Trajectory generation for robotics via optimization

Advanced Algorithms Spring 2023

Luke Raus

Overview

The field of optimal control is a rich and deep subject concerned with controlling dynamic systems, from chemical plants to humanoid robots. Naturally, seeking control schemes which are (in some sense) optimal leads to formulations of problems as maximizations (or minimizations) of some objective function under some sets of constraints. Thus, optimal control builds heavily on the theory and algorithms of mathematical and computational optimization invigorated by Dantzig's Linear Programming, as we have discussed in Advanced Algorithms.

However, LP alone doesn't begin to suffice for solving worthwhile optimal control problems. At one basic level, the dynamics of worthwhile systems almost always contain vital nonlinearities in their governing equations, such as trigonometric or quadratic relationships, thus preventing the system from being formulated as a linear system. Just as crucially, the dynamics of physical systems evolve in continuous time, whereas LP is developed around discrete sets of decision variables. The nonlinearities at play additionally tend to result in nonconvex programs, thus losing the convexity on which LP relies to guarantee optimality.

Fortunately, myriad techniques have been developed which extend the core ideas and techniques of LP to handle more general nonlinear programs (abbreivated to NLP; sorry, natural language processing), albeit generally without strict or provable guarantees of achieving global optimality. With these tools, the task of transforming an optimization problem over continuous-time functions representing complicated physical systems like robots and its interactions with its environment into a tractable nonlinear program with some finite set of decision variables - a process known as *transcription* - remains a difficult task and an immensely active field of research.

In this project, I seek to understand transcription methods for formulating trajectory generation problems in robotics, as well as the broader context of this topic in the broader field of optimal control. I then formulate a relatively simple but quite rich trajectory generation problem: namely, inverted pendulum swing-up - as a tractable nonlinear program and implement the optimization in MATLAB. As we will see, the generated control trajectories successfully execute the swing-up in an external simulation given reasonable parameters!

Global vs. local control

Broadly, approaches to optimal control can be divided into 2 main categories: global and local.

Global approaches tend to rely on an equilibrium equality equation that a control policy must satisfy to be optimal. They thus follow in the tradition of Dynamic Programming as chiefly formulated by Bellman. In the context of continuous-time dynamics and controls, this equality is known as the Hamilton-Jacobi-Bellman (HJB) equation, which is a partial differential equation (PDE).

Managing to compute a policy which satisfies the equilibrium condition is especially powerful because this policy is a closed-loop function of the system's state; that is, no matter what state the system is currently in, we will know exactly what control action to take to optimally move towards the goal. Unfortunately, this strong global result is expensive to compute, and all current methods for exactly solving an HJB take exponential time [Wensing et al]. This relegates their usefulness to relatively low-dimensional problems, although researchers (like my advisor for this upcoming summer) are working on methods to solve this PDE more efficiently via e.g. neural networks, to allow for better scaling to higher-dimensional problems [Bansal].

Local methods forego a complete global control policy for a specific open-loop control plan (e.g. as a function of time) to make the system evolve from one particular state to another. To arrive at such a trajectory, local methods perform a direct optimization as discussed previously by starting with some initial candidate trajectory and then iteratively evolving it to reduce its cost while achieving feasibility, hopefully arriving at a global minimum [Tedrake]. This tends to be a much more tractable computational task, at the downside of being a less general method resulting in open-loop control schemes which are not provably optimal.

We focus on local methods due to their relative popularity in the robotics community and their much tighter connection to what we have covered in Advanced Algorithms.

Formalizing optimal control for direct optimization

In an optimal control problem, we always have 3 main ingredients: system state, dynamics, and control. The state $\mathbf{x}$ captures all relevant information about our system; the control u describes how we influence the system; and the dynamics capture how these are coupled: generally, the derivative of the state is a certain function g (derived for that particular system) of the current state and the current control: $\dot{\mathbf{x}}(t) = g(\mathbf{x}(t), u(t))$.

Thus, for a trajectory generation problem, we will want to find functions for control and state with respect to time which minimize some cost function while obeying the system's dynamics as well as constraints relating to the initial and final conditions of state and any other physical constraints we might have (e.g. actuators which cannot supply infinite force/torque).

Our concrete cart-pole example will help clarify!

The cart-pole swing-up problem

My example scenario for implementing NLP-based trajectory optimization is the cart-pole (or inverted pendulum) swing-up problem. The cart-pole is a canonical dynamic system [Tedrake], [Kelly] consisting of a freely-swinging pendulum or "pole" with a pivot on a horizontally-translating "cart" on which we can apply a horizontal force, as with a motor. This idealized system model excludes nonconservative forces like friction or damping.

In introductory controls classes (such as ESA at Olin), a common task is to control the cart so as to balance the pendulum upright, starting from an initial condition near upright. The swing-up problem is more difficult: with the pendulum initially hanging downwards at rest, we must apply a sequence of forces to the cart such that the pendulum swings up to the upright position and stops there (at which point we could activate a simpler balancing controller).

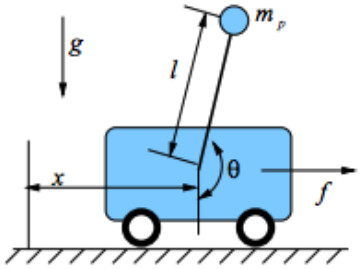


Figure 1: The cartpole system. Applied forces f influence the system's horizontal position x and pendulum angle θ according to its dynamics $\dot{\mathbf{x}} = g(\mathbf{x}, u)$. Image from [Tedrake].

This system has second-order dynamics as usual, so its state $\mathbf{x}$ is a column vector containing not only the horizontal position x and the pendulum angle θ, but also their time derivatives:

$$\text{State: } \mathbf{x} = \begin{bmatrix} x, & \theta, & \dot{x}, & \dot{\theta} \end{bmatrix}^T$$

Since we can only control the horizontal force f applied to the cart, our control is simply:

$$\text{Control: } u(t) = f(t)$$

This system is "underactuated" [Tedrake] since it has 2 configurational degrees of freedom (x and θ) but have only 1 control input: the force on the cart, u. We can imagine the system being "fully actuated" if we could directly apply a torque to the pendulum.

As mentioned, interesting systems like this tend to have quite nonlinear dynamics. Indeed, the linear acceleration of the cart and angular acceleration of the pole are highly nonlinear functions of the state $\mathbf{x}$ and control u as shown here, as derived by [Tedrake] and [Kelly]:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x} \\ \dot{\theta} \\ \ddot{x} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{\theta} \\ \Big(u + m_p \sin\theta (l\dot{\theta}^2 + g\cos\theta)\Big) \,/\, \big(m_c + m_p \sin^2\theta\big) \\ \Big(-u\cos\theta - m_p l\dot{\theta}^2 \cos\theta \sin\theta - (m_c + m_p) g \sin\theta\Big) \,/\, \big(l(m_c + m_p \sin^2\theta)\big) \end{bmatrix}$$

Cost function

Our cost function allows us to shape the definition of *optimality* for the particular problem at hand. In this case, we would like the system to perform the swing-up in as efficient a manner as possible; that is, without exerting unnecessary control effort. As in [Kelly], we will thus attempt to find the function of state and control which minimize the total squared force exerted on the cart (where squaring ensures any effort positively contributes to the total which we minimize) for our time window $t \in [0, T]$:

$$\text{minimize}_{\mathbf{x}(t), u(t)} \int_0^T u(\tau)^2 d\tau$$

Constraints

Our most important constraint is that the chosen sequence of state and control must respect the system's dynamics:

$$\dot{\mathbf{x}}(t) = g(\mathbf{x}(t), u(t)) \quad \forall t$$

But almost as importantly, in order for our trajectory generation to actually accomplish our goals, we need to specify our initial and final states. We want the system to both start and end at rest, so the derivatives $\dot{x} = \dot{\theta} = 0$ in both cases. Then, we can initialize the cart at $x(0) = 0$ with the pendulum hanging down at $\theta(0) = 0$ and then end with the cart at some specified horizontal position $x(T) = d$ with the pendulum upright at $\theta(T) = \pi$.

$$\mathbf{x}(0) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{x}(T) = \begin{bmatrix} d \\ \pi \\ 0 \\ 0 \end{bmatrix}$$

Additionally, we can impose realistic physical constraints such as limiting the cart's horizontal motion to some finite track and limiting the magnitude of the applied force to represent our hypothetical motor's physical limitations (also known as actuator saturation) [Tedrake]:

$$x_{min} \leq x(t) \leq x_{max} \quad \forall t$$
$$u_{min} \leq u(t) \leq u_{max} \quad \forall t$$

This set of constraints is relatively tame for the cartpole swing-up problem, since it has effectively no interaction with the external environment. More complicated physical systems such as walking robots necessitate incorporation of deeply sophisticated constraints. For example, a walking robot's feet must not penetrate the ground, its contact forces with the ground must fit within the "cone of friction" which dictates maximum static friction, and it also has sets of joint-angle states which are infeasible since they would make the various segments of its legs self-intersect. Encoding these types of constraints on complex dynamical systems - as well as encoding the actual dynamics of e.g. feet colliding with the ground - in a computationally-tractable manner is essentially an entire field of research unto itself! See [Wensing et al.]'s review for more.

Discretization via collocation

Now that we have formulated the problem, we run into perhaps *the* problem which makes optimal control most difficult: we are attempting to yield continuous-time, a.k.a. infinite-dimensional, functions! To use an NLP solver, we need to find a way to represent our output with some finite set of decision variables; that is, we need to discretize the problem.

This discretization problem is very much nontrivial, as it is difficult to get any sort of guarantees for whether our resulting approximations of e.g. $\mathbf{x}(t)$ and $u(t)$ will adequately represent the underlying functions in a way that transfers to, say, a real-life robot (where time and dynamics are indeed continuous). A host of approaches exist for performing this discretization [Wensing et al], which we can broadly categorize into *collocation* and *shooting* methods.

Collocation broadly refers to representing the relevant functions using polynomial splines which can be described by a finite set of real-number decision variables. At the most basic level, this entails simply sampling N evenly-spaced points in time in our range $[0, T]$ and treating the values of state and control at each of those time points as decision variables; we then linearly interpolate between these sampled points to produce continuous functions as output [Yadav]. More advanced collocation methods get more creative about the polynomial spline representation, but this direct sampling method is simple to implement and offers a very intuitive method for tuning the resolution of our outputs [Kelly]. Increasing N will give us a finer-resolution sampling across time and thus hopefully a more accurate and dynamically-feasible solution at the cost of a higher-dimensional, and thus more computationally difficult, optimization.

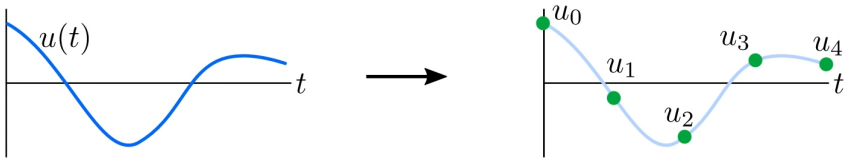


Figure 2: Collocation via direct sampling. Image from [Kelly].

Shooting methods offer an alternative approach, instead forward-simulating the dynamics over certain segments of time [Wensing et al]. This comes from the idea that we can theoretically determine the evolution of the system state (and thus find the constrained final state) using *only* the initial state and the control input by forward-simulating the system, thus removing the state from our set of decision variables [Tedrake]. While this does significantly reduce the dimensionality of the decision set, it does make our nonlinear constraints significantly more cumbersome to directly represent.

Under our chosen direct collocation approach, our continuous functions for state and control thus become discrete sequences $\mathbf{x}_k \in [\mathbf{x}_0, \ldots, \mathbf{x}_{N-1}]$ and $u_k \in [u_0, \ldots, u_{N-1}]$. Since our state has 4 elements, we get 5 sequences of N points for a total of $5N$ decision variables.

Since this sequence contains a direct representation of the initial and final state which we seek

to constrain, we can immediately apply our constraints on these decision variables:

$$\mathbf{x}_0 = \mathbf{x}(t=0), \quad \mathbf{x}_{N-1} = \mathbf{x}(t=T)$$

Likewise, our constraints on the cart's position and the applied force carry over just as readily, now being applied to each of the finite decision variables in the relevant sequences:

$$x_{min} \leq x_k \leq x_{max} \quad \forall k$$
$$u_{min} \leq u_k \leq u_{max} \quad \forall k$$

Translating the continuous-time integral defining our cost function and the derivative defining the dynamics constraint to this discrete domain is not as trivial but still very doable. Here, the mathematical subdiscipline of numerical methods provides guidance.

A simple Reimann sum lets us approximate our integral expression for the total squared control effort. Since we linearly interpolate the control effort between sampled points, a trapezoidal sum perfectly describes its signed area [Kelly]. Thus, instead of simply summing $(u_k^2 \Delta t)$ for all k, we average the squares of neighboring sample points and multiply that value by the timestep. This lets us express our objective statement as:

$$\text{minimize}_{[\mathbf{x}_k],[u_k]} \sum_{k=0}^{N-2} \left(\frac{u_k^2 + u_{k+1}^2}{2} \cdot \Delta t \right)$$

For the dynamics, which relate the derivative of the system state to the current state and control, we can similarly approximate this relationship with finite differences between sampled state values. At the heart of this method is the definition of the derivative, which is equivalent to Euler's method: $\mathbf{x}_(t + \Delta t) \approx \mathbf{x}(t) + \dot{\mathbf{x}}(t)\Delta t$. Since we have linear interpolation between sampled points, it is most accurate to approximate the derivative at that point by averaging the 2 slopes immediately before and after that point [Kelly]:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \left(\frac{\dot{\mathbf{x}}_k + \dot{\mathbf{x}}_{k+1}}{2} \right) \Delta t \qquad \forall k$$

And that's all! With our finite set of decision variables and our objective function and all constraints formulated in these finite terms, we are prepared to implement the NLP.

Implementation in MATLAB

Managing to actually solve an NLP often has as much to do with one's choice of solver as one's formulation of the problem. Since the nonlinearity and nonconvexity of NLPs makes it essentially impossible to offer strict guarantees of optimality, NLP solvers rely on various **heuristics** and specialized methods for seeking optima. These "tricks of the trade" are often wrapped in closed-source, proprietary, but very high-performance libraries such as GPOPS2 in C++. The open source community seems to have a bit of catching up to do!

I chose to use MATLAB's **fmincon()** method, bundled in the Optimization toolbox, because it seems like a well-regarded solver with good documentation and a low barrier to entry. MATLAB is also well-suited to expressing functions on numerical vectors, such as with its many built-in numerical methods like trapezoidal integration.

Once I wrapped my head around it, expressing the problem in MATLAB for **fmincon** was relatively straightforward (although the interim certainly featured plenty of refactoring and debugging). The cost function was trivial to compute, and both the initial/final state constraints and the max position/force constraints could be encoded in the "lower/upper bound" vectors which map directly to bounds on the decision variables. This meant we did not use the matrix/vector combination inputs with which linear equalities and inequalities could be specified as matrix-vector products.

The dynamics constraints were imposed by the specific nonlinear constraint input, which lets us specify a vector as any function of the decision variables which the solver will then try to constrain to zero. Simply subtracting the righthand side of the dynamics constraint from both sides gave us expressions equal to zero, one per state variable per sample for a total of $4N$ nonlinear constraints.

Specifying initial guess

A final important implementation detail is our initial guess at the solution, which is especially important in the realm of NLP due to the prevalence of local optima induced by convexity. Fortunately, the cartpole swing-up is a relatively kind problem in this regard, and [Kelly] suggests that even very simple initializations should suffice. We initialize with no control effort and a simple linear interpolation between the initial and final states.

Results

We successfully arrived at (locally) optimal trajectories!

The following result shows the sequence of applied force and resulting linear and angular positions of the cartpole for system configuration **A**: $N = 15$ samples, $T = 2$ s, $d = x_{final} = 1$ m, $x_{max} = \pm 2$ m, $u_{max} = \pm 20$ N, $m_{cart} = 1$ kg, $m_{pole} = 0.3$ kg, and $L_{pole} = 0.5$ m.

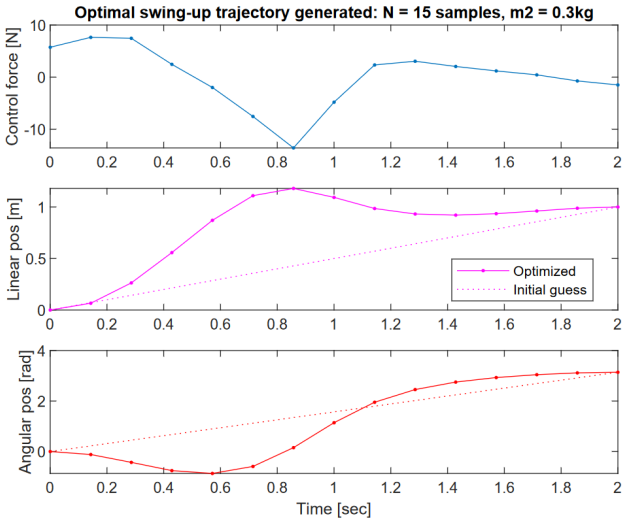


Figure 3: Trajectory generation results from configuration **A**.

This relatively low-resolution trajectory still shows the optimization's emergent approach to the swing-up problem: the cart first swings the pendulum in the negative angular direction to build momentum before rapidly accelerating in the other direction to swing the pendulum up and quickly reversing to "catch" it from below before balancing. Beautiful!

Keeping the system configuration identical but increasing the number of samples to $N = 50$ gives a higher-resolution view of what ends up being the same solution:

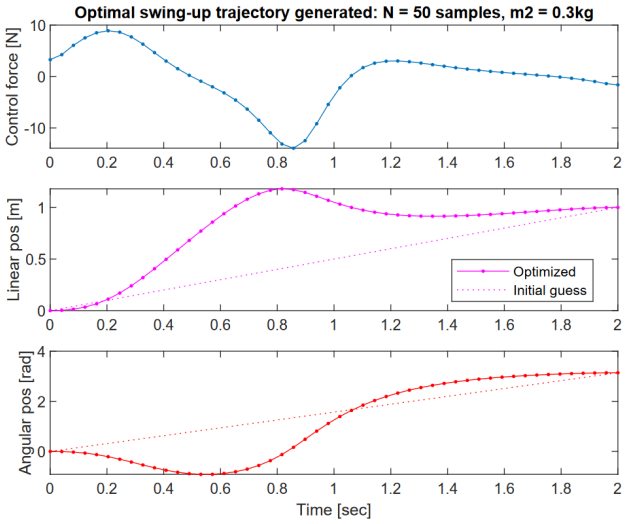


Figure 4: Configuration **B**: Same as A but more samples.

Changing the configuration to dramatically increase the pendulum weight to $m_{pole} = 2$ kg from 0.3 kg makes this a significantly harder problem from a physical perspective. As we see, the general gist of the solution stays the same but the forces are larger and more aggressive. The optimization confronts but successfully respects the control force limits of 20 N. The pendulum's initial backwards swing nearly takes it to horizontal!

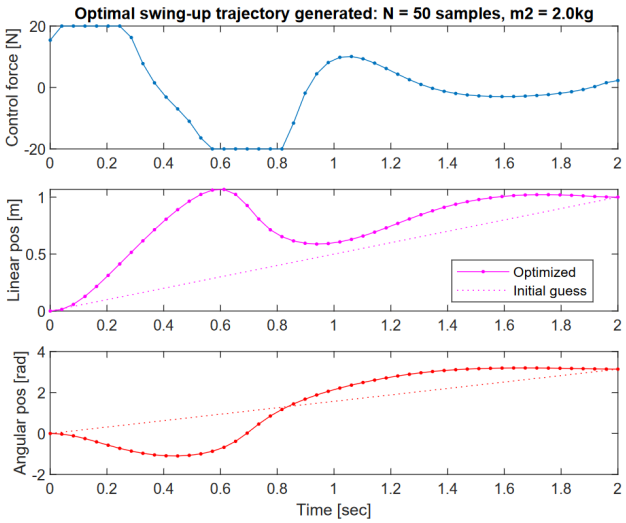


Figure 5: Configuration **C**: Heavier pendulum demonstrates force bounds!

Validation with simulation

An important consideration in trajectory optimization is determining whether the open-loop control plans achieved will actually work on the system. While the dynamics - as approximated in the constraints - are satisfied from the solver's perspective, whether these approximate dynamics will hold up on the actual system must be tested.

To do this, we carry out these open-loop control schemes in a simulation environment which independently calculates the dynamics of the system as a function of state and control, where the control is linearly interpolated from our decision vector as a function of time. We then compare these simulated state trajectories with those computed by the optimization.

Unsurprisingly, the low-resolution sampling in configuration **A** resulted in noticeable discrepancies with the simulation as seen in Figure 6. The control sequence and approximate dynamics were simply too coarse to accurately represent the system's evolution, so the dynamics diverge after about 1.4 seconds.

However, the higher-resolution solution to the same physical configuration in **B** does much better. Here, the pendulum in simulation has reached 3 radians: a mere 8 degrees from the desired angle of π radians. This is absolutely in the realm where a closed-loop linear controller could activate and stabilize the system in the long term; this is precisely the goal of solving the swing-up problem. Thus, we can declare **success**, noting that the larger number of samples indeed helps the approximated dynamics match those of the actual system.

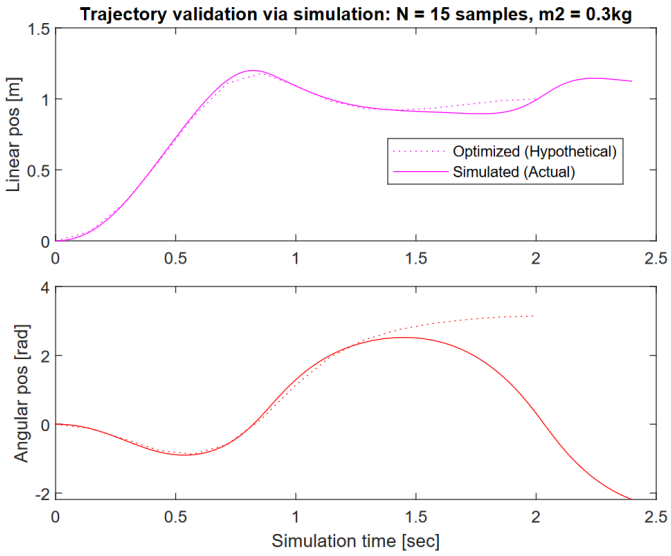


Figure 6: Configuration **A**'s low resolution harms accuracy, causing the optimized and simulated dynamics to diverge.

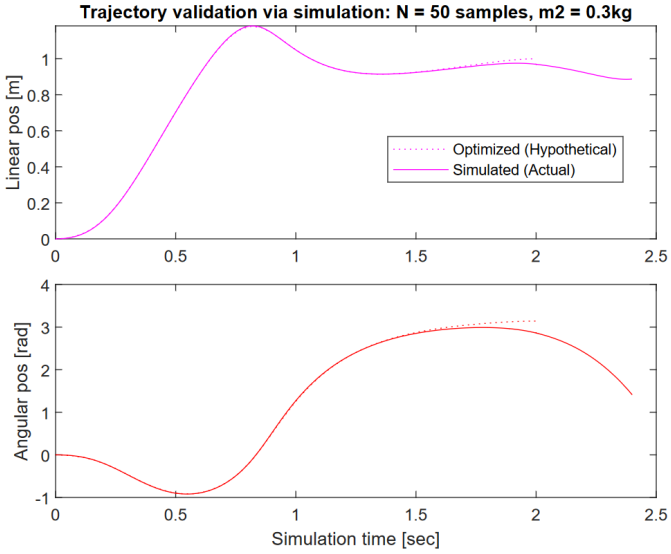


Figure 7: Configuration **B**'s higher resolution helps the control sequence get the pole very close to upright. Success!

For the much more physically difficult problem posed by configuration **C**, we again see the dynamics diverge in Figure 8, despite this solution having the same number of samples as B. We can attribute this to the difficulty of the problem and the aggressiveness of the physical problem, which necessitates very intense accelerations and changes thereof. The approximate nature of the optimized dynamics find it difficult to adequately represent and constrain this motion. However, they perform admirably for the first 1.2 seconds or so.

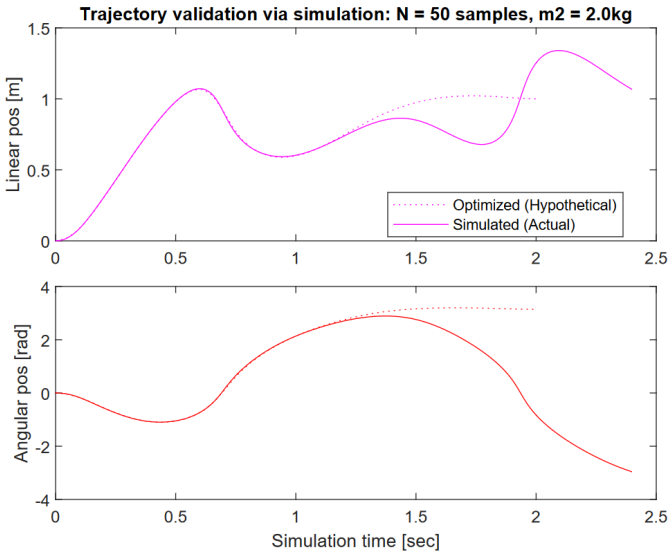


Figure 8: The harsher control and more aggressive dynamics required in configuration **C** make the approximate dynamic constraints insufficient.

Discussion & takeaways

Despite our relatively naive initial guesses, our solver was nonetheless able to find (approximately) dynamically-feasible trajectories satisfying our constraints. The more accurate these dynamics were - whether by simply having more samples or not having harsh conditions which are difficult to capture - the better they help up to simulation.

However, it is worth mentioning that I had better success with my trajectory optimization by enabling a "Feasibility Mode" of fmincon, which uses some interior point method to prefer maintaining feasibility while seeking optimality. In reflection, it is interesting to note that in the set of all possible candidate solutions, the subset which is actually dynamically feasible is quite small. In this lens, it broadly seems to ring true that in optimal control, respecting the dynamic constraints of the problem matters far more than finding something perfectly optimal, as an "optimal" solution which doesn't actually work is pretty useless.

Takeaway: Dynamic feasibility trumps optimality for control!!

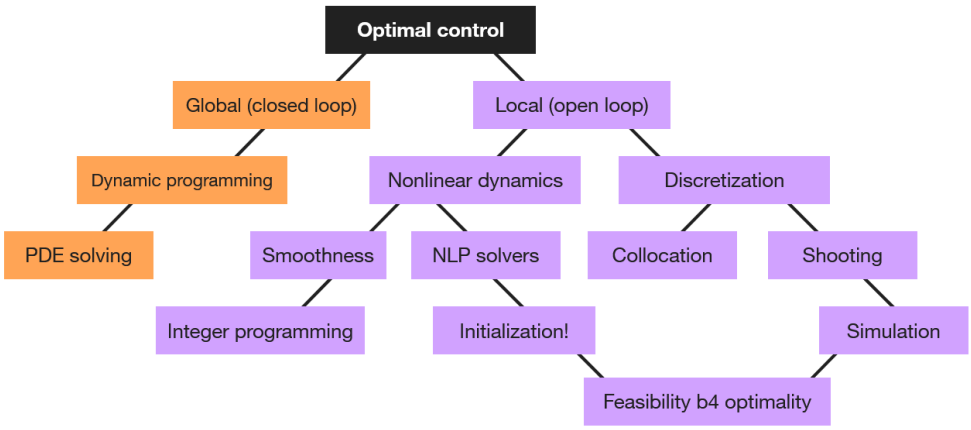
Harder problems also sometimes struggle to find feasible solutions in the first place. Here, initializing the solution guess is key to eventually reaching both feasibility and optimality. One interesting approach with our direct collocation method is starting with a lower-resolution version of the same problem and using its solution to seed a higher-resolution version, thus letting the first problem quickly "get the gist" of the solution before refining it. Alternatively, highly constrained problems might relax constraints to find an initialization and gradually add them back in. For now, such approaches remain largely more of an application-specific art than a science [Kelly].

Finally, we have thusfar only discussed improving the usefulness of our open-loop control plan

by improving accuracy such as increasing resolution. However, an interesting and very popular modern approach is to explicitly acknowledge the weaknesses of such open-loop plans. Thus, one commits only to executing the beginning of the plan, hopefully before the dynamics diverge, and then recomputes an optimal trajectory from the future state in quasi-closed-loop fashion [Tedrake]. This approach, called Model Predictive Control (MPC), has only been enabled by recent advances in fast and reliable NLP optimization. Having a good grasp on trajectory optimization, I'm excited to dig into MPC in the near future and uncover some of the difficulties that come with trying to solve optimization problems online in real-time.

Concept tree

Having journeyed through both the theory and implementation of trajectory generation using optimization, we can appreciate the following concept tree highlighting the various branches of the field of optimal control as we've discussed.



References

- Bansal, Somil et al. "Hamilton-Jacobi Reachability: A Brief Overview and Recent Advances," 2017. https://arxiv.org/abs/1709.07523
- Hargraves, C.R. and S.W. Paris. "Direct trajectory optimization using nonlinear programming and collocation," 1986. https://arc.aiaa.org/doi/abs/10.2514/3.20223? journalCode=jgcd
- Kelly, Matthew. "Introduction to Trajectory Optimization," 2016. https://www.youtube.com/watch?v=wlkRYMVUZTs
- Tedrake, Russ. "Underactuated Robotics course notes," 2023. http://underactuated.mit.edu/index.html
- Wensing, Patrick et al. "Optimization-Based Control for Dynamic Legged Robots," 2022. https://arxiv.org/abs/2211.11644
- Yadav, Vivek. "Direct collocation for optimal control," 2016. https://mec560sbu.github.io/2016/09/30/direct_collocation/