# Verified Cruise Control System on RC Vehicle

**Shashank Ojha**
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
shashano@cmu.edu

**Yufei Wang**
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
yufeiw2@andrew.cmu.edu

## Abstract

A lot of work has been done to model an efficient and provably safe cruise control system for vehicles. Nevertheless, there is still a large gap between the theoretical models and the physical reality of implementation that allows us to actually see the benefits of the research in the real world. We strive to take an intelligent cruise control model and implement it in an RC Vehicle traveling on a one-dimensional path. Due to limitations in hardware and time constraints, however, we had to modify the models from previous research and focus on a static obstacle model. We have proofs of our modified models and a working demo of the static obstacle model on an RC car. The work done in this project sheds light on some of the challenges in implementation and hardware when converting a theoretical model to a viable product. While our project focused on a static model, the work is not too far from a dynamic obstacle model.

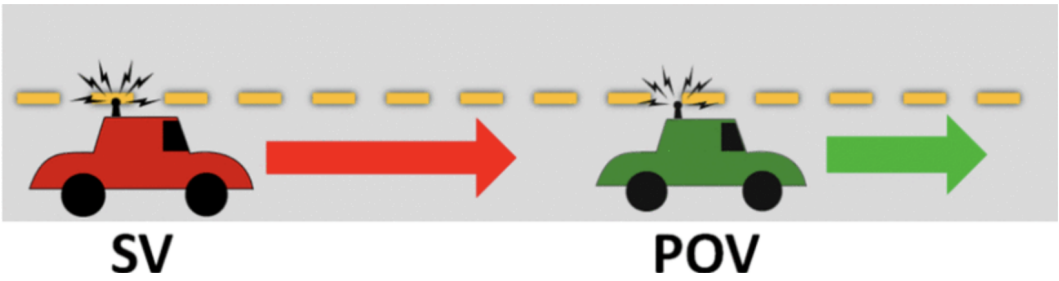


Figure 1: Cruise control image taken from Aréchiga et al. [1]

# 1 Introduction

A lot of work has been done in Cyber Physical Systems to prove the safety of dynamic systems that interact with the world. This is especially true for autonomous vehicles. One such example is an intelligent cruise controlled system. This system involves two vehicles: a lead vehicle denoted as the primary other vehicle (POV) and a follower vehicle denoted as the subject vehicle (SV). The objective of this system is to allow the SV to follow behind the POV efficiently while also ensuring the safety of both vehicles. The POV vehicle is free to accelerate and brake as it wishes. There already exists models and proofs for this system, but there is a significant amount of additional work needed to actually implement this in real hardware.

The motivation of this project is that it is not enough to simply have proven models. There is generally a gap between theory and real systems that must be closed before we can enjoy the benefits of the research already done. Implementing the theoretic models in real hardware has its own challenges. For example, the sensors might not be perfect and might only have access to some of the information required in the model. Besides, there are also challenges on the embedded systems side to manage all the low level hardware needed to operate the vehicle. It's clear that there are many challenges to overcome.

We choose the theme of this project to be the Cruise Control System because of its broad applicability in self driving vehicles. There are various levels of autonomy ranging from level 0 (L0) to level 5 (L5). L0 vehicles have no automation capabilities, whereas L5 vehicles are perfectly autonomous. Everything in between is a stepping stone towards L5. A vehicle capable of some automation in certain circumstances is generally referred to as L2. This will be most similar to the SV in the project. On the other hand, a very capable autonomous vehicle but restricted to certain types of roads and weather is referred to as L4. This is most similar to what some of the most popular self driving companies are trying to build today and to the POV vehicle in our project (Hyat and Paukert [3]).

It is generally a lot more expensive to build a higher level autonomous vehicle due to the additional amount of sensors and other hardware required. Therefore, one common cost cutting strategy is to build one L4 vehicle capable of analyzing all the road conditions and traffic signals to make decisions, while having a much cheaper L2 vehicle simply follow the lead vehicle only making decisions based on what the acceleration of lead vehicle is. This is especially valuable in trucking where a lot of the travel is on long straight highways. There is no need to build expensive L4 trucks if some of the trucks can just make do by following a lead. Beyond shipping, a verified cruise control system is also valuable for commercial use in long road trips.

Our work is done on a small RC robot vehicle as a proof of concept for the verified cruise control system. It is a stepping stone towards the larger sedan and trucking models. The main content of our work is summarized as follows:

- We setup a real RC vehicle, and derived a model for it to safely avoid a static obstacle, while incorporating the asynchronous feature of the sensor updates into the model;
- We proved the derived model to be safe using KeYmaera X;
- We implemented the derived model on the real RC vehicle, adjusting all the gaps between the theoretic model and the real hardware on the car, and recorded a demonstration of the implemented car successfully detecting the obstacle and safely stopping.

# 2 Related Work

## 2.1 Formal Model

$$ICC \equiv (ctrl; dyn)^*$$
$$ctrl \equiv POV_{ctrl} || SV_{ctrl};$$
$$POV_{ctrl} \equiv (a_{pov} := *;\ ?(-B \leq a_{pov} \leq A))$$
$$SV_{ctrl} \equiv (a_{sv} := *;\ ?(-B \leq a_{sv} \leq -b))$$
$$\cup(?\mathbf{Safe}_{\epsilon};\ a_{sv} := *;\ ?(-B \leq a_{sv} \leq A))$$
$$\cup(?v_{sv} = 0);\ a_{sv} := 0)$$

$$\mathbf{Safe}_\epsilon \equiv p_{sv} + \frac{v_{sv}^2}{2b} + (\frac{A}{b} + 1)(\frac{A}{2}\epsilon^2 + \epsilon v_{sv}) < p_{pov} + \frac{v_{pov}^2}{2B}$$
$$dyn \equiv (t := 0;\ t' = 1,$$
$$p'_{sv} = v_{sv}, v'_{sv} = a_{sv}, p'_{pov} = v_{pov}, v'_{pov} = a_{pov}$$
$$\&(v_{sv} \geq 0 \wedge v_{pov} \geq 0 \wedge t \leq \epsilon))$$

Formal dL model from Aréchiga et al. [1].

A great amount of research on intelligent cruise control systems has already been done by Aréchiga et al. [1]. They have a formal model of the scenario in dL - a modeling language for cyber physical systems - and have also proved it to be safe. The exact model is shown above. In this model, we have a lead vehicle denoted as the primary other vehicle (POV) and a follower vehicle denoted as the subject vehicle (SV). The model is simply an infinite loop between a discrete controller and the dynamics of the physical world. The controller allows the POV to brake at a maximum rate of $-B$ or accelerate at a maximum rate of $A$. These constraints on the POV model the real world limits in our motor and braking systems. The controllers allow for the SV vehicle to brake or accelerate within the same limits, but also enforce the condition that the chosen acceleration is safe and the velocity is non-negative.

If we assume that sensor update delay is bounded by $\epsilon$, the safety condition we check for is that if we were to accelerate at a given acceleration, then after $\epsilon$ time we can still apply the maximum brakes such that we do not collide with the POV. It essentially ensures there is enough distance to accelerate for $\epsilon$ time and then brake to a stop.

There is no efficiency condition in the above model, but if there were one, it'd essentially ensure that the distance between the POV and SV vehicle never gets greater than some large max distance $\delta$. Another assumption of this model is that is exists in a 1 dimensional space. This simplification is valid as most applicable conditions of cruise control is on long straight roads.

After the controls have set the accelerations of the POV and SV respectively, the model simulates the dynamics using ordinary differential equations (ODEs) obeying the laws of physics. This process loops indefinitely.

### 2.2 Assumptions

A list of all the assumptions made in the above model is summarized below for convenience:

- The vehicles move on a 1-dimensional straight road.
- The POV vehicle can choose to accelerate or brake arbitrarily.
- Both vehicles can only accelerate within a range $-B < 0 < A$.
- SV has a sensor that can get access of POV's position and velocity, but not acceleration directly. The sensor is not perfect, and has an update delay of $\epsilon$.

Most of the above assumptions are made to model the real world limitations of hardware. However, the last assumption does not hold for our real RC vehicle, as its LIDAR sensor actually only gives us the distance to the obstacle instead of POV's position. Besides, since we could not find another car, we do not need all assumptions for the POV and we focus on a static obstacle model.

## 3 The Remote Control Vehicle

### 3.1 Hardware

The RC vehicle we decided to use for this project is pictured in Figure 1. It was taken from Professor David Held's lab in Carnegie Mellon's Robotics Institute. The following is a basic description for the hardware of the RC vehicle [4]:

> The rear-wheel drive front steering vehicle is capable of driving with speed of up to 6 m/s forward and 3 m/s backwards and achieving steering angles of 30 degrees

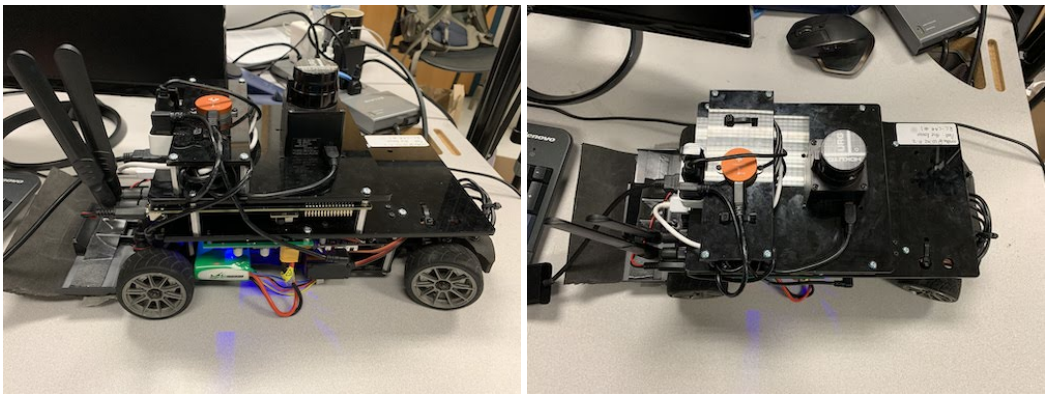

Figure 2: Side and Top Views of the RC Car

on either side. It achieves localization reasonably well using data from the Hall effect sensors on the motor as odometry (ODOM), IMU, LIDAR and the camera on the Jetson developer kit with optic flow. It can be teleoperated or controlled by a software giving it navigation commands.

This RC car will act as the SV. Unfortunately, we were not able to get another vehicle to act as the POV. Additionally, we were not able to use some random other object because it is too hard to enforce that a random object to move at some constant acceleration with limits on its braking and acceleration rates. Thus, we focused on the static obstacle case using a stack of boxes as the POV. This proved to be challenging enough with the time we had.

Some important values that affect our controller are summarized below:

- **LIDAR sensor update frequency:** 10Hz
- **LIDAR sensor scan range:** 5.6 meters
- **ODOM sensor update frequency:** 30Hz

Unfortunately, the vehicle was recently involved in a serious crash, and as a result a lot of the sensors and controls became error-prone. The heaviest hit was on the ODOM sensors. The data sensed by the ODOM sensors were too noisy to be of any use. As a result, we estimated the velocity of the vehicle using other means in our code (more details provided in section 5.2). In addition, the steering linkage of the vehicle was also damaged resulting in the car taking biased left turn even when we commanded it to go straight. We had to correct for this manually as well. See comments in code for details on how this was done.

### 3.2 Software

The vehicle runs Robot Operating System (ROS), which allows us to connect to the Nvidia chip on the vehicle and command it. In order to remotely control the vehicle, both the remote computer and the vehicle need to be connected to the same WIFI network. More instructions on software dependencies and the setup process can be found in the GitHub repository at Khurana [4].

A lot of the software we needed were already setup in the repository. The communication between different sensors and the central ROS system are modeled as a subscriber-publisher model. For example, with a couple of command line commands, we can subscribe to the ODOM sensor topic which gives us the car's localization information such as the relative position of the vehicle to where the system boots up and linear velocity on each axis of the space. In addition, we can also subscribe to the LIDAR sensor topic to get the distance to objects within the LIDAR sensor's effective range. Lastly, there is also topic that we can send commands to control the vehicle's velocity. Unfortunately, currently there is no way to command the acceleration of the vehicle directly. This was a major limitation that we have to deal with (more details in section 5.1), and it again shows the gap between theory and implementation.

# 4 The Proposed Model

## 4.1 Detailed Model in dL

There are a lot differences between the model in the research by Aréchiga et al. [1] and our model. The most important difference is that Aréchiga et al. [1] does not take into account different feedback rates of the car's different sensors, e.g., the LIDAR sensor and the ODOM sensor. The different update rate of the sensors essentially make the whole system running asynchronously, and we have to ensure safety under this practical setting. Another difference is that Aréchiga et al. [1] looks at the system with respect to the absolute positions of the SV and POV. In contrast, our sensors give the distance to an obstacle relative to the car's current position. To fix these differences, we propose the following model.

```
/* Exported from KeYmaera X v4.7.4 */

Theorem "Cruise Control Static"

Functions
  Real A;           /* Robot's max acceleration */
  Real B;           /* Robot's max braking */
  Real ctrlT;       /* Time-trigger limit on evolution */
  Real scanT;       /* Time interval to update obstacle position from sensors */
  Real vPublishT;   /* Time interval to publish robot's velocity from sensors */
  Real distBuffer;  /* Buffer distance between robot and obstacle */

  /* Max range of LIDAR sensor. This is used as the default value of the
     obstacle range when an object isn't detected. */
  Real sensorRange;

End.

ProgramVariables
  Real v;               /* Linear velocity of robot */
  Real a;               /* Linear acceleration of robot */

  Real obstacleDist;    /* Distance to obstacle */
  Real sensedDist;      /* Estimated distannce to obstacle based on sensor */
  Real sensedV;         /* Estimated velocity of robot based on sensor */


  Real t;               /* Time */

  Real worstCaseV;      /* Temporary variable to upper bound current velocity */
  Real worstCaseDist;   /* Temporary variable to lower bound current obstacle distance */

End.

Problem
  (
    A > 0  &
   -B < 0  &
    ctrlT > 0  &
    scanT > 0  &
    vPublishT > 0 &
    distBuffer > 0  &
    sensorRange > 0  &
    v >= 0  &
    sensedV = v & /* Assume initial sensor value is accurate */
    scanT < ctrlT &
```

```
  vPublishT < ctrlT &
  /* The obstacle must be further than the buffer + current stopping distance */
  obstacleDist >= v^2 / (2*B) + distBuffer &
  sensorRange >= v^2 / (2*B) + distBuffer

) /* Initial conditions */
->
[
  {
      {
        {?(obstacleDist <= sensorRange);
          sensedDist := *; ?((sensedDist-obstacleDist) <= (v * scanT + 0.5 * A * scanT^2)
                           & obstacleDist < sensedDist);
        }
        ++
        {?(obstacleDist > sensorRange);
          sensedDist := sensorRange;
        }
      }



    sensedV := *; ?((a > 0 & (v-sensedV) <= A * vPublishT & sensedV < v) |
                    (a < 0 & (sensedV - v) <= B * vPublishT & sensedV > v) |
                    (a = 0 & sensedV = v));

    worstCaseV := (sensedV + A*vPublishT);
    worstCaseDist := sensedDist - (worstCaseV * scanT + 0.5 * A * scanT^2)

    /* Discrete control of robot */
    {
        ?(worstCaseDist > (worstCaseV * ctrlT + 0.5 * A * ctrlT^2) +
                          (worstCaseV + A*ctrlT)^2 / (2*B) +
                           distBuffer);
          a := A;
        ++
        ?(worstCaseDist > (worstCaseV * ctrlT) +
                          (worstCaseV^2 / (2*B)) +
                           distBuffer);
          a := 0;
        ++
          a := -B;
    }

    t := 0;
    /* Distance between robot and obstacle is decreasing at the same rate the
       robot is moving at */
    {
     obstacleDist' = -v,
     v' = a,
     t' = 1 &
     t <= ctrlT & v >= 0  /* Differential Equations */
    }


  }*@invariant(obstacleDist >= (v)^2 / (2*B) + distBuffer &
               sensorRange >= (v)^2 / (2*B) + distBuffer &
               v >= 0) /* Loop Invariant */
](obstacleDist >= (v)^2 / (2*B) + distBuffer & v >= 0) /* Post Condition */
```

`End.`

The model above is written in `dL`, the differential dynamic logic. The benefits of `dL` is that we can prove our programs to be safe using the KeYmaera X theorem prover developed at Carnegie Mellon University. Please refer to Platzer [6] for more information. We now explain each part of the model in detail.

### 4.2 Explanation of the Model

#### 4.2.1 Asynchronous Updates Between the Sensors and the Controller

One key feature, illustrated in Figure 3, of the model that makes it more practical is that it considers the asynchronous updates between different sensors and the control (which is common in real world), and the time delays between these events. Specifically, we will have two sets of values, one set is the sensed values (e.g., sensed obstacle distance `sensedDist` and sensed car velocity `sensedV`), and another set is the real values (e.g. `obstacleDist` and `v`). Our controller is solely based on the sensed values, and the safety conditions are based on the real values. Therefore, one key thing for the model is to use the sensed values to derive correct bounds for the real values, and design a safe controller based on these bounds. We explain this in more detail below.

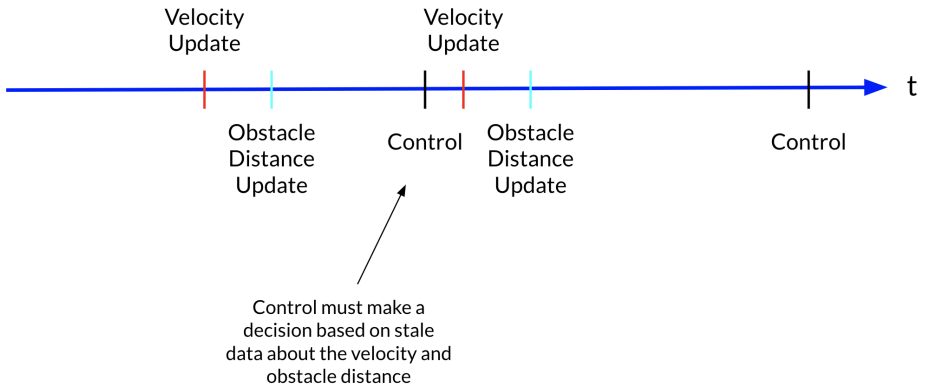


Figure 3: Timeline of events

As shown, the model is based on three critical time intervals: `ctrlT`, `scanT`, and `vPublishT`. Each time interval is used for a different asynchronous event. The `ctrlT` interval denotes the time passed between consecutive control decisions. `scanT` denotes the interval that the car's scanner sensor updates the distance to the obstacle. `vPublishT` denotes the interval that the car's odom sensor updates the car's velocity. It's important to note that the control time interval is the largest of the three, and the controller loop is based on that interval.

The model aligns with the real implementation that the robot stores the sensed distance and velocity in memory, denoted as `sensedDist` and `sensedV`. At the beginning of each control loop, the robot first reads the `sensedDist` variable from the memory and uses it to bound the real distance to the obstacle, denoted as `obstacleDist`. There are two cases. The first case is that there is no obstacle within the LIDAR sensor's effective range (denoted as `sensorRange`). In that case, the LIDAR sensor has stored nothing in `sensedDist`, and we simply assume the worst case that the obstacle is right at the edge of the sensor's range, i.e., we set `sensorDist` exactly to be `sensorRange`.

The second case is that the obstacle is within the sensor's effective range, so a certain value is stored in the variable `sensedDist`. However, since the LIDAR sensor updates `sensedDist` asynchronously with the controller, the value in `sensedDist` is stale or outdated at the time of the control decision. As the time interval for the LIDAR sensor update is `scanT`, we know that `sensedDist` is at most `scanT` out-of-date, and during this time the vehicle is still moving forward. Therefore, the actual distance between the vehicle and the obstacle must be strictly less than `sensedDist`, i.e., we have `obstacleDist` $<$ `sensedDist`. In addition, we can also derive a lower bound for `obstacleDist`

since in scanT time the car could have moved at most (when it's accelerating) $\texttt{v} * \texttt{scanT} + 0.5 * \texttt{A} * \texttt{scanT}^2$, so we have $\texttt{sensedDist} - \texttt{obstacleDist} \leq \texttt{v} * \texttt{scanT} + 0.5 * \texttt{A} * \texttt{scanT}^2$.

Similarly, the value of sensedV stored by the car's odom sensor is also stale at the time of the control decision, but we can use it to bound the car's real velocity v. Recall that the time interval that odom sensor updates sensedV is vPublishT, so sensedV is at most vPublishT time out-of-date. During that time, if the car is accelerating (i.e., a > 0), then its real velocity v could be bounded as $\texttt{sensedV} \leq \texttt{v} \leq \texttt{sensedV} + A \cdot \texttt{vPublishT}$; if the car is braking, then the real velocity is bounded as $\texttt{sensedV} - B \cdot \texttt{vPublishT} \leq \texttt{v} \leq \texttt{sensedV}$; and if the car is either not braking or accelerating, the real velocity should just be the sensed velocity and we have $\texttt{v} = \texttt{sensedV}$.

### 4.2.2 Controller Design

For the controller, we need to make sure that the control decision is safe **for the true obstacle distance and true vehicle velocity based on the sensed values**. The key idea here is just to use the bounds we derived above in the control strategy. In other words, in the controller, we use $\texttt{worseCaseV} = \texttt{sensedV} + \texttt{A} \cdot \texttt{vPublishT}$ as a upper bound for the real velocity v by assuming the car is accelerating with rate $A$ for the vPublishT time since we last got the sensed value sensedV. We then use $\texttt{worseCaseDist} = \texttt{sensedDist} - (\texttt{worstCaseV} \cdot \texttt{scanT} + 0.5 \cdot A \cdot \texttt{scanT}^2)$ as a lower bound for the real obstacle distance obstacleDist, again by assuming the car is accelerating with rate $A$ for the scanT time since we last read the sensed data sensedDist. Figure 3 shows the asynchronous behaviour between the sensor updates and the controller decisions more visually.

The rest of the logic for the controller is pretty simple: for acceleration ($a = A$) or moving at a constant velocity ($a = 0$), we check that the vehicle would still be able to break safely after ctrlT time. See Figure 4 for pictorial explanation. To write this mathematically, we use kinematics to predict the distance traveled by the car in ctrlT time using our worstCaseV, so we can get an upper bound for the real distance the car would have traveled. We then add the distance required to safely stop after we have accelerated with the desired acceleration rate ($A$ or 0) for the control time ctrlT. This calculation is also based on kinematics. Lastly, we add the desired buffer distance between the vehicle and obstacle to the previous two distances to get the total required distance in order to safely accelerate or remain at a constant velocity. If the total required distance as calculated is smaller than the lower bound for the obstacle distance worstCaseDist, then we can safely command the desired acceleration rate.

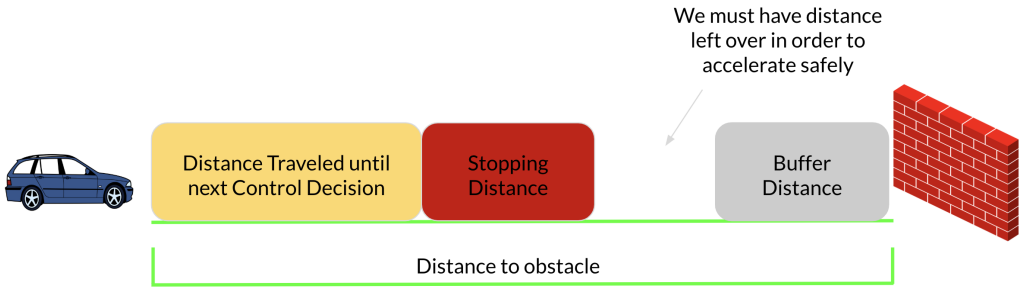


Figure 4: Safety condition required

The control is then followed by the ODE dynamics, which models the physics of the situation. This part is fairly simple. The vehicle is moving forward with velocity $v$. Since the obstacle is not moving, this is the same thing as viewing the distance between the vehicle and obstacle is decreasing with velocity $-v$. In addition, we know $v$ is changing at the commanded acceleration rate $a$. Time is of course increasing at unit rate. We then add the domain constraint $v \geq 0$ to ensure that in real world braking does not make the vehicle travel backwards. Lastly, we add another constraint $t \leq \texttt{ctrlT}$ for our time-triggered controller.

### 4.3 Proof

As mentioned above, the proof of the model was done using KeYmaera X, a theorem proving software (Platzer [6]). Most of the proof was able to auto prove, but we had to split up the cases in order to speed things up. Specifically, we used induction to prove our invariants in the model which implied the postcondition. We proved the invariants for the acceleration $(a = A)$, constant velocity $(a = 0)$, and braking $(a = -B)$ cases separately. The braking case was able to prove automatically. For the other two cases, depending on whether we were in range or not, we had to cut in information. For example, if we were in range, the `sensedDist` was set to some value less than `sensorRange`, since `obstacleDist` $\leq$ `sensorRange`. Thus `sensedDist` was bounded by a function of `obstacleDist`. Now, when we tried to prove the invariant `sensorRange` $\geq \frac{v}{2B} +$ `distBuffer`, KeYmaera X was having trouble since the safety condition we checked was with respect to `sensedDist` which again was bounded by a function of `obstacleDist`. However, since we know the robot is in range, we know that `sensorRange` is just a strictly higher bound. So we cut in the same safety condition but replacing `obstacleDist` with `sensorRange`. This allowed KeYmaeraX to prove the invariant. A similar cut was used when trying to prove the other invariant `obstacleDist` $\geq \frac{v}{2B} +$ `distBuffer` when the robot was not in range. Beyond these cuts, the only other trick we used was hiding irrelavent assumptions to speed up the auto prover. The full proof can be found in our deliverables.

## 5 Implementation on Real RC vehicle

In this section, we first talk about two differences in our real implementation and the proved model, and then present the demo of the implemented system.

### 5.1 Approximation of Acceleration Control by Velocity Control

One of the major issues we hit when implementing our models on the real RC car is that we could not control its acceleration directly. Instead, we could only command the velocity of it. A model based on pure velocity control of the car is impractical and less useful. Thus, we tried our best to command the acceleration of the car based solely on the ability to control its velocity to implement our acceleration based model. We approximate the acceleration control by high-frequency velocity control. Specifically, if the controller decides to accelerate with a value A, then every $\epsilon$ seconds we send a command to the car to update its velocity based on the following equation:

$$v_{command} = v_{old\ commanded} + A \cdot \epsilon \tag{1}$$

This equation essentially approximates the acceleration with a piece-wise constant velocity, as shown in Figure 5.

Theoretically, such an approximation can achieve arbitrary small error compared with the real acceleration as long as the velocity update interval $\epsilon$ is small enough. However, in practice, we found the car's acceleration approximated in such a way seems always being smaller than the real one. This is perhaps due to that we are updating the velocity at a very high frequency, e.g., we set $\epsilon$ to be $0.05$ seconds in our program, thus the car is required to increase the velocity by $A \cdot \epsilon$ in that short period of time, which might be unachievable by the physical hardwares of the car. Nevertheless, the approximation trick makes it possible to implement our acceleration-based model using just velocity control and produce a demo.

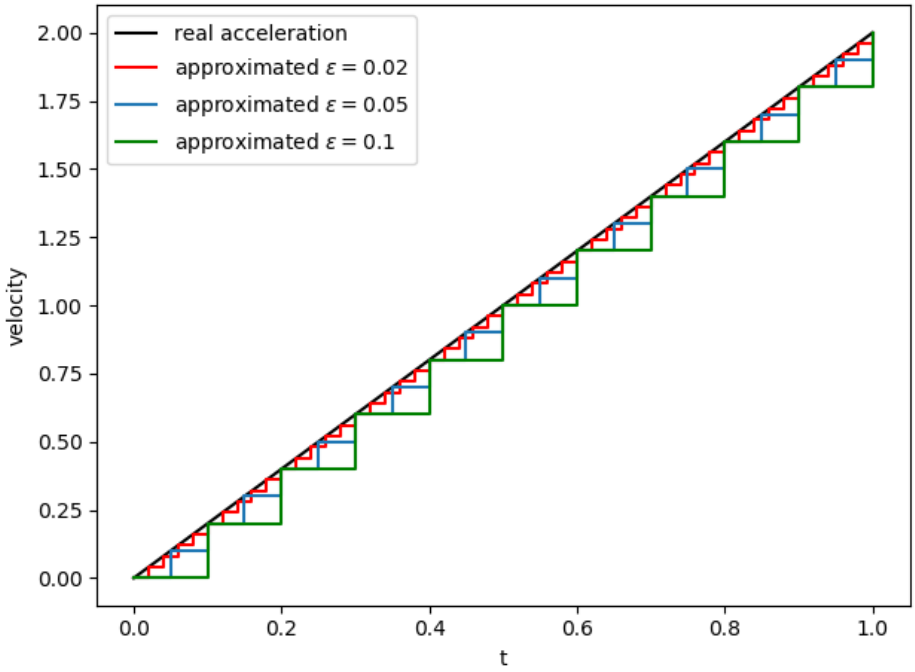


Figure 5: The visual of using a piece-wise constant function (periodic update of the velocity) to approximate the linear acceleration. The approximation is more precise with smaller $\epsilon$.

.

### 5.2 Noisy ODOM Sensors

As mentioned in section 3.1, the RC car was recently in a crash, so its ODOM sensor is partially damaged. As a result, we found the sensed velocity returned by the ODOM sensor to be very noisy and inaccurate. To compensate this, we decided to not use the ODOM sensed velocity at all. Instead, we maintain an analytic velocity of the car based on our acceleration approximation technique (i.e., $v_{command}$ in equation 1). We update this analytic velocity every $\epsilon$ seconds, based on the current acceleration chosen by the controller. The controller's decision is then based on this analytic value of velocity. Since the car cannot really accelerate fast enough to the commanded analytic velocity, the analytic velocity is always a upper bound for the car's real velocity (i.e., we always have $v_{command} > v_{real}$), so the controller should still be safe. If we are breaking, then the reverse is true: $v_{command} < v_{real}$. To account for this, we compute $v_{command} - B * vPublishT$ to get an upper bound for the velocity. Note that the entire run of the robot relies on these theoretical bounds instead of the sensors. Any differences in the true velocity and and theoretical bounds add up over time, which may have led to slightly poor behavior.

### 5.3 Demonstration of the Implemented Model

We implemented the model with the above two adjustments to suit the real hardware we have and recorded a demo of the RC car successfully detecting the obstacle in front of it and safely stopping. The clipped pictures from the demo is shown in Figure 6. The full video can be found in our deliverables. From Figure 6, we can see that while our robot does stop before hitting the obstacle, it doesn't exactly obey the model as it gets closer to the obstacle than desired. This is due to the acceleration approximation technique described earlier, as the car cannot really brake fast enough as we command it to be. (To ensure safety, we compensate this drawback by using an empirically large enough distance buffer). This shows the importance of correct hardware and attention to all details in the implementation when building a provably safe vehicle in the real world.

Figure 6: Clips from demo video (read left to right followed by top to bottom)

## 6 Discussion

This project was a great demonstration highlighting the gap between theory and implementation. Even with a modified model, we had trouble implementing exactly that on real hardware due to limitations on what was available. We hit many roadblocks such as the inability to control acceleration directly, unusable odometry feed, and biased steering controls. Despite these difficulties, we were still able to successfully deploy a model that could prevent collision autonomously.

Nevertheless, its also important to acknowledge some of the objectives we were not able to meet. As seen in the demo, our robot violated the desired model by stopping too close to the obstacle. There should have been a much larger buffer distance between the two objects. In addition, we had set out to solve the cruise control problem with a dynamic obstacle in front of the robot. Due to the limited access to hardware we had, however, we couldn't implement this model. It's also important to note that we ran into unexpected challenges in the static obstacle model itself, so we were also constrained on time had we even had to access another lead vehicle.

The challenges faced in the static obstacle model shined lots of light about the work that still needs to be done to build this research into a viable product. Primarily, safety critical systems need to be made starting at the hardware stage. Even our full sized vehicles on the road today aren't programmed to maintain a constant acceleration. This makes it hard to implement provably safe models. Thus, as this projects demonstrates, certain abilities need to be implemented at the embedded systems layer in order to better align the theoretical model with implementation details. In addition, more accurate odometry and steering can also play a huge role during implementation.

Lastly, we found that the discrepancy between the commanded controls and the true behavior can play a big role in the amount of error we see. There may always be other external factors such as friction or drag constantly leading to different dynamics than those commanded. For example, the controller might command the vehicle to travel at $5$ m/s, but due to the external factors it may only reach a velocity of $4.9$ m/s. We actually saw this in our vehicle because of of some parts on the rear dragging on the floor. In this case, the ODE of the model does not perfectly capture all the physics occurring in the real world. This is even more important for larger vehicles traveling outdoors in windy and/or snowy conditions. Therefore, it is very important to look into *runtime validation*, i.e., to inject monitors, including control monitors, plant monitors, and fallback controls to our execution. One approach to do so may be to leverage the the ModelPlex [5] feature of KeYmaera X to generate a verified sandbox controller with such monitors [2]. This is definitely an area of work that needs to be looked further into.

Nevertheless, the work done so far is promising. It suggests that with enough research, more complex models can also be achievable. There are just a few milestones in hardware and modeling that need to be solved first.

## 7 Deliverables

In summary, the deliverables for this project were:

- A model of the static obstacle system in KeYmaera X along with its proof
- The code implementing the above model
- A video demo of the above code running on a physical RC vehicle

## 8 Work Division

Equal work was performed by both project members.

## References


[1] N. Aréchiga, S. M. Loos, A. Platzer, and B. H. Krogh. Using theorem provers to guarantee closed-loop system properties. *2012 American Control Conference (ACC)*, 2012.

[2] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O Myreen, and André Platzer. Veriphy: Verified controller executables from verified cyber-physical system models. In *ACM SIGPLAN Notices*, volume 53, pages 617–630. ACM, 2018.

[3] Kyle Hyat and Chris Paukert. Self-driving cars: A level-by-level explainer of autonomous vehicles.

[4] Aman Khurana. Assured autonomy car. `https://github.com/amankh/assured_autonomy_car`, 2019.

[5] Stefan Mitsch and André Platzer. Modelplex: Verified runtime validation of verified cyber-physical system models. *Formal Methods in System Design*, 49(1-2):33–74, 2016.

[6] André Platzer. URL `http://www.ls.cs.cmu.edu/KeYmaeraX/`.