Intro to Optimal Control without Proofs

Anshuman Medhi

May 14, 2021

Contents

1 Introduction

Optimal control, as it is usually taught, is a post-graduate level field of mathematics that involves a lot of theory and proofs. For mathematical rigor, you must understand differential equations and optimization solvers. They try to solve continuous-time control problems which inherently requires a lot of complicated maths and logical thinking to understand how you get from one place to another. They try to prove properties of dynamical systems and optimization solvers to fully understand the nitty gritty details of how you are solving the problem and when some algorithm will work.

In the real world however, we almost always have to work in the discrete time world, and we have to rely on simulation or testing to understand if the controller works. Theoretical results or predictions are largely useless. Finally, we don't need to understand the rigorous mathematical proof of why a solver works. There are very few tutorials on Optimal Control that focus on someone who just wants to use these methods and algorithms, with just enough understanding to be able to debug when things go wrong.

This article attempts to strip away the theory and proofs from Optimal Control and give a working mans understanding of methods like multiple shooting and direct collocation, along with examples (with code) for implementing some of these methods in popular libraries like CasADi.

1.1 Structure of this Article

First we will cover defining the system and the problem we are trying to solve in a mathematical way. This would involve continuous time equations, so the next section will be about

complete this

2 Defining the Problem

Optimal Control is about designing controllers that optimize (minimize) some metric. In contrast, classical control has often only worried about stabilizing and regulating, ie the system eventually gets to a goal state. Another way of thinking about control is Robust control, which wants to ensure the controller works even with uncertainty in our modelling or measurement of the system. However optimal control techniques are still used in robust control, as we are simply optimizing for a metric that includes robustness.

2.1 The Cost Function

The main feature at the heart of an optimal control problem is the objective/cost function. We generally define cost function as a function of 2 other functions of time $x(t), u(t)$. These may also be called the state trajectory and control trajectory respectively. This means its actually a cost **functional**, and analysis or optimization of it requires **Calculus of Variations**. But

that is only if we want to solve it directly (in continuous time), instead we will discretize the problem we so that we get an approximation of the cost functional as a function of a finite number of variables. It can come in many forms but one typical form is as follows:

$$J(x(\cdot), u(\cdot)) = \int_0^\infty g(x(t), u(t))dt$$

This defines the cost functional as the integration of a normal function of the state and control action over time.

We would then want to find the trajectories that minimize the cost function

$$\underset{x,u}{\operatorname{argmin}} J(x, u)$$

This requires some boundary conditions and constraints, which comes more under the differential equation section, because without it the optimal solution should be do nothing.

$$\begin{aligned} x(0) &= x_0 \\ x(t_{end}) &= x_{end} \\ \textbf{subject to } C_{ineq}(x, u) &\leq 0 \\ C_{eq}(x, u) &= 0 \end{aligned}$$

We may not have line 2 if our goals/trajectories do not have a defined end point.

section on examples

2.1.1 Examples

Minimize the time of the trajectory

$$J(x(\cdot), u(\cdot)) = \int_0^\infty \begin{cases} 1 & t \leq t_{end} \\ 0 & t > t_{end} \end{cases} dt = \int_0^{t_{end}} dt \quad (1)$$

Minimize the distance travelled (in state space)

$$\begin{aligned} J(x(\cdot), u(\cdot)) &= \int_0^\infty \left\| \frac{dx(t)}{dt} \right\|_2 dt \\ &= \int_0^\infty \|f(x(t), u(t))\|_2 \, dt \end{aligned}$$

This is closely related to minimizing the total energy required, for each of these you would take a subset of the state variable.

Minimize the energy required (different form)

$$J(x(\cdot), u(\cdot)) = \int_0^\infty \|u(t)\|_2 \, dt$$

Add more details and explanation

2.2 The Differential Equation

Given the cost function that takes a state and action trajectory we can ask how those are calculated/related to each other. This is essentially asking us to mathematically model a system, while there are methods such as the Lagrangian that model a system as an optimization (principle of least action), the most common form in Optimal control is to use differential equations. We can define the evolution of the system as a function $x(t)$ defined by a differential equation with boundary conditions

$$\begin{aligned} \dot{x} &= f(x, u) \\ x(0) &= x_0 \\ x(t_{end}) &= x_{end} \\ \textbf{subject to } C_{ineq}(x, u) &\leq 0 \\ C_{eq}(x, u) &= 0 \end{aligned}$$

The third line is not necessary but you would usually want it. The constraints model the limits on the system and the function $f(x, u)$ model how the system evolves, this is mostly used as part of the optimization process, rather than when solving for the state trajectory itself. As expected, different choices for the function $u(t)$ give different state functions/trajectories $x(t)$.

Keep in mind $x(t), u(t)$ are vector functions, and thus $f(x, u)$ is also vector-valued.

Notice that all these equations are living in continuous time. Real world is a continuous time system [1]. However our controllers generally live in discrete time

some more details/examples here

3 Solving the problem

There are two main meanings to the phrase "solving the optimal control problem" that we have defined in the previous sections. We may solve it for a specific

[1] Lets revisit this sentence if the quantum physicists every finally prove anything about the nature of our universe, regardless the planck timescale is far below anyone computation capabilities (for now)

set of initial conditions, and get one specific optimal trajectory. Or we may try to solve it for all initial conditions and get a global control low.

3.1 Trajectory Optimization

This solves a single instance of the problem, ie for some specific initial conditions. This can be thought of as optimizing an open loop plan for controlling the robot. Open-loop is not generally considered a good, robust method of controlling robots or any other kind of system that has to interect with the real world. However, by developing a closed loop controller for following the optimized trajectory plan, you may be able to get the best of both worlds.

There is also the approach of continually resolving the trajectory optimization problem, with the initial/boundary conditions being the real (measured) state of the system. This gives us feedback, brings the optimization into the control loop of the system, but it is only feasible if the problem can be solved very quickly, and more importantly predictably quickly.

3.2 Global controllers

What if we wanted to solve the problem for any and all initial conditions. Well besides the fact that however we formulate the trajectory optimization problem it should be able to be solved for any initial condition. What if there were a more efficient way, either at solve time or at run time? The ideal goal is a simple, mostly constant time algorithm that gives the optimal control action from just the current state.

The well known result is that for linear systems and quadratic cost functions we can create a linear feedback law for optimally controlling the system. This is known as the **Linear-Quadratic-Regulator**. I won't cover the derivation or even the equations for the LQR, you can look at just about any optimal control lecture notes to find this.

The way to solve the optimal control globally is to find a **Lyapunov Function**

$$L(x) = \min_u J(x(\cdot), u)$$

This is basically a function (of the state) that gives you the how much cost/energy/time is required to reach the goal. From this function it is easy enough to recover optimal action. As we know the gradient of a function is the direction of steepest descent. In this context that means that ∇L is a vector that describes a direction in state space that would minimize the cost left as fast as possible. So from the equation $\nabla L = f(x, u)$ we can solve for u to get the optimal action.

This is obviously locally optimal, if we go in this direction, the amount of time/energy/cost left will be minimized. Is this definitely globally optimal? is it possible that although we are making progress the goal as fast as possible right now, that a short term less profitable action would end up taking less time/cost/energy in total? Suffice to say no, to understand this you must understand **Pontryagin's Maximum Principle**

The problem with this is either you solve it in continuous time, which is only analytically possible for certain classes of problems, and requires some creative thought to find what the Lyapunov function would look like. Or you solve it in discrete time, which mostly comes down to having a large Look Up Table of precomputed optimal costs for every possible boundary condition. This is expensive in terms of computational time as well as memory. The field of reinforcement learning is in some sense trying to solve this global controller problem by representing the Lyapunov function as a deep neural network or some other huge parameterized function, and then 'learning' the specific parameters according to the system model,

Trajectory optimization as a method is general for any class of system, but each solution applies to only one set of boundary conditions. On the other hand, Lyapunov Functions are general for any set of boundary conditions, but usually apply to only one class of systems.

4 Just enough theory about optimization solvers

In the next section we will convert from optimizing in continuous time, over the uncountably infinitely space of all possible functions, to optimizing over a discrete number of decision variables. But lets take a moment to talk about how this kind of problem can be solved. There are many optimization algorithms that have different properties and work for different kinds of problems.

At its most general we have nonlinear optimization or

nonlinear programming. This works for any kind of problem, but it also means we can say little about how well we can really solve the problem. We can't be sure that we will find the global optimal solution, or any solution at all. Within this we have two major classes of optimization solvers: gradient-free/derivative-free optimizers and the opposite, which doesnt really have a name (gradient-full? gradient-based?). The most effective way we have for solving optimization problems uses the gradients (or hessians, the second derivative). This is because the gradient is the direction of steepest descent, this makes it a good direction to search for the next guess of the optimal solution. Different solvers have different specifics but they all use the gradient in a similar way, as a search direction.

Gradient-free solvers come into play when the derivative is not known or hard to compute. Historically methods that used gradient required you to derive the gradient by had and code it into a function manually, with all the error-prone nature that that suggests. However the field of automatic or algorithmic differentiation has made strides in recent years, mostly for machine learning applications in order to use gradient descent methods even with completely arbitrary and huge neural networks. Optimal Control problems are another good application of AD, and many general purpose optimal control toolkits make use of this technique. CasADi is one such library, letting you write the optimization problem as expressions of symbolic variables that is then used to generate functions for the solver, including the derivatives required.

The next important class of optimization problems is of Convex Optimization. These are the set of optimization problems where the objectives are convex functions. Among many things this means we can solve the problems faster, be guaranteed of making progress, and most importantly the global minimum is the only local minimum. If you're system admits being posed as a convex optimization problem it is usually a good idea to do so.

A nonlinear optimization solver requires some initial guess, and only guarantees you'll find **a** locally optimal solution in the neighbourhood of the initial guess. Convex Optimizations be solved entirely from scratch and guarantee a globally optimal solution. If you want to use a nonlinear solver you'll want to think of a good way to find an initial guess, maybe even analyze the problem to find a way to always fall into the globally optimal solution.

To learn all about convex optimization problems you should read **Convex Optimization by Stephen Boyd**. Within convex optimization you have different classes of problems with different levels of 'difficulty', although as a user of blackbox optimization libraries they are all about the same to us. See 1

In terms of solving convex optimization problems: the techniques used may also be employed by nonlinear optimization solvers. This would/may give them the guarantee that the solution find is the optimal for the convex subset of the objective function that the initial guess lies in. It also means that if your problem does happen to be (globally) convex the general nonlinear solver would work quite well, and give you a global solution. Using a general nonlinear solver may be desirable as they may accept the objective function and constraints in a more flexible format. check

The other thing you can do is use a dedicated convex optimization solver, see ECOS, OSQP, GUROBI, etc. These would require you to really understand the so-called 'standard' form of the class of convex optimization problems that each solver targets. You would usually provide the objective function implicitly as vectors and matrices that represent parameters of the restricted class of objective functions and constraints you may use. Unfortunately this standard form is different between every solver, and require some manual manipulation to get from the more human-readable and natural formulation of the problem you may come up with.

Most programming languages have optimization problem 'modelling' libraries that let you write the optimization problem using natural looking expressions of symbolic variables. The libraries then convert this to the expected format of different solvers, taking some once-off computation time but saving you a lot of trouble. See CVXPY and CVX.jl for examples of such libraries. These libraries technically target a subset of convex optimization problems **Disciplined Convex Programming**, this is the subset of convex optimization problems that can be built from a few building block atomic functions such that the convexity of the resulting problem is statically guaranteed. An arbitrary expression, even consisting of convex functions may not itself be convex, DCP formalizes a way of

	Objective	Equality Constraints
Linear Programming	Affine	Affine
Quadratic Programming	Symmetric Positive Definite Quadratic	Linear
Second Order Cone Programming	Symmetric Positive Definite Quadratic	Conic
Semidefinite Programming	Linear	Positive Semidefinite Cone

Figure 1: Kinds of Convex Optimization problems

checking that a problem is still convex, by restricting the problems you can build.

5 Discretizing the problem

As mentioned before solving the problem directly is intractable as we are optimizing over the space of all functions, which is isomorphic to $\mathbb{R}^\infty$. This is not solvable by computers, and is only analytically solvable for certain classes of objective functions and constraints. We can keep applying analysis and reach other forms of the problem (for example the **hamilton-jacobi-bellman** equation). We could then discretize that, but it is simplest to discretize the cost functional and state/control trajectories directly.

To reiterate our goal in this section, we want to take the optimization problem from optimizing over the the space of all functions to a finite dimensional subspace of functions, this is equivalent to solving over a finite number of variables. Fundamentally the approach is to use a method for solving differential equations numerically to turn the continuous time integral in J into an expression involving the values of $x(t)$ and $u(t)$ at only a few discrete points in time. This also then involves solving the differential equation $\dot{x} = f(x, u)$ at those points in time. Our decision variables for the actual optimization would then be the values of the $x(t), u(t)$ at those points in time, usually denoted as $x_0, x_1, \ldots, x_n, u_0, u_1, \ldots, u_n$.

This is necessarily an approximation and so a large part of theory is understanding the kinds and bounds on the error introduced by different approximations. But we can skip over that and just use some rules of thumb for deciding on how to discretize.

For example we could do the following by using first order methods with a fixed timestep size h

$$\dot{x} = x + u \qquad \implies x_{i+1} = x_i + (x + u)h$$
$$J = \int_0^T \left(x(t)^2\right) dt \qquad \implies J = \sum_0^n x_i^2 h$$

If we were to optimize J now it would be an optimizing over a finite number of decision variable x_i, u_i

While we are technically discretizing a optimization problem, the optimization problem consists of a integration (cost function) and a differential equation solution. These are the parts that we are really trying to discretize, so we take clues from how to discretize and solve integration problems and differential equations. There problems are actually highly similar to each other. We can use almost any and all methods for integrating and solving differential equations as part of this step.

Lets focus on the differential equation problem first as it is the most important part. We would like to solve for $x(t)$ (assuming we have $u(t)$). Because the starting point and ending point are generally constrained we have a boundary value problem. The first class of methods for solving boundary value problems are called the shooting methods. These lift initial value problem solvers (essentially integrators for differential equations) into solving the boundary value problem. The other class of methods is the Collocation methods, which assume a structure/parameterization of the solution and then solving for the specific solution/parameters.

5.1 Shooting Methods

This whole section: So much maths

Shooting methods push the result forward in time from the initial value, evaluating $f(x, u)$ at specific points and using some approximation method to get the next point. All of the method of solving differential equation initial value problems that you know can be used here.

For solving boundary value problems (which generally add a constraint at the end as well as the beginning) we just need to a little bit of extra work.

$$\dot{x} = f(x, u)$$
$$x(0) = x_0$$
$$x(t_{end}) = x_{end}$$

Suppose $x_p(t)$ was the solution $x(t_{end})$ of solving the IVP defined by

$$\begin{aligned} \dot{x} &= f(x, u) \\ x(0) &= x_0 \\ x'(0) &= p \end{aligned}$$

Define $F(p) = x_p(t_{end}) - x_{end}$, we then solve for p such that $F(p) = 0$, ie find the root of F.

Within shooting methods we can make a choice, single shooting or multiple shooting, single shooting is mostly as I have described above, while multiple shooting makes one key change to improve the performance and accuracy of the solver.

5.2 Collocation Methods

This whole section: So much maths

6 Time Optimal Path Tracking is Convex

It turns out if we have a 'path' that we want to track, we can generate a trajectory that optimizes the time for following that path, by solving a convex optimization problem (atleast the objective function will be convex, the convexity of constraints depends on the system that you are controlling)