11 Trajectory Optimization for Mechanical Systems with Hybrid Dynamics

In last lecture, we saw how we can represent trajectory optimization problems as finite-dimensional optimization problems. We briefly touched on some methods for computing trajectories for a couple specific problem instances. Particularly, we saw how control policies can be computed for the linear dynamics, quadratic cost problem (LQR) by formulating it as a Quadratic Program. Similarly, we saw how we can solve for locally optimal trajectories for systems with nonlinear dynamics by iteratively solving Quadratic Programs that represent Linear-Quadratic approximations of our problem. We showed that this method was nearly equivalent to Sequential Quadratic Programming.

In this lecture, we will expand upon these ideas to show how we can use Sequential Quadratic Programming to handle problems with more general constraint and cost terms, including unilateral constraints. We will demonstrate how, by using solvers based on SQP and similar ideas, we can compute trajectories for complicated systems. Specifically, we will dive into a particular application of generating trajectories for mechanical systems with hybrid dynamics. This example will bring together many of the components we have seen so far in the course in a very practical way.
11.1 Sequential Quadratic Programming: Continued

In last lecture, we presented Iterative-LQR as a method for generating trajectories for systems with nonlinear dynamics. This was a natural extension to what we knew how to do with systems with linear dynamics, in the LQR case. We showed that since LQR can be thought of as solving a Quadratic Program, that Iterative-LQR can be thought of as sequentially solving quadratic programs. We saw that a very similar method, Sequential Quadratic Programming, solves nearly identical problems. However, we will see that SQP is able to optimize a much larger class of optimization problems, and will prove to be a relatively general optimization method. Specifically, we will show how it can be used to optimize problems which have more complicated constraints than the type we saw in the Iterative-LQR case.

Consider the general nonlinear-optimization problem

$$\min_z f(z)$$

s.t.  
$$h_1(z) = 0, \ldots, h_m(z) = 0$$
$$g_1(z) \leq 0, \ldots, g_r(z) \leq 0$$

(1)

The Lagrangian for this problem is given by

$$L(z, \lambda, \mu) = f_0(z) + \lambda^T h(x) + \mu^T g(x).$$

The KKT necessary conditions of optimality for this problem are given by the following:

$$\nabla_z L(z^*, \lambda^*, \mu^*) = 0$$
$$h_j(z^*) = 0, \ j \in 1, \ldots, m$$
$$g_j(z^*) \leq 0, \ j \in 1, \ldots, r$$
$$A(z^*) = \{i : g_i(z^*) = 0\}$$
$$\mu_j^* \geq 0, \ j \in 1, \ldots, r$$
$$\mu_j^* = 0, \ j \notin A(z^*)$$

(2) (3) (4) (5) (6) (7)

Here $A(z)$ represents the active set of unilateral constraints at $z$. The constraint (7) essentially says that if a constraint is inactive at the optimum, it can be left-out in the problem description and the resulting solution will be unchanged. This leads to an important observation: if we know the set of constraints that will be active at the optimum, we can convert our problem to an equality-only constrained problem and solve using the method presented at the end of last lecture. This approach is called an active-set method, and is used in solving Quadratic Programs with unilateral constraints.

11.1.1 Active-set Quadratic Programming

Let $W$ be the best guess of the constraints which will be active at the (unknown) optimal point $z^*$, i.e. $W^* = A(z^*)$. Solve an equality-constrained quadratic program for those constraints, in the space of deviations from the previous solution iterate. If the optimal solution for this problem is $z_0^*$, then let $z_1 = z_0^*$. The iteration continues, replacing $z_k$ by $z_k^*$ and solving the active-set quadratic program for those constraints.
step is 0, then check the value of each of the Lagrange multipliers associated with the constraint terms. If any of these terms are 0, remove the index \( i \) corresponding to the constraint with multiplier equal to 0 from the set \( W \), and resolve the problem. Otherwise, terminate with the previous iterate as the optimal solution.

If the optimal step is non-zero, then the previous iterate is non-optimal given the current constraint set. Update the solution iterate in the direction of the optimal step. If any inequality constraints become violated in this update, add the corresponding indices to the working active set \( W \). Otherwise, keep the working active set the same. Repeat this process until convergence [1].

11.1.2 Integrating into SQP

To incorporate this process into Sequential Quadratic Programming, we simply form an inequality-constrained Quadratic Program by linearizing all of the nonlinear constraints in problem (1), forming a quadratic approximation of the Lagrangian, and solving the corresponding active-set Quadratic Program that results. The final active-set of constraints for this quadratic program are then used as the initial guess of the subsequent quadratic program approximation. There is an excellent presentation of such methods in the books [1] and [2], which I recommend reading if you are interested.

There are many subtle details that need to be accounted for in order to make these methods converge reliably and maintain feasibility throughout the approximations made at each iteration. Therefore, unless your problem has very demanding run-time requirements, in practice it is recommended to use an off-the-shelf professional solver when solving similar optimization problems. Professional solvers are typically very robust to all sorts of edge-cases that can arise in practice, and will save your brain-power for thinking about your problem at hand. One such SQP solver which you will use your next homework is SNOPT [4].

11.2 Case Study: Mechanical Systems with Hybrid Dynamics

We will now turn to applying trajectory optimization to a particular type of hybrid system. We have seen in earlier lectures that a very simple model of a bouncing ball can be represented as a hybrid system with two discrete modes: a flight mode and a contact mode. This (dynamically regularized) model is recalled in Figure 1. This model can be thought of as having a very stiff spring at the point where the ball hits the ground, with spring constant \( k = \frac{1}{\epsilon} \). Consider a slightly more complicated model of this ball, one where the ball is able to jump by applying some force during the contact phase, and alter it’s flight by applying some thrust in the air. In this model, the dynamics of mode \( q_1 \) in Figure 1 would be such that \( \dot{x}_2 = -x_1/\epsilon + u \), and the dynamics of mode \( q_2 \) would be such that \( \dot{\theta}_2 = -g + v \), where \( u \) and \( v \) are the jump and thrust forces, respectively. We could also impose some limits on the control signal applied, such that \( 0 \leq u \leq u_{\text{max}} \) and
\(-v_{max} \leq v \leq v_{max}\). How might we go about optimizing a trajectory for this actuated ball?

Because this model is so simple, we could probably hand-design a control strategy on pen and paper for most tasks. However, we will formulate the trajectory optimization problem for this system as a non-convex optimization problem and leverage the tools we have learned about to solve it. In order to do so, we must make some adjustments to the model to account for the hybrid dynamics. Because we are unsure what mode each discrete time-step of our trajectory will be in at any given time, it is unclear which dynamic constraint should be imposed at that time. This is the problem of mode-scheduling. Traditionally, when optimizing trajectories for hybrid systems, the mode sequence had to be pre-specified. For our simple system here, that would not be such a difficult problem - we know the system alternates between flight and contact modes, and the duration of each mode can easily be computed. However, for systems with many discrete modes, such as a robotic hand manipulating objects, trying to manually identify the optimal mode sequence is an unreasonable task.

One way you could imagine handling this problem is to use a slightly modified differential equation which has an ‘if’ statement checking if the ball position is below the ground, and then applying the corresponding spring force if so. However when solving numerically for such a system, we run into issues. In order to ensure that the constraint violation is small, a very large spring constant would need to be applied so as to quickly alleviate the constraint violation. This then requires that the time-discretization is correspondingly very small so the dynamics do not become unstable. This large spring constant turns our model into what is called a ‘stiff’ ODE. A nice way to handle this problem is to use an implicit integration scheme, which simply evaluates the derivative of the state variables at the future location instead of the current location,
i.e. \( x_{k+1} = x_k + hf(x_{k+1}) \) instead of \( x_{k+1} = x_k + hf(x_k) \). A very nice, intuitive example of using an implicit method to handle stiff ODEs can be found in [5].

The spring model also has some additional drawbacks. It is not clear how to handle certain physical phenomenon such as friction using such models. For our simple bouncing ball example, this does not apply, but we will still consider a more general form of implicit dynamics which can handle more general systems. The method comes from a technique from computer graphics, called ‘implicit time-stepping for rigid bodies’, first introduced in [6]. To understand this method, we need to recall from physics the equations of motion which can describe the contact interactions we wish to simulate and optimize trajectories over.

### 11.2.1 Manipulator Equations

Recall from lecture 8 that the equations of motion for rigid-body systems can be found through the calculus of variations. It turns out that to find the equations of motion for a rigid-body which comes into contact with another object, we can also use the calculus of variation where the performance index is subject to some additional unilateral constraints. We will omit the derivation here, and simply present the resulting equations of motion. If you are interested in this derivation, you can find a nice presentation in [8] and [9]. We have, for an articulated rigid body system with a single contact point, the following equations of motion:

\[
M(q)\ddot{q} + C(q, \dot{q}) = \tau + J^T\lambda \\
0 \leq \phi(q) \perp \lambda \geq 0
\]

Here, \( \phi \) is the signed-distance function to the contact point, \( M \) is the mass matrix, \( q, \dot{q} \) and \( \ddot{q} \) are the generalized coordinates, velocities and accelerations of our system, respectively, \( C \) represents the Coriolis, centrifugal and gravitational forces, \( \tau \) represents the inertial generalized forces, \( \lambda \) represents a normal force acting at the contact point, and \( J \) is the constraint Jacobian \( \frac{\partial \phi}{\partial q} \) evaluated at the contact point. The complementarity condition in Eq. 9 represents the constraint that the contact surface can not be penetrated, and normal forces can only be applied when in contact. Ignoring this constraint and the normal force in Eq. 8, the equations revert to Newton's second Law, \( f = ma \) in the generalized coordinates \( q \).

These equations give us a form of dynamics that is valid independent of whether our ball is in the flight phase or contact phase. The additional complementarity constraints implicitly define the distinction between the two modes. In order to use these equations to generate trajectories for our bouncing ball, we need to first discretize. Following with our insight regarding the stiff dynamics, we use backward-Euler integration to
discretize our dynamics. This gives
\[ \frac{1}{h} M_{k+1} (q_{k+1} - q_k) + C_{k+1} = \tau_{k+1} + J^T \lambda_{k+1} \]
(10)
\[ 0 \leq \phi(q_{k+1}) \perp \lambda_{k+1} \geq 0 \]
(11)

We can now use Eq. 10 as a dynamic constraint at every discrete-stage in our trajectory. By jointly optimizing over the states \((q, \dot{q})\), controls \((\tau)\), and contact forces \((\lambda)\), we optimize trajectories without knowing beforehand what discrete mode of the dynamics will be active at a given time! Using this strategy for trajectory optimization, denoted ‘Contact-implicit Trajectory Optimization’, was first introduced in [10].

We do have to make one slight adjustment to our bouncing ball model to make it easier to directly apply the strategy outlined. Because we no longer allow any penetration of our contact surface, we will assume that the ball is modeled by the Pogo Stick model, as in [13]. A graphical depiction of this model can be seen in Figure 2. In this model, there are two point-masses attached to a spring, and the contact point is defined to be between the bottom mass and the ground surface. The thrust force which can be applied during the flight phase affects both masses in the same direction, while the jumping force applied during contact phase pushes the masses apart. Using this model, we are now able to define our optimization problem. Consider the problem of starting from some initial height, and trying to jump to a particular goal position while minimizing total control effort. We can formulate the optimization as follows:

\[ \min_{\tau} \sum_{k=0}^{N-1} \tau_k^T R \tau_k \]
\[ \text{s.t.} \quad \frac{1}{h} M_{k+1} (q_{k+1} - q_k) + C_{k+1} = \tau_{k+1} + J^T \lambda_{k+1} \]
\[ q_{k+1} = q_k + h \dot{q}_{k+1} \]
\[ 0 \leq \phi(q_{k+1}), \quad \lambda_{k+1} \geq 0, \quad \phi(q_{k+1})^T \lambda_{k+1} = 0 \]
\[ 0 \leq \phi(q_{k+1}), \quad u_{k+1} \geq 0, \quad \phi(q_{k+1})^T u_{k+1} = 0 \]
\[ \tau_{\text{min}} \leq \tau_k \leq \tau_{\text{max}} \]
\[ [q_0^T \quad \dot{q}_0^T]^T = x_{\text{init}} \]
\[ [q_N^T \quad \dot{q}_N^T]^T = x_{\text{goal}} \]

We can transcribe the above problem 12 into a solver such as SNOPT and generate locally optimal trajectories for this system. In homework 3, you will work through an example which does exactly that.

References

Figure 2: Pogo Stick Model of Bouncing Ball


