15-424/15-624/15-824 Recitation 11
Project Modeling + Proving: Baseball

1 Final Exam

I know I gave you a poll asking whether you wanted to talk about the final, but then I realized next week is Carnival which changes the recitation scheduling equations. So if you want to talk about exam this week please talk to me offline.

2 Project Modeling: Baseball

There is now actually nothing left for you to do in the class this semester except the final projects. Luckily for you, that's still a lot of stuff to do! So in this recitation we will work through part of the process of modeling and proving a project. Obviously we won't go through all the work of doing a project (that should take you more than 80 minutes), but we can reinforce some of the core things you'll have to think about and do during your project:

Key Recitation Concepts:

- After choosing a problem domain, identify specific safety (and liveness!) questions to answer.
- After choosing a problem domain, decompose the question into a series of progressively more complex (starting with very simple) models.
- Decompose problems both in series and in *parallel*: to model a complex system, first model components separately and then compose them together.
- After deciding on the physics for a model, use *logical model-predicitive control*[1]

Overall, you seem to already be doing a great job with identifying questions and breaking it up into a series of models. Perhaps a few of you are so ambitious with your model ideas that you want to make sure you have enough ways to make it simpler and not just ways to make it more complex. We'll do some examples of that in this recitation though.[2]

We have a lot of theory and implementation projects this year (which is exciting!), but sadly it's not to give three recitations at the same time. So if you're doing a theory/implementation

[1] this is, to the best of my knowledge, entirely unrelated to model-predictive control from control theory, except for super high-level philosophy.

[2] Reminder: previous years' projects should also contain examples of this

project and you're finding that this recitation isn't super helpful for you, please come talk to us offline and I should be able to put together some kind of notes that I can share with everyone who's doing theory or implementation. For the rest of you, this recitation specifically aims to address the following which come up in several your projects:

- Combining discrete+continuous control
- Extending physics seen already in the course
- Liveness proofs
- Combining circular motion with non-circular motion
- Making sure Your-Model-Means-What-You-Think-It-Means when things get More Complicated. Specifically, some of you have projects where the system has multiple "phases" do X for a while then start doing Y. We'll look at how to make sure our safety theorem says both safety while X-ing and safety while Y-ing. It's super important and easier to mess up than you think.

Circular motion in particular might be very hard, and we might not get around to it properly during this recitation. This also implies that you should make sure things that involve circular motion are stretch goals in your project: you don't want to try the hardest model right away and then realize you have nothing to show at the end of the semester.

2.1 Picking Problems

Many of you have picked problem domains that are potentially very ambitious (the judges may be surprised that it's even possible to model some of your projects in $\mathsf{d}\mathcal{L}$ at all!). Because the potential scope is quite broad, it's important to take your vaguely defined problem area and narrow in on something that's not only precise, but achievable. Most of you are already doing a great job of this, but it's so important that we'll do it again.

Let's think about the game of baseball (for the uninitiated, a variant of cricket played in the US and Japan), and what properties might be interesting to us and what kind of models might be appropriate for proving them.

Any time you've got an actual game like baseball, you now have multiple players interacting with each other competitively, and you'll probably start thinking to yourself that you could model the scenario as a hybrid game. However, you also need to consider the complexity of the problem at hand: baseball is a game with a bunch of players, each with different roles and rules. $\mathsf{dG}\mathcal{L}$ only talks about two player games, and while it's actually possible to model team games that have exactly two teams, it's not going to be remotely easy. If we're going to look at games at all, it would be much more tractable to look at games that have exactly one kind

of interaction between exactly two players: e.g. a pitcher throwing a ball strategically and a batter hitting the ball strategically, or a batter hitting the ball strategically and then an outfielder catching it. Even then, start simple: we should start with hybrid systems proofs (about interactions between two players at most) and then move on to games if we find out that they're important.

Before we even discuss how to model baseball, we should think about what kinds of properties we want to prove. This will actually inform the design of the model: a model should talk about all the quantities and concepts that we need in order to state and prove the properties that we find interesting, whereas properties that don't show up in the theorems might be skippable. For example, even though a baseball experiences wind resistance, we can state and pretty much anything you want to know about baseball without talking about wind resistance, so we should probably leave that to our most advanced models: we can get really far without it.

Roughly speaking, the properties you'll want to prove for a project fall into three categories: "Basics", "Results", and "Lemmas". When you first create a model, it's really important both to convince yourself that the model actually makes some basic level of sense (validating the model) and to build some experience with it before tackling bigger challenges. These are your basics. For example in Labs 3 and 4 we always proved that we stayed on a circle. This both gave us some confidence that our model wasn't too nuts and made a nice warmup for a much harder proof. But once you have basics, you also want to be able to go to a judge (or a boss, or an audience for a research talk) and say "because I proved X, you should now believe that Y is now safe or otherwise correct". These are your results. This is why people care. Your results will likely be a lot morce difficult to prove that then basics. Along the way, you will probably discover more properties that need to be proved. In the labs, some of these were also the basics. But you might discover other things too. These are your lemmas. They are important because (a) when you do a big proof, you're going to want and need to break it up into more manageable tasks and (b) if you don't get through the proof that you were hoping for, these can often serve as watered-down results (not as cool obviously, but way better than having nothing).

Without further ado, what are some properties for baseball?

Basics: The ball never leaves the stadium. The players never leave the field. The ball is always above (or worst case, on) the ground. The ball and bat never move too fast. The bat never leaves the batting area.

Results: If there is a small child watching the game, our batter does not hit the small child with the ball. Our batter can hit a home run (hit the ball out of the field). Our batter can hit the ball wherever they want as long as the bat is strong enough. A good pitcher can throw a strike (make the batter miss the ball). A good outfielder can catch the ball if it's not thrown too far.

Lemmas: (these are made guesses, the real ones will come up during the proof!): If the bat is X strong, the ball goes Y far. If the outfielder is X fast, they can run Y far. If the batter turns fast enough, they can hit the ball at any angle they want.

Pay good attention to the kinds of results we picked. Remember baseball is a complex game with many people involved. Yet each result only talks about one or maybe two people: this makes it much more likely that we will succeed with our models and our proofs.[3] And at the same time they're somehow relevant to the full game: if I put all these results together I now feel like I know something about how to play (robot?) baseball. And if everything goes really well and I have enough time, maybe I can put it all together to really model and prove a game with lots of people.

2.2 Picking Models

Now that we have a bunch of interesting problems to solve we can think about what's really essential to the models. All the properties I came up with talk about the motion of the ball and the motion + actions of players. So my physics will most likely involve the players (and a bat, for that matter), whereas the controls will be the players' decisions.

Important Simplification: Dimensionality Any time you have physical objects and physical motion, you should ask yourself how many dimensions you need and how complex the motion has to be. In general if getting the motion *just right* is important to safety/liveness, then you should eventually do a fancy model of the motion, but you should still start out simple. Surprisingly often, you can get some decent basic results using a totally 1-dimensional model. This will be a much easier way to start out.

Note that the batter and pitcher don't move, so we don't need to model their motion. This suggests to me that we should start out with modeling the interactions between the batter and the pitcher. There are two actions in the interaction: throwing the ball and hitting the ball. How do you chose where to start?

Important Design Principle: Obviousness If you are picking between modeling+proving situation X vs. Y, and you kinda know how X works but not Y, you should do X first.

I don't know about you, but I don't really know how the pitcher's arm works: the human body is incredibly complicated. But a bat is "just" a stick, and I sorta know how a stick works.[4] So I'd start by modeling the batter hitting the ball. And I'd start out with the most

[3] Unless you constantly focus on how to make your model simpler, most of the time the first model you write down for your project will be far too complicated

[4] Note the bat is being swung by a person, so there's still a lot of complicated human motion. But the important part is that we can simplify the person away and just talk about how the bat moves. Don't be afraid of modeling a complex problem. What you should be afraid of is if your model is as complex as the problem is.

basic batter motion I can think of and then build up to something more complicated. And thus we get our list of models for today:

- 1-dimensional ball-hitting (ignore vertical motion)
- 2-dimensional ball-hitting (model vertical motion)
- 3-dimensional cylindrical ball-hitting (also consider rotation of the bat)

Important Simplification: Dimensions vs. Degrees of Freedom The number of *dimensions* in a system can refer to either (a) the total number of variables or (b) the number of physical dimensions. The thing about dimensions is higher dimensions make problems harder, but often you can't make the model fully realistic without using multiple dimensions. The number of *degrees of freedom* is the number of different parameters your controller can set, which might or might not be the same as the number of dimensions. Both high dimensions and high degrees of freedom make the problem harder, but the number of degrees of freedom is often easier to reduce. For example in 3D rotational motion, a cylindrical motion model is often a good compromise: even though we have 3 dimensions, our only degrees of freedom are 1 dimension of vertical motion + 1 axis of rotational motion.[5] Not only that, but this is far fewer degrees of freedom that the body of the human controlling the bat, e.g. some bipedal robots have 28 degrees of freedom which is a lot more than 2.

2.3 Initial Model: 1D Ball-Hitting

After all that thinking we are now ready to start writing a real $\mathsf{d}\mathcal{L}$ model. First let's design a hybrid program and then we can pick some basic properties. Starting top-down, we can say that a hitting-the-ball scenario has two phases, a "throw" phase and a "hit" phase:

$$\alpha \equiv \alpha_t; \alpha_h$$

Each phase, has a discrete part (throw the ball, hit the ball) and a continuous part (ball moves, bat moves). I'll write **d**elta δ for **d**iscrete programs and ~~e~~**g**amma γ for **c**ontinuous programs.

$$(\alpha_t; \alpha_h) \equiv ((\delta_t; \gamma_t); (\delta_h; \gamma_h))$$

Since everything is one-dimensional, we'll model the ball and bat as points x_o ("o" is a ball) and x_-, ("-" is a bat) respectively. For simplicity's sake (while not being totally boring), we'll say that the ball moves with a fixed velocity (depends how hard the pitcher throws)

[5] The same simplifications can actually be used to reduce dimensionality as well depending on the problem: ask me about ACAS X.

while the bat starts out with acceleration chosen by the batter. The bat's velocity is chosen at the beginning, at the same time as the ball is thrown:

$$\delta_t \equiv v_o := *; ?(v_0 < 0); a_- := *; ?(goodAcc)$$

We're using nondeterministic assignment to keep the model as general as possible. We don't know what the pitcher is going to do, so let's just assume they're throwing the ball toward the batter (toward 0, let's say). Maybe we're trying to prove that the batter can hit the ball good, so we'll need to say something interesting about what the batter does when they hit the ball (i.e. there's interesting control happening to pick the acceleration). But we don't even know what we're proving yet so let's leave the controller totally abstract for now: we'll fill in a controller by defining *goodAcc* some time later.

The physics for the "we just threw the ball" stage will say that the bat moves parabolically while the ball moves in a straight line. But! Our domain constraint says we'll stop when they hit each other ($x_o = x_-$), because we'll want to do our "I hit the ball" logic next:

$$\gamma_t \equiv \{x_o' = v_o, x_-' = v_-, v_-' = a_- \& x_o \geq x_-\}$$

When we hit the ball, we want to update the velocities. Since the bat weighs much more than the ball, it's reasonable to say the ball takes on the velocity of the bat. The bat's velocity is complicated to model if we want to be accurate because the player is swinging it. We can look at then if/when we get around to a circular model. For now we'll say that it's (a) hard to do right and (b) not actually interesting yet so we'll ignore the bat's motion in the second phase:

$$\begin{aligned} \delta_h &\equiv v_o := v_- \\ \gamma_h &\equiv \{x_o' = v_o\} \end{aligned}$$

We now have a complete hybrid program, but what should we prove? Let's start with some *basics*: We're modeling a perfect batter, so the ball should never be behind the bat:

$$\begin{aligned} Pre &\equiv x_- < x_o \wedge v_- = 0 \\ Post &\equiv x_- \leq x_o \\ \phi &\equiv (Pre \rightarrow [\alpha]Post) \end{aligned}$$

Proof Sketch: During the hit phase, v_o is positive because v_- is positive. At the start of the hit phase $x_o = x_-$, so at the end $x_o \geq x_-$ as desired.

There's only one flaw in this argument: we don't know that v_o was positive! That depends on the acceleration, and we totally haven't defined what makes a "goodAcc" good! At this point, when your proof breaks down, you can go back and fill in a new precondition:

Important Technique: Assumption+Control Discovery When you first develop your model, there will be some details that you're not sure about, whether they're tricky preconditions or tricky control ideas. Sometimes the easiest way to fill in these gaps is to just try a proof and see what happens. In this case we filled in:

$$goodAcc \equiv (a_- \geq 0)$$

2.4 Logical Model-Predictive Control

You might dissatisfied with the "prove and update" approach described above, because maybe it seems too ad-hoc. The good news is there's an alternate approach, which we call Logical Model-Predictive Control (LMPC). The idea behind LMPC is: If you want your controller to ensure that the system is safe, just write "the system is safe" as your control decision!:

$$goodAcc \equiv [\gamma_t; \delta_h; \gamma_h]Post$$

Plugging in to δ_t we get:

$$\delta_t \equiv v_o := *; ?(v_0 < 0); a_- := *; ?([\gamma_t; \delta_h; \gamma_h]x_- \leq x_o)$$

If this hasn't already made you feel uncomfortable, now would be a great time to start. Recall the First Law of Tests:

First Law of Tests: Thou shalt never write a non-exhaustive test, for thy theorem will be totally bogus. [6]

This is a serious violation of the First Law. We just wrote "uhh... the system is safe" for our test condition, and since we're in the middle of trying to *prove* the system is safe, we've got absolutely no reason to believe that this test will ever be true! To drive this home, look at the following safety proof for our system! (where $\delta_t' \equiv v_o := *; ?(v_0 < 0); a_- := *$, i.e. everything but the test):

$$\cfrac{\cfrac{\cfrac{\cfrac{*}{\vdash ([\gamma_t; \delta_h; \gamma_h]x_- \leq x_o) \to ([\gamma_t; \delta_h; \gamma_h]x_- \leq x_o)}\,\to R,\text{id}}{\vdash [?(goodAcc)][\gamma_t; \delta_h; \gamma_h]x_- \leq x_o}\,[?]}{Pre \vdash [\delta_t'][?(goodAcc)][\gamma_t; \delta_h; \gamma_h]x_- \leq x_o}\,\text{G}}{Pre \vdash [\alpha]x_- \leq x_o}\,[;]$$

Important Sanity Check: If it seems too good to be true, it is. If you're trying to model and prove something hard, but it proves immediately by `master`, chances are you

[6] Excepting when that test be a guard for a nondeterministic assignment, in which case it shall be satisifiable at least.

made a serious modeling mistake. Try stepping through the proof more slowly to identify any mistakes.

But note the *only* problem here was that we don't know whether the test ?(*goodAcc*) will pass, whereas with the old definition ($goodAcc \equiv a_- > 0$) it was obvious that it would pass. Essentially, if we have a proof that $(a_- > 0) \rightarrow ([\gamma_t; \delta_h; \gamma_h]x_- \leq x_o)$ then we can turn our bad proof of a bad theorem above into a good proof of a good theorem. Except our whole point was that sometimes the test isn't as simple as $a_- > 0$ so we want a way to *derive* $a_- > 0$ from $[\gamma_t; \delta_h; \gamma_h]x_- \leq x_o$

LMPC: The Useful Part Now's the part where LMPC can actually help us figure out the right controller. Since we're not already convinced that $[\gamma_t; \delta_h; \gamma_h]x_- \leq x_o$ will ever be true, we should do what we already do when we're not convinced: *prove it!* Since this isn't true *all the time*, just *some of the time*, our proof attempt should leave us with a simpler, *obviously-sometimes-true* condition we can put in the test:

In the following proof, note the following equivalences (due to the solutions of the ODEs):

$$
\begin{aligned}
x_{-,mid} &= x_{-,init} + a_- \cdot t_1^2/2 \\
x_{o,mid} &= x_{o,init} + v_{o,init} \cdot t_1 \\
x_{-,mid} &\leq x_{o,mid} \\
v_{-,mid} &= a_- \cdot t_1 \\
v_{o,mid} &= v_{-,mid}
\end{aligned}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{t_1, t_2 \geq 0 \vdash 0 \leq t_1 \cdot t_2 \cdot a_-}\,\mathbb{R},\text{CE}}{t_1, t_2 \geq 0, x_{-,mid} \leq x_{o,mid} \vdash x_{-,mid} \leq x_{o,mid} + t_1 \cdot t_2 \cdot a_-}\,\mathbb{R},\text{CE}}{t_1, t_2 \geq 0, x_{-,mid} \leq x_{o,mid}, v_{-,mid} = t_1 \cdot a_- \vdash x_{-,mid} \leq x_{o,mid} + t_2 \cdot v_{-,mid}}\,\mathbb{R},\text{CE}}{\vdash \forall t_1 \geq 0. x_{-,mid} \leq x_{o,mid} \wedge v_{-,mid} = t_1 \cdot a_- \rightarrow \forall t_2 \geq 0. x_{-,mid} \leq x_{o,mid} + t_2 \cdot v_{-,mid}}\,\rightarrow R, \wedge L, \forall R}{\vdash [\gamma_t]\forall t_2 \geq 0. x_{-,mid} \leq x_{o,mid} + t_2 \cdot v_{-,mid}}\,[\text{`}]}{\vdash [\gamma_t][\delta_h]\forall t_2 \geq 0. x_{-,mid} \leq x_{o,mid} + t_2 \cdot v_{o,mid}}\,[:=]}{\vdash [\gamma_t][\delta_h][\gamma_h]x_- \leq x_o}\,[\text{`}]}{\vdash [\gamma_t; \delta_h; \gamma_h]x_- \leq x_o}\,[;]
$$

2.5 Sanity Check 2: Compare (Desired) Model vs. Proof

Doing a proof for a model in detail (as opposed to letting `master` do the proof) is a good way to assess whether our model makes sense. Above when introducing LMPC, doing a proof allowed us to see that the controller was truly bogus because the proof ignored the

entire model. Note that even when the proof is interesting/isn't trivialized by the model, inspecting the proof more closely can help reveal mistakes in the model.

Specifically, notice anything strange in the proof above? A student noticed something strange during recitation: At no point in the proof do we actually learn that $x_- = x_o$ when the bat hits the ball. That is, we never learn that the bat hits the ball. Whenever we see this happen in a proof, there's a distinct possibility that the fact of interest (we hit the ball) is simply not true, though there's also a compelete possibility that the fact is true and we just couldn't see it from the facts we had in front of us.

In this case, the fact is false: there's a bug in the model! This model allows for the possibility that γ_t stops before the ball and bat hit each other, at which time the ball will reverse direction and magically fly away from the bat! This is totally unrealistic, but was not caught by our proofs so far! Sometimes proving the basics will not be enough to catch every modeling mistake: a mixture of proving things, thinking hard about the models and looking at the proofs closely is the best way to catch your mistakes. In this case, there's an "easy" fix[7], which is to add a test saying we made sure the ODE ran as long as possible:

$$\delta_h \equiv ?(x_o = x_-); v_o := v_-$$

2.6 Sanity Check 3 and Multi-Phase Modeling

The test above is problematic for another reason: it deletes a lot of interesting cases. Recall from all the way back in Recitation 3 (and from grading feedback throughout the semester) that we need to be extremely careful whenever we add a test to a model. KeYmaera X can only tell you whether your model was true, not whether it was the model you wanted, so we need to have a very clear understanding of what exactly we proved ourselves. Generally speaking, for a safety property, your safety property should say "no matter what everybody else does, if I make the right control decisions, I'm safe at all times t". The "for all t" part is important: If I'm unsafe at time 0 and safe at time 1, that doesn't help because I'm already dead at time 0.

This problem comes up in a big way when we add the above test because we're now ignoring the entire phase where the ball was moving toward us: we assume the ODE always runs to completion. And previously the only reason that our safety theorem implied "safety at all t" was *because of* that nondeterministic ODE duration.

Though now that you mention it, we have an even bigger problem here. No matter how long we run γ_h, we are always always running δ_h as well. What this means is that the end states

[7]This is not entirely satisfactory for the same reason that testing the safety theorem is not satisfactory: you don't know whether the formula is satisfiable. Can you think of a more satisfying way to resolve the issue?

of γ_t are never the end states of α, which also means we never get to prove that the system is safe "during γ_h. We only prove that it's safe "during γ_t."

To summarize, if you have a model with multiple phases (such as a "throw" phase followed by a "hit" phase), your safety theorem will not mean "safe at all t" unless you're careful.

The Easy Way to be Careful The good news is there are several solutions. The absolute lowest-tech simplest solution is to simply write two models and do two proofs, where one proof means "I'm safe during phase 1" and the second means "safe during phase 2", e.g. $Pre \rightarrow [\alpha_t]Safe$ and $Pre \rightarrow [\alpha]Safe$. The downside here is that you might have to do the same proof multiple times for multiple models.

The Medium-Complexity Way to be Careful Another option is to have only one model but add the option to skip the second phase, like

$$\alpha \equiv \alpha_t; (\alpha_h \cup ?(true)))$$

This is arguably a bit more complicated because you had to change the model, and now it's not as obvious what the model's saying. But it's nice because you only have one theorem to prove and you might be able to reuse more proof effort. It can also get a little gross syntactically if you have more than 2 phases, e.g.

$$\alpha \equiv \alpha_1; (?true \cup (\alpha_2; (?true \cup (\alpha_3; (?true \cup \alpha_4)))))$$

The High-Complexity Way If your formula means the wrong thing and you don't want to change the formula, you could... change the meaning! There is actually a variant of dℒ called dTL (differential temporal logic) where $[\alpha]\phi$ really *does* mean "true at all times", with a totally different semantics and totally different proof rules to match. Given sufficient student interest we could cover this logic in lecture, but sadly it's not implemented in KeYmaera X, so you won't be able to use it on your projects.