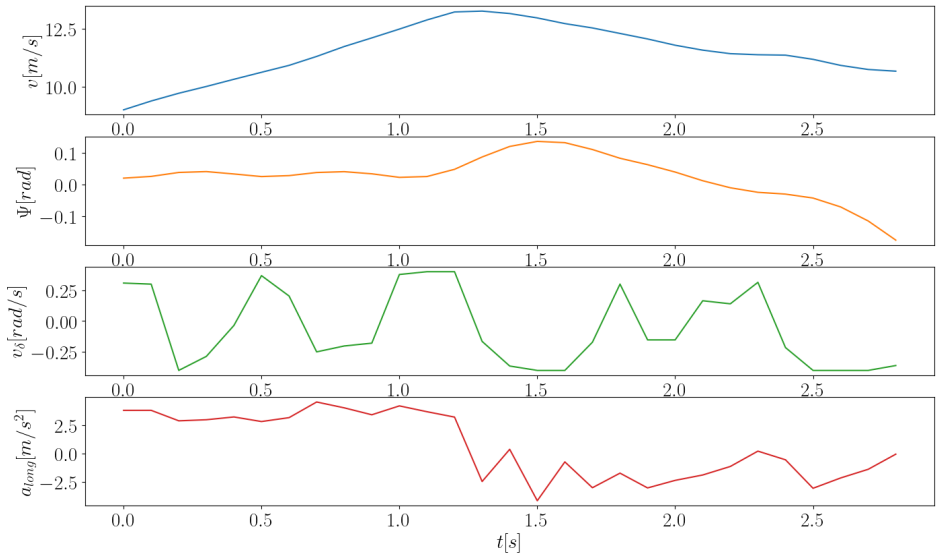


Figure A.11.: The motion profiles $(v, \Psi, v_\delta, a_{long})$ in scenario "Narrow Passage". v_δ change the sign several times, because the ego vehicle needs to keep Ψ in a reasonable interval.

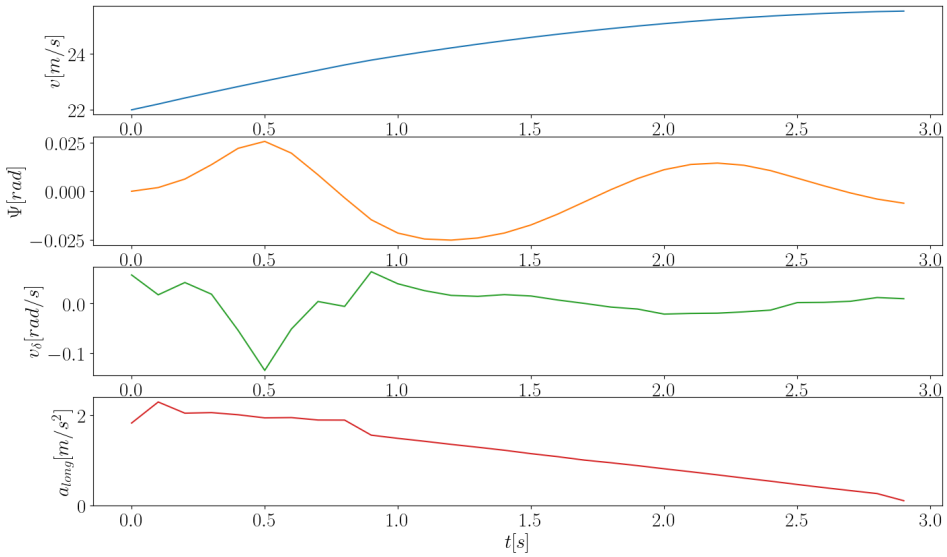


Figure A.12.: The motion profiles $(v, \Psi, v_\delta, a_{long})$ in scenario "Straight with Dynamic Obstacles". In this scenario, the front vehicle is far from the ego vehicle, while the behind vehicle has a higher speed than the ego vehicle. Therefore, the ego vehicle accelerate slowly.

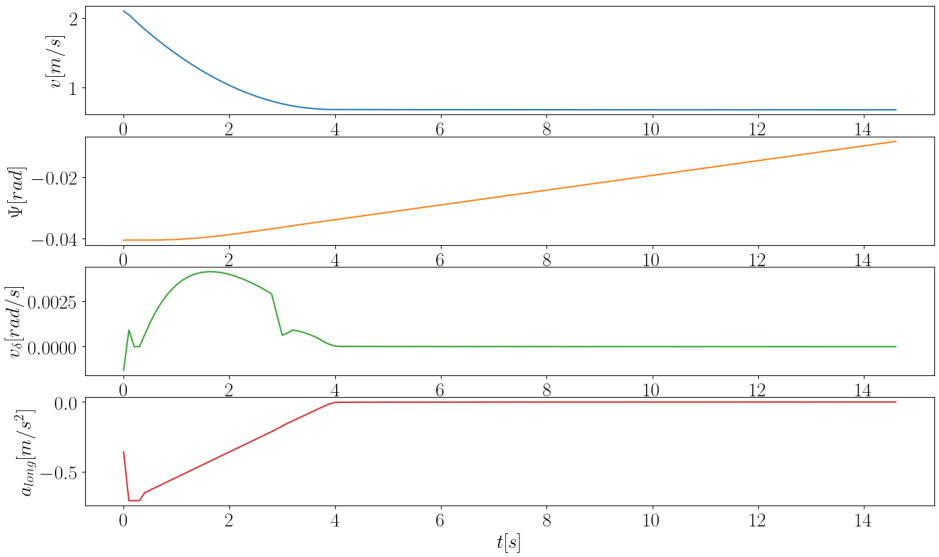


Figure A.13.: The motion profiles $(v, \Psi, v_\delta, a_{long})$ in scenario "T-Intersection". The trajectory is energy saving after reaching the goal region, namely control inputs $a_{long} = 0$ and $v_\delta = 0$.

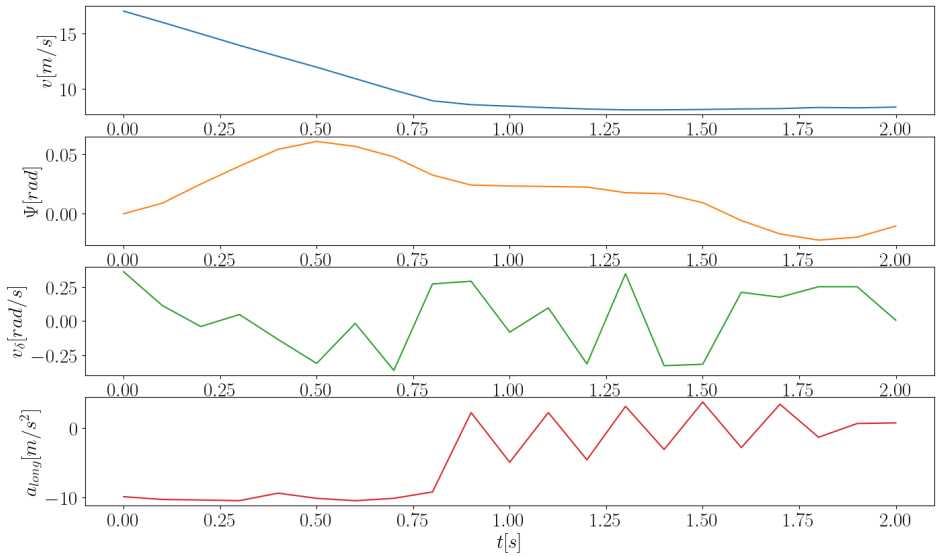


Figure A.14.: The motion profiles $(v, \Psi, v_\delta, a_{long})$ in scenario "Keep Safety Distance". The ego vehicle takes a brake, and then keeps a constant speed. There is some fluctuation in v_δ due to the inherent curvature in the scenario.

A.3.1. Success Rate on Selected Scenarios of CommonRoad

The IDs of 23 scenarios are: ['RUS_Bicycle-10_1_T-1', 'RUS_Bicycle-11_1_T-1', 'RUS_Bicycle-12_1_T-1', 'RUS_Bicycle-13_1_T-1', 'RUS_Bicycle-14_1_T-1', 'RUS_Bicycle-1_1_T-1', 'RUS_Bicycle-1_2_T-1', 'RUS_Bicycle-2_1_T-1', 'RUS_Bicycle-2_2_T-1', 'RUS_Bicycle-3_1_T-1', 'RUS_Bicycle-3_2_T-1', 'RUS_Bicycle-3_3_T-1', 'RUS_Bicycle-4_1_T-1', 'RUS_Bicycle-4_2_T-1', 'RUS_Bicycle-5_1_T-1', 'RUS_Bicycle-5_2_T-1', 'RUS_Bicycle-6_1_T-1', 'RUS_Bicycle-6_2_T-1', 'RUS_Bicycle-7_1_T-1', 'RUS_Bicycle-7_2_T-1', 'RUS_Bicycle-8_1_T-1', 'RUS_Bicycle-8_2_T-1', 'RUS_Bicycle-9_1_T-1']

List of Figures

List of Tables

Bibliography

[1] S. Mallavarapu. "Autonomous Driving Cars: Future Prospects, Obstacles, User Acceptance and Public." In: *Advances in Human Aspects of Transportation: Proceedings of the AHFE 2018 International Conference on Human Factors in Transportation, July 21-25, 2018, Loews Sapphire Falls Resort at Universal Studios, Orlando, Florida, USA*. Vol. 786. Springer. 2018, p. 318.

[2] M. Verucchi, L. Bartoli, F. Bagni, F. Gatti, P. Burgio, and M. Bertogna. "Real-Time clustering and LiDAR-camera fusion on embedded platforms for self-driving cars." In: *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*. IEEE. 2020, pp. 398–405.

[3] W. Schwarting, J. Alonso-Mora, and D. Rus. "Planning and decision-making for autonomous vehicles." In: *Annual Review of Control, Robotics, and Autonomous Systems* 1 (2018), pp. 187–210.

[4] paper with code. *Autonomous Driving on CARLA Leaderboard*. [EB/OL]. `https://paperswithcode.com/sota/autonomous-driving-on-carla-leaderboard` Accessed December 16, 2022.

[5] S. Ren, K. He, R. Girshick, and J. Sun. "Faster r-cnn: Towards real-time object detection with region proposal networks." In: *Advances in neural information processing systems* 28 (2015).

[6] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao. "Yolov4: Optimal speed and accuracy of object detection." In: *arXiv preprint arXiv:2004.10934* (2020).

[7] S. E. Shladover, C. A. Desoer, J. K. Hedrick, M. Tomizuka, J. Walrand, W.-B. Zhang, D. H. McMahon, H. Peng, S. Sheikholeslam, and N. McKeown. "Automated vehicle control developments in the PATH program." In: *IEEE Transactions on vehicular technology* 40.1 (1991), pp. 114–130.

[8] V. Sobal, A. Canziani, N. Carion, K. Cho, and Y. LeCun. "Separating the world and ego models for self-driving." In: *arXiv preprint arXiv:2204.07184* (2022).

[9] P. Wolper. "The tableau method for temporal logic: An overview." In: *Logique et Analyse* (1985), pp. 119–136.

[10] G. E. Fainekos and G. J. Pappas. "Robustness of temporal logic specifications for continuous-time signals." In: *Theoretical Computer Science* 410.42 (2009), pp. 4262–4291.

[11] Y. E. Sahin, R. Quirynen, and S. Di Cairano. "Autonomous vehicle decision-making and monitoring based on signal temporal logic and mixed-integer programming." In: *2020 American Control Conference (ACC)*. IEEE. 2020, pp. 454–459.

[12] A. Rizaldi, F. Immler, B. Schürmann, and M. Althoff. "A formally verified motion planner for autonomous vehicles." In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2018, pp. 75–90.

[13] K. Esterle, L. Gressenbuch, and A. Knoll. "Formalizing traffic rules for machine interpretability." In: *2020 IEEE 3rd Connected and Automated Vehicles Symposium (CAVS)*. IEEE. 2020, pp. 1–7.

[14] S. Maierhofer, A.-K. Rettinger, E. C. Mayer, and M. Althoff. "Formalization of interstate traffic rules in temporal logic." In: *2020 IEEE Intelligent Vehicles Symposium (IV)*. IEEE. 2020, pp. 752–759.

[15] C. Belta and S. Sadraddini. "Formal methods for control synthesis: An optimization perspective." In: *Annual Review of Control, Robotics, and Autonomous Systems* 2 (2019), pp. 115–140.

[16] V. Kurtz and H. Lin. "Mixed-Integer Programming for Signal Temporal Logic with Fewer Binary Variables." In: *IEEE Control Systems Letters* 6 (2022), pp. 2635–2640.

[17] L. Lindemann and D. V. Dimarogonas. "Control barrier functions for signal temporal logic tasks." In: *IEEE control systems letters* 3.1 (2018), pp. 96–101.

[18] J. Ma, Z. Cheng, X. Zhang, M. Tomizuka, and T. H. Lee. "Alternating direction method of multipliers for constrained iterative LQR in autonomous driving." In: *IEEE Transactions on Intelligent Transportation Systems* 23.12 (2022), pp. 23031–23042.

[19] K. Leung, N. Aréchiga, and M. Pavone. "Backpropagation through signal temporal logic specifications: Infusing logical structure into gradient-based methods." In: *The International Journal of Robotics Research* (2020), p. 02783649221082115.

[20] A. Domahidi. *Methods and tools for embedded optimization and control*. ETH Zurich, 2013.

[21] M. Werling, S. Kammel, J. Ziegler, and L. Gröll. "Optimal trajectories for time-critical street scenarios using discretized terminal manifolds." In: *The International Journal of Robotics Research* 31.3 (2012), pp. 346–359.

[22] V. M. Becerra. "Solving optimal control problems with state constraints using nonlinear programming and simulation tools." In: *IEEE Transactions on education* 47.3 (2004), pp. 377–384.

[23] F. Kuhne, W. F. Lages, and J. G. da Silva Jr. "Model predictive control of a mobile robot using linearization." In: *Proceedings of mechatronics and robotics*. Vol. 4. 4. Citeseer. 2004, pp. 525–530.

[24] Y. Mao, M. Szmuk, and B. Açıkmeşe. "Successive convexification of non-convex optimal control problems and its convergence properties." In: *2016 IEEE 55th Conference on Decision and Control (CDC)*. IEEE. 2016, pp. 3636–3641.

[25] Y. Mao, M. Szmuk, X. Xu, and B. Açikmese. "Successive convexification: A superlinearly convergent algorithm for non-convex optimal control problems." In: *arXiv preprint arXiv:1804.06539* (2018).

[26] M. Szmuk and B. Acikmese. "Successive convexification for 6-dof mars rocket powered landing with free-final-time." In: *2018 AIAA Guidance, Navigation, and Control Conference*. 2018, p. 0617.

[27] Y. Mao, B. Acikmese, P.-L. Garoche, and A. Chapoutot. "Successive Convexification for Optimal Control with Signal Temporal Logic Specifications." In: *25th ACM International Conference on Hybrid Systems: Computation and Control*. 2022, pp. 1–7.

[28] M. Althoff, M. Koschi, and S. Manzinger. "CommonRoad: Composable benchmarks for motion planning on roads." In: *Proc. of the IEEE Intelligent Vehicles Symposium*. 2017, pp. 719–726.

[29] S. Manzinger, C. Pek, and M. Althoff. "Using reachable sets for trajectory planning of automated vehicles." In: *IEEE Transactions on Intelligent Vehicles* 6.2 (2020), pp. 232–248.

[30] L. Schäfer, S. Manzinger, and M. Althoff. "Computation of solution spaces for optimization-based trajectory planning." In: *IEEE Transactions on Intelligent Vehicles* (2021).

[31] Y. Shtessel, C. Edwards, L. Fridman, A. Levant, et al. *Sliding mode control and observation*. Vol. 10. Springer, 2014.

[32] E. F. Camacho and C. B. Alba. *Model predictive control*. Springer science & business media, 2013.

[33] H.-G. Lee. *Linearization of Nonlinear Control Systems*. Springer, 2022.

[34] T. Kailath. *Linear systems*. Vol. 156. Prentice-Hall Englewood Cliffs, NJ, 1980.

[35] S. Ghosh, D. Sadigh, P. Nuzzo, V. Raman, A. Donzé, A. L. Sangiovanni-Vincentelli, S. S. Sastry, and S. A. Seshia. "Diagnosis and repair for synthesis from signal temporal logic specifications." In: *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*. 2016, pp. 31–40.

[36] Y. Gilpin, V. Kurtz, and H. Lin. "A smooth robustness measure of signal temporal logic for symbolic control." In: *IEEE Control Systems Letters* 5.1 (2020), pp. 241–246.

[37] S. P. Han and O. L. Mangasarian. "Exact penalty functions in nonlinear programming." In: *Mathematical programming* 17 (1979), pp. 251–269.

[38] H. Attouch, J. Bolte, and B. F. Svaiter. "Convergence of descent methods for semi-algebraic and tame problems: proximal algorithms, forward–backward splitting, and regularized Gauss–Seidel methods." In: *Mathematical Programming* 137.1-2 (2013), pp. 91–129.

[39] J.-B. Tomas-Gabarron, E. Egea-Lopez, and J. Garcia-Haro. "Vehicular trajectory optimization for cooperative collision avoidance at high speeds." In: *IEEE Transactions on Intelligent Transportation Systems* 14.4 (2013), pp. 1930–1941.

[40] S. Sadraddini and C. Belta. "Formal synthesis of control strategies for positive monotone systems." In: *IEEE Transactions on Automatic Control* 64.2 (2018), pp. 480–495.

[41] R. Tedrake and the Drake Development Team. *Drake: Model-based design and verification for robotics*. 2019.

[42] S. P. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[43] M. Bénichou, J.-M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent. "Experiments in mixed-integer linear programming." In: *Mathematical Programming* 1 (1971), pp. 76–94.

[44] wikipedia. *Special ordered set*. 2023. URL: `https://en.wikipedia.org/wiki/Special_ordered_set` (visited on 05/11/2023).

[45] A. Wächter and L. T. Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming." In: *Mathematical programming* 106 (2006), pp. 25–57.

[46] P. Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." In: *Nature Methods* 17 (2020), pp. 261–272.

[47] Y. Mao, M. Szmuk, and B. Açıkmeşe. "A tutorial on real-time convex optimization based guidance and control for aerospace applications." In: *2018 Annual American Control Conference (ACC)*. IEEE. 2018, pp. 2410–2416.

[48] J. Gondzio. "Interior point methods 25 years later." In: *European Journal of Operational Research* 218.3 (2012), pp. 587–601.

[49] L. Blum, L. A. BLUM, F. Cucker, M. Shub, and S. Smale. *Complexity and real computation*. Springer Science & Business Media, 1998.

[50] A. Richards and J. How. "Mixed-integer programming for control." In: *Proceedings of the 2005, American Control Conference, 2005.* IEEE. 2005, pp. 2676–2683.

[51] G. A. Cardona, D. Kamale, and C.-I. Vasile. "Mixed Integer Linear Programming Approach for Control Synthesis with Weighted Signal Temporal Logic." In: *Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control*. 2023, pp. 1–12.

[52] J. Ziegler and C. Stiller. "Fast collision checking for intelligent vehicle motion planning." In: *2010 IEEE intelligent vehicles symposium*. IEEE. 2010, pp. 518–522.

[53] S. Diamond and S. Boyd. "CVXPY: A Python-embedded modeling language for convex optimization." In: *Journal of Machine Learning Research* 17.83 (2016), pp. 1–5.

[54] A. Domahidi, E. Chu, and S. Boyd. "ECOS: An SOCP solver for embedded systems." In: *European Control Conference (ECC)*. 2013, pp. 3071–3076.