EXPLORING FORMAL METHODS FOR PROVABLY SAFE AUTONOMOUS CYBER-PHYSICAL SYSTEMS

Dylan Le

Submitted in Partial Fullfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Approved by:
Carlos A. Varela, Chair
Radoslav Ivanov, Co-Chair
Stacy Patterson

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York

[May 2025]

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Listings

ABSTRACT

The increasing complexity of autonomous cyber-physical systems (CPS) necessitates rigorous verification techniques to ensure safety and reliability. Safety-critical systems, such as those in aerospace, automotive, and medical domains, operate under strict constraints where failures can have catastrophic consequences. Modelling autonomous CPS as a hybrid system allows for verification of safety properties. Current verification methods, including model checking and Satisfiability Modulo Theories (SMT) solvers, face scalability challenges when dealing with large state spaces and complex non-linear dynamics. This thesis investigates theorem proving as a means to provide more scalable and explainable safety guarantees for cyber-physical systems.

As a case study, we analyze the Mountain Car problem. We develop a discrete and continuous model for the dynamics. In this thesis we provide a formal proof using theorem proving and physical properties that shows that a simple controller for Mountain Car, under specific conjectures, ensures it reaches the goal position from any valid initial condition and without a finite time bound for verification.

The findings of this thesis highlight the potential for theorem proving to enhance the explainability and scalability of safety verification in autonomous cyber-physical systems.

1. INTRODUCTION

1.1 Motivation

With the rise of Artificial Intelligence, more and more companies are pushing for their products to be smart. Most notable are *safety-critical* products, which are those that can have direct implications on the health and safety of humans. Failure of operation or improper operation outside expected behavior can be fatal. Safety-critical systems exist in many industries, e.g., automotive, aerospace, medical. However, complexity in these cyber-physical systems and the environments in which they operate brings uncertainty. The automotive maker Tesla is well known for their car with a Full Self-Driving (FSD) feature, yet US regulators are investigating how FSD was involved in multiple fatal crashes [1]. Advances in Reinforcement Learning (RL) have allowed for models to be created that can learn about complex dynamics in an unknown environment and teach themselves strategies to interact with them. RL models will increasingly be used for safety-critical applications as control tasks become more complex. Complex tasks, such as self-driving cars [2], air-traffic control [3], and space-ship docking [4, 5] have garnered significant research interest. Formal verification of safety properties for these models face scalability and explainability issues. These models may behave well while training and testing but when deployed they may experience out-of-distribution inputs that cause failures. Many verifiers used in the robustness training such as Reluplex [6], a Satisfiability Modulo Theories (SMT) [7] based solver, do not scale well with deep neural networks due to their large number of variables or complexity of the verification property. As more safety-critical autonomous systems are deployed, the importance of verification scalability becomes greater. In this thesis, we consider the state of the art and explore possible ways to approach improving scalability and explainability. We survey existing techniques and develop some sample proofs.

1.2 Contributions

In this thesis we provide the following contributions:

- A brief survey of the state of the art for model checking cyber-physical systems.
- Generalizations of Mountain Car for discrete and continuous dynamics.

- A formal proof for a simple controller for Mountain Car which shows, under the assumption of our conjectures, the controller is able to reach the top of the hill for any valid initial condition.
- Using physical principles, we prove Mountain Car with continuous dynamics can be solved with a simple controller for all valid initial conditions.

1.3 Thesis Organization

This thesis is organized into 6 chapters. In Chapter 2, we go over model checking cyber-physical systems and how it can be applied to neural network controllers and go over its limitations. In Chapter 3, we introduce the Mountain Car problem, as a toy problem for us to analyze using multiple different techniques. In Chapter 4, we explore theorem proving approaches including manual and automated reasoning to produce direct proofs of safety. In Chapter 5, we use physical principles to analyze the toy problem and its system directly. Finally, in Chapter 6, we discuss the takeaways from this research.

2. MODEL CHECKING

2.1 Cyber-Physical Systems and Hybrid Systems

Cyber-Physical Systems (CPS) are systems that both exist in the cyber domain (algorithms, software, control) and the physical domain (time, space, motion). In this thesis, we focus on autonomous cyber-physical systems. *Hybrid systems* can be used to model many Cyber-Physical Systems. A hybrid system is a dynamical system that can be used to model complex controllers or devices with continuous and discrete dynamics [8]. Model checking performs exhaustive search of the state to verify that a hybrid system satisfies some property. That is, model checking can be used to check all possible state spaces and validate that a hybrid system has some proposed property. Because of this exhaustive search, model checking can be expensive [9]. We consider different approaches for verification of CPS through modelling as hybrid systems.

2.2 SMT Approaches to Verifying Cyber-Physical Systems

Gao et al. [10] address the undecidability of SMT solvers for reals with Trigonometric functions by introducing *dReal*. dReal given first-order logic formula ϕ produces either ϕ is unsatisfiable or ϕ^δ is satisfiable along with a proof. Where ϕ^δ is some over approximation of ϕ within some $\delta \in \mathbb{Q}^+$. *dReal* can perform model checking for a hybrid system by encoding safety properties in ϕ. Kong et al. [11] build on top of dReal to create *dReach*, a bounded reachability analysis tool for hybrid systems. In both tools, if dReal outputs a satisfiable proof for ϕ the safety property is guaranteed. When dReal shows that there is an assignment such that ϕ^δ is satisfiable, then the system is unsafe. If δ is sufficiently small, then the guarantee that the system is unsafe is stronger. Like previous model checking tools, dReal and dReach are limited to a finite time horizon for hybrid system verification.

2.2.1 The Need for Neural Network Controllers

While dReach works well with many non-linear systems that rely on the real numbers and transcendental functions, as control problems become more complex more designers will use *neural networks* (NN) to train controllers to accomplish tasks. Neural networks rely on *activation functions* that may be non-linear and discontinuous to create universal

approximation functions. These functions are able to adapt to highly non-linear and complex distributions. Many of these neural networks are trained with *reinforcement learning* (RL). The optimal action at any state may be unknown, so RL uses an overall performance metric that can be optimized, also known as the reward function. Controllers then must optimize reward, but in doing so there is no guarantee that the optimal reward decision path leads to a safe decision path. In addition, *deep neural networks* (DNN), compositions of many layers of activation functions, make analysis of these networks much more challenging. We will examine two methods that use model checking in order to validate safety properties of CPS with NN controllers.

2.3 Reachability Analysis for Neural Networks

Ivanov et al. [12] develop *Verisig* an approach that converts closed-loop neural network controllers into hybrid systems that model checking tools, such as Flow* [13], can then verify. In the paper, the authors do a case-study on Mountain Car. The safety problem is formulated as having an average reward greater than 90 over 100 trials. Their results showed that Verisig+Flow* scaled linearly with the number of layers. This was possible because limits were placed on the types of activation functions that were permissible, namely sigmoid activation functions. One drawback of this method is a finite verification time. Verisig cannot validate that for all time a controller may not reach the goal. That is a controller that may have less reward but still reach the goal cannot be validated with a finite time horizon.

Jin et al. [14] propose extracting policy from a DNN that comes from a discretized state space made up of finite decision-making units (DMU) for use with standard model checking. Because the policy extracted is finite and the DNN is treated as a black box, the verification time is architecture independent. This policy can also be validated using standard model checking tools. This allows them to evaluate more complex temporal based properties. However, due to the coarseness of sampling the DMU's, the method cannot verify safety conditions and is limited to liveness conditions for other types of problems.

2.4 State of Model Checking

The current state of the art is simultaneously not scalable nor able to validate safety for universal time. This is case regardless of choice of controller for a given hybrid system.

In the following chapters we will explore the toy problem Mountain Car in the context of theorem proving as a possibly more explainable and scalable method of verification.

3. MOUNTAIN CAR

3.1 A Toy Problem: Mountain Car

In reinforcement learning, one simple toy problem that can be modelled with continuous dynamics and discrete control is Mountain Car. In Mountain Car, a controller observes state, position and velocity, within a one-dimensional valley and has the goal of escaping the valley by reaching the top of the hill. See Figure 3.1 for illustration. The amount of thrust that the controller can put into the system is always less than the amount needed to exit the valley by just applying thrust in a single direction, so the controller must make use of the valley's slopes to help it reach the goal. The challenge is if given any controller, can we prove that the controller is capable of eventually getting the car to the goal.

Many sound but incomplete methods exist to show that a constrained controller can solve this goal. *Verisig* [12] is able to prove for a neural network controller with only *sigmoid* activation that up to a finite time horizon a controller can reach the goal. We lay out the requirements for a complete proof for a specific toy controller in Chapters 3 and 4.

3.1.1 Mountain Car with Continuous Dynamics

We consider Mountain Car, with modifications as a generalized problem. In Mountain Car the continuous variables x position and v velocity change with respect to time according

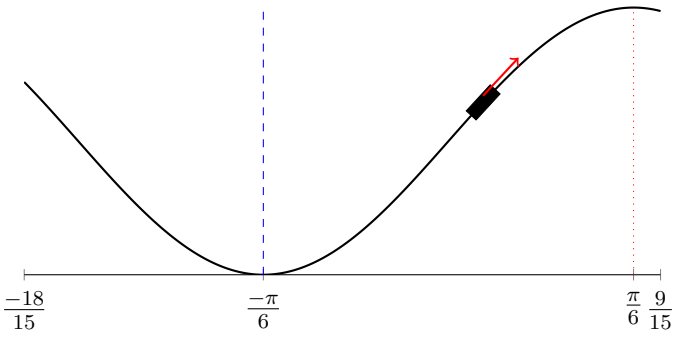


Figure 3.1: An illustration of the Mountain Car Problem with $w = 3$. The car is the rectangular black box. Its thrust direction is indicated by the red arrow. The blue dashed line indicates the bottom of the valley, while the red dotted line is the goal.

to the following set of coupled differential equations,

$$v'(t) = Fu(x(t), v(t)) - G\cos(wx) \tag{3.1}$$

$$x'(t) = v(t) \tag{3.2}$$

Where F, G, w are positive constants corresponding to thrust force, gravity, and the width and steepness of the valley respectively. The function $u(x(t), v(t))$ is the control function, which can output values $[-1, 1]$ for control. Similarly, we define the Mountain Car as a discrete formulation.

3.1.2 Mountain Car with Discrete Dynamics

Formally Mountain Car's discrete dynamics are described in the following equations.

$$v_t = v_{t-1} + Fu(x_{t-1}, v_{t-1}) - G\cos(wx_{t-1}) \tag{3.3}$$

$$x_t = x_{t-1} + v_{t-1} \tag{3.4}$$

Where F, G, w are positive constants corresponding to thrust force, gravity, and the width of the valley respectively. The function $u(x_t, v_t)$ is the control function, which can output values $-1, 0, 1$ for control. This equates to accelerate left, do nothing, and accelerate right respectively. The position is clipped to the domain $x \in \left[\frac{-18}{5w}, \frac{9}{5w}\right]$, approximately one period of the cos. The velocity is also clipped, $v \in [-70F, 70F]$. The goal is $x_f = \frac{\pi}{2w}$. The initial state variables are defined to be the following

$$x_0 \in \left[\frac{-9}{5w}, \frac{-6}{5w}\right] \tag{3.5}$$

$$v_0 = 0 \tag{3.6}$$

Note that this generalized version of Mountain Car is equivalent to the more popular setup when $F = 0.001$, $G = 0.0025$ and $w = 3$ [15]. We use these constants going forward as well.

Formally the goal of Mountain Car is: given x_0 and v_0, there exists a sequence of controls $u(x_t, v_t)$ such that there exists some t_f such that $x_{t_f} \geq x_f$ subject to the dynamics in (3.3) and (3.4). Succinctly, there is a controller that from any valid initial condition can reach the goal in a finite amount of time.

$$
\begin{aligned}
x_f &\geq \pi/6 \\
\pi/6 &\leq x_{t_{f-1}} + v_{t_{f-1}} \\
&\leq x_{t_{f-1}} + v_{t_{f-2}} + Fu(x_{t_{f-2}}, v_{t_{f-2}}) - G\cos(wx_{t_{f-2}}) \\
&\vdots \\
&\leq x_{t_0} + v_{t_0} + \dots
\end{aligned} \tag{3.7}
$$

So for any x_{t_0} in the initial position set, does there exist a sequence of $u(x_t, v_t)$ such that the inequality holds. However, it is not clear how many time steps there are, so checking satisfiability of the inequality for a fixed number of steps is not sufficient.

3.1.3 Controller Description

We design a simple controller $u(x_t, v_t)$ to have the following control:

$$
u(x_t, v_t) = \begin{cases} -1 & v < 0 \\ 1 & v \geq 0 \end{cases} \tag{3.8}
$$

Intuitively, this controller only looks at the velocity component of the state and tries to either continue going right, or continue going left until the car starts "falling" back into the valley. Throughout this thesis, we examine different verification techniques and what they can prove for this discrete controller. We will examine this controller for both discrete dynamics (Chapter 4) and continuous dynamics (Chapter 4,Chapter 5).

4. THEOREM PROVING

To address the challenges in scalability with model checking approaches, in this chapter we explore manual and automated reasoning through the use of theorem proving. Specifically, we study two formal logics with an application example and a formalization of Mountain Car in their respective languages.

4.1 Lean 4

Lean 4 is a functional programming language and proof language for interactive theorem proving that is based on dependent type theory and the Calculus of Constructions [16]. Lean4 has a strong mathematical reasoning library developed by its community, Mathlib [17]. This strong math library will help us to develop our proofs.

4.1.1 Real Numbered Reasoning

To demonstrate the capabilities of Lean 4 for analysis we will prove the following:

$$\forall x \in \mathbb{R} : \ \sin(\cos(x)) \leq \cos(\sin(x)) \tag{4.1}$$

It is not immediately clear that the inequality is true, but we will prove it formally using Lean 4. To do this, we will first note that the functions on either side of the inequality are even functions and periodic over 2π. See Figure 4.1 for a visualization of 4.1. So we will prove this formally for only $[0, \pi]$. We then divide the proof into two cases: $x \in [0, \pi/2]$ or $x \in [\pi/2]$. In this section we will only go through the proof for $x \in [0, \pi/2]$, see Appendix A for the full proof. Breaking into cases, we define a theorem we want to prove in Lean as seen in Listing 4.1. We then want to show that the inequality holds in either case. For $x \in [0, \pi/2]$, we will show that $\sin(\cos(x)) \leq \cos x \leq \cos(\sin(x))$ via transitivity of the $\leq$ in Listing 4.2. We focus on the proof for the left hand side of the inequality. So in Lean we get the following in Listing 4.3 In Listing 4.3 we apply multiple techniques. First on line 1, we create an axiom that states $\sin x \leq x \forall x \geq 0$. Showing this can be approached in two ways, using Calculus or proving $x - \sin x$ is non-decreasing. For brevity, we leave this as an axiom to demonstrate reasoning with assumptions. Secondly on line 6, we use an automatic tactic

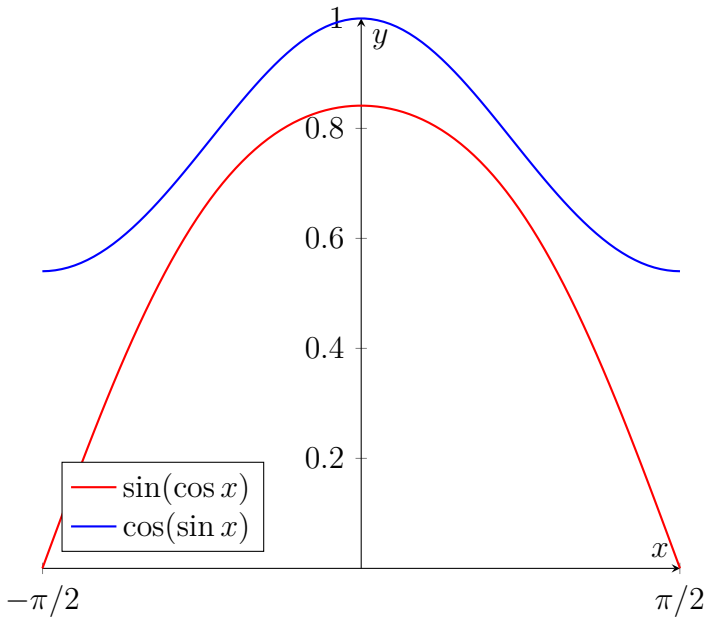


Figure 4.1: Plot of $\sin(\cos x)$ (red) and $\cos(\sin x)$ over $[-\pi/2, \pi/2]$.

```
theorem sin_cos_cos_sin_period (h: x ≥ 0 ∧ x ≤ Real.pi): Real.sin (Real.cos x) ≤
    Real.cos (Real.sin x) := by
{
  cases le_or_lt x (Real.pi / 2) with
  | inl h1 => -- Case: x ≤ π/2
  | inr h2 => -- Case: x > π/2, so x is in [π/2, π]
}
```

Listing 4.1: A skeleton proof for $\sin(\cos(x)) \leq \cos(\sin(x))$

linarith to prove that $x \geq -\pi/2$. In this example it trivially follows from the hypothesis h, but in general the tactic searches for a contradiction between the hypotheses in the proof context which are inequalities or equalities. Finally, another technique we use is rewriting which modifies the goal of the current step by matching the pattern from some result. See Appendix A for more.

By planning our proofs top-down and building assumptions bottom-up and leveraging automated reasoning libraries to assist in tedious verification tasks; Lean 4 allows us to reason and formalize real valued systems.

```
lemma goal_left_half (x : ℝ) (h: x ≥ 0 ∧ x ≤ Real.pi/2): Real.sin (Real.cos x)
    ≤ Real.cos (Real.sin x) := by
{
  obtain lhs := sin_cos_lt_cos x h
  obtain rhs := cos_lte_cos_sin_x x h
  apply le_trans lhs rhs
}
```

Listing 4.2: Applying the transitivity rule for $\sin(\cos(x)) \leq \cos x \leq \cos(\sin(x))$

```
1  axiom sin_x_lte_x (x: ℝ ) (h: x ≥ 0): Real.sin x <= x -- Well known from
       calc.
2  lemma sin_cos_lt_cos (x : ℝ) (h: x ≥ 0 ∧ x ≤ Real.pi/2):
3    Real.sin (Real.cos x) ≤ Real.cos x := by
4  {
5      have x_gt_neg_pi_div_2 : x ≥ -Real.pi/2 := by {
6        linarith
7      }
8      have x_in_bounds : x ∈ (Set.Icc (-Real.pi/2) (Real.pi/2)) := by {
9        rw [Set.mem_Icc]
10       apply And.intro x_gt_neg_pi_div_2 h.right
11     }
12     have h_cos : Real.cos x ≥ 0 := by {
13       apply Real.cos_nonneg_of_mem_Icc
14       rw [neg_div] at x_in_bounds
15       exact x_in_bounds
16     }
17     apply sin_x_lte_x (Real.cos x)
18     exact h_cos
19 }
```

Listing 4.3: Applications of axioms and automated reasoning for $\sin(\cos(x)) \leq \cos x$

4.2 Differential Dynamic Logic and KeYmaera X

Differential Dynamic Logic is a logical system for describing hybrid systems, systems with that combine continuous and discrete dynamics, and verifying them [18]. The logic system, through logical rules following a sequent calculus, is able to transform the hybrid descriptions into real number satisfiability problems.

KeYmaera X is theorem prover for Differential Dynamic Logic written in Java [8]. In addition to the logical tools developed from Differential Dynamic Logic, it also includes tactics which search and help to prove logical statements in Differential Dynamic Logic (dL).

We look at one example using both dL and KhymeraX to understand its benefits.

4.2.1 A Damped Bouncing Ball

We consider a modified bouncing ball system from "Logical Foundations of Cyber-Physical Systems" with damping [19]. Consider a ball that starts at some initial height $H \geq 0$ and falls with gravity according to $x' = v$ and $v' = -g$. While you can model this system in physics, there is a discontinuity in the velocity when a ball hits the floor. Specifically, when x becomes 0 the velocity experiences a discontinuous jump to $-cv$ where $0 \leq c \leq 1$ is some damping constant. This fits a model for a hybrid system. So we will approach it as such. See Figure 4.2 for the full proof. We break the proof into three loop cases: before the loop, at an arbitrary step in the loop, and after the loop. Apart from the step in the loop, we can conclude from quantifier elimination that no constraints are violated. In the loop step, we are able to use the solution to the differential equations to satisfy the constraints.

While possible to derive the calculus by hand it is much easier to verify in KeymaeraX. KeymaeraX is also able to make use of SAT solvers to validate the proofs once the problem is moved up from a hybrid system and into a real number satisfiability problem. For the full proof see Appendix A.

4.3 Theorem Proving Applications to Mountain Car

In this section, we apply the theorem provers to our toy problem and discuss their results.

4.3.1 Lean 4 Model for Mountain Car

To prove that our simple velocity controller u is able to solve the Mountain Car problem we consider breaking up the problem into swings. We define a swing, as some time interval $[t_i, t_j]$ for which the control of the car is constant. Because the controller is discrete, and its outputs differ in sign, the entire run of the controller can be modelled as a series of swings.

We consider the last swing to x_f over $[t_i, t_j]$ and $x_{t_i} < x_f$. There could be many possible swings that could reach the goal with varying velocities and positions. We consider one specific swing where $v_{t_i} = 0$ and x_{t_i} is as close to goal as possible while on the left side of the valley. In other words, $\cos(wx_{t_i}) < 0$. Clearly, $u = 1$, as the car must travel right to

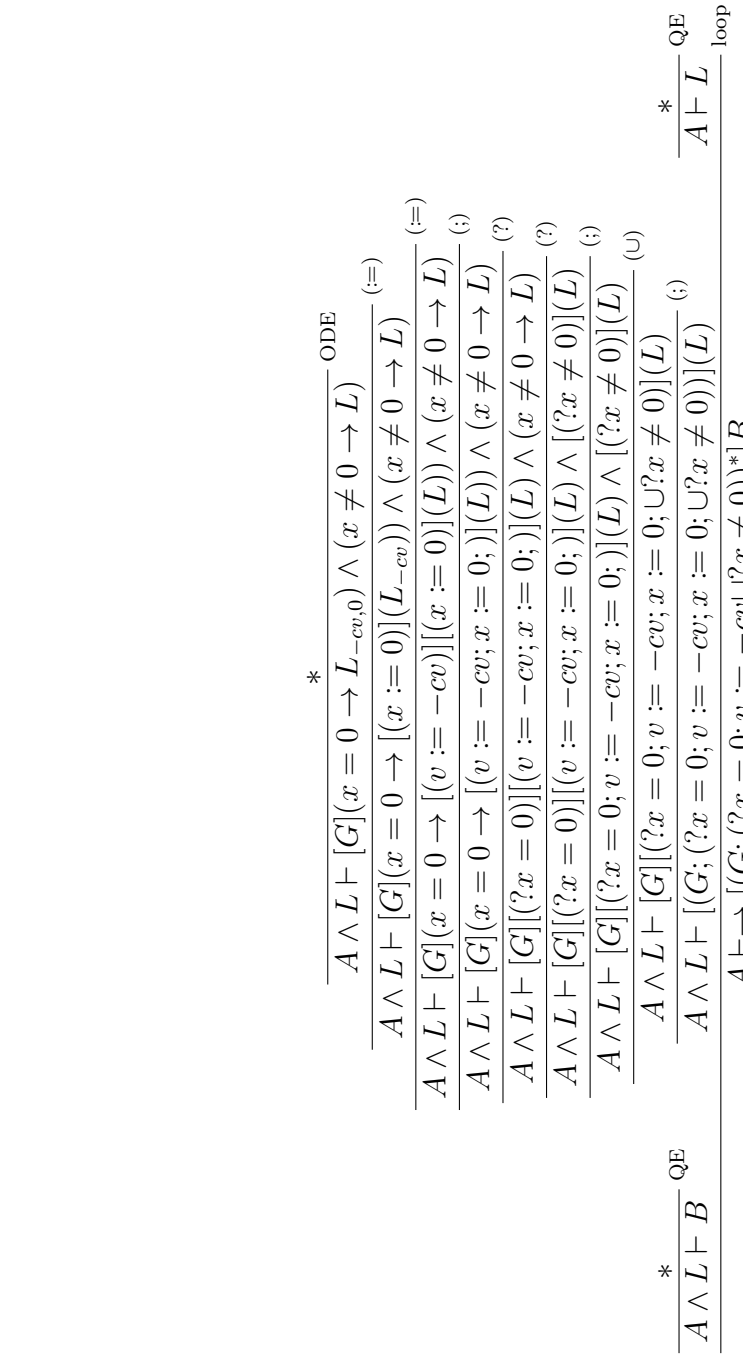


Figure 4.2: Full Proof of a Damped Bouncing Ball in Sequent Calculus for dL.
We define $A \equiv x \geq 0 \land x = H \land v = 0 \land g \geq 0 \land 1 \geq c \geq 0$, $B \equiv 0 \leq x \leq H$,
$G \equiv \{x' = v, v' = -g \land x \geq 0\}$. **We use the loop invariant**
$L \equiv 2gx \leq 2gH - v^2 \land x \geq 0 \land v^2 \leq 2gh$. **We let**
$L_{-cv} \equiv 2gx \leq 2gH - (-cv)^2 \land x \geq 0 \land (-cv)^2 \leq 2gh$. **We let**
$L_{-cv,0} \equiv 0 \leq 2gH - (-cv)^2 \land (-cv)^2 \leq 2gh$.

reach the goal. Suppose that from x_{t_i} we can reach the goal, because the slope of the valley is negative then any point $x < x_{t_i}$ should also be able to reach the goal x_f. Intuitively this is because the car is higher up at x than x_{t_i}, so it has more "potential energy" that allows the car to travel further. So any swing that starts with $x \leq x_{t_i}$ can reach the goal if x_{t_i} exists. We let these be called regions.

Formally a region is some set of contiguous positions X that for some fixed control u starting in with initial position $x \in X$ with velocity 0 can reach x_g where $x \leq x_g$ if $u = 1$ and $x > x_g$ if $u = -1$.

Let the region that gets the controller to x_f be swing_{goal}. We can then ask if there is some other region that can bring us to swing_{goal}. We can repeat this tactic of finding regions, until we reach the initial state space. If we can from the initial state reach any of these regions in the chain, we can thus conclude we can reach the goal state.

For visualization see Figure 4.3.

4.3.1.1 Function Definitions

While Lean4 allows us to reason about real numbers, in order for the proof to be feasible we need to have a computable cos function, so we can evaluate it for checking if the inequality in 3.7 is satisfied.

4.3.1.2 Tightening Conjectures

To more formally prove that our controller reaches the goal, we consider two conjectures where x_i is some position to the left of the center of the valley or to the right respectively.

Conjecture 1 (Region from the left) *If from some x_i and $v_i = 0$ a controller $u(x, v)$ can reach x_f, where $x_i < x_f$. Then the controller with initial position $x_j \leq x_i$ and initial velocity $v_j \geq v_i$ can reach $x_k \geq x_f$.*

Conjecture 2 (Region from the right) *If from some x_i and $v_i = 0$ a controller $u(x, v)$ can reach x_f, where $x_i > x_f$. Then the controller with initial position $x_j \geq x_i$ and initial velocity $v_j \leq v_i$ can reach $x_k \leq x_f$.*

For any specific run, the velocity may never be equal to zero exactly except at the initial state. So we need to more tightly bound the conditions such that moving between regions encodes changing signs. Which is what Conjectures 1 and 2 cover.

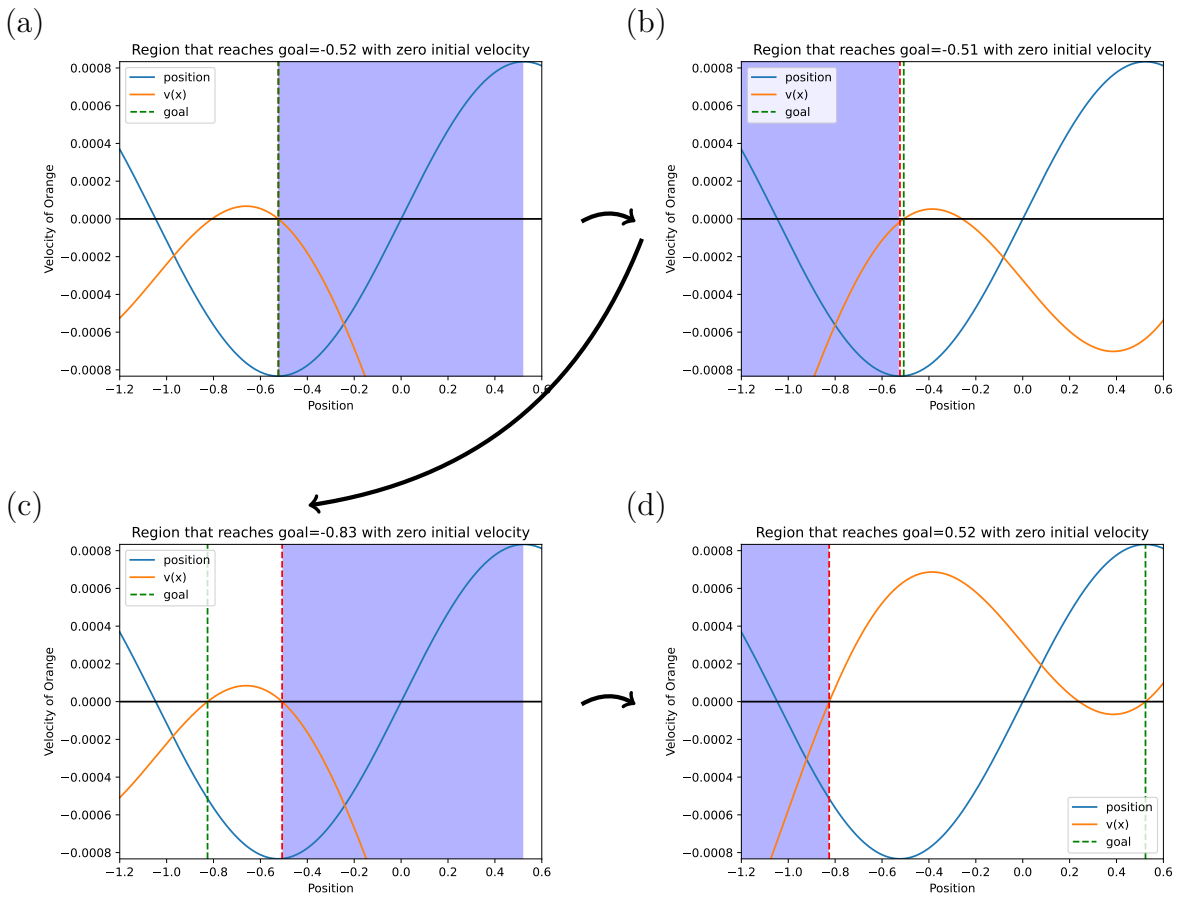


Figure 4.3: In all plots, any point in the shaded regions can reach the green dashed line. If you start in the blue region in (a) then you can reach some point in (b)'s blue region. The controller that goes through swing phases from an initial position may go through all regions $(a) \to (b) \to (c) \to (d)$ but may also skip regions. If a controller can get to any of these regions, then the controller following the swinging behavior will reach the goal. It suffices to show that any controller either starts in (a) or (b) or can get to a region in (a).

In Lean4 we take these conjectures as axioms, so for Conjecture 1 we would have the following:

Like in the sketch we need a single point then that reaches a goal. We construct a proof that verifies we know what that point is and how long it takes to reach the goal. The point can be found using a conservative approximation using a continuous model of the problem or through binary search. Using this single point proof and the axiom we can then show the proof for (a) in Figure 4.3.

Assuming you either start in initial conditions for swing_4 or you can from your

```
def gravity' : ℝ := 0.0025
def force' : ℝ := 0.001
def w' : ℝ := 3
def max_velocity' : ℝ := 0.07
def min_velocity' : ℝ := -0.07
def max_x' : ℝ := 0.6
def min_x' : ℝ := -1.2
-- def goal' : ℝ := -0.52359942
def goal' : ℝ := 0.5

def cos (x : ℝ) : ℝ := -- Cannot use polynomials or rationals as that becomes
-- uncomputable
   1 - 0.5*x^2 + 0.041666*x*x*x*x - 0.00138888888*x*x*x*x*x*x +
   0.00002480158*x*x*x*x*x*x*x*x - 2.75573192e-7*x*x*x*x*x*x*x*x*x*x -- However
    much precision you want.

mutual
   def velocity (t : ℕ) (x₀ v₀ u: ℝ) : ℝ :=
      if t = 0 then max (min v₀ max_velocity') (min_velocity') -- Clip velocity
      else
      max (
         min max_velocity'
         (
            velocity (t - 1) x₀ v₀ u +
            (u * force') -
            gravity' * cos (w' * position (t - 1) x₀ v₀ u)
         )
      ) min_velocity'


   def position (t : ℕ) (x₀ v₀ u : ℝ) : ℝ :=
      if t = 0 then max (min x₀ max_x') min_x'
      else
      max (
         min max_x'
         (
            position (t - 1) x₀ v₀ u + (velocity t x₀ v₀ u) * 0.001
         )
      ) min_x'
end
```

Listing 4.4: Definitions of functions for Mountain Car simulation in Lean 4.

```
axiom reach_region_further_left (t₀ : ℕ) (p₀ p₁ v₀ v₁ u p v: ℝ)
   (h₀ : p₁ >= p₀) (h₁ : v₁ <= v₀) (h₂ : u = -1):
   (position t₀ p₀ v₀ u = p ∧ velocity t₀ p₀ v₀ u = v) → (∃ t₁ : ℕ, position t₁
   p₁ v₁ u <= p ∧ velocity t₁ p₁ v₁ u >= v)
```

Listing 4.5: Conjecture 1 as an axiom in Lean

```
def swing4_start' : ℝ := -0.52359778
lemma swing_4_single_point (p₀ v₀ u : ℝ)
   (h₀ : p₀ = swing4_start') (h₁ : v₀ = 0) (h₂ : u = -1):
   ∃ t : ℕ, position t p₀ v₀ u <= swing3_start'
   ∧ velocity t p₀ v₀ u >= 0 := by {
      rw [h₀, h₁, h₂, swing4_start', swing3_start']
      use 2 -- Explicit choice of t to prove existence.
      repeat (
         first
         |
            unfold position
            dsimp only [max_x', min_x']
            dsimp [if_pos]
         |
            unfold velocity
            dsimp only [max_velocity', min_velocity', gravity', force', w']
            dsimp [if_pos]
      )
      dsimp only [cos]
      norm_num
   }
```

Listing 4.6: Automated proof to validate that a swing can reach the goal point from a single start point. norm_num is a tactic which searches for solutions to (in)equalities and supports most arithmetic operators over multiple fields. We use it to validate the inequality for position is correct.

starting position reach an initial condition in swing_4 you can chain all the swings together from the initial position such that you reach the goal.

If there exists a proof for the Conjectures 1 and 2, then the result follows as shown.

4.3.1.3 Limitations

In Listing 4.6, our verifier runs in exponential time of t. This is due to the unfolding of the definition of the functions in Listing 4.4. Because we unfold from $t \to 0$ then each velocity term multiplies the number of terms by 2.

```
theorem swing_4 (p₀ v₀ u : ℝ)
   (h₀ : p₀ >= swing4_start') (h₁ : v₀ <= 0) (h₂ : u = -1):
   ∃ t : ℕ, position t p₀ v₀ u <= swing3_start'
   ∧ velocity t p₀ v₀ u >= 0 := by {
      let p_s := swing4_start'
      have h_p : p_s = swing4_start' := by {rfl}
      let v_s : ℝ := 0
      have h_v : v_s = 0 := by {rfl}
      have h_0 : p₀ ≥ p_s := by {simp [p_s]; exact h₀}
      have h_1 : v_s ≥ v₀ := by {simp [v_s]; exact h₁}
      obtain ⟨t_1, swing4_result_single⟩ := swing_4_single_point _ _ _ h_p h_v h₂
      let p_t := position t_1 p_s v_s u
      let v_t := velocity t_1 p_s v_s u
      have p_f : p_t ≤ swing3_start' := by {
         apply And.left swing4_result_single
      }
      have v_f : v_t ≥  0 := by {
         apply And.right swing4_result_single
      }
      obtain res := reach_region_further_left t_1 p_s p₀ v_s v₀ u p_t v_t h_0 h_1
    h₂
      simp [p_t, v_t] at res
      obtain ⟨t_f, statement_t_f⟩ := res
      use t_f
      and_intros
      apply le_trans statement_t_f.left p_f
      apply le_trans v_f statement_t_f.right
   }
```

Listing 4.7: We validate that given a point in the swing region, all points higher up on the mountain will also reach the same position.

If the verifier were to instead compute the position and velocity from $0 \to t$ then the verifier may run in polynomial time of t. Because we can just compute the values elsewhere, the computation that produces the values used as boundary points for the regions proves their existence. So for the purposes of our proof, we can take these other region boundaries as conjectures.

The full proof provided relies on 4.5, while we do not provide the proof it is most likely true over the restricted domain of the problem up to the goal. Similar to the problems in our verifier in Listing 4.6, we would have an exponential number of terms to unfold had we have any specific points to evaluate.

We leave this as a future works.

```
obtain ⟨t₁, initial_swing_result⟩ := initial_swing p₀ v₀ u h₀ h_1' h_2'
have p_1 := initial_swing_result.left
have v_1 := initial_swing_result.right
let u_1 : ℝ := -1 -- Just assuming control.
have u_1' : u_1 = -1 := by {
   rfl
}
obtain ⟨t_2, swing4_result⟩ := swing_4 _ _ u_1 p_1 v_1 u_1'
have p_2 := swing4_result.left
have v_2 := swing4_result.right
let u_2 : ℝ := 1 -- Just assuming control.
have u_2' : u_2 = 1 := by {
   rfl
}
obtain ⟨t_3, swing3_result⟩ := swing_3 _ _ u_2 p_2 v_2 u_2'
have p_3 := swing3_result.left
have v_3 := swing3_result.right
let u_3 : ℝ := -1 -- Just assuming control.
have u_3' : u_3 = -1 := by {
   rfl
}
obtain ⟨t_4, swing2_result⟩ := swing_2 _ _ u_3 p_3 v_3 u_3'
have p_4 := swing2_result.left
have v_4 := swing2_result.right
let u_4 : ℝ := 1 -- Just assuming control.
have u_4' : u_4 = 1 := by {
   rfl
}
obtain ⟨t_5, swing1_result⟩ := swing_1 _ _ u_4 p_4 v_4 u_4' -- This states
the goal
```

Listing 4.8: Chaining multiple swings together to show that a set of controls brings the Mountain Car to the goal.

4.3.2 KeYmaera X Model for Mountain Car

To prove that Mountain Car can be solved we can try to prove that for all runs of the hybrid program, that we will always reach the goal. To do this we define the Mountain Car program as a hybrid system in KeYmaera X. Note that due to limitations in KeYmaera X, Trigonometric functions and π cannot be represented like in Lean 4. So we must use a rational approximate for both. For our Trigonometric functions we use the MacLaurin series. With high enough degree polynomials, the error is negligible for the system. See Listing 4.9.

Listing 4.9: KeYmaera X program description for Mountain Car

```
Theorem "Mountain Car V3"
Definitions
  Real C = -3.1415/6;
  Real G = 3.1415/6;
  Real sin(Real x) = x - (x^3)/6 + (x^5)/120;
  Real cos(Real x) = 1 - (x^2)/2 + (x^4)/24;
End.
ProgramVariables  /* program variables may change their value over
    time */
    Real p;       /* position of car xvalue. */
    Real v;       /* velocity */
    Real u;       /* current acceleration chosen by controller */
End.
Problem
    p >= -0.6 & p <= 0.4 & v = 0
    ->
    [
        {
            {
                {?(v >= 0); u := 1;} ++
                {?(v < 0); u := -1;}
            }
            {
                {?(p < -1.2); p := -1.2; v := 0;}
                {?(p > 0.6); p := 0.6; v := 0;}
            }
            {
                {?(v < -0.07); v := -0.07;}
                {?(v > 0.07); v:= 0.07;}
            }
            {p' = v}
            {v'= u*0.0015 - 0.0025*cos(3*p)}
        }* @invariant(????)
    ] p >= G
End.
```

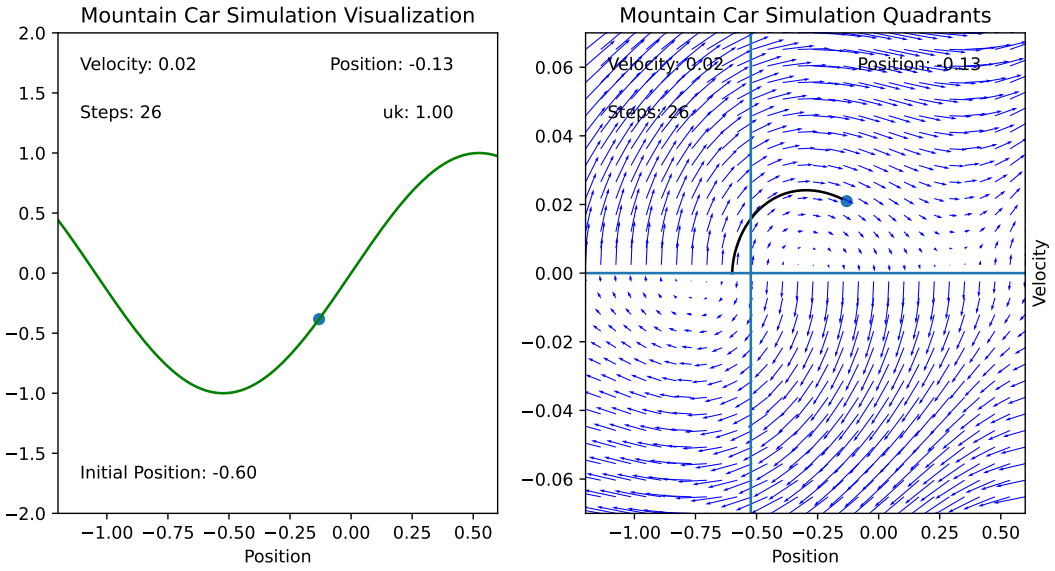


Figure 4.4: Position view and phase diagram for the simple controller M. Arrows follow differential equations' evolution. The top-right quadrant could have multiple invariants as the vector for change in position and velocity varies. Whereas, for example in the top-left quadrant we can generally see both the velocity and position component are positive, so a different invariant may work here.

4.3.2.1 Limitations

Proving this system is much harder than the system of the damped bouncing ball. Because there is no closed form solution to the differential equation we rely on finding a good loop invariant. However, due to complexities in the coupled differential equations, there are a lot of non-linearities. See Figure 4.4. Each quadrant in Figure 4.4 has possibly different invariants and some may have multiple. So finding the invariant for the system as a whole would be difficult. If we were able to come up with an invariant then this proof could tell us constraints about F, w, and G. If u is a controller that has non-deterministic choice, and we can find a sufficiently strong invariant to prove its safety, then we may learn more about what u could solve Mountain Car.

5. PHYSICAL PRINCIPLES AND ANALYSIS

In this chapter, we detail efforts to analytically solve Mountain Car. We hope to gain insight through physical properties as results from this may lend to explaining or verifying more complex controllers.

5.1 Approximations and Assumptions

Note that because the domain is restricted to a single period, we could consider using a local approximation for the transcendental functions sin and cos. This is specifically useful for when we will try to solve the transcendental equations to find their roots. And is of particular usefulness when working with other theorem provers. We use the MacLaurin series for both sin and cos with k terms.

$$\sin(x) \approx \sum_{n=0}^{k} (-1)^n \frac{x^{(2n+1)}}{(2n+1)!} \tag{5.1}$$

We could use the remainder theorem to find how much error this approximation gives us, and choose k such that it is locally accurate. So far we are using $k = 5$.

5.2 Proof of Correctness For Continuous Dynamics

We prove that the simple controller in (3.8) will always reach the goal of the generalized Mountain Car. That is there exists some $t > 0$ such that $x(t) \geq \frac{\pi}{2w}$ with initial position (3.5).

To do this, we will show that for a given setup of the Mountain Car problem, there exists some way to reach the goal with final velocity of $v_f = 0$ when beginning in the initial position space, and that our controller can reach the goal.

For now, we will assume the following values:

$$F = 0.001 \qquad G = 0.0025 \qquad w = 3 \tag{5.2}$$

Thus our positions and initial position x_0 will be in domain:

$$x \in \left[\frac{-6}{5}, \frac{9}{15}\right] \qquad\qquad x_0 \in \left[\frac{-3}{5}, \frac{-2}{5}\right] \tag{5.3}$$

We wish to show that there exists some way to get to $x_f = \frac{\pi}{6}$ with $v_f = 0$. There may be many possible points, including if the velocity is capped. We will find a point where if the velocity is $v = 0$ then it is possible to reach x_f. Recall the differential equation for $v'(t)$ in (3.1). We will treat $u(x(t), v(t))$ as a constant. We will then rewrite (3.1) via the chain rule. Note that:

$$\begin{aligned} v' = \frac{dv}{dt} &= \frac{dv}{dx}\frac{dx}{dt} \\ v' &= v\frac{dv}{dx} \end{aligned} \tag{5.4}$$

Note that $\frac{dx}{dt} = v$ from (3.2). We then take the integral of (5.4) with respect to x, we get:

$$\begin{aligned} \int v\frac{dv}{dx}dx &= \int (Fu - G\cos(3x))dx \\ \int v\ dv &= \int Fu\,dx - \int G\cos(3x)\,dx \\ \frac{v^2}{2} &= (Fu)x - \frac{G}{3}\sin(3x) + C \end{aligned} \tag{5.5}$$

Assuming we have some initial velocity v_i and some initial position x_i we can solve for the integration constant to be:

$$C = \frac{v_i^2}{2} - (Fu)x_i + \frac{G}{3}\sin(3x_i) \tag{5.6}$$

Thus we can solve for the final equation for velocity in terms of position as:

$$v(x) = \pm\sqrt{(2Fu)x - \frac{2G}{3}\sin(3x) + 2C} \tag{5.7}$$

From this equation we can determine what initial position will take us to x_f. So

plugging in our values for $v_i = 0, v = 0$, and x_f we get:

$$\begin{aligned} v(x) &= \pm\sqrt{(2Fu)x - \frac{2G}{3}\sin(3x) + 2C} \\ 0 &= \pm\sqrt{2Fux_f - \frac{2G}{3}\sin(3x_f) + 2(\frac{G}{3}\sin(3x_i) - Fux_i)} \\ \frac{2G}{3}\sin(3x_f) - 2Fux_f &= 2(\frac{G}{3}\sin(3x_i) - Fux_i) \\ \frac{G}{3}\sin(3x_f) - Fux_f &= \frac{G}{3}\sin(3x_i) - Fux_i \end{aligned} \tag{5.8}$$

Note that (5.8) is a transcendental equation, plugging in $x_f = \pi/6$ and assuming the Mountain Car is coming from the left, hence $u = 1$:

$$\frac{G}{3} - \frac{F\pi}{6} = \frac{G}{3}\sin(3x_i) - Fx_i \tag{5.9}$$

There is no closed form solution to this problem for x_i. We can iteratively use a constrained Newton's method over the domain to find regions where we can reach the goal. To do this we use SciPy's [20] implementation for a truncated Newton algorithm and minimize the squared (5.8) set equal to zero. Note that the region that is shaded will also reach the goal but with a higher final velocity. See Figure 5.1.

Note that because we framed the question as finding the position where we may end up at the goal with zero velocity, we can do it again for the previous point we found. We do this iteratively until the entire space of initial positions is covered. See Figure 5.2.

If the goal is to the right and velocity of the car is greater than zero at the initial point then the car will go further. Also, if the position is further to the left from the initial point there will be more velocity at the original initial point when it is reached. So any points to the opposite side of the reachable point will reach the goal point. Any point in the domain will thusly lead to another goal point. Thus the controller will escape after finite time.

5.2.1 Discrete Regions

The goal points for the discrete problem in (3.3) can be found in Figure 5.3

To find these regions we use binary search with 20 refinement steps where. Each point that is found during the binary search is then used as the next goal point. All code for this can be found in Appendix B.

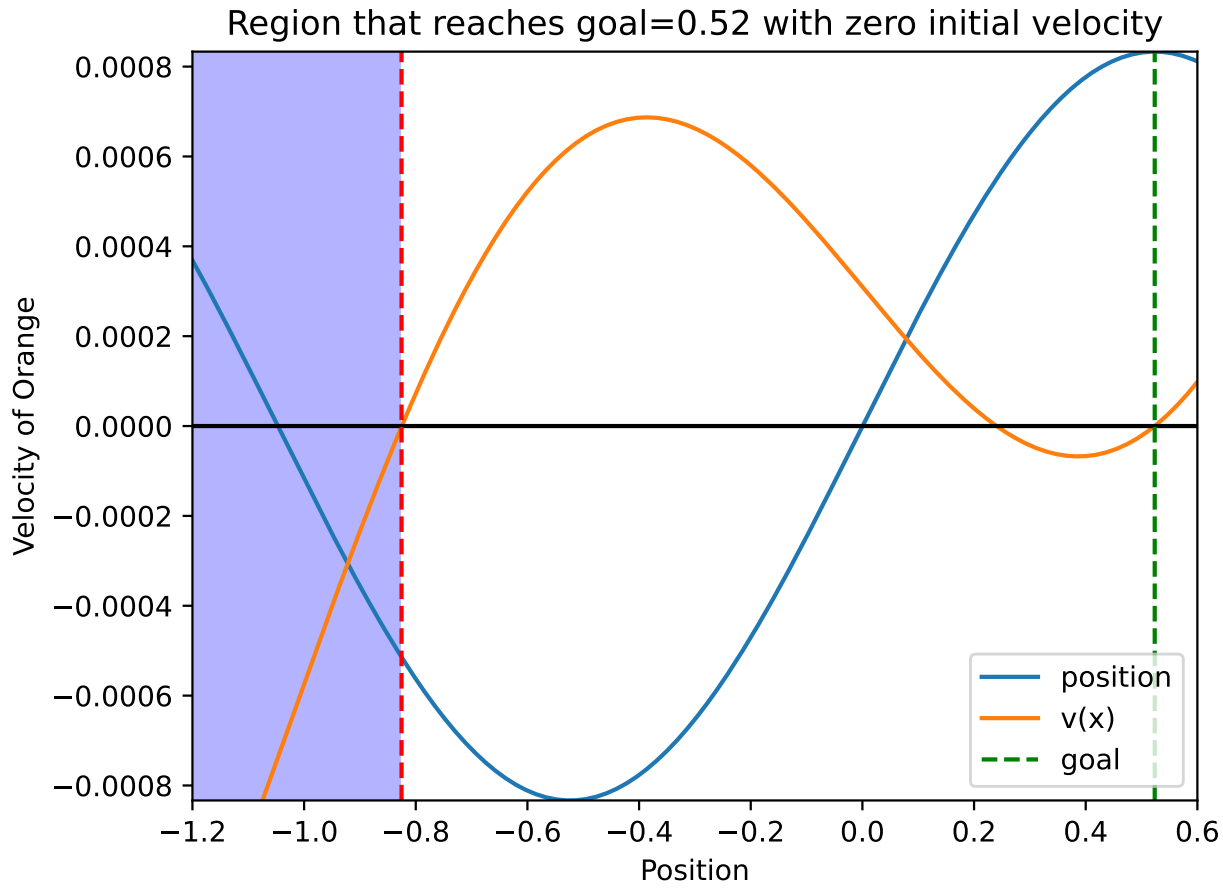
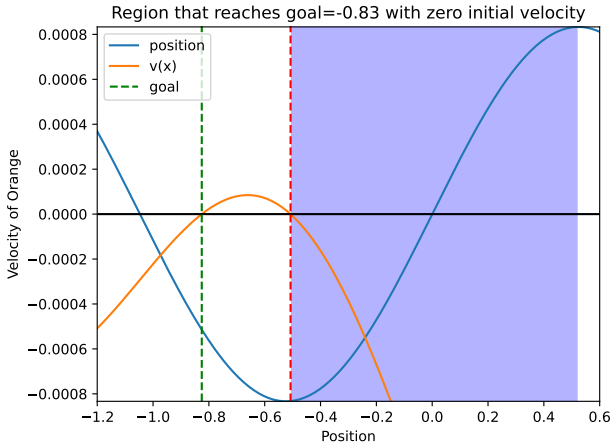
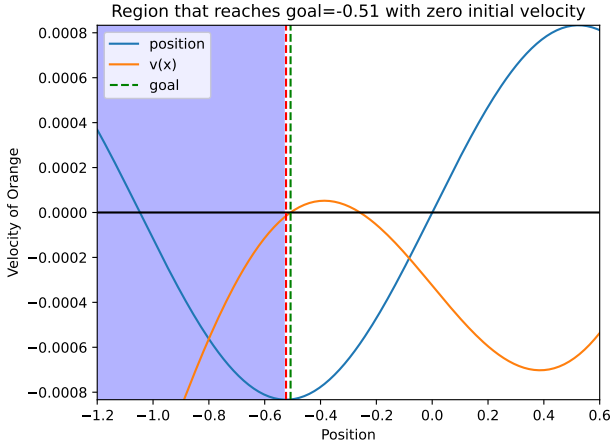


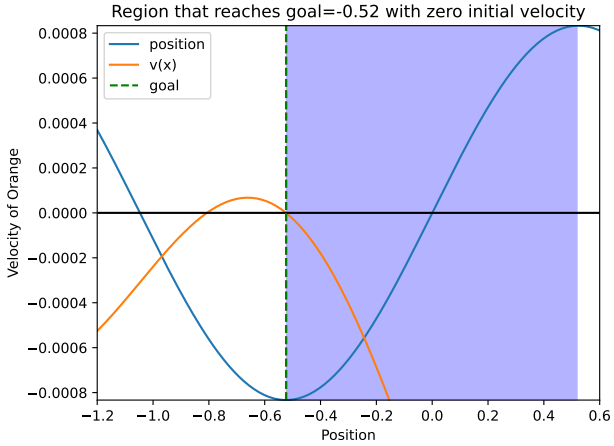
Figure 5.1: Region that can reach the goal



(a) Reaches Region in Figure 5.1



(b) Reaches Region in Figure 5.2a



(c) Reaches Region in Figure 5.2b

Figure 5.2: Other Regions in Mountain Car

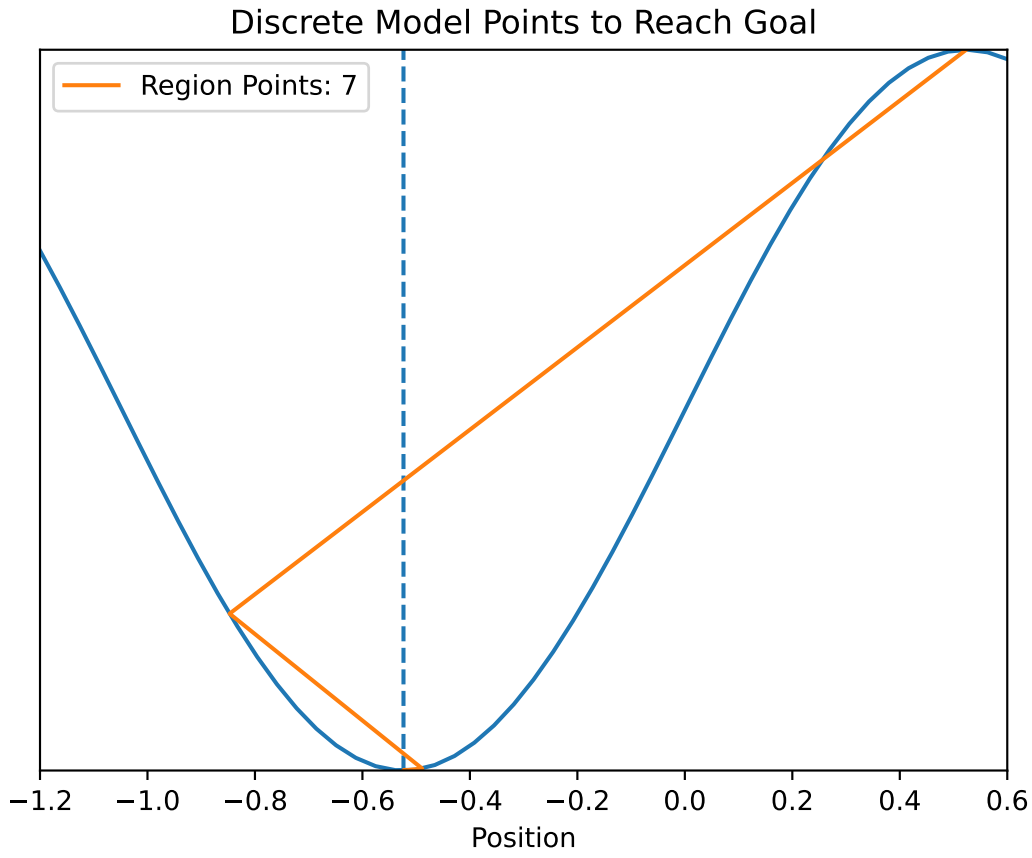


Figure 5.3: Swing Path for Discrete Mountain Car

6. CONCLUSION

6.1 Discussion

In this work, we surveyed multiple state-of-the-art tools and different analytical methods to help us prove properties about the toy problem. For a simple toy problem such as Mountain Car there are still many non-linearities and complexities that make it difficult to verify. We used theorem proving and physical principles to prove that the simple controller, under our conjectures, is able to safely reach the goal. While model checking approaches can show this controller is able to reach the goal, they lack explainability. See Table 6.1 for a summary of results. The approaches of model checking and theorem proving are fundamentally different. Model checking relies on exhaustive data-driven approaches, but theorem proving can make use of physical principles. While physical principles may be harder to show they give more insight into the dynamics and constraints of the problem. The state of the art is capable of verifying many properties but as we discussed in Chapter 2, that comes at a cost of scalability: finite time horizon, or lack of safety guarantees.

6.2 Future Work

We provided a formal proof that can show for a simple controller that the Mountain Car problem can be solved for any valid initial condition given some assumptions on natural properties.

This work explored using automated and manual theorem proving as a scalable approach to verifying controllers are safe to address challenges with the state of the art. One of the limitations is the exponential length inequalities that we must verify. A future work would be to rewrite the verifiers to prove bottom up to run in polynomial time in the size of t.

	Model Checking	Theorem Proving
Approach	Data-driven	Physical Principles
Scalability	No	Yes
Expressiveness	Bounded Time	1st order logic, differential equations
Explainability	No	Yes

Table 6.1: Summary of Survey Results

We examined a very simple controller, so it is of interest to apply these techniques to a class of controllers. One could examine the family of all linear controllers, where the decision boundary is a line in the state space.

We do not examine any noise in the perception component of our controller as it acts directly on the states. One could model some noise on the sensors and create more conservative swing regions that still solve the problem.

We limit our selves to a simple controller based on the sign of the velocity, but it would be more useful to examine larger and more complex controllers. Specifically, it would be of interest to look at neural network controllers. If we are able to find constraints on some controllers that can solve our toy-problem, then those constraints may be useful in training safe neural network controllers.

LITERATURE CITED

[1] Guardian staff and agencies, "Tesla Autopilot feature was involved in 13 fatal crashes, US regulator says," *The Guardian*, April 2024, Accessed Feb. 2, 2025. [Online]. Available: https://www.theguardian.com/technology/2024/apr/26/tesla-autopilot-fatal-crash

[2] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. A. Sallab, S. Yogamani, and P. Pérez, "Deep Reinforcement Learning for Autonomous Driving: A Survey," *IEEE Trans. Intell. Transp. Syst.*, vol. 23, no. 6, pp. 4909–4926, 2022. [Online]. Available: https://www.doi.org/10.1109/TITS.2021.3054625

[3] Z. Wang, W. Pan, H. Li, X. Wang, and Q. Zuo, "Review of Deep Reinforcement Learning Approaches for Conflict Resolution in Air Traffic Control," *Aerosp.*, vol. 9, no. 6, 2022. [Online]. Available: https://www.doi.org/10.3390/aerospace9060294

[4] N. Hamilton, K. Dunlap, and K. L. Hobbs, " Investigating the Impact of Choice on Deep Reinforcement Learning for Space Controls ," in *2024 IEEE 10th Int. Conf. Space Mission Chall. Inf. Technol. (SMC-IT)*. Los Alamitos, CA, USA: IEEE Computer Society, Jul. 2024, pp. 56–66. [Online]. Available: https://www.doi.org/10.1109/SMC-IT61443.2024.00014

[5] M. Tipaldi, R. Iervolino, and P. R. Massenio, "Reinforcement learning in spacecraft control applications: Advances, prospects, and challenges," *Annu. Rev. Control*, vol. 54, pp. 1–23, 2022. [Online]. Available: https://doi.org/10.1016/j.arcontrol.2022.07.004

[6] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks," in *Comput. Aided Verif.*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 2017, pp. 97–117.

[7] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Commun. ACM*, vol. 54, no. 9, p. 69–77, Sep. 2011. [Online]. Available: https://doi.org/10.1145/1995376.1995394

[8] J.-D. Quesel, S. Mitsch, S. Loos, N. Aréchiga, and A. Platzer, "How to model and prove hybrid systems with KeYmaera: a tutorial on safety," *Int. J. Softw. Tools Technol. Transf.*, vol. 18, no. 1, pp. 67–91, Feb 2016. [Online]. Available: https://doi.org/10.1007/s10009-015-0367-0

[9] M. Schmalz, H. Völzer, and D. Varacca, "Model Checking Almost All Paths Can Be Less Expensive Than Checking All Paths," in *Found. Softw. Technol. Theor. Comput. Sci. (FSTTCS)*, V. Arvind and S. Prasad, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 532–543.

[10] S. Gao, S. Kong, and E. M. Clarke, "dreal: An smt solver for nonlinear theories over the reals," in *Proc. 24th Int. Conf. Autom. Deduct. (CADE)*, M. P. Bonacina, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 208–214.

[11] S. Kong, S. Gao, W. Chen, and E. Clarke, "dReach: δ-Reachability Analysis for Hybrid Systems," in *Proc. Tools Alg. Constr. Anal. Syst. (TACAS)*, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 200–205.

[12] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Verisig: verifying safety properties of hybrid systems with neural network controllers," in *Proc. 22nd ACM Int. Conf. Hybrid Syst. Comput. Control*, ser. HSCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 169–178. [Online]. Available: https://doi.org/10.1145/3302504.3311806

[13] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An Analyzer for Non-linear Hybrid Systems," in *Comput. Aided Verif.*, N. Sharygina and H. Veith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 258–263.

[14] P. Jin, Y. Wang, and M. Zhang, "Efficient LTL Model Checking of Deep Reinforcement Learning Systems using Policy Extraction," in *34th Int. Conf. Softw. Eng. Knowl. Eng. (SEKE)*, 07 2022, pp. 357–362. [Online]. Available: https://www.doi.org/10.18293/SEKE2022-029

[15] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," 2016. [Online]. Available: https://doi.org/10.48550/arXiv.1606.01540

[16] L. d. Moura and S. Ullrich, "The Lean 4 Theorem Prover and Programming Language," in *Proc. 28th Int. Conf. Autom. Deduct. (CADE)*, A. Platzer and G. Sutcliffe, Eds. Cham: Springer International Publishing, 2021, pp. 625–635.

[17] The mathlib Community, "The lean mathematical library," in *Proc. 9th ACM SIGPLAN Int. Conf. Certif. Programs Proofs*, ser. CPP 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 367–381. [Online]. Available: https://doi.org/10.1145/3372885.3373824

[18] A. Platzer, "Differential Dynamic Logic for Verifying Parametric Hybrid Systems." in *TABLEAUX*, ser. LNCS, N. Olivetti, Ed., vol. 4548. Springer, 2007, pp. 216–232.

[19] ——, *Logical Foundations of Cyber-Physical Systems*. Cham: Springer, 2018, ch. 7.

[20] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nat. Methods*, vol. 17,

pp. 261–272, 2020. [Online]. Available:
https://www.doi.org/10.1038/s41592-019-0686-2

APPENDIX A
PROOFS

```
theorem sin_cos_cos_sin_period (h: x ≥ 0 ∧ x ≤ Real.pi): Real.sin (Real.cos x) ≤
    Real.cos (Real.sin x) := by
{
  cases le_or_lt x (Real.pi / 2) with
  | inl h1 => -- Case: x ≤ π/2
    exact goal_left_half x ⟨h.left, h1⟩
  | inr h2 => -- Case: x > π/2, so x is in [π/2, π]
    have h3 : x ≥ Real.pi/2 := by {linarith}
    exact goal_right_half ⟨h3, h.right⟩
}
```

Listing A.1: Lean 4 Proof for $\sin(\cos x) \leq \cos\sin(x)$ over one period (1/5). The proof begins by breaking the theorem into two cases. The case of $x \leq \pi/2$ and $x > \pi/2$. Because the functions are periodic, it is sufficient to show that over one period if the inequality is true then it is true for all x.

```
lemma goal_left_half (x : ℝ) (h: x ≥ 0 ∧ x ≤ Real.pi/2): Real.sin (Real.cos x)
   ≤ Real.cos (Real.sin x) := by
{
  obtain lhs := sin_cos_lt_cos x h
  obtain rhs := cos_lte_cos_sin_x x h
  apply le_trans lhs rhs
}
lemma goal_right_half (h: x ≥ Real.pi/2 ∧ x ≤ Real.pi): Real.sin (Real.cos x) ≤
   Real.cos (Real.sin x) := by
{
  obtain lhs := sin_cos_lte_zero x h
  obtain rhs := cos_sin_gte_zero x
  linarith
}
```

Listing A.2: Lean 4 Proof for $\sin(\cos x) \leq \cos\sin(x)$ over one period (2/5). Continuing, we break each subgoal into two parts. We start with the left goal.

```
axiom sin_x_lte_x (x: ℝ ) (h: x ≥ 0): Real.sin x <= x -- Well known from calc.
lemma sin_cos_lt_cos (x : ℝ) (h: x ≥ 0 ∧ x ≤ Real.pi/2):
  Real.sin (Real.cos x) ≤ Real.cos x := by
{
    have x_gt_neg_pi_div_2 : x ≥ -Real.pi/2 := by {
      linarith
    }
    have x_in_bounds : x ∈ (Set.Icc (-Real.pi/2) (Real.pi/2)) := by {
      rw [Set.mem_Icc]
      apply And.intro x_gt_neg_pi_div_2 h.right
    }
    have h_cos : Real.cos x ≥ 0 := by {
      apply Real.cos_nonneg_of_mem_Icc
      rw [neg_div] at x_in_bounds
      exact x_in_bounds
    }
    apply sin_x_lte_x (Real.cos x)
    exact h_cos
}
```

Listing A.3: Lean 4 Proof for $\sin(\cos x) \leq \cos\sin(x)$ over one period (3/5). We prove this using the linear arithmetic automated tactic. This searches our hypotheses set in order to prove our goal. We rely on an axiom to show that $\sin(x) \leq x \; \forall x \geq 0$. But this can be shown from the definitions of $\sin(x)$ as well.

```
lemma cos_le_cos_of_nonneg_of_le_pi_div_2 {x y : ℝ} (hx₁ : 0 ≤ x)
(hy₂ : y ≤ Real.pi/2) (hxy : x ≤ y): Real.cos y ≤ Real.cos x := by {
  have y_leq_pi_div_2 : y ≤ Real.pi := by {linarith}
  exact Real.cos_le_cos_of_nonneg_of_le_pi hx₁ y_leq_pi_div_2 hxy
}

lemma cos_lte_cos_sin_x (x : ℝ) (h: x ≥ 0 ∧ x ≤ Real.pi/2):
Real.cos x ≤ Real.cos (Real.sin x) := by
{
    have x_in_bounds : x ∈ Set.Icc 0 Real.pi := by {
      rw [Set.mem_Icc]
      have x_leq_pi : x ≤ Real.pi := by {linarith}
      apply And.intro h.left x_leq_pi
    }
    have sin_x_geq_0 : Real.sin x ≥ 0 := by {
      apply Real.sin_nonneg_of_mem_Icc
      exact x_in_bounds
    }
    obtain res := sin_x_lte_x x h.left
    exact cos_le_cos_of_nonneg_of_le_pi_div_2 sin_x_geq_0 h.right res
}
```

Listing A.4: Lean 4 Proof for $\sin(\cos x) \leq \cos\sin(x)$ over one period (4/5). We now apply simple rewriting tactics for the right goal.

```
lemma sin_cos_lte_zero (x : ℝ) (h: x ≥ Real.pi/2 ∧ x ≤ Real.pi):
Real.sin (Real.cos x) <= 0 := by
{
  apply Real.sin_nonpos_of_nonnpos_of_neg_pi_le
  apply Real.cos_nonpos_of_pi_div_two_le_of_le
  linarith
  linarith
  have cos_x_geq_neg_1 : Real.cos x ≥ -1 := by {apply Real.neg_one_le_cos}
  have pi_gt_one : Real.pi > 1 := by {
    obtain _ := Real.one_le_pi_div_two
    linarith
  }
  linarith
lemma cos_sin_gte_zero (x : ℝ):
Real.cos (Real.sin x) ≥ 0 := by {
  apply Real.cos_nonneg_of_neg_pi_div_two_le_of_le
  obtain _ := Real.neg_one_le_sin x
  obtain _ := Real.one_le_pi_div_two
  linarith
  obtain _ := Real.sin_le_one x
  obtain _ := Real.one_le_pi_div_two
  linarith
}
```

Listing A.5: Lean 4 Proof for $\sin(\cos x) \leq \cos\sin(x)$ over one period (5/5). Finally, we use automated tactics again to finish the remaining portions of the proof of the right goal. Thus proving the inequality.

```
def gravity' : ℝ := 0.0025
def force' : ℝ := 0.001
def w' : ℝ := 3
def max_velocity' : ℝ := 0.07
def min_velocity' : ℝ := -0.07
def max_x' : ℝ := 0.6
def min_x' : ℝ := -1.2
-- def goal' : ℝ := -0.52359942
def goal' : ℝ := 0.5

def cos (x : ℝ) : ℝ := -- Cannot use polynomials or rationals as that becomes
-- uncomputable
   1 - 0.5*x^2 + 0.041666*x*x*x*x - 0.00138888888*x*x*x*x*x*x +
   0.00002480158*x*x*x*x*x*x*x*x - 2.75573192e-7*x*x*x*x*x*x*x*x*x*x -- However
    much precision you want.

mutual
   def velocity (t : ℕ) (x₀ v₀ u: ℝ) : ℝ :=
      if t = 0 then max (min v₀ max_velocity') (min_velocity') -- Clip velocity
      else
      max (
         min max_velocity'
         (
            velocity (t - 1) x₀ v₀ u +
            (u * force') -
            gravity' * cos (w' * position (t - 1) x₀ v₀ u)
         )
      ) min_velocity'


   def position (t : ℕ) (x₀ v₀ u : ℝ) : ℝ :=
      if t = 0 then max (min x₀ max_x') min_x'
      else
      max (
         min max_x'
         (
            position (t - 1) x₀ v₀ u + (velocity t x₀ v₀ u) * 0.001
         )
      ) min_x'
end
```

Listing A.6: Lean 4 proof for Mountain Car (1/5). We start with definitions, due to the need to compute the boundary positions of each region, we need a computable version of the Trigonometric functions. Thus, we use a Taylor approximation for cosine. We define the velocity and position as the recurrence relations in (3.4).

```
    obtain ⟨t₁, initial_swing_result⟩ := initial_swing p₀ v₀ u h₀ h_1' h_2'
    have p_1 := initial_swing_result.left
    have v_1 := initial_swing_result.right
    let u_1 : ℝ := -1 -- Just assuming control.
    have u_1' : u_1 = -1 := by {
       rfl
    }
    obtain ⟨t_2, swing4_result⟩ := swing_4 _ _ u_1 p_1 v_1 u_1'
    have p_2 := swing4_result.left
    have v_2 := swing4_result.right
    let u_2 : ℝ := 1 -- Just assuming control.
    have u_2' : u_2 = 1 := by {
       rfl
    }
    obtain ⟨t_3, swing3_result⟩ := swing_3 _ _ u_2 p_2 v_2 u_2'
    have p_3 := swing3_result.left
    have v_3 := swing3_result.right
    let u_3 : ℝ := -1 -- Just assuming control.
    have u_3' : u_3 = -1 := by {
       rfl
    }
    obtain ⟨t_4, swing2_result⟩ := swing_2 _ _ u_3 p_3 v_3 u_3'
    have p_4 := swing2_result.left
    have v_4 := swing2_result.right
    let u_4 : ℝ := 1 -- Just assuming control.
    have u_4' : u_4 = 1 := by {
       rfl
    }
    obtain ⟨t_5, swing1_result⟩ := swing_1 _ _ u_4 p_4 v_4 u_4' -- This states
  the goal
```

Listing A.7: Lean 4 proof for Mountain Car (2/5). This is a top-down view of the proof tactic. We must chain the regions together, starting from the initial position set. Until we reach the final swing to the goal.

```
def swing1_start' : ℝ  := -0.84715057
theorem swing_1 (p₀ v₀ u : ℝ)
   (h₀ : p₀ <= swing1_start') (h₁ : v₀ >= 0) (h₂ : u = 1):
   ∃ t : ℕ, position t p₀ v₀ u >= goal' := by {
      sorry
   }

def swing2_start' : ℝ  := -0.4890173
theorem swing_2 (p₀ v₀ u : ℝ)
   (h₀ : p₀ >= swing2_start') (h₁ : v₀ <= 0) (h₂ : u = -1):
   ∃ t : ℕ, position t p₀ v₀ u <= swing1_start'
   ∧ velocity t p₀ v₀ u >= 0 := by {
      sorry
   }

def swing3_start' : ℝ := -0.52359942
theorem swing_3 (p₀ v₀ u : ℝ)
   (h₀ : p₀ <= swing3_start') (h₁ : v₀ >= 0) (h₂ : u = 1):
   ∃ t : ℕ, position t p₀ v₀ u >= swing2_start'
   ∧ velocity t p₀ v₀ u <= 0 := by {
      sorry
   }
```

Listing A.8: Lean 4 proof for Mountain Car (3/5). The template for each individual swing is given to show the conditions necessary, but we will show only the first swing proof as all other proofs are the same with varying verification time.

```
def swing4_start' : ℝ := -0.52359778
lemma swing_4_single_point (p₀ v₀ u : ℝ)
   (h₀ : p₀ = swing4_start') (h₁ : v₀ = 0) (h₂ : u = -1):
   ∃ t : ℕ, position t p₀ v₀ u <= swing3_start'
   ∧ velocity t p₀ v₀ u >= 0 := by {
      rw [h₀, h₁, h₂, swing4_start', swing3_start']
      use 2 -- Explicit choice of t to prove existence.
      repeat (
         first
         |
            unfold position
            dsimp only [max_x', min_x']
            dsimp [if_pos]
         |
            unfold velocity
            dsimp only [max_velocity', min_velocity', gravity', force', w']
            dsimp [if_pos]
      )
      dsimp only [cos]
      norm_num
   }
```

Listing A.9: Lean 4 proof for Mountain Car (4/5). This is a verifier, using results from external computation, one can find the number of steps to complete the swing from the boundary position. After which it becomes unfolding and inequality verification. This is the boundary point verification lemma.

```
axiom reach_region_further_left (t₀ : ℕ) (p₀ p₁ v₀ v₁ u p v: ℝ)
   (h₀ : p₁ >= p₀) (h₁ : v₁ <= v₀) (h₂ : u = -1):
   (position t₀ p₀ v₀ u = p ∧ velocity t₀ p₀ v₀ u = v) → (∃ t₁ : ℕ, position t₁
   p₁ v₁ u <= p ∧ velocity t₁ p₁ v₁ u >= v)
theorem swing_4 (p₀ v₀ u : ℝ)
   (h₀ : p₀ >= swing4_start') (h₁ : v₀ <= 0) (h₂ : u = -1):
   ∃ t : ℕ, position t p₀ v₀ u <= swing3_start'
   ∧ velocity t p₀ v₀ u >= 0 := by {
      let p_s := swing4_start'
      have h_p : p_s = swing4_start' := by {rfl}
      let v_s : ℝ := 0
      have h_v : v_s = 0 := by {rfl}
      have h_0 : p₀ ≥ p_s := by {simp [p_s]; exact h₀}
      have h_1 : v_s ≥ v₀ := by {simp [v_s]; exact h₁}
      obtain ⟨t_1, swing4_result_single⟩ := swing_4_single_point _ _ _ h_p h_v h₂
      let p_t := position t_1 p_s v_s u
      let v_t := velocity t_1 p_s v_s u
      have p_f : p_t ≤ swing3_start' := by {
         apply And.left swing4_result_single
      }
      have v_f : v_t ≥  0 := by {
         apply And.right swing4_result_single
      }
      obtain res := reach_region_further_left t_1 p_s p₀ v_s v₀ u p_t v_t h_0 h_1
   h₂
      simp [p_t, v_t] at res
      obtain ⟨t_f, statement_t_f⟩ := res
      use t_f
      and_intros
      apply le_trans statement_t_f.left p_f
      apply le_trans v_f statement_t_f.right
   }


theorem initial_swing (p₀ v₀ u : ℝ)
   (h₀ : p₀ >= -0.6 ∧ p₀ <= -0.4) (h₁ : v₀ <= 0) (h₂ : u = -1 ∨ u = 1 ∨ u = 0):
   ∃ t : ℕ, position t p₀ v₀ u >= swing4_start'
   ∧ velocity t p₀ v₀ u <= 0 := by {
      sorry
   }
```

Listing A.10: Lean 4 proof for Mountain Car (5/5). We conjecture that if you are higher up on the slope, with a higher velocity in the direction going down, you will swing further than a reference point with a lower position or velocity. Using this axiom and the result of the boundary point verification lemma, we can then prove that for the entire region of the swing, we can reach the next goal.

Listing A.11: Hybrid Description for Damped Bouncing Ball. The bouncing ball's safety property is that the ball doesn't go higher than it initially began and always is above the ground. The invariant comes from the differential equation solution directly.

```
ArchiveEntry "Damped Bouncing Ball"
  Description "Quantum the acrophobic bouncing ball, modified
  from Chapter 7 of 'Logical Foundations of Cyber-Physical Systems
     ' with
  additional damping".
  Title "Bouncing ball".

Definitions          /* function symbols cannot change their value
   */
  Real H;             /* initial height */
  Real g;             /* gravity */
  Real c;             /* damping coefficient */
End.

ProgramVariables  /* program variables may change their value over
    time */
  Real x;             /* height */
  Real v;             /* velocity */
End.

Problem
  x>=0 & x=H
  & v=0 & g>0 & 1>=c & c>=0
 ->
  [
    {
      {x'=v,v'=-g&x>=0}
      {?x=0; v:=-c*v; x:=0;  ++  ?x!=0;}
    }*@invariant(2*g*x <= 2*g*H-v^2 & x>=0 & v^2<=2*g*H)
  ] (x>=0 & x<=H)
End.

End.
```

Listing A.12: Tactics to Prove Damped Bouncing Ball. The tactics, described here are the same as those described in the sequent calculus as seen in Figure 4.2.

```
unfold;
loop("2*g()*x<=2*g()*H()-v^2&x>=0&v^2<=2*g()*H()", 'R=="[{{x'=v,v
    '=-g()&x>=0}{?x=0;v:=-c()*v;x:=0;++?x!=0;}}*](x>=0&x<=H())");
    <(
  "Init":
    QE,
  "Post":
    QE,
  "Step":
    composeb('R=="[{x'=v,v'=-g()&x>=0}{?x=0;v:=-c()*v;x:=0;++?x
        !=0;}](2*g()*x<=2*g()*H()-v^2&x>=0&v^2<=2*g()*H())");
    choiceb('R=="[{x'=v,v'=-g()&x>=0}]#[?x=0;v:=-c()*v;x:=0;++?x
        !=0;](2*g()*x<=2*g()*H()-v^2&x>=0&v^2<=2*g()*H())#");
    composeb('R=="[{x'=v,v'=-g()&x>=0}](#[?x=0;v:=-c()*v;x:=0;](2*
        g()*x<=2*g()*H()-v^2&x>=0&v^2<=2*g()*H())#&[?x!=0;](2*g()*x
        <=2*g()*H()-v^2&x>=0&v^2<=2*g()*H()))");
    testb('R=="[{x'=v,v'=-g()&x>=0}]([?x=0;][v:=-c()*v;x:=0;](2*g
        ()*x<=2*g()*H()-v^2&x>=0&v^2<=2*g()*H())&#[?x!=0;](2*g()*x
        <=2*g()*H()-v^2&x>=0&v^2<=2*g()*H())#)");
    testb('R=="[{x'=v,v'=-g()&x>=0}](#[?x=0;][v:=-c()*v;x:=0;](2*g
        ()*x<=2*g()*H()-v^2&x>=0&v^2<=2*g()*H())#&(x!=0->2*g()*x
        <=2*g()*H()-v^2&x>=0&v^2<=2*g()*H()))");
    composeb('R=="[{x'=v,v'=-g()&x>=0}]((x=0->#[v:=-c()*v;x
        :=0;](2*g()*x<=2*g()*H()-v^2&x>=0&v^2<=2*g()*H())#)&(x
        !=0->2*g()*x<=2*g()*H()-v^2&x>=0&v^2<=2*g()*H()))");
    assignb('R=="[{x'=v,v'=-g()&x>=0}]((x=0->#[v:=-c()*v;][x
        :=0;](2*g()*x<=2*g()*H()-v^2&x>=0&v^2<=2*g()*H())#)&(x
        !=0->2*g()*x<=2*g()*H()-v^2&x>=0&v^2<=2*g()*H()))");
    assignb('R=="[{x'=v,v'=-g()&x>=0}]((x=0->#[x:=0;](2*g()*x<=2*g
        ()*H()-(-c()*v)^2&x>=0&(-c()*v)^2<=2*g()*H())#)&(x!=0->2*g
        ()*x<=2*g()*H()-v^2&x>=0&v^2<=2*g()*H()))");
    ODE('R=="[{x'=v,v'=-g()&x>=0}]((x=0->2*g()*0<=2*g()*H()-(-c()*
        v)^2&0>=0&(-c()*v)^2<=2*g()*H())&(x!=0->2*g()*x<=2*g()*H()-
        v^2&x>=0&v^2<=2*g()*H()))")
)
```

APPENDIX B
CODES

In this appendix, code is provided to find the boundary points used for swing regions in Mountain Car. We provide two versions, one for discrete dynamics and one for continuous dynamics.

Listing B.1: Discrete region solver (1/3), imports and constants. We use a Taylor approximation for the cosine to match the cosine we use in our downstream proofs.

```
from numpy import sin, pi
import matplotlib.pyplot as plt
import numpy as np

G = 0.0025
F = 0.001
w = 3
VELOCITY = [-0.07,0.07]
CENTER = -pi/(2*w)
GOAL = pi/(2*w)
DOMAIN = [-18/(5*w), 9/(5*w)]


# Problem constants:
PK_MIN = -1.2
PK_MAX = 0.6
VK_MIN = -0.07
VK_MAX = 0.07
UK_MIN = -1
UK_MAX = 1
GOAL = pi/6
VALLEY_CENTER = -pi/6

def cos(x):
    return 1 - 0.5*x**2 + 0.041666*x**4 - 0.00138888888*x**6 +
       0.00002480158*x**8 - 2.75573192e-7*x**10
```

Listing B.2: Discrete region solver (2/3), simulation function for Mountain Car. Mountain Car has inelastic collisions when colliding with the boundaries. Using inelastic collisions allows for less conservative discrete regions.

```
def get_next_stat(pk: float, vk: float, uk: float) -> list[float,
    float]:
    """ Given the input of uk and current position,
        return the new position and velocity.
        pk is in the range [-1.2, 0.6]
        vk is in the range [-0.07, 0.07]
        uk is in the range [-1, 1]
    """
    FORCE = 0.001
    GRAVITY = 0.0025

    # What are my preconditions?
    # if uk > UK_MAX or uk < UK_MIN:
    #     logging.warning("uk is out of input range, limiting.")
    uk = max(min(uk, UK_MAX), UK_MIN)
    vk1 = vk + FORCE*uk - GRAVITY * cos(3*pk)
    vk1 = max(min(vk1, VK_MAX), VK_MIN)
    pk1 = pk + vk1
    pk1 = max(min(pk1, PK_MAX), PK_MIN)

    if (pk1 == PK_MAX) or (pk1 == PK_MIN):
        # Inelastic
        vk1 = 0
        # Elastic bounce
        # vk1 *= -1
    return [pk1, vk1]
```

Listing B.3: Discrete region solver (3/3), finds points that are the borders of each swing region. For the case where the car goes to the right, we bound the possible solutions to be within the left wall and the bottom of the valley. We do not check for all solutions just one.

```
MAX_REFINE=20
x_f = GOAL
pk = PK_MIN
goal_points = [x_f]
steps_li = []
interval_gr = [PK_MIN, CENTER] # interval going right
interval_gl = [CENTER, GOAL] # interval going left
vk = 0
best = None
right = True
for j in range(6):
    interval_gr_temp = interval_gr.copy()
    interval_gl_temp = interval_gl.copy()
    best_steps = 0
    for _ in range(MAX_REFINE):
        vk = 0
        steps = 0
        if right:
            # Find pk within interval that works
            inital_pk = np.mean(interval_gr_temp)
            pk = inital_pk
            while vk >= 0:
                pk,vk = get_next_stat(pk=pk,vk=vk,uk=1)
                steps += 1
                if pk >= x_f and vk <= 0:
                    break
            if pk >= x_f and vk <= 0:
                interval_gr_temp[0] = inital_pk
                best_steps = steps
                best = inital_pk
            else:
                interval_gr_temp[1] = inital_pk
        else:
            # Do the equivalent of right but for left.
    goal_points.append(best)
    steps_li.append(best_steps)
    x_f = goal_points[-1]
    right = not right
goal_points = np.array(goal_points)
```

Listing B.4: Continuous region solver (1/2), imports and definitions of transcendental functions.

```
# %%
from scipy.optimize import fmin_tnc
from numpy import sin, cos, pi
from numpy.random import uniform
import matplotlib.pyplot as plt
import numpy as np

# %%
G = 0.0025
F = 0.001
w = 3
VELOCITY = [-0.07,0.07]
CENTER = -pi/(2*w)
GOAL = pi/(2*w)
DOMAIN = [-18/(5*w), 9/(5*w)]

def f(x_i,x_f,G,F,u):
    return G/3*sin(3*x_f) - F*u*x_f - G/3*sin(3*x_i) +F*u*x_i

def fprime(x_i,G,u):
    return - G*cos(3*x_i) +F*u
```

Listing B.5: Continuous region solver (2/2), uses a constrained Newton solver to find solutions to a minimization equivalent problem. When a solution is found, the solver reinitializes and solves for the next swing from the previously found solution. This process repeats until Mountain Car is solved.

```
bounds = 0
initial_bounds = (-9/(5*w), -6/(5*w))
bounds = (DOMAIN[0],CENTER) # Initially from left wall to center
    line.
x_f = GOAL
u = 1
swing_bounds = []
break_rule = 0
max_iter = 0
fs = []
while max_iter<10:
    fs.append( lambda x,x_f=x_f,u=u : f(x, x_f, G, F, u))
    fsq = lambda x: fs[-1](x) ** 2
    fsqprime = lambda x : 2 * f(x, x_f, G, F, u) * fprime(x, G, u)
    x_i, iterations, rc = fmin_tnc(fsq, uniform(bounds[0],bounds
        [1]), fsqprime, pgtol=0.0, bounds=[bounds])
    x_i = x_i[0]
    print(bounds)
    if u == 1:
        swing_bounds.append((DOMAIN[0],x_i))
        bounds = (CENTER, x_f)
    else:
        swing_bounds.append((x_i,DOMAIN[1]))
        bounds = (x_f, CENTER)

    u = -u
    if (x_i >= initial_bounds[0] and x_i <= initial_bounds[1] and
        x_f >= initial_bounds[0] and x_f <= initial_bounds[1]):
        break_rule+=1
    if break_rule == 2:
        break
    x_f = x_i
    max_iter += 1
print(swing_bounds)
```