Underactuated Robots
Lecture 3: Dense QP and DDP

Nicola Scianca

December 2024

solving the optimization problem

- we can control a system using the solution to an **optimization** problem
- the question now is: how do we solve this optimization problem?
- a straightforward way is to formulate a **dense quadratic program**: easy but usually inefficient
- a solution that better exploits the structure of the problem: **differential dynamic programming**

dense QP

- consider a linear system

$$x_{k+1} = Ax_k + Bu_k$$

- as an example, let's write a cost function to **regulate** the state x to the origin

$$\begin{aligned} \min \quad & \sum_{i=0}^{k+N-1} \left(x_i^T Q x_i + u_i^T R u_i \right) + x_N^T Q x_N \\ \text{s.t.} \quad & x_{k+1} = Ax_k + Bu_k \\ & x_0 = \bar{x} \end{aligned}$$

- for the moment, we have no constraints, aside from the linear dynamics and the initial state

dense QP

- in this problem we have no constraints, aside from the linear dynamics and the initial state
- this means that if we adopt a **single shooting** approach we can eliminate all the constraints
- to do this, we need to perform a series of substitutions so that the **dynamic constraints** disappear

dense QP

- the dynamic constraints are

$$x_{k+1} = Ax_k + Bu_k, \quad \text{for } i = 0, \dots, N-1$$

- starting from the initial state x_0, we can write

$$\begin{aligned}
x_1 &= Ax_0 + Bu_0 \\
x_2 &= Ax_1 + Bu_1 = A(Ax_0 + Bu_0) + Bu_1 \\
&= A^2x_0 + ABu_0 + Bu_1 \\
x_3 &= Ax_2 + Bu_2 = A(A^2x_0 + ABu_0 + Bu_1) + Bu_2 \\
&= A^3x_0 + A^2Bu_0 + ABu_1 + Bu_2 \\
&\vdots
\end{aligned}$$

dense QP

- if we keep going until the end of the horizon, we get

$$
\begin{aligned}
x_1 &= Ax_0 + Bu_0 \\
x_2 &= A^2x_0 + ABu_0 + Bu_1 \\
x_3 &= A^3x_0 + A^2Bu_0 + ABu_1 + Bu_2 \\
&\vdots \\
x_N &= A^Nx_0 + A^{N-1}Bu_0 + \cdots + ABu_{N-2} + Bu_{N-1}
\end{aligned}
$$

dense QP

- we can express this as matrix multiplication

$$\underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_N \end{pmatrix}}_{X_{k+1}} = \underbrace{\begin{pmatrix} A \\ A^2 \\ A^3 \\ \vdots \\ A^N \end{pmatrix}}_{\bar{T}} x_0 + \underbrace{\begin{pmatrix} B & 0 & 0 & \dots & 0 \\ AB & B & 0 & \dots & 0 \\ A^2B & AB & B & \dots & 0 \\ \vdots & & & & \\ A^{N-1}B & A^{N-2}B & A^{N-3}B & \dots & B \end{pmatrix}}_{\bar{S}} \underbrace{\begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-1} \end{pmatrix}}_{U_k}$$

- we find an expression for the **predicted state** X_{k+1} in terms of the **current state** x_0 and the **predicted inputs** U_k

$$X_{k+1} = \bar{T}x_0 + \bar{S}U_k$$

dense QP

- recall the cost function that we want to minimize

$$J = \frac{1}{2} \sum_{i=k}^{k+N-1} \left(x_i^T Q x_i + u_i^T R u_i \right) + x_{k+N}^T Q x_{k+N}$$

- this too we can write it in terms of the entire prediction

$$\frac{1}{2} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix}^T}_{X_{k+1}^T} \underbrace{\begin{pmatrix} Q & 0 & \dots & 0 \\ 0 & Q & \dots & 0 \\ \vdots & & & \\ 0 & 0 & \dots & Q \end{pmatrix}}_{\bar{Q}} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix}}_{X_{k+1}} + \frac{1}{2} \underbrace{\begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{pmatrix}^T}_{U_k^T} \underbrace{\begin{pmatrix} R & 0 & \dots & 0 \\ 0 & R & \dots & 0 \\ \vdots & & & \\ 0 & 0 & \dots & R \end{pmatrix}}_{\bar{R}} \underbrace{\begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{pmatrix}}_{U_k}$$

dense QP

- we can eliminate the state from the cost function we just wrote by substituting in the expression for the **prediction**

$$
\begin{aligned}
J &= \frac{1}{2} X_{k+1}^T \bar{Q} X^{k+1} + \frac{1}{2} U_k^T \bar{R} U_k \\
&= \frac{1}{2} (\bar{T} x_k + \bar{S} U_k)^T \bar{Q} (\bar{T} x_k + \bar{S} U_k) + \frac{1}{2} U_k^T \bar{R} U_k \\
&= \frac{1}{2} x_k^T \bar{T}^T \bar{Q} \bar{T} x_k + \frac{1}{2} U_k^T \bar{S}^T \bar{Q} \bar{S} U_k + U_k^T \bar{S}^T \bar{Q} \bar{T} x_k + \frac{1}{2} U_k^T \bar{R} U_k \\
&= \underbrace{\cancel{\frac{1}{2} x_k^T \bar{T}^T \bar{Q} \bar{T} x_k}}_{\text{constant term}} + \frac{1}{2} U_k^T \underbrace{(\bar{S}^T \bar{Q} \bar{S} + \bar{R})}_{H} U_k + U_k^T \underbrace{\bar{S}^T \bar{Q} \bar{T}}_{F} x_k
\end{aligned}
$$

- we can eliminate the linear term as it only changes the value of the minimum, not where the minimum is

dense QP

- we managed to express the problem as

$$\min_{U_k} \frac{1}{2} U_k^T H U_k + U_k^T F x_k$$

- if there are **no constraints**, we can solve this easily by zeroing the gradient

$$\nabla \left(\frac{1}{2} U_k^T H U_k + U_k^T F x_k \right) = 0 \quad \Longrightarrow \quad H U_k + F x_k = 0$$

$$U_k = -H^{-1} F x_k$$

dense QP

- since we want to only apply the first input, let's use a matrix $I_{\text{sel}} = (I, 0, 0, \dots)$ to select it

$$u_k = -I_{\text{sel}} H^{-1} F x_k$$

- after having applied the input u_k, we measure the new state, and then repeat the process once again

- as it turns out, with **no constraints**, linear MPC is a form of **linear state feedback**

dense QP

- we can also write a multiple shooting problem as a dense QP
- in this case, instead of performing substitutions, we keep all the state variables inside the problem
- let's call W_k the vector of decision variables, which now includes inputs and states

$$W_k = (x_k, u_k, x_{k+1}, u_{k+1}, \dots, x_{k+N-1}, u_{k+N-1}, x_{k+N})^T$$

dense QP

- we can write the **dynamics constraint** and the **initial state constraint** on the vector of decision variables as

$$\begin{pmatrix} I & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ -A & -B & I & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & -A & -B & I & \dots & 0 & 0 & 0 \\ \vdots & & & & & & & & \\ 0 & 0 & 0 & 0 & 0 & \dots & -A & -B & I \end{pmatrix} \begin{pmatrix} x_k \\ u_k \\ x_{k+1} \\ u_{k+1} \\ \vdots \\ x_{k+N-1} \\ u_{k+N-1} \\ x_{k+N} \end{pmatrix} = \begin{pmatrix} \bar{x} \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

- this is a very **sparse** constraint: its matrix has lots of zeros

dense QP

- the cost function is very easy to write

$$J = \frac{1}{2} \begin{pmatrix} x_k \\ u_k \\ x_{k+1} \\ u_{k+1} \\ \vdots \\ x_{k+N-1} \\ u_{k+N-1} \\ x_{k+N} \end{pmatrix}^T \begin{pmatrix} Q & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & R & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & Q & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & R & \dots & 0 & 0 & 0 \\ \vdots & & & & & & & \\ 0 & 0 & 0 & 0 & \dots & Q & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & R & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & Q \end{pmatrix} \begin{pmatrix} x_k \\ u_k \\ x_{k+1} \\ u_{k+1} \\ \vdots \\ x_{k+N-1} \\ u_{k+N-1} \\ x_{k+N} \end{pmatrix}$$

- we can't solve the multiple shooting formulation with a simple matrix inverse
- we typically use a solver (e.g., OSQP, HPIPM, PROXQP, ...)

dense QP

- these solutions require, in some way or the other, to **invert** a **large matrix** (unless the solver is smart and exploits the sparsity of the problem)
- this can be done for a few prediction samples, but scales badly if we want to predict many samples
- however, these matrices have a lot of zeros: they are **sparse**
- **dynamic programming** will offer us a solution technique that scales **linearly** with the size of the horizon!

introduction to dynamic programming

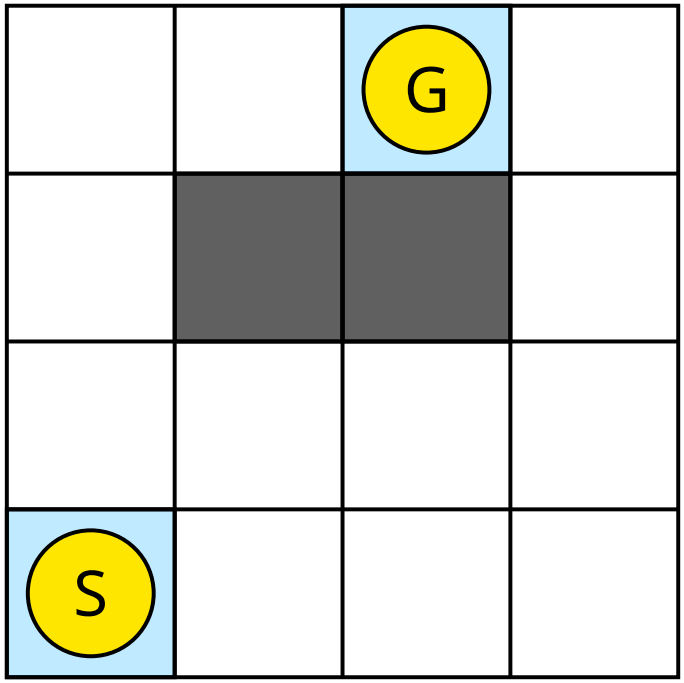
- **dynamic programming** solves complex problems by breaking them into smaller subproblems
- originally developed by Richard Bellman (Bellman, "The Theory of Dynamic Programming", Bulletin of the American Mathematical Society, 1954)
- has broad applications in fields like robotics, reinforcement learning, economics, etc...

foundational concepts

- the fundamental idea of dynamic programming is that some optimization problems can be broken down into a sequence of **recursive sub-problems**
- finding the optimal solution to the original problem comes down to finding the optimal solution to each sub-problem
- this idea is called Bellman's **principle of optimality** and it's the underpinning of dynamic programming methods

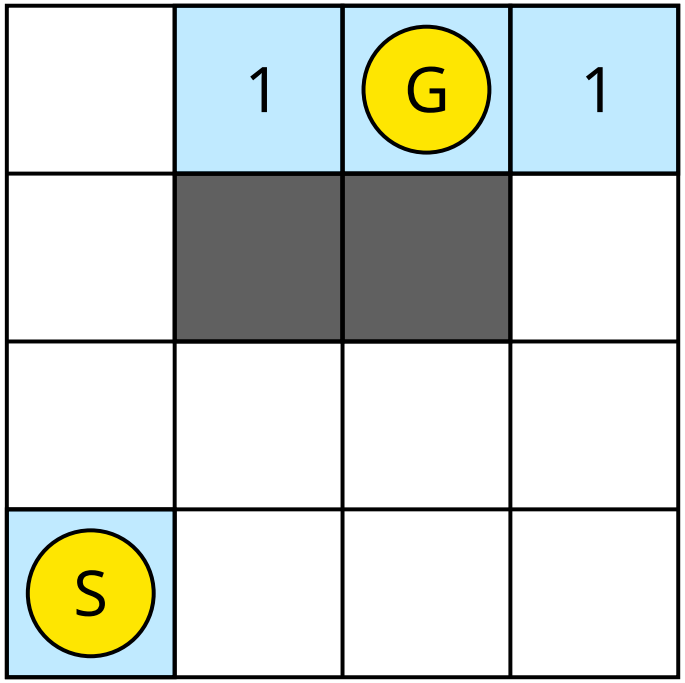
grid world example

- we have a simple **grid world** environment, in which an agent must start from the goal S and reach the goal G
- the agent can move in 4 directions ($\uparrow$, $\rightarrow$, $\downarrow$, $\leftarrow$) and each action costs 1; black squares represent walls



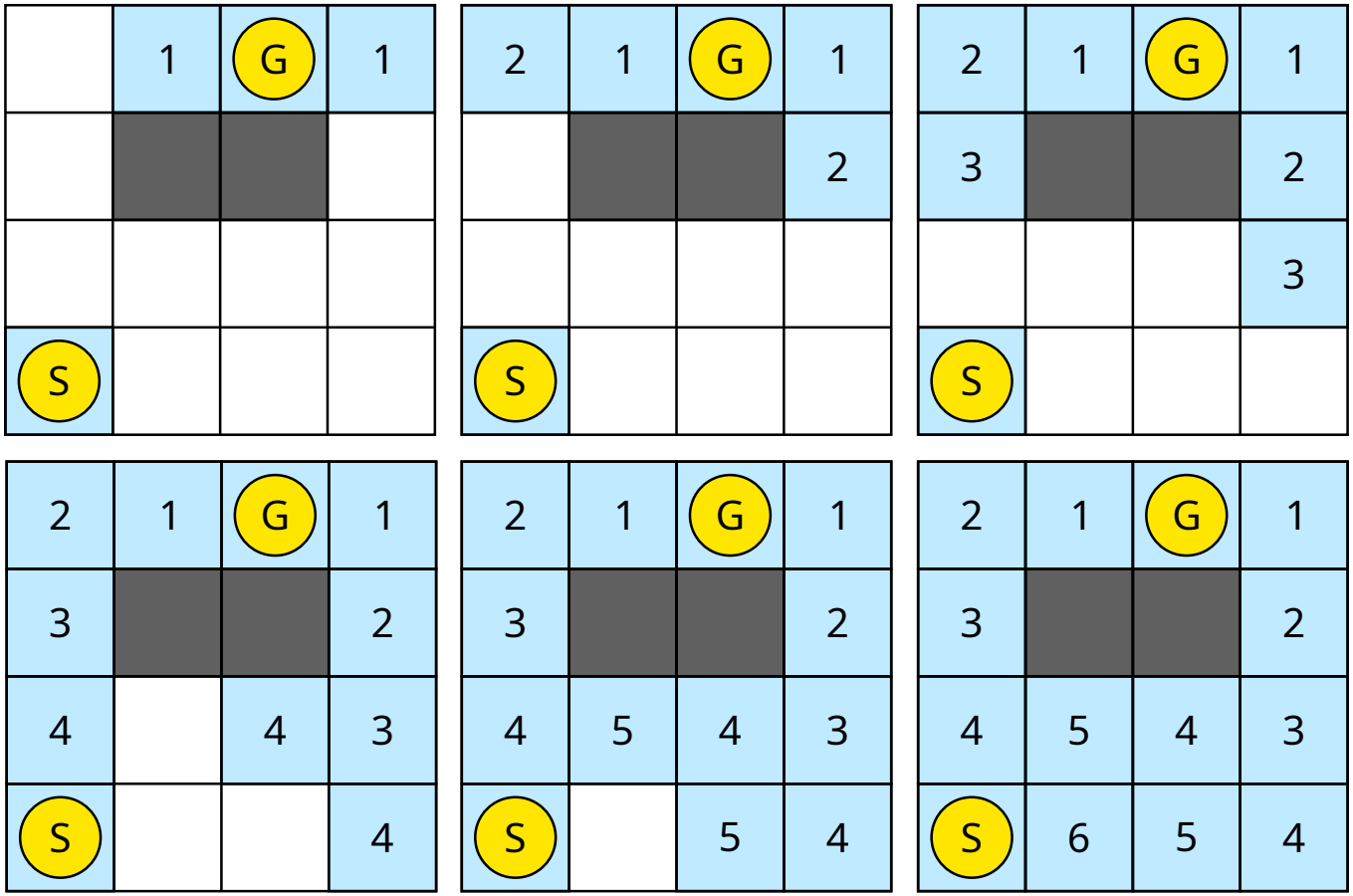
grid world example

- if we are next to the goal, all we need to do is take a step in the direction of the goal, which costs 1
- let's take not of this by writing 1 inside the squares adjacent to the goal



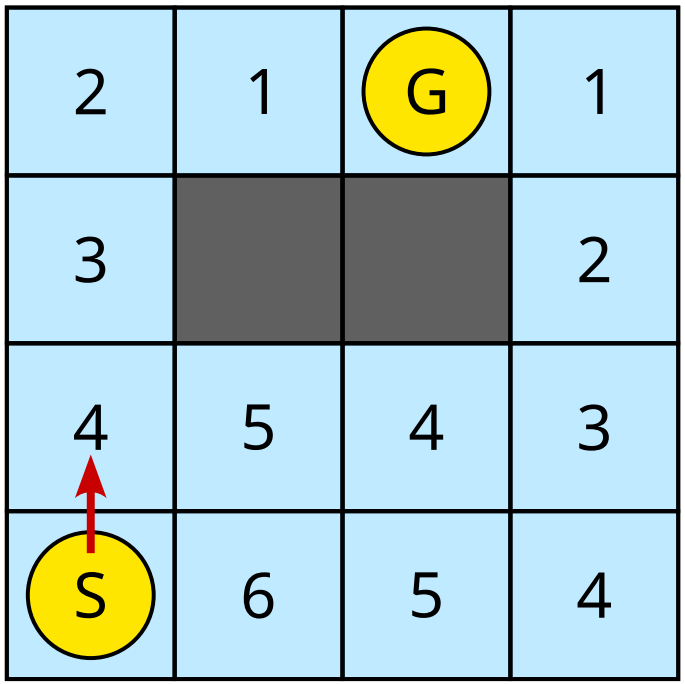
grid world example

- next to a 1 we can write 2, because we know it takes us one more action to get to the 1, and so on
- by repeating this, we can populate the entire grid world



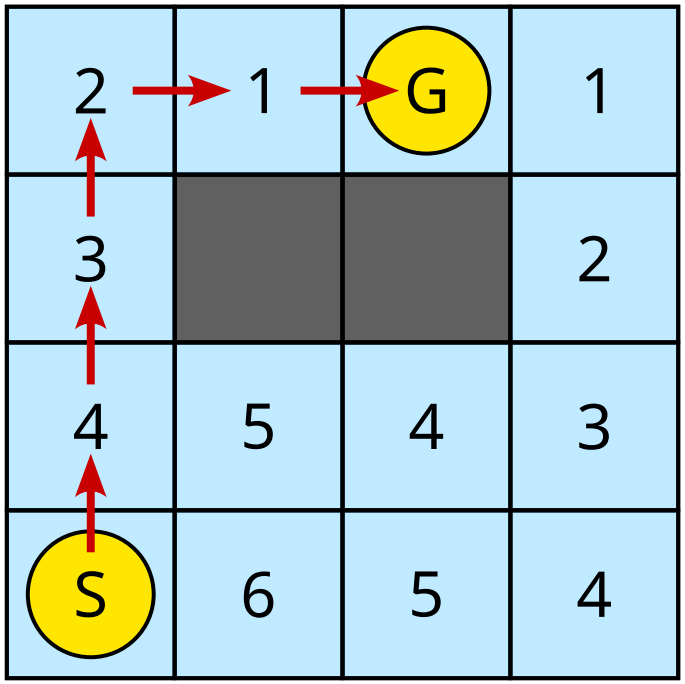
grid world example

- these numbers represent how many actions it takes us (at least), from a given square, to reach the goal
- when we are at the start, it doesn't make sense to go right $\rightarrow$, because then we need to take take at least 6 more actions; the best thing to do is to go up $\uparrow$



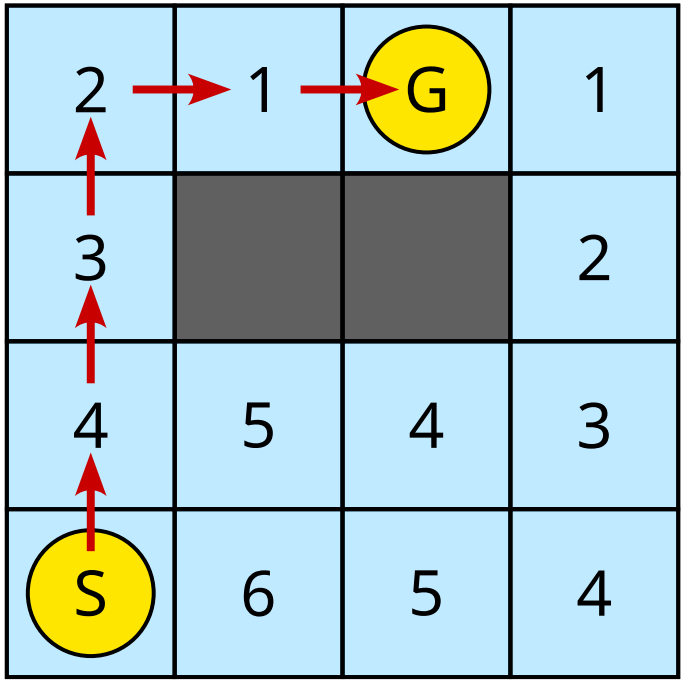
grid world example

- we can repeat this reasoning and find an optimal path from the start to the goal
- all we have to do, is to always go in the direction where we find the smallest number



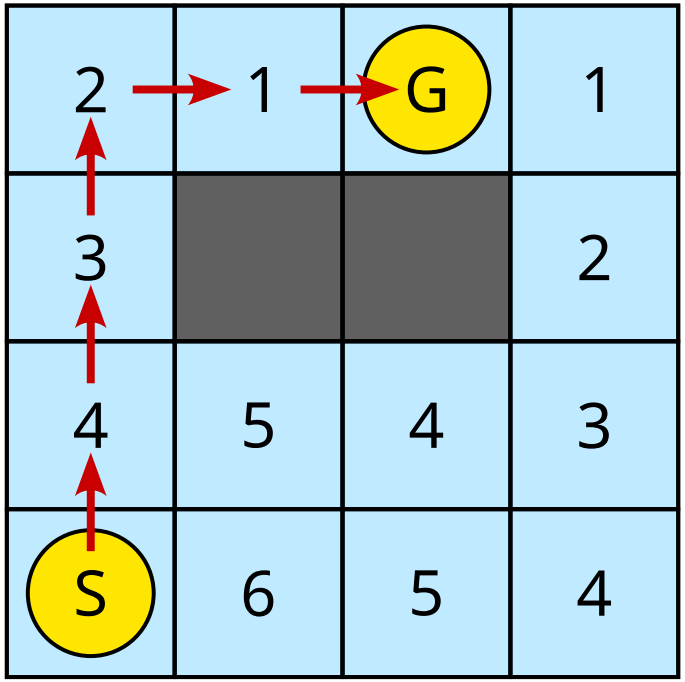
grid world example

- the numbers we wrote in the grids represent the minimum cost we have to pay, from a given state, to reach the goal: the **cost-to-go**
- in maximization problems it represents an obtainable value, which is why it's often called the **value function**



grid world example

- this example is simple practical application of dynamic programming
- we have moved the complexity of solving an optimization problem on the entire path, to a simpler problem in which we just have to find the neighbor with the smallest cost-to-go



the value function

- let's now formulate a more general optimization problem for a dynamic system

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

- the cost function is composed of a sum of **stage costs** $l_k(x_i, u_i)$ and a **terminal cost** $l_N(x_N)$
- constraint enforce the **initial state** and the **system dynamics**

the value function

$$V_k(x_k) = \min_{\substack{x_{k+1},\dots,x_N \\ u_k,\dots,u_{N-1}}} \left(\sum_{i=k}^{N-1} l_k(x_i, u_i) + l_N(x_N) \right)$$
$$\text{s. t.} \quad x_{k+1} = f(x_k, u_k)$$

- the **value function** $V_k(x_k)$ tells us the cost we will pay if we always take optimal actions in the future
- the cost is a reward in maximization problems, hence the name "value function"
- at time k, it depends on the state x_k, but not on the input (we are choosing the best possible input sequence)

the Bellman equation

- now, take out the first item of the sum

$$
\begin{aligned}
V_k(x_k) &= \min_{\substack{x_{k+1},\dots,x_N \\ u_k,\dots,u_{N-1}}} \left(l_k(x_k,u_k) + \sum_{i=k+1}^{N-1} l_k(x_i,u_i) + l_N(x_N) \right) \\
&= \min_{u_k} \left(l_k(x_k,u_k) + \boxed{\min_{\substack{x_{k+2},\dots,x_N \\ u_{k+1},\dots,u_{N-1}}} \left(\sum_{i=k+1}^{N-1} l_k(x_i,u_i) + l_N(x_N) \right)} \right) \\
&= \min_{u_k} \left(l_k(x_k,u_k) + V_{k+1}(x_{k+1}) \right)
\end{aligned}
$$

- since $\boxed{}$ is the value function at $k+1$, we get a nice recursive expression, in which the value function at x_k is expressed in terms of the value function at x_{k+1}

the Bellman equation

- remember: this minimization must take into account the dynamic constraint $x_{k+1} = f(x_k, u_k)$

- if we substitute this inside the recursive expression, we get

$$V(x_k) = \min_{u_k} \left(l_k(x_k, u_k) + V_{k+1}(f(x_k, u_k)) \right)$$

which is the **Bellman equation** in the discrete case

the Bellman equation

$$V(x_k) = \min_{u_k} \left(l_k(x_k, u_k) + V_{k+1}(f(x_k, u_k)) \right)$$

- the Bellman equation relates the value of V_k to the value of V_{k+1} in a recursive way
- if we had perfect knowledge of V, this would make it easy to derive an optimal control law, but usually we cannot explicitly solve for V and we have to approximate
- however, there is at least one case in which we can find an explicit solution: the **Linear Quadratic Regulator** (LQR)

linear quadratic regulator

- in the LQR we are trying to regulate the state x to zero, thus the cost function to minimize looks like

$$J = \sum_{k=0}^{N-1} \left(x_k^T Q x_k + u_k^T R u_k \right) + x_N^T Q_f x_N$$

where $Q \succeq 0$, $R \succ 0$, and $Q_f \succeq 0$ are weight matrices

- the system dynamics equation is linear

$$x_{k+1} = A x_k + B u_k$$

linear quadratic regulator

- the value function at time step k is

$$V_k(x_k) = \min_{u_k,\dots,u_{N-1}} \sum_{i=k}^{N-1} \left(x_i^T Q x_i + u_i^T R u_i\right) + x_N^T Q_f x_N$$

- using the Bellman equation we can write it recursively as

$$V_k(x_k) = \min_{u_k} \left(x_k^T Q x_k + u_k^T R u_k + V_{k+1}(x_{k+1})\right)$$

with system dynamics $x_{k+1} = Ax_k + Bu_k$.

linear quadratic regulator

- let's assume that the value function is **quadratic**

$$V_k(x_k) = x_k^T P_k x_k$$

- the value function at the next time-step $k+1$ can be related to the current x_k and u_k via the system dynamics

$$V_{k+1} = x_{k+1}^T P_{k+1} x_{k+1} = (Ax_k + Bu_k)^T P_{k+1}(Ax_k + Bu_k)$$

- substitute into the Bellman equation:

$$V_k(x_k) = \min_{u_k} \Big(x_k^T Q x_k + u_k^T R u_k + (Ax_k + Bu_k)^T P_{k+1}(Ax_k + Bu_k) \Big)$$

linear quadratic regulator

- to compute the minimum, derive with respect to u_k and set equal to zero

$$\frac{d}{du_k}\left(x_k^T Q x_k + u_k^T R u_k + (Ax_k + Bu_k)^T P_{k+1}(Ax_k + Bu_k)\right)$$
$$= \cancel{2}Ru_k + \cancel{2}B^T P_{k+1}(Ax_k + Bu_k)$$
$$= (R + B^T P_{k+1} B)u_k + B^T P_{k+1} A x_k = 0$$
$$\implies \quad u_k = -(R + B^T P_{k+1} B)^{-1} B^T P_{k+1} A x_k$$

- substituting into the Bellman equation yields the **Riccati recursion**

$$P_k = Q + A^T P_{k+1} A - A^T P_{k+1} B (R + B^T P_{k+1} B)^{-1} B^T P_{k+1} A$$

linear quadratic regulator

- now we start from the final state with $P_N = Q_f$, and going backwards with the Riccati recursion we can compute P_k at each time step

$$P_k = Q + A^T P_{k+1} A - A^T P_{k+1} B (R + B^T P_{k+1} B)^{-1} B^T P_{k+1} A$$

- once we have full knowledge of P_k (which means full knowledge of the value function), the optimal control law is simply given by

$$u_k = -(R + B^T P_{k+1} B)^{-1} B^T P_{k+1} A x_k$$

differential dynamic programming

- if the systems dynamics are nonlinear, we usually can't directly solve for the value function
- if we discretize the state space we can compute V numerically everywhere, but for large systems this is impossible due to the **curse of dimensionality**
- what we can do is find a **quadratic approximation** of the value function: **Differential Dynamic Programming** (DDP)

quadratic approximation of the value function

- let's recall the Bellman equation

$$V(x_k) = \min_{u_k} \left(l_k(x_k, u_k) + V_{k+1}(f(x_k, u_k)) \right)$$

- call Q the argument of the minimization

$$Q(x_k, u_k) = l_k(x_k, u_k) + V_{k+1}(f(x_k, u_k))$$

quadratic approximation of the value function

- a quadratic approximation Q, around the point $(\bar{x}_k, \bar{u}_k)$, would look like

$$Q(x_k, u_k) \simeq Q(\bar{x}_k, \bar{u}_k) + \left.\frac{\partial Q}{\partial x_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta x_k + \left.\frac{\partial Q}{\partial u_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta u_k$$
$$+ \frac{1}{2}\Delta x_k^T \left.\frac{\partial^2 Q}{\partial x_k^2}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta x_k + \frac{1}{2}\Delta u_k^T \left.\frac{\partial^2 Q}{\partial u_k^2}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta u_k$$
$$+ \Delta u_k^T \left.\frac{\partial^2 Q}{\partial x_k \partial u_k}\right|_{\substack{\bar{x}_k \\ \bar{u}_k}} \Delta x_k$$

- let's use a more compact notation

$$\tilde{Q}(x_k, u_k) = \bar{Q}_k + \bar{Q}_k^x \Delta x_k + \bar{Q}_k^u \Delta u_k$$
$$+ \frac{1}{2}\Delta x_k^T \bar{Q}_k^{xx} \Delta x_k + \frac{1}{2}\Delta u_k^T \bar{Q}_k^{uu} \Delta u_k + \Delta u_k^T \bar{Q}_k^{ux} \Delta x_k^T$$

quadratic approximation of the value function

- let's compute, as an example, $\bar{Q}_k^x$

$$\bar{Q}_k^x = \frac{\partial}{\partial x_k}\Big(l_k(x_k, u_k) + V_{k+1}(f(x_k, u_k))\Big) = \frac{\partial l}{\partial x_k}\bigg|_{\substack{\bar{x}_k \\ \bar{u}_k}} + \frac{dV_{k+1}}{dx_k}\bigg|_{\substack{\bar{x}_k \\ \bar{u}_k}}$$

- since V_{k+1} is the value function at $k+1$, it doesn't depend on x_k directly but through the dynamics $f(x_k, u_k)$; therefore, we must use the chain rule

$$\bar{Q}_k^x = \frac{\partial l}{\partial x_k}\bigg|_{\substack{\bar{x}_k \\ \bar{u}_k}} + \frac{dV_{k+1}}{dx_{k+1}}\bigg|_{\substack{\bar{x}_k \\ \bar{u}_k}} \frac{\partial f}{\partial x_k}\bigg|_{\substack{\bar{x}_k \\ \bar{u}_k}} = \bar{l}_k^x + V_{k+1}^x \bar{f}_k^x$$

quadratic approximation of the value function

- similarly we can compute all the other terms

$$\begin{aligned}
\bar{Q}_k^x &= l_k^x + V_{k+1}^x f_k^x \\
\bar{Q}_k^u &= l_k^u + V_{k+1}^x f_k^u \\
\bar{Q}_k^{xx} &= l_k^{xx} + (f_k^x)^T V_{k+1}^{xx} f_k^x + \cancel{V_{k+1}^x f_k^{xx}} \\
\bar{Q}_k^{uu} &= l_k^{uu} + (f_k^u)^T V_{k+1}^{xx} f_k^u + \cancel{V_{k+1}^x f_k^{uu}} \\
\bar{Q}_k^{ux} &= l_k^{ux} + (f_k^u)^T V_{k+1}^{xx} f_k^x + \underbrace{\cancel{V_{k+1}^x f_k^{ux}}}_{\text{second derivative, usually neglected}}
\end{aligned}$$

- the second-order derivatives of the dynamics are often neglected because they take a lot of time to compute (this variant of DDP is sometimes called iLQR or iLQG)

quadratic approximation of the value function

- remember that Q is the argument of the minimization; to find the value function approximation we derive it with respect to Δu_k and set it equal to zero

$$\frac{\partial \tilde{Q}(x_k, u_k)}{\partial \Delta u_k} = \bar{Q}_k^u + \bar{Q}_k^{uu} \Delta u_k + \bar{Q}_k^{ux} \Delta x_k = 0$$

- the optimal Δu_k is therefore

$$\Delta u_k = -(\bar{Q}_k^{uu})^{-1} \left(\bar{Q}_k^u + \bar{Q}_k^{ux} \Delta x_k \right)$$

- this is a **local feedback** control law, which we can write as

$$\Delta u_k = k_k + K_k \Delta x_k \qquad \begin{aligned} k_k &= -(\bar{Q}_k^{uu})^{-1} \bar{Q}_k^u \\ K_k &= -(\bar{Q}_k^{uu})^{-1} \bar{Q}_k^{ux} \end{aligned}$$

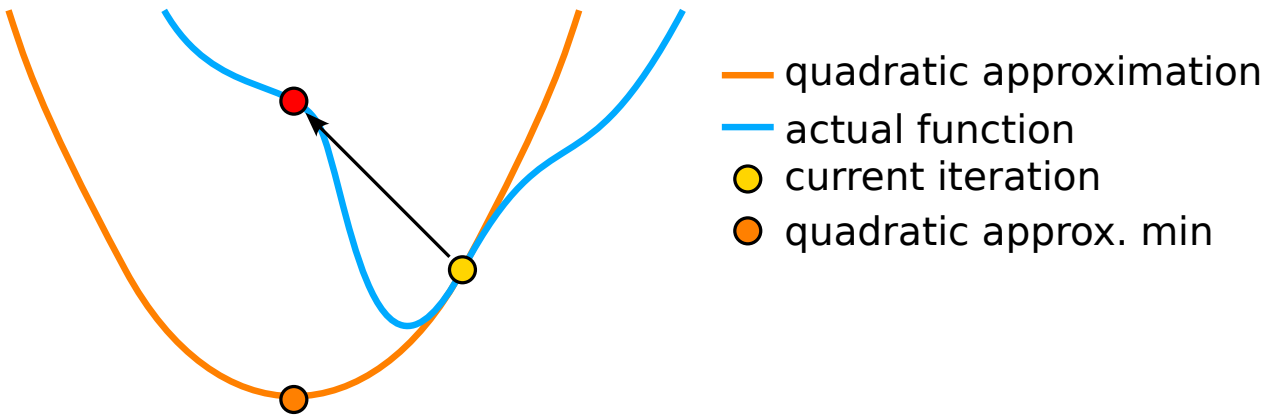
k_k is a feedforward term, and K_k is a local optimal gain

DDP algorithm

- start with an initial guess trajectory $(\bar{x}, \bar{u})$
- backward pass:
  - compute the value function approximation in the terminal state x_N
  - compute the approximation in the previous state x_{N-1}
  - in the process, you also obtain the optimal control law (k_k, K_k)
  - repeat until reaching the start of the horizon
- forward pass:
  - start from the current state $x_0 = \bar{x}_0$
  - compute the new input using the local gains $u_k = \bar{u}_k + k_k + K_k \Delta x_k$
  - propagate the dynamics forward $x_{k+1} = f(x_k, u_k)$
  - repeat until reaching the end of the horizon
- we now have a new guess trajectory $(\bar{x}, \bar{u}) \leftarrow (x, u)$
- repeat backward pass and forward pass until convergence

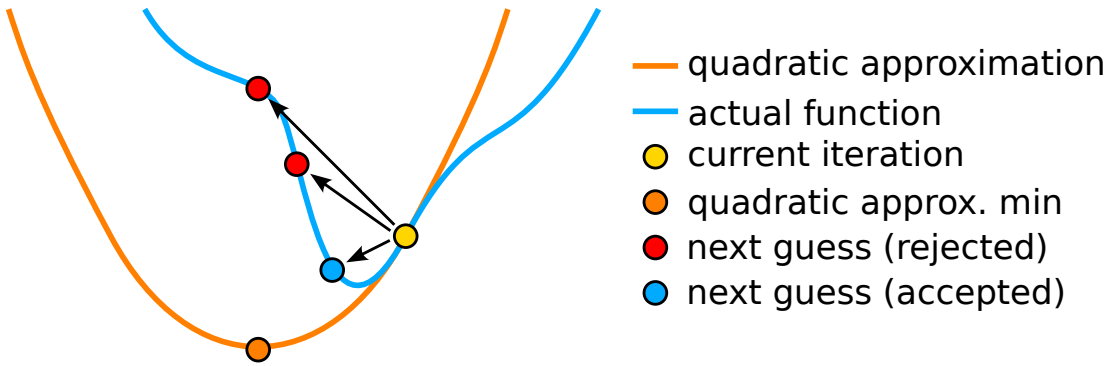
line search

- the local optimal control law $\Delta u_k = k_k + K_k \Delta x_k$ is only valid for a quadratic approximation, not for the true value function
- in reality, the value function is not quadratic, so this adjustment might overshoot the local minimum, and increase the cost of the new guess instead of decreasing it



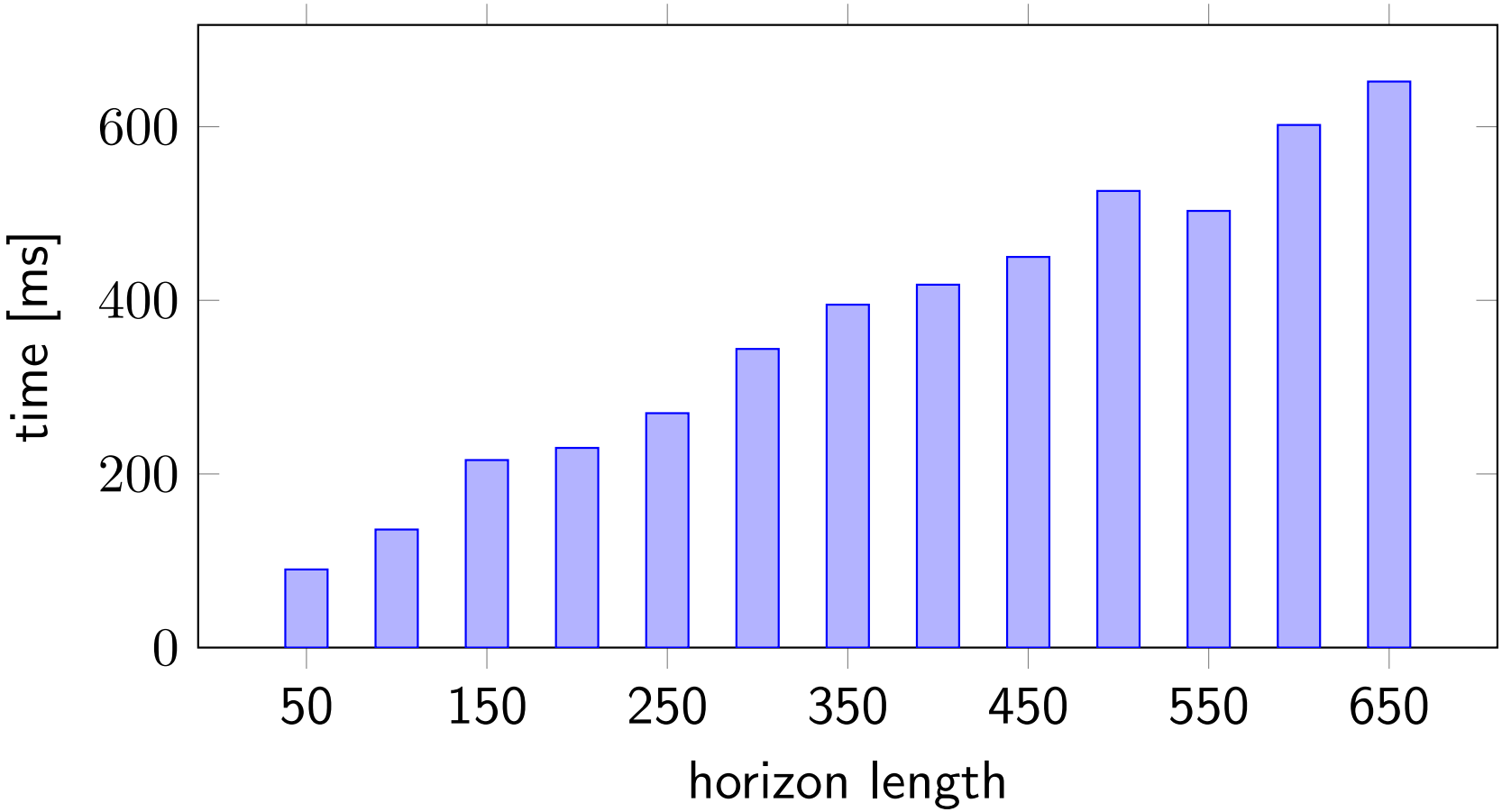
line search

- therefore, we modify the forward pass and perform something similar to a **line search** procedure:
  - start with an $\alpha = 1$
  - propagate the dynamics $x_{k+1} = f(x_k, \bar{u}_k + \alpha k_k + K_k \Delta x_k)$
  - evaluate the cost function over the new trajectory
  - if it increased, choose a smaller α and repeat the forward pass
  - if it decreased, accept the new trajectory (line search finished)



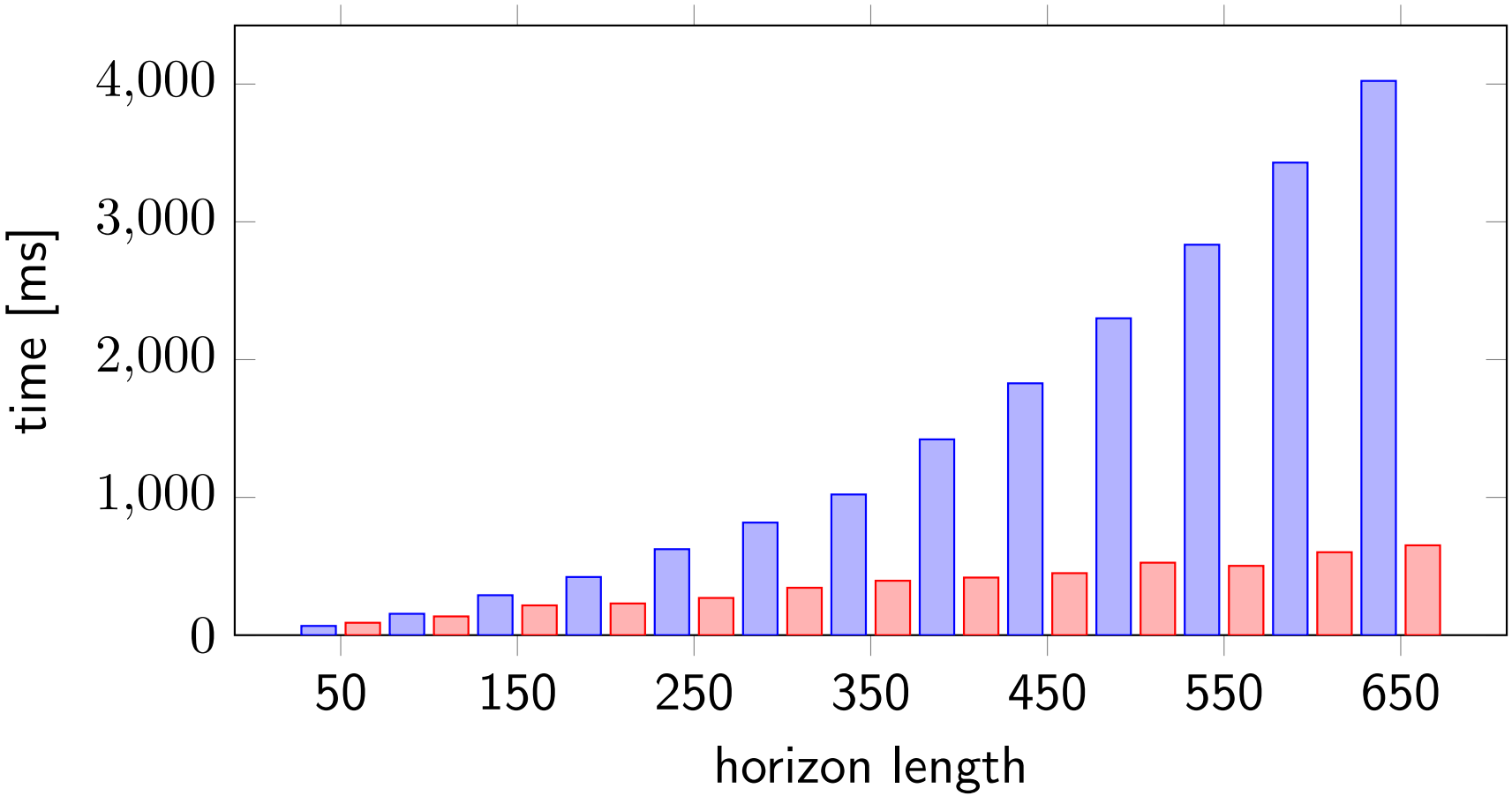
computational complexity

- no large matrix factorization is required! DDP scales **linearly** with the number of variables



computational complexity

- compared with the way SQP scales, we have a clear advantage when operating with a long horizon



pros and cons

- pro: the computations are propagated forward and backward along the horizon: if you double the horizon length you just double the number of computations

- this means that the computational cost is **linear** with the horizon length

- by contrast, the complexity of solving a dense QP depends on the solver used (we don't know it apriori)

pros and cons

- con: it does not directly allow to set **constraints**
- con: it is a **single shooting** method, which means that when integrating the initial input guess you could get a diverging trajectory
- pro: there are variants of DDP that use an initial guess on the state, incorporating defects in a way that is similar to **multiple shooting** [Mastalli et al., 2020]

Mastalli et al., "Crocoddyl: An Efficient and Versatile Framework for Multi-Contact Optimal Control", Robotics and Automation Letters 2020