Underactuated Robots
Lecture 2: Trajectory Optimization

Nicola Scianca

December 2024

why optimization

- with complex systems, such as underactuated robots with many degrees of freedom, we often can't explicitly write a control law $u = f(x)$ as a closed-form expression

- what we can do instead is to write an **optimization problem** that encodes our control objectives, and its solution will implicitly define our control law

- these lectures will be mostly focused on optimization-based techniques, which we collectively call **trajectory optimization**

introduction to optimization

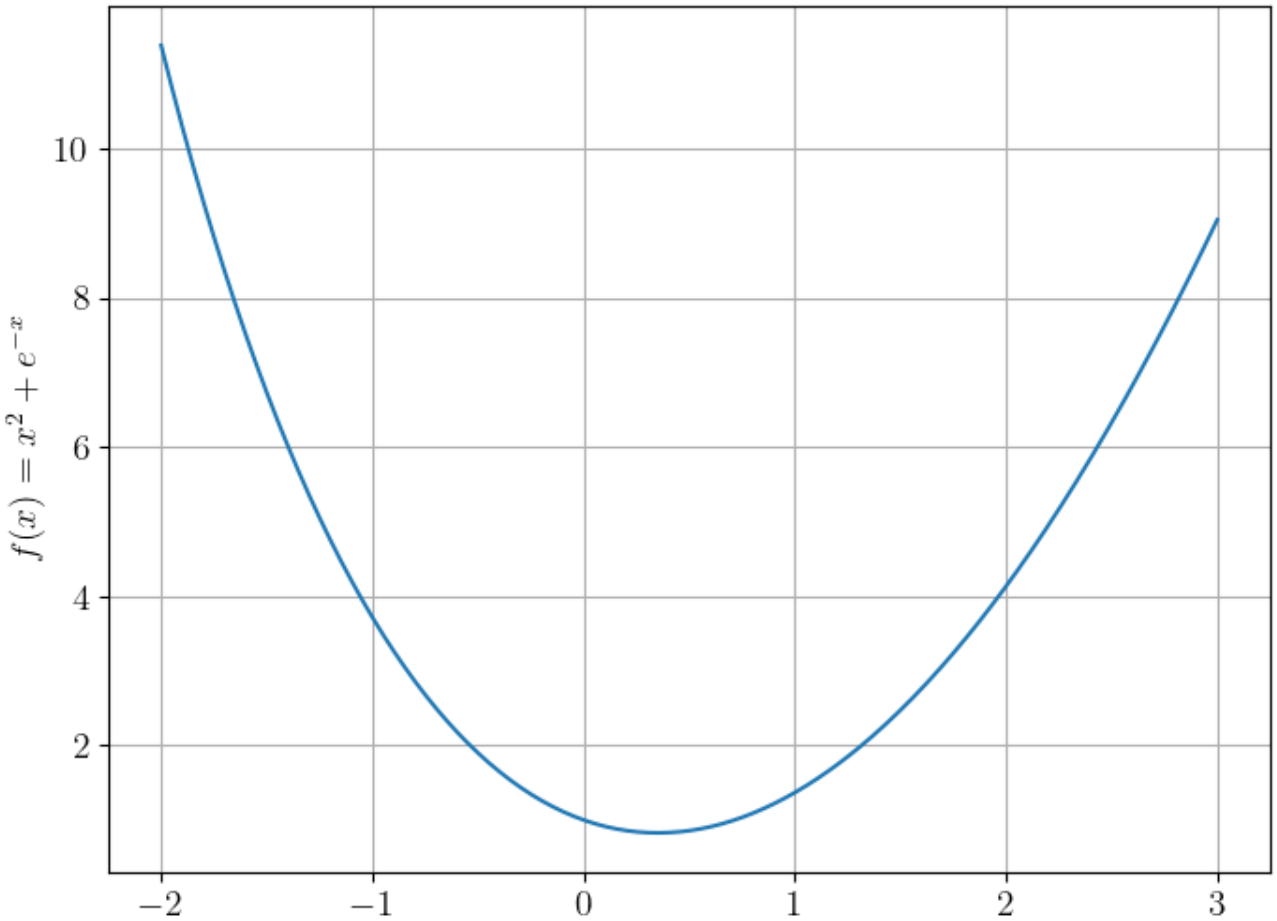
- optimization is the process of finding the **minimum** (or the maximum) of some **function**, possibly in the presence of **constraints**

- key concepts:
  - constrained / unconstrained optimization
  - convexity
  - gradient descent
  - newton's method

unconstrained optimization

- let's say we want to find the minimum of the function

$$f(x) = x^2 + e^{-x}$$



gradient descent

- we learned in calculus class that we can compute the derivative and set it equal to zero

$$\frac{df}{dx} = 2x - e^{-x} = 0 \quad \Longrightarrow \quad 2x = e^{-x}$$

  but solving this equation is not straightforward: we must zero it **numerically**

- the basic idea of **gradient descent**: pick some point and check the derivative
  - if **positive**, the function decreases to the left
  - if **negative**, the function decreases to the right

- if we just keep going in the direction in which the function **decreases**, we should, at some point, find a (local) minimum

gradient descent

- in gradient descent we are computing a **linear** approximation of the function around our guess $\bar{x}$

$$f(\bar{x} + \Delta x) = f(\bar{x}) + \nabla f|_{\bar{x}} \cdot \Delta x$$

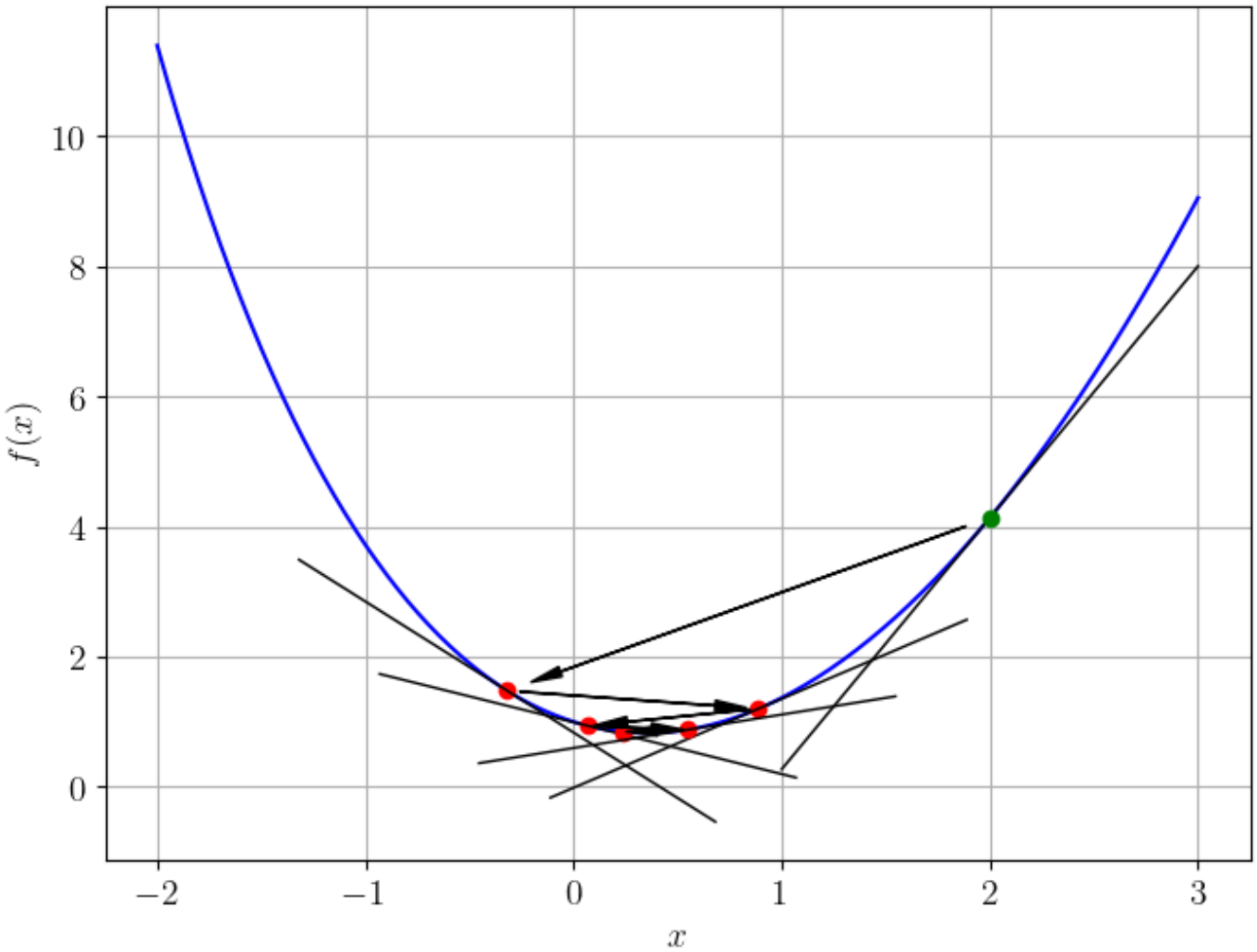
- then we take a step in the direction in which this linear approximation decreases, which means, the negative of its gradient

$$\Delta x = -\alpha \nabla f|_{\bar{x}}$$

- α is the **step size**

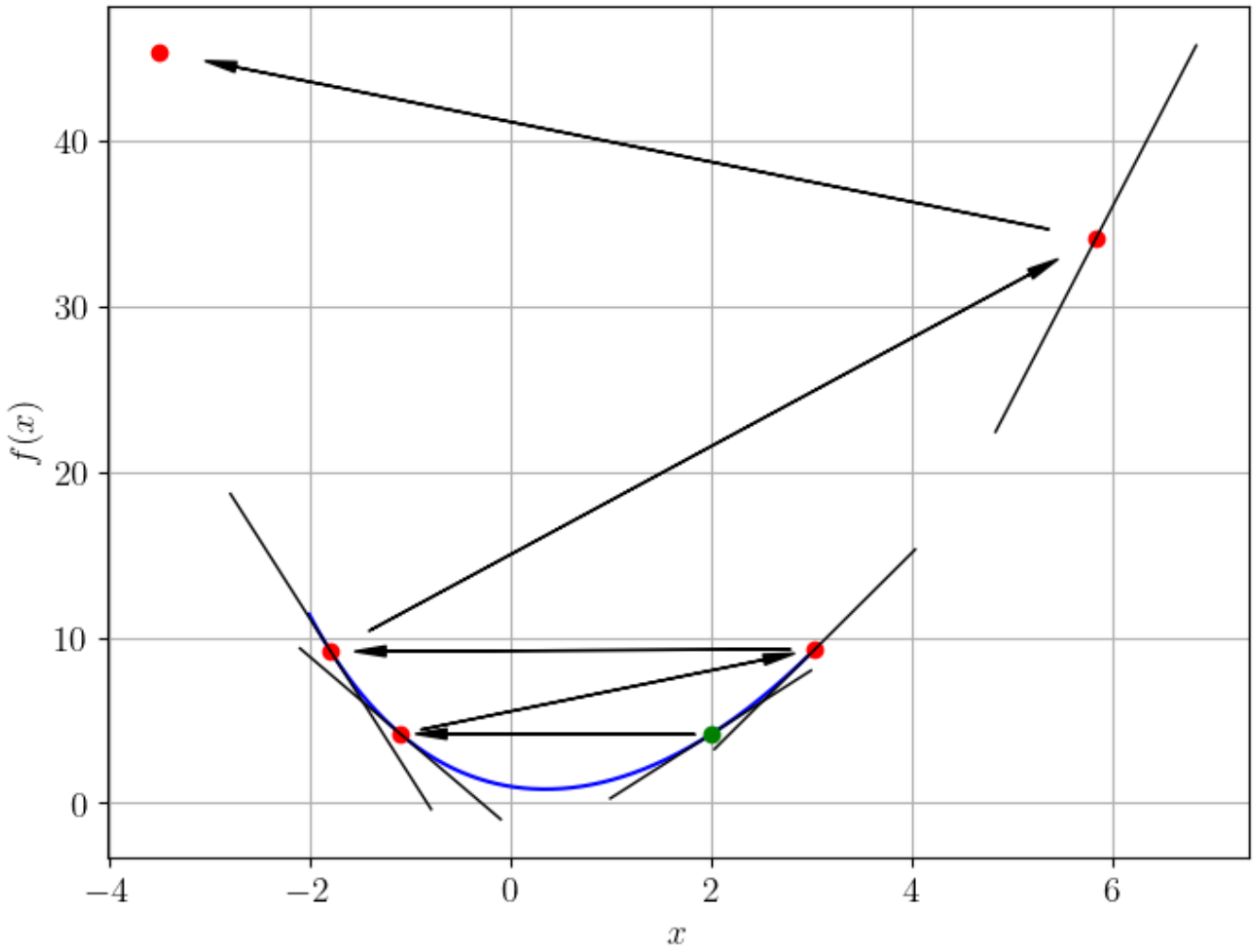
gradient descent

- for our example function, if we start from the point $x = 2$ and iterate with step size $\alpha = 0.6$, we converge quickly



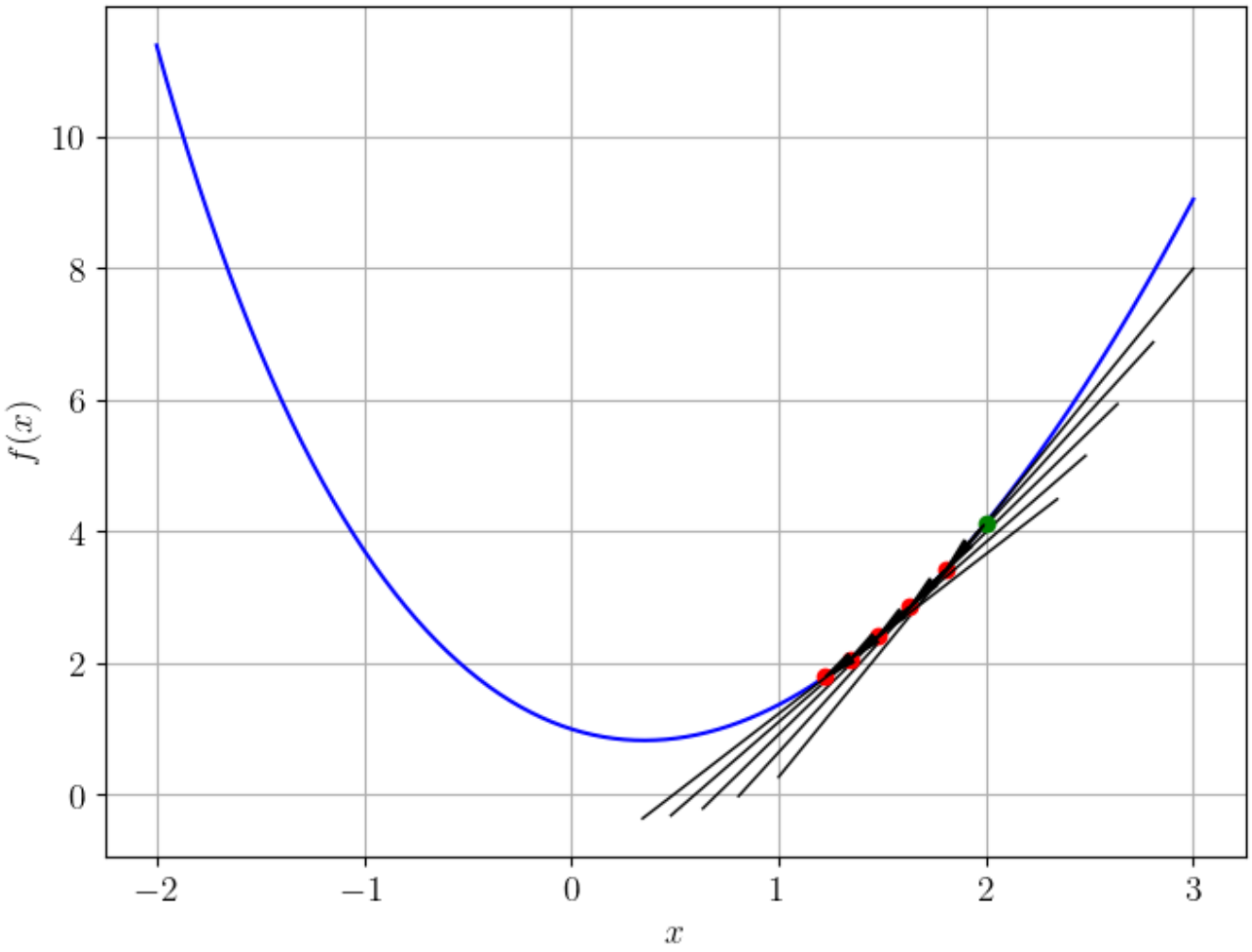
gradient descent

- with just a slightly larger step size ($\alpha = 0.8$) we actually diverge from the minimum!



gradient descent

- if we play it safe and choose a small step size ($\alpha = 0.05$) then the convergence can be very slow!



line search

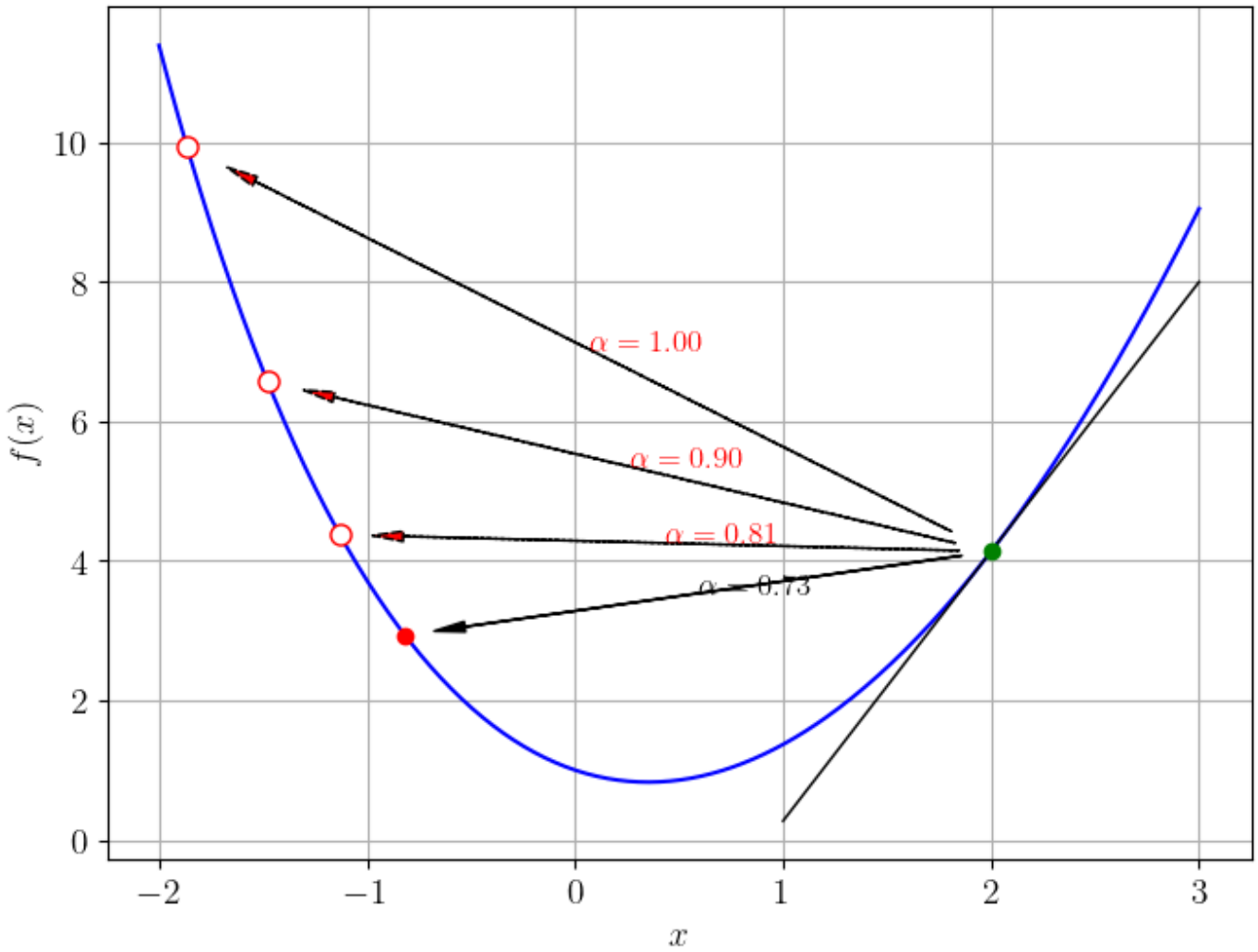
- one way to avoid these problems is to do a **line search**
- the basic idea is that we want to move **along the descent direction** (the negative gradient), but we want to choose the step size small enough to have the function decrease
- the simplest way: starting from an initial step size, say $\alpha = 1$ we check if our function decreases with this step size

$$f(\bar{x} - \alpha \nabla f|_{\bar{x}}) \leq f(\bar{x})$$

- if it does, then we **accept** the step, if it doesn't, we reduce α and try again

gradient descent

- in this example we reduce α by a factor of 0.9 each time (usually it is reduced by 0.5, this is just for illustration)



Newton's method

- Netwon's method is another technique for numerically finding the minimum of a function
- instead of just looking at the descent direction, at each iteration we approximate the function $f(x)$ as a **quadratic function**, then jump to the minimum of the approximation
- it uses more information (the second derivative), but usually performs much better

Newton's method

- first we have to compute both the first and second derivative

$$\frac{df}{dx} = 2x - e^{-x}, \quad \frac{d^2f}{dx^2} = 2 + e^{-x}$$

- this gives us a **quadratic** approximation of our function (second order Taylor expansion)

$$f(\bar{x} + \Delta x) = f(\bar{x}) + \left.\frac{df}{dx}\right|_{\bar{x}} \Delta x + \frac{1}{2} \left.\frac{d^2f}{dx^2}\right|_{\bar{x}} \Delta x^2$$

- the minimum of this approximation is easy to find

$$\left.\frac{df}{dx}\right|_{\bar{x}} + \left.\frac{d^2f}{dx^2}\right|_{\bar{x}} \Delta x = 0 \quad \Longrightarrow \quad \Delta x = -\left(\left.\frac{d^2f}{dx^2}\right|_{\bar{x}}\right)^{-1} \left.\frac{df}{dx}\right|_{\bar{x}}$$

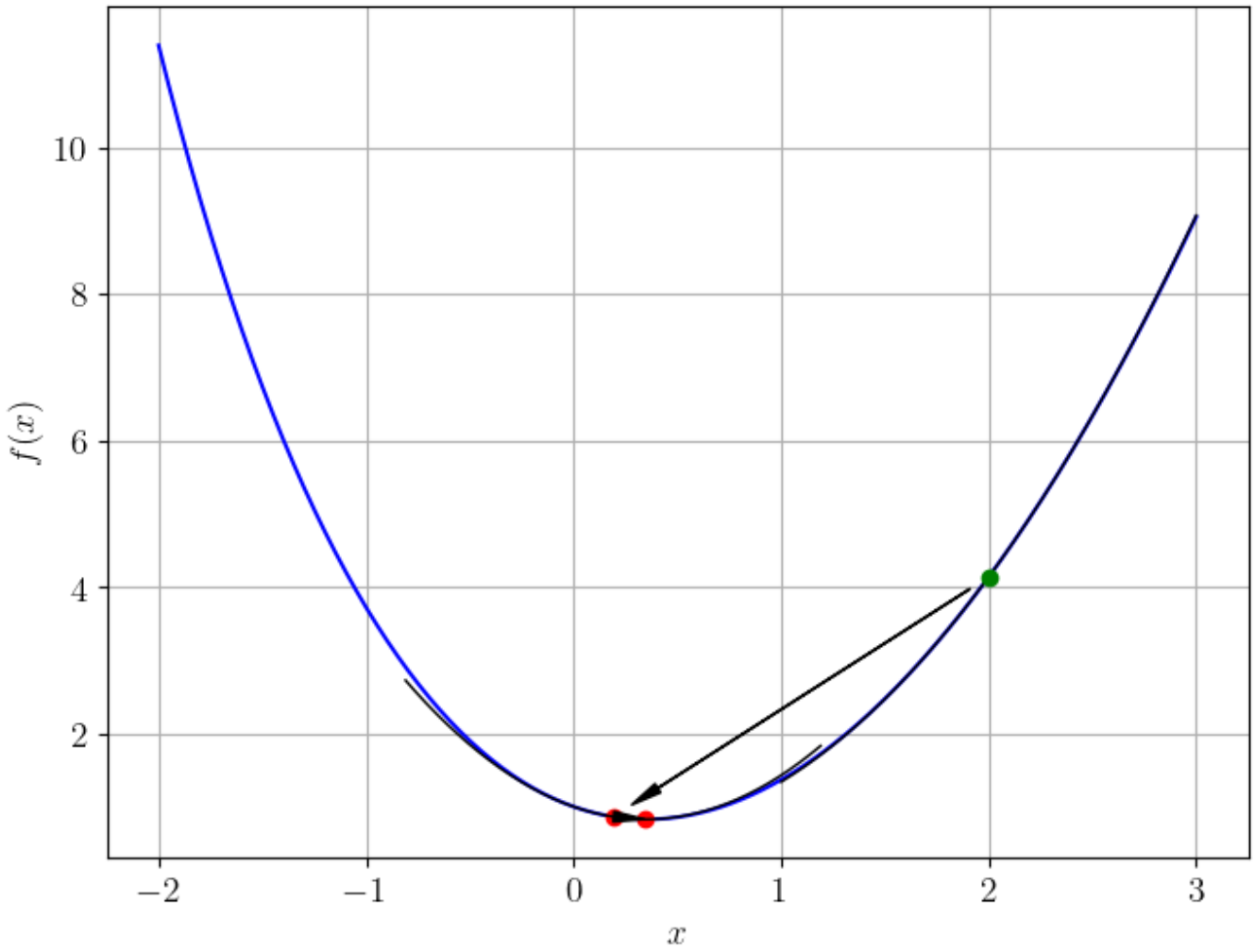
Newton's method

$$\Delta x = -\left(\nabla^2 f|_{\bar{x}}\right)^{-1} \nabla f|_{\bar{x}}$$

- note that, if $\left(\nabla^2 f|_{\bar{x}}\right)^{-1}$ is positive, Newton's method also follows the direction of the negative gradient, but it does so in a smarter way
- however, if $\left(\nabla^2 f|_{\bar{x}}\right)^{-1}$ is negative, we actually go up the gradient, and we might end up in a maximum!
- this should make sense because it means that our quadratic approximation is upside down, so Newton's method actually takes us to the maximum

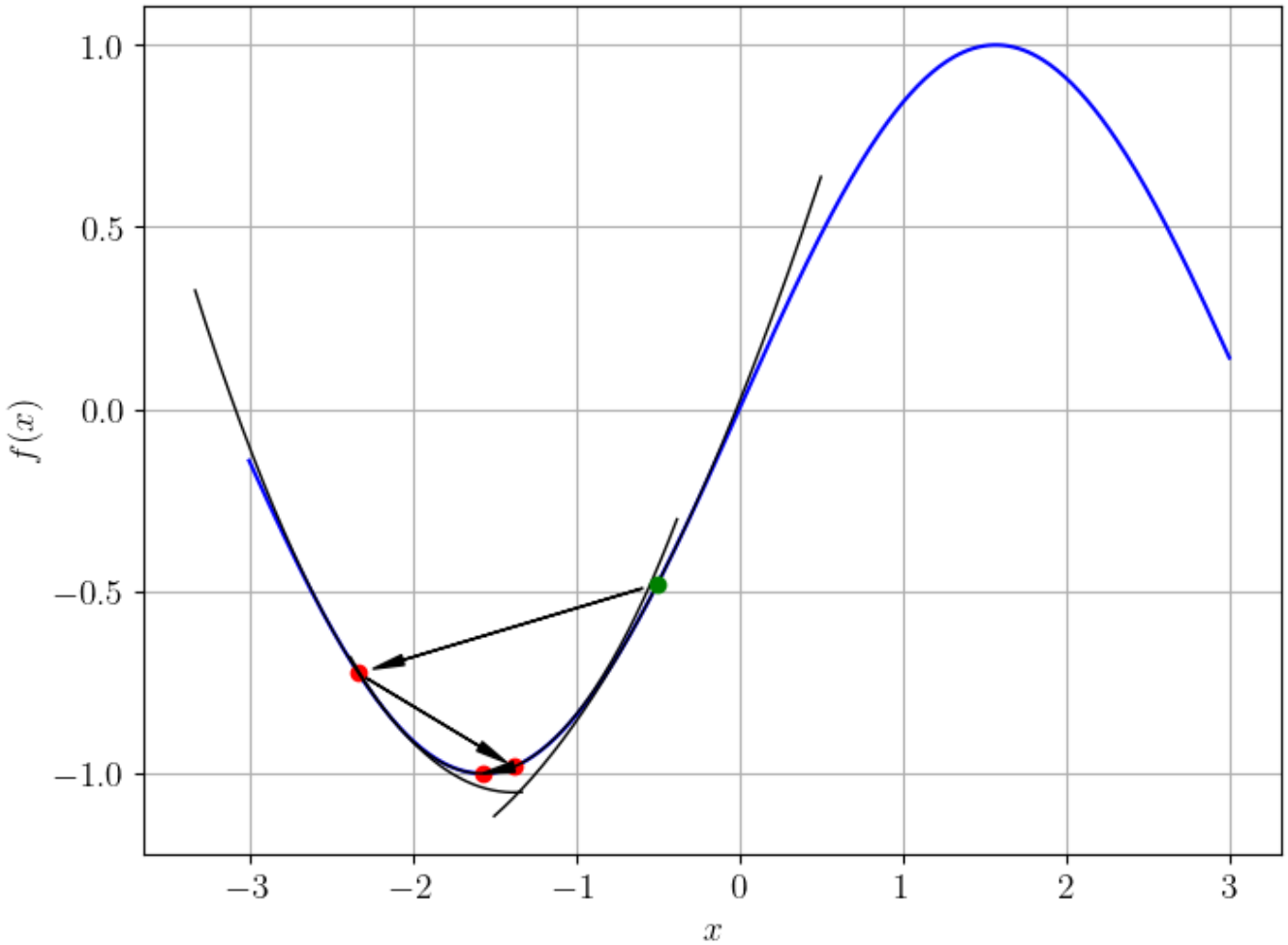
Newton's method

- in our example, Newton's method converges very fast, and we don't even have to choose a step size!



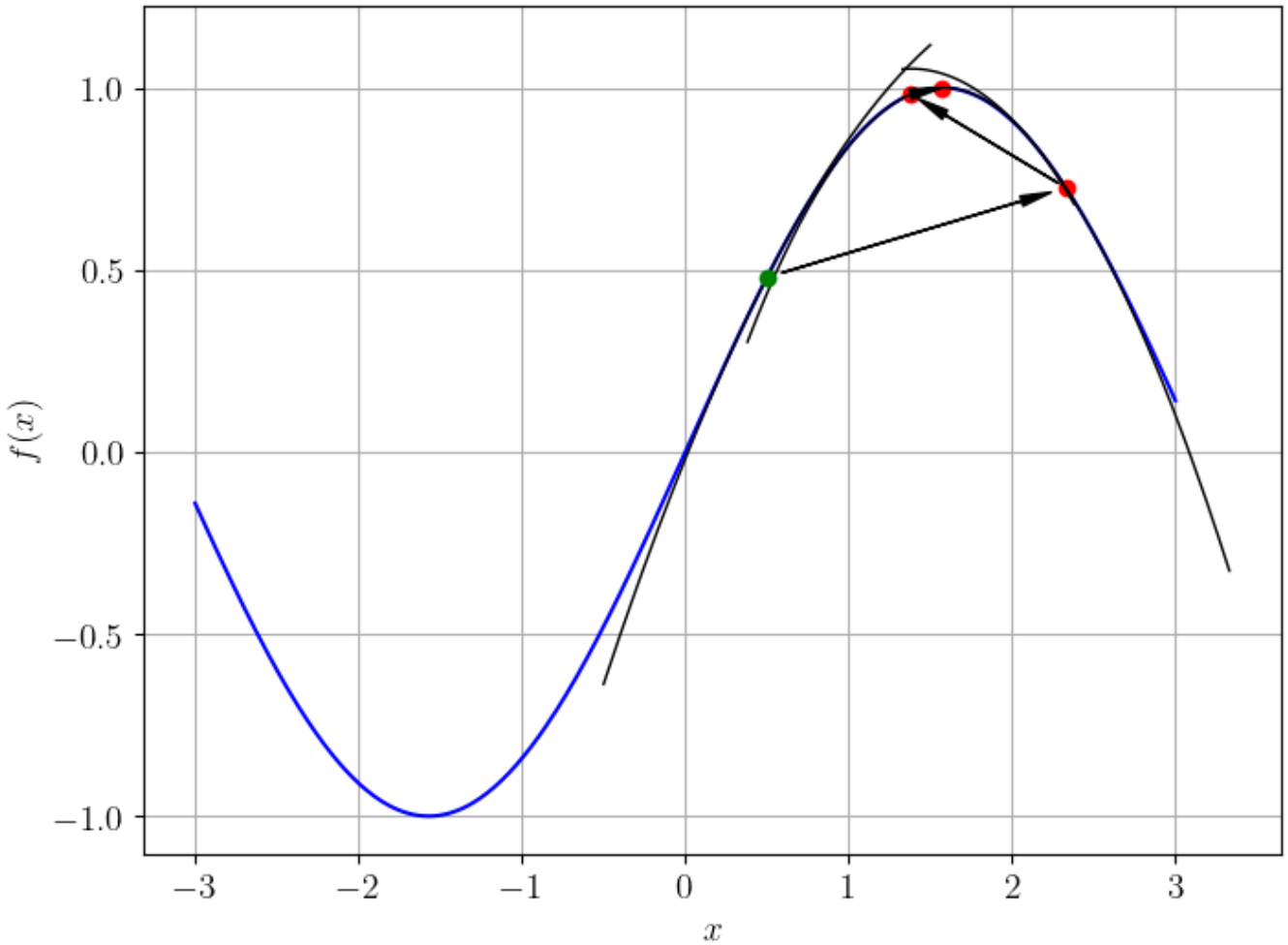
Newton's method

- however, look at what happens if we have a **non-convex** function: if we start close to the minimum, everything is fine



Newton's method

- but if we start at a point where the function is not **convex**, we actually end up in a maximum!



convexity

- **convexity**: if we pick any two points x_a and x_b, and draw a line in between them, the function $f(x)$ is **below** this line

$$\forall x_a, x_b \in \mathbb{R}, \alpha \in (0,1)$$
$$f(\alpha x_a + (1-\alpha)x_b) < \alpha f(x_a) + (1-\alpha) f(x_b)$$

- convex functions have a **single minimum**, thus if we find a solution we can always be sure that it's the **global optimum**

- also, convex functions have a Hessian $\nabla^2 f$ that is always positive, which means that we never run into trouble with Newton's method (if not: regularization, Gauss-Newton approximation, etc...)

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

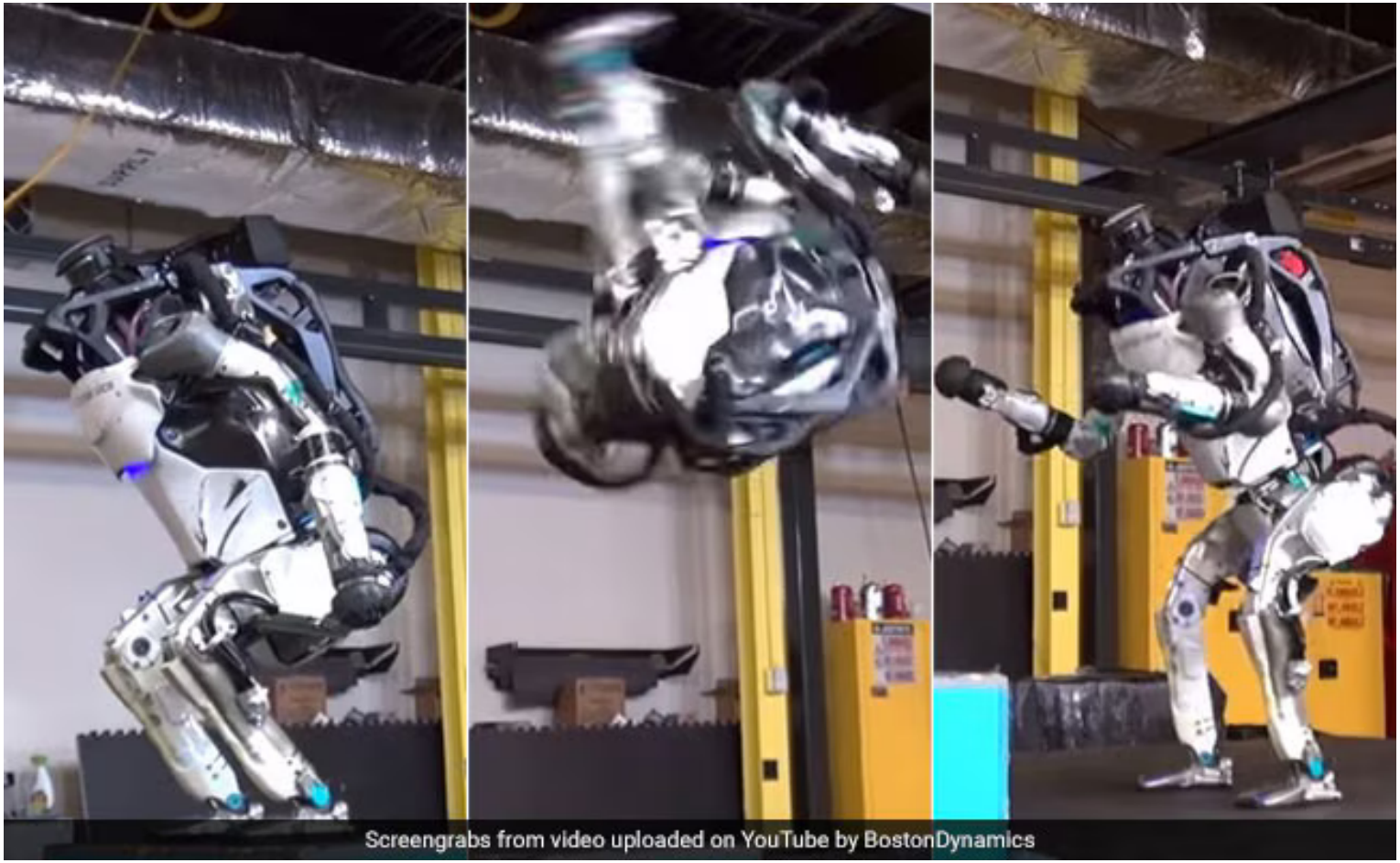
trajectory optimization: examples

- 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



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

trajectory optimization

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

$$
\begin{aligned}
&\min \int_{t_0}^{t_f} l(x(t), u(t), t)dt + l_t(x(t_f)) \\
&\text{subject to} \\
&\dot{x} = f(x(t), u(t)), \quad \text{for} \quad t \in [t_0, t_f], \\
&x(t_0) = \bar{x}_0, \\
&h(x(t), u(t)) = 0, \quad \text{for} \quad t \in [t_0, t_f], \\
&g(x(t), u(t)) \leq 0, \quad \text{for} \quad t \in [t_0, t_f],
\end{aligned}
$$

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

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

- 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

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

- 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

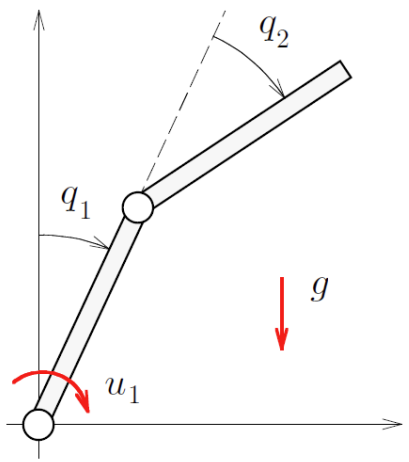
- if we can't explicitly write out a feedback law $u(x)$, maybe we can still solve the optimization problem, starting from one initial state $\bar{x}_0$, and find a particular trajectory as solution

- 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

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

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

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

example

- the dynamics of the pendubot can be written as

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

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

$$\dot{x} = \begin{pmatrix} \dot{q} \\ M(q)^{-1} \left(\begin{pmatrix} u \\ 0 \end{pmatrix} - n(q, \dot{q}) \right) \end{pmatrix} = f(x, u)$$

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

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

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

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

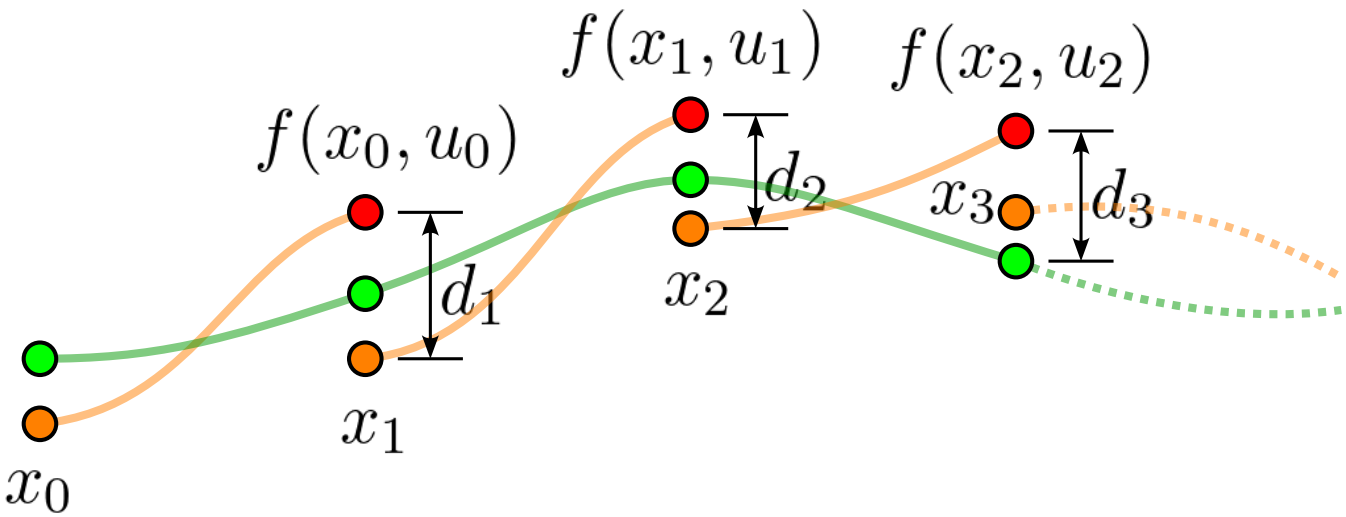
- **multiple shooting** formulation

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

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

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