Underactuated Robots
Lecture 1: Trajectory Optimization

Nicola Scianca

September 2024

info

- these slides were made by **Nicola Scianca**, researcher at Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza University of Rome, Italy
email: scianca@diag.uniroma1.it

- the slides and the examples are available at this **repository**:
`https://github.com/DIAG-Robotics-Lab/underactuated`

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

system dynamics

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

$$x_0, x_1, \ldots, x_{N-1}, x_N$$
$$u_0, u_1, \ldots, u_{N-1}$$

- 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, ...)

cost function

- 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

constraints

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

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

- 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

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

- 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

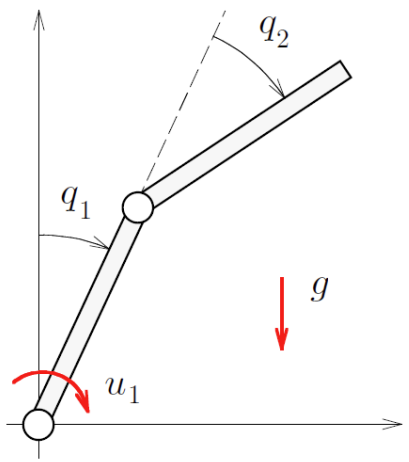
- if we can't explicitly write out a feedback law $u(x)$, maybe we can still solve the optimization problem, starting from one initial state $\bar{x}_0$, and find a particular trajectory as solution

- 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

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

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

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

example

- the dynamics of the pendubot can be written as

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

- 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 methods

- **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

transcription methods

- 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** $u0, \dots, u_{N-1}$ are decision variables: we eliminate states from the cost function by substituting in the dynamics

$$
\begin{aligned}
x_1 &= f(\bar{x}_0, u_0) \\
x_2 &= f(x_1, u_1) = f(f(\bar{x}_0, u_0), u_1) \\
&\dots \\
x_k &= f(f(\dots f(\bar{x}_0, u_0), u_1), u_2), \dots), u_{k-1})
\end{aligned}
$$

- 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)

$$
\begin{aligned}
x_1 &= f(x_0, u_0) \\
x_2 &= f(x_1, u_1) \\
&\dots \\
x_k &= f(x_{k-1}, u_{k-1})
\end{aligned}
$$

- 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

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

transcription methods: example

- **single shooting** formulation

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

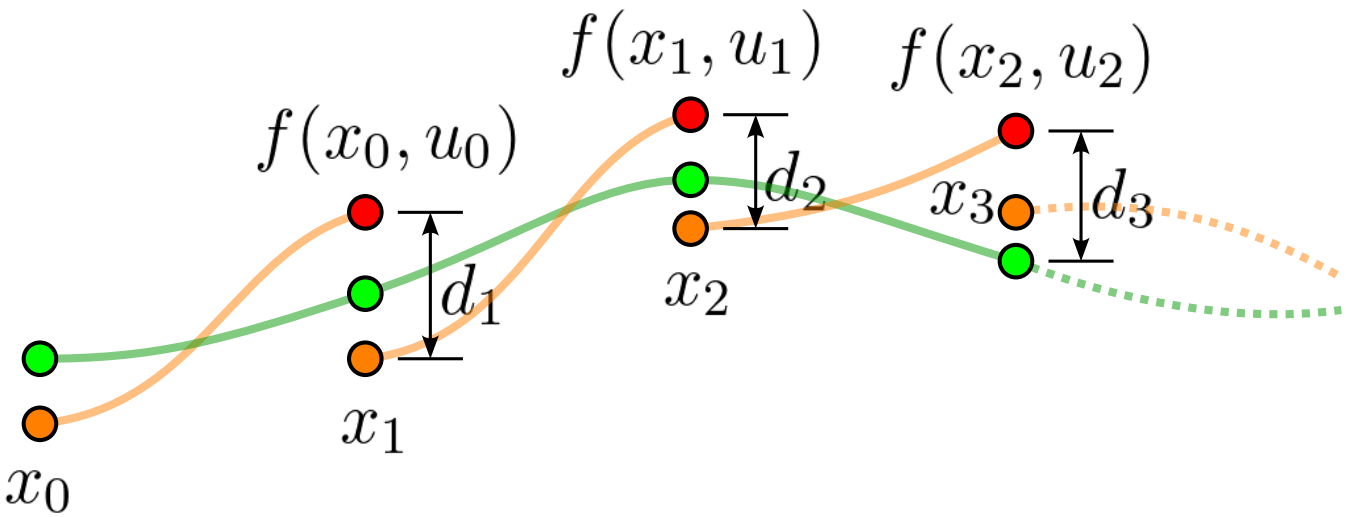
- **multiple shooting** formulation

$$\min_{\substack{u_0,u_1,\\ x_0,x_1,x_2}} \left(u_0^2 + u_1^2 + x_0^2 + x_1^2 + x_2^2 \right)$$

$$\begin{aligned} \text{s. t.} \quad & x_0 = \bar{x}_0 \\ & x_1 = f(x_0, u_0) \\ & x_2 = f(x_1, u_1) \end{aligned}$$

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