

Underactuated Robots

Lecture 2: Trajectory Optimization

Nicola Scianca

December 2024

trajectory optimization

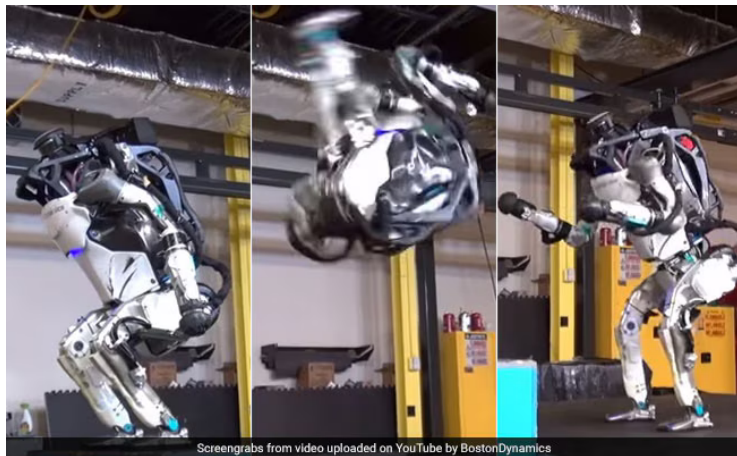
- **trajectory optimization** is a mathematical approach to compute the best trajectory for a system to follow, while satisfying **constraints** and minimizing a **cost function**
- it involves optimizing the states and control inputs over time to achieve a specific goal, such as minimizing energy, time, effort, or following a desired path
- trajectory optimization can be used, in many forms, for tasks like motion planning, navigation, and control

trajectory optimization: examples

- controlling a **fixed-base manipulator** to minimize input torques while avoiding workspace obstacles
- controlling an **autonomous vehicle** to minimize the distance from the center of the road while avoiding incidents
- controlling a **legged robot** to follow a given reference velocity and making sure it doesn't fall



trajectory optimization: examples



Screenshots from video uploaded on YouTube by BostonDynamics

<https://www.youtube.com/embed/fRj34o4hN4I>

- in the continuous case, we can formulate a basic trajectory optimization problem as

$$\min \int_{t_0}^{t_f} l(x(t), u(t), t) dt + l_t(x(t_f))$$

subject to

$$\dot{x} = f(x(t), u(t)), \quad \text{for } t \in [t_0, t_f],$$

$$x(t_0) = \bar{x}_0,$$

$$h(x(t), u(t)) = 0, \quad \text{for } t \in [t_0, t_f],$$

$$g(x(t), u(t)) \leq 0, \quad \text{for } t \in [t_0, t_f],$$

trajectory optimization

- in this problem we have a **cost function** composed of a **running cost** integrated along the trajectory, and a **terminal cost** on the final state
- we also have constraints on the **initial state** and on the **system dynamics**
- we can also add other **equality constraints** $h(x(t), u(t)) = 0$ and **inequality constraints** $g(x(t), u(t)) \leq 0$, for example to perform obstacle avoidance, or to limit control effort

system dynamics discretization

- we consider a **discrete** sequence of input and states over N time-steps, covering the entire duration of the task

$$x_0, x_1, \dots, x_{N-1}, x_N$$

$$u_0, u_1, \dots, u_{N-1}$$

- these states and inputs are connected via the systems **dynamics**, which we write in discrete form

$$x_{k+1} = f(x_k, u_k)$$

- for physical systems, which have continuous dynamics, we assume to have discretized them in some way (e.g., Euler, Runge-Kutta, ...)

- to formulate a trajectory optimization problem, we choose a **cost function**, which is something that we want to minimize

$$\min \sum_{i=0}^{N-1} \underbrace{l_k(x_i, u_i)}_{\text{running cost}} + \underbrace{l_N(x_N)}_{\text{terminal cost}}$$

- for example, the cost function might be a measure of energy consumption, or the time necessary to complete the task, or the distance to some reference point or trajectory
- the cost function is usually constituted by a **running cost**, evaluating the cost of states and inputs along the trajectory, and a **terminal cost** evaluating the cost of the final state

- this cost function must be minimized while taking into account some necessary **constraints**

$$x_{i+1} = f(x_i, u_i) \quad \text{for } i = 0, \dots, N - 1$$
$$x_0 = \bar{x}_0$$

- the input and state sequences should satisfy the dynamics, which makes it so that the resulting input and state sequences constitute legitimate **trajectories** for the system
- the initial state should correspond to the initial state of the system \bar{x}_0
- we could add additional constraints (e.g., max input, obstacle avoidance, ...)

trajectory optimization

- a basic trajectory optimization problem has the following form

$$\begin{aligned} \min \quad & \sum_{i=0}^{N-1} l_k(x_i, u_i) + l_N(x_N) \\ \text{s. t.} \quad & x_{i+1} = f(x_i, u_i) \quad \text{for } i = 0, \dots, N-1 \\ & x_0 = \bar{x}_0 \end{aligned}$$

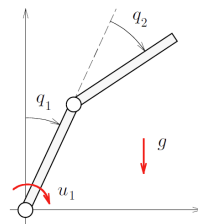
- the optimal input that we get by solving this problem depends on the current state \bar{x}_0
- if we could solve this problem for **every** state, we would have found an **optimal control law** $u(x)$

trajectory optimization

- if we can't explicitly write out a feedback law $u(x)$, maybe we can still solve the optimization problem, starting from one initial state \bar{x}_0 , and find a particular trajectory as solution
- by doing this we don't get an optimal control law, but rather a particular **optimal trajectory**, i.e., a sequence of inputs and states which satisfy the system dynamics
- we can perform this optimization **offline**, and then do trajectory tracking to keep the system as close as possible to this optimal trajectory

example

- consider a **pendubot**: a double pendulum with one actuated joint
- the state variables are
 - ▶ θ_1 angle of the first link
 - ▶ θ_2 angle of the second link (relative)
 - ▶ $\dot{\theta}_1$ angular velocity of the first link
 - ▶ $\dot{\theta}_2$ angular velocity of the second link
- goal: swing up the pendubot to its upright configuration $(0, 0, 0, 0)$
- initial condition: stable equilibrium $(\pi, 0, 0, 0)$ in which both arms of the pendulum are pointing down



example

- first, we discretize states and inputs

$$u_0, u_1, \dots, u_{N-1}$$

$$x_0, x_1, \dots, x_{N-1}, x_N$$

- then, we define a cost function, which in our case is just to minimize the sum of the squares of the input

$$\min \sum_{i=0}^{N-1} u_i^2$$

- and we define our initial state (given) and our final state (target) as constraints

$$x_0 = (\pi, 0, 0, 0)$$

$$x_N = (0, 0, 0, 0)$$

- the dynamics of the pendubot can be written as

$$M(q)\ddot{q} + c(q, \dot{q})\dot{q} + g(q) = \underbrace{(u, 0)^T}_{\text{zero component due to the underactuation}}$$

zero component due to the underactuation

- first we rewrite it as $\dot{x} = f(x, u)$

$$\underbrace{\frac{d}{dt} \begin{pmatrix} q \\ \dot{q} \end{pmatrix}}_{\dot{x}} = \underbrace{\begin{pmatrix} 0_{2 \times 2} & I_{2 \times 2} \\ 0_{2 \times 2} & -M^{-1}c \end{pmatrix} \begin{pmatrix} q \\ \dot{q} \end{pmatrix} + \begin{pmatrix} 0_{2 \times 2} \\ -M^{-1}g \end{pmatrix} + \begin{pmatrix} 0_{2 \times 2} \\ M^{-1} \end{pmatrix} \begin{pmatrix} u \\ 0 \end{pmatrix}}_{f(x, u)}$$

- we must discretize the dynamics, e.g. with Euler integration

$$x_{k+1} = x_k + \Delta \cdot f(x_k, u_k)$$

example

```
import ... [numpy, casadi, model, etc ...]

f = model.get_pendubot_model()

# parameters
N = 200
delta = 0.01

# set up optimization problem
opt = cs.Opti()
opt.solver("ipopt", p_opts, s_opts)
X = opt.variable(4,N+1)
U = opt.variable(1,N+1)

for i in range(N):
    opt.subject_to( X[:,i+1] == X[:,i] + delta * f(X[:,i], U[0,i]) )

opt.subject_to( X[:,0] == (math.pi, 0, 0, 0) )
opt.subject_to( X[:,N] == (0, 0, 0, 0) )

cost_function = cs.sumsqr(U)
opt.minimize(cost_function)

# get solution
sol = opt.solve()
u = sol.value(U)
x = sol.value(X)
```

how do we solve the optimization problem

- in the example we used `casADi` to set up a problem for `ipopt`, which is a very popular off-the-shelf optimization algorithm
- however, our problem is not a generic optimization problem, it is a problem with a particular structure
- sometimes it is possible to use techniques that exploit the particular structure of an optimal control problem (e.g., DDP)

- **transcription** is the particular way in which we choose we write our optimization problem so that we can then solve it
- there are many transcription methods: shooting methods, collocation methods, ...
- we will discuss the two main paradigms that characterize shooting methods: **single shooting** and **multiple shooting**
- aside from very simple cases, the solution process will always involve **numerical iteration** that hopefully **converge** to an optimal solution

- the same problem can be **transcribed** in different ways: we will consider the two main possibilities (but there are others)
- **single shooting** methods: we perform substitutions to have less variables and constraints, but we increase the nonlinearity
- **multiple shooting** methods: we have a larger number of variables held together by constraints, so the problem is larger but less nonlinear

transcription methods: single shooting

- in **single shooting** methods, the **inputs** u_0, \dots, u_{N-1} are decision variables: we eliminate states from the cost function by substituting in the dynamics

$$x_1 = f(\bar{x}_0, u_0)$$

$$x_2 = f(x_1, u_1) = f(f(\bar{x}_0, u_0), u_1)$$

...

$$x_k = f(f(\dots f(\bar{x}_0, u_0), u_1), u_2), \dots), u_{k-1})$$

- we also substitute x_0 with \bar{x}_0 to eliminate the initial state constraint
- we get a problem that is only dependent on the input sequence, and has **no constraints**

transcription methods: multiple shooting

- in **multiple shooting** methods, the **inputs** and **states** are decision variables: they are linked together by constraints (the dynamic equation)

$$x_1 = f(x_0, u_0)$$

$$x_2 = f(x_1, u_1)$$

...

$$x_k = f(x_{k-1}, u_{k-1})$$

- now we have to solve a constrained problem, but the advantage is that we have a simpler cost function
- if we stop the solution before convergence, the dynamics might not be satisfied exactly

transcription methods: example

consider this example with horizon length $N = 2$

- discrete inputs and states

$$u_0, u_1,$$

$$x_0, x_1, x_2$$

- cost function

$$\sum_{i=0}^1 \left(u_i^2 + x_i^2 \right) + x_2^2$$

- dynamics

$$x_{k+1} = f(x_k, u_k)$$

- **single shooting** formulation

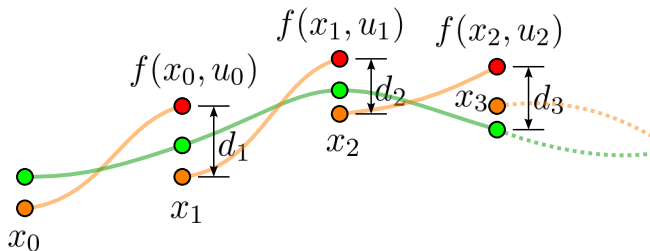
$$\min_{u_0, u_1} \left(u_0^2 + u_1^2 + \bar{x}_0^2 + f(\bar{x}_0, u_0)^2 + f(f(\bar{x}_0, u_0), u_1)^2 \right)$$

- **multiple shooting** formulation

$$\begin{aligned} \min_{\substack{u_0, u_1, \\ x_0, x_1, x_2}} & \left(u_0^2 + u_1^2 + x_0^2 + x_1^2 + x_2^2 \right) \\ \text{s. t.} \quad & x_0 = \bar{x}_0 \\ & x_1 = f(x_0, u_0) \\ & x_2 = f(x_1, u_1) \end{aligned}$$

multiple shooting: defects

- in a multiple shooting formulation, the constraints are not necessarily satisfied until we reach convergence
- the difference between the forward integration and the next variable is called the **defect** $d_{k+1} = x_{k+1} - f(x_k, u_k)$



single shooting: numerical problems

- in single shooting, the entire trajectory is determined by forward integration from an initial guess for the control inputs
- this can lead to **numerical instability**, especially for long time horizons or highly nonlinear systems
- the initial trajectory might be too far away from the optimum, or in the worst case we might even have a software failure (numbers get too big)

single shooting vs. multiple shooting

- let's recap the main differences between single and multiple shooting methods

single shooting	multiple shooting
less variables	more variables
cost function is very nonlinear	cost function is usually quadratic
less constraints	more constraints
initial guess could diverge	initial guess is stable
result is always a feasible trajectory	result could have defects

- if possible, it is usually better to do multiple shooting rather than single shooting

sequential quadratic programming

- **Sequential Quadratic Programming** (SQP) is an iterative method for nonlinear optimization, solving a sequence of optimization subproblems
- at each iteration, SQP solves a **Quadratic Programming** (QP) approximation of the original problem, where the objective function is quadratic, and the constraints are linearized
- despite its simplicity, SQP can be quite effective; for example it is used to control the **MIT Humanoid** [Khazoom et al., 2023]

Khazoom et al., "Tailoring Solution Accuracy for Fast Whole-Body Model Predictive Control of Legged Robots", Robotics and Automation Letters, 2024

- we want to solve a **NonLinear Program** (NLP) with the following structure

$$\begin{aligned} \min_{\substack{x_0, \dots, x_N \\ u_0, \dots, u_{N-1}}} \quad & \sum_{i=0}^{N-1} l_k(x_i, u_i) + l_N(x_N) \\ \text{s. t.} \quad & x_{k+1} = f(x_k, u_k) \quad \text{for } k = 0, \dots, N-1 \\ & x_0 = \bar{x}_0 \end{aligned}$$

outline of the algorithm

- start from an **initial guess** for the trajectory (\bar{x}, \bar{u})
- find a **linear-quadratic approximation** of the NLP
- this gives us a QP, if we solve it we find a **descent direction**
- do a **line search** to find a step size in this direction
- find a new guess $(\bar{x}, \bar{u}) \leftarrow (x, u)$, **repeat** from step 2

linear-quadratic approximation

- we want to find approximate expressions for the cost function and constraints
- first, we define Δx_k and Δu_k as the variations of the k -th state and input with respect to some solution guess \bar{x}_k and \bar{u}_k

$$\Delta x_k = x_k - \bar{x}_k$$

$$\Delta u_k = u_k - \bar{u}_k$$

- by computing Δx_k and Δu_k we can update our guess, and iterate to get closer and closer to the optimal solution

linear-quadratic approximation

- we first find a quadratic approximation of the cost function

$$\begin{aligned}l(x_k, u_k) &\simeq l(\bar{x}_k, \bar{u}_k) + \left. \frac{\partial l}{\partial x_k} \right|_{\bar{x}_k, \bar{u}_k} \Delta x_k + \left. \frac{\partial l}{\partial u_k} \right|_{\bar{x}_k, \bar{u}_k} \Delta u_k \\&\quad + \frac{1}{2} \Delta x_k^T \left. \frac{\partial^2 l}{\partial x_k^2} \right|_{\bar{x}_k, \bar{u}_k} \Delta x_k + \frac{1}{2} \Delta u_k^T \left. \frac{\partial^2 l}{\partial u_k^2} \right|_{\bar{x}_k, \bar{u}_k} \Delta u_k \\&\quad + \Delta u_k^T \left. \frac{\partial^2 l}{\partial x_k \partial u_k} \right|_{\bar{x}_k, \bar{u}_k} \Delta x_k \\l_N(x_N) &\simeq l_N(\bar{x}_k) + \left. \frac{dl_N}{dx_k} \right|_{\bar{x}_k} \Delta x_k + \frac{1}{2} \Delta x_k^T \left. \frac{d^2 l_N}{dx_k^2} \right|_{\bar{x}_k} \Delta x_k\end{aligned}$$

- and a linear approximation of the constraints

$$\bar{x}_{k+1} + \Delta x_{k+1} \simeq f(\bar{x}_k, \bar{u}_k) + \left. \frac{\partial f}{\partial x_k} \right|_{\bar{x}_k, \bar{u}_k} \Delta x_k + \left. \frac{\partial f}{\partial u_k} \right|_{\bar{x}_k, \bar{u}_k} \Delta u_k$$

linear-quadratic approximation

- let's write them in more compact form

$$\begin{aligned} l(x_k, u_k) &\simeq \bar{l}_k + \bar{l}_k^x \Delta x_k + \bar{l}_k^u \Delta u_k \\ &\quad + \frac{1}{2} \Delta x_k^T \bar{l}_k^{xx} \Delta x_k + \frac{1}{2} \Delta u_k^T \bar{l}_k^{uu} \Delta u_k + \Delta u_k^T \bar{l}_k^{ux} \Delta x_k \end{aligned}$$

$$l_N(x_N) \simeq \bar{l}_N + \bar{l}_N^x \Delta x_k + \frac{1}{2} \Delta x_k^T \bar{l}_N^{xx} \Delta x_k$$

$$f(x_k, u_k) \simeq \bar{f}_k + \bar{f}_k^x \Delta x_k + \bar{f}_k^u \Delta u_k$$

the quadratic subproblem

- now we set up a **quadratic subproblem**, approximating the original nonlinear problem around the current guess (\bar{x}, \bar{u})

$$\min_{\substack{\Delta x_0, \dots, \Delta x_N, \\ \Delta u_0, \dots, \Delta u_{N-1}}} \sum_{k=0}^N \left(\bar{l}_k^x \Delta x_k + \bar{l}_k^u \Delta u_k + \frac{1}{2} \Delta x_k^T \bar{l}_k^{xx} \Delta x_k \right. \\ \left. + \frac{1}{2} \Delta u_k^T \bar{l}_k^{uu} \Delta u_k + \Delta u_k^T \bar{l}_k^{ux} \Delta x_k \right) \\ \left. + \bar{l}_N^x \Delta x_k + \frac{1}{2} \Delta x_k^T \bar{l}_N^{xx} \Delta x_k \right)$$

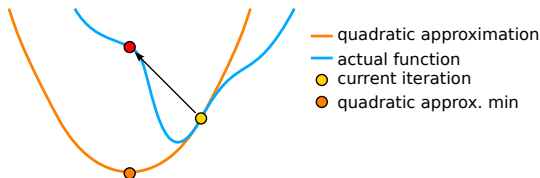
$$\text{s.t.} \quad \bar{x}_{k+1} + \Delta x_{k+1} = \bar{f}_k + \bar{f}_k^x \Delta x_k + \bar{f}_k^u \Delta u_k$$

$$\Delta x_0 = 0$$

- no need to include the constant terms \bar{l}_k and \bar{l}_N because they only alter the value of the function, not the descent direction

line search

- the solution to the previous problem represents a local **descent direction** for the NLP
- taking a full step ($\Delta x_k, \Delta u_k$) in this direction takes us to the minimum of the local approximation, which might overshoot the true local minimum and increase the cost instead of decreasing it



line search

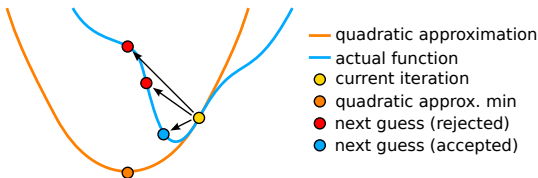
- instead of taking a full step, we perform a **line search**:

- ▶ start with an $\alpha = 1$
- ▶ take a step of size α in the descent direction

$$\bar{x} \leftarrow \bar{x} + \alpha \Delta x$$

$$\bar{u} \leftarrow \bar{u} + \alpha \Delta u$$

- ▶ evaluate the cost function on the new trajectory
- ▶ if the cost is increased, choose a smaller α and retry
- ▶ if the cost is decreased, accept the step (line search finished)



- a point on the descent direction satisfies the linearized constraints, not necessarily the constraints of the NLP
- if we evaluate a line search step by looking at the reduction of the cost, we might end up **increasing constraint violation**
- therefore, it is sometimes useful to add a term that evaluates violation of the nonlinear constraints during the line search

$$M = \underbrace{C}_{\text{cost function}} + \sum_{i=0}^N \underbrace{(x_{k+1} - f(x_k, u_k))^T (x_{k+1} - f(x_k, u_k))}_{\text{square of the defect}}$$

this is called a **merit function**

inequality constraints

- the previous formulation can easily be extended to include **inequality constraints**
- for example, a general constraint

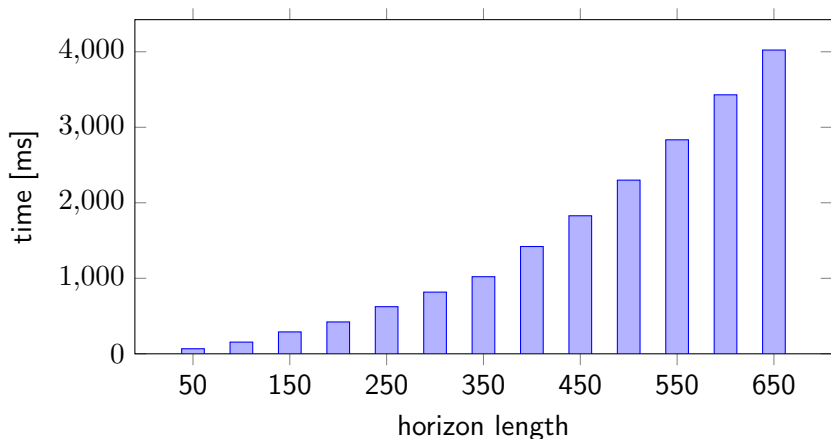
$$g(x_k, u_k) \leq 0$$

could be approximated in a similar fashion

$$g(\bar{x}_k, \bar{u}_k) + \left. \frac{dg}{dx_k} \right|_{\bar{x}_k, \bar{u}_k} \Delta x_k + \left. \frac{dg}{du_k} \right|_{\bar{x}_k, \bar{u}_k} \Delta u_k \leq 0$$

computational complexity

- the expensive step is the **factorization** of a large matrix, which usually scales badly with the number of variables



why is SQP good?

- it allows to define **constraints**, which is good because otherwise every objective would be in the cost function (many weights to tune!)
- the number of iterations can be set to a low number to get a suboptimal solution: good for **real-time** operation
- having constraints facilitates writing a **multiple shooting** formulation

why is SQP bad?

- we don't know apriori how it scales with the size of the horizon: it depends on the solver we use for the QP subproblem
- in particular, if we use a dense solver, we usually have to factor a large matrix, which is bad if we have a lot of variables
- if the solver properly exploits the **sparsity** of the problem, it could scale a lot better [Jordana et al., 2023]

Jordana et al., "Stagewise implementations of sequential quadratic programming for model-predictive control", Online preprint, 2023