Notes of the Course

Advanced Optimization-Based Robot Control

prof. Del Prete Andrea

andrea.delprete@unitn.it

November 17, 2023

Authors:

Matteo Dalle Vedove (*matteodv99tn@gmail.com*), Nicolo Cavalieri, Andrea Del Prete

Contents

Chapter 1

Robot Dynamic Modeling (review)

1.1 Modeling Robot Manipulators

A set of rigid bodies (called links), connected by joints (see Fig. 1.1). The most common types of joints are:

1. revolute: allow rotation around a fixed axis;
2. prismatic: allow translation along a fixed axis.

Joints are typically actuated by a motor and a transmission system, e.g. gearbox, cable-driven transmission, pulleys. We typically need a transmission system because we do not want to have the motor located at the joint (such as at the wrist joint, which is far from the base and would result in inefficient motions), but we want the motor as close as possible to the robot base. Moreover, the problem with almost all actuators/motors is that they provide high velocities and small torques. On robots we need the opposite: high torques and small velocities: with a gear-box (gear ratio) we are able to reduce the velocity and get a higher torque.

The positions and angles of the n joints of a robot are encoded by the variables $q_i \quad \forall i = 0,..,n-1$. These **joint coordinates** can be collected in an n-dimensional vector q.

1.1.1 Homogeneous Transformations

When working with a robot very often we are not interested in the joint angles, but in the position and orientation (pose) of the end-effector (e.e.), which is where the robot behavior can be more easily specified. To define the pose of the e.e. we need a reference frame RF_0 (fixed) and a body (rigid) (with reference frame RF_1). We define the pose of this body w.r.t. RF_0 (see Fig. 1.2). H_1^0 is the TRANSFORMATION MATRIX defining the pose of the body w.r.t. RF_0. R_1^0 is the rotation matrix; it contains three vectors, which are the axes of RF_1 expressed in RF_0. p is a 3D vector containing the x, y, z coordinates of RF_1 w.r.t. RF_0.
Properties:

1. $R_1^0 = (R_0^1)^{-1} = {R_0^1}^T$ (inverse = transpose)
2. $R_0^1 {R_0^1}^T = I$ (that's not a property strictly speaking but just a consequence)
3. $|R_1^0| = 1$

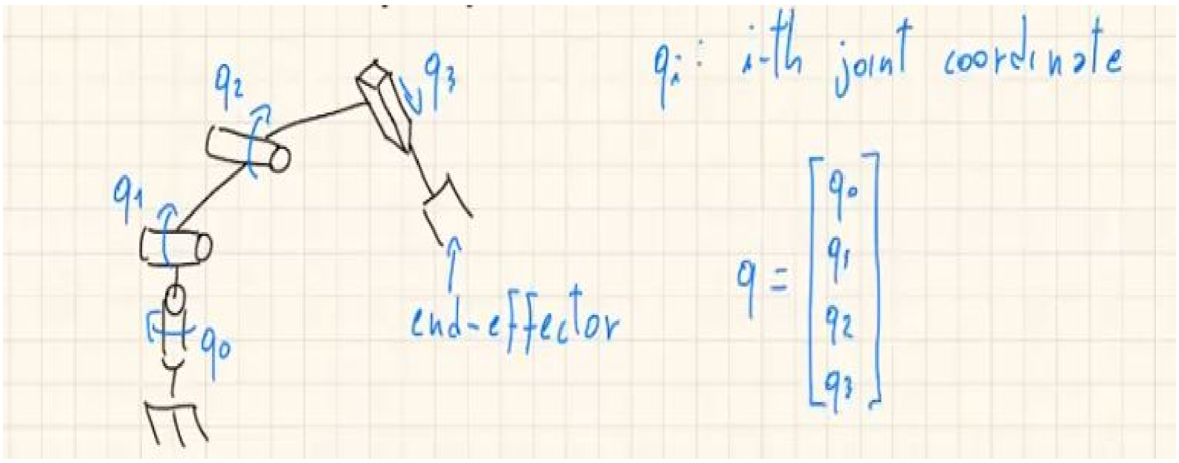

q_i: i-th joint coordinate
q_0
q_1
q_2
q_3
end-effector
$q = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$

Figure 1.1: Robot Manipulator Model

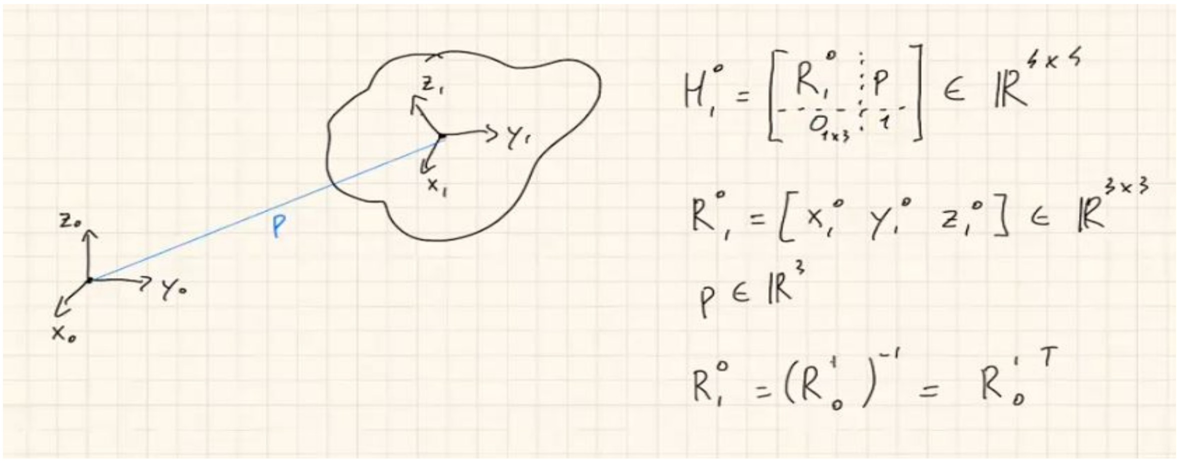

z_1
y_1
x_1
z_0
y_0
x_0
p
$H_1^0 = \begin{bmatrix} R_1^0 & p \\ 0_{1\times3} & 1 \end{bmatrix} \in \mathbb{R}^{4\times4}$
$R_1^0 = [x_1^0 \; y_1^0 \; z_1^0] \in \mathbb{R}^{3\times3}$
$p \in \mathbb{R}^3$
$R_1^0 = (R_0^1)^{-1} = R_0^{1\,T}$

Figure 1.2: Homogeneous Transformation

1.2 Forward Geometry

The forward geometry problem consists in computing the pose (i.e. position and orientation) of the end-effector given the joint angles.

Suppose to have a sequence of RF's, and to know the homogeneous matrices H_i^{i-1} defining the pose of the i^{th} RF w.r.t. the $(i-1)^{th}$ one $(i = 1, ..., n)$. Then we can easily compute the transformation from RF 0 to RF n as:

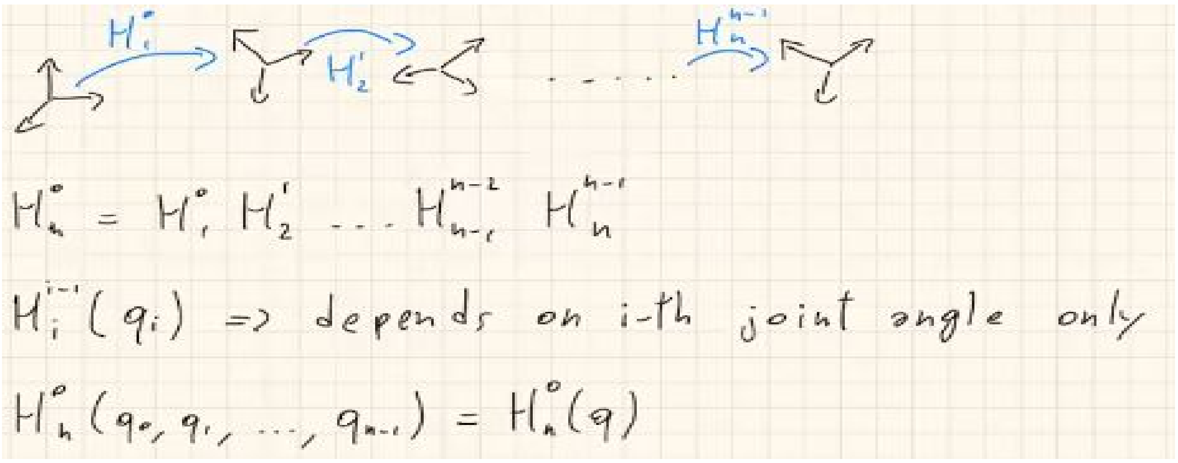
$$H_n^0 = H_1^0 \, H_2^1 \, ... \, H_{n-1}^{n-2} \, H_n^{n-1} = \prod_{i=1}^{n} H_i^{i-1}$$

These RF's refer to the joints: we associate a RF to each joint, and we compute the E.E. pose (RF n) w.r.t. the robot base (RF 0): we use this chain of transformations, each of which depends on a single joint angle q_i. So the total transformation H_n^0 depends on all the joint angles, so on q (see Fig. 1.3).

The R matrix represents the frame orientation. However, there are other ways to represent **orientations**:

1. Rotation matrices (dim = [3x3])

2. Euler Angles (dim = [3x1])

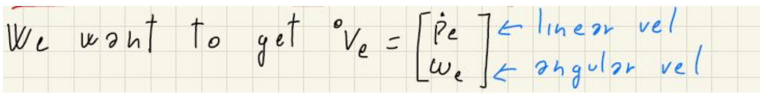


$H_n^0 = H_1^0 H_2^1 \dots H_{n-1}^{n-2} H_n^{n-1}$

$H_i^{i-1}(q_i) \Rightarrow$ depends on i-th joint angle only

$H_n^0(q_0, q_1, \dots, q_{n-1}) = H_n^0(q)$

Figure 1.3: Composition of transformations



We want to get $^0V_e = \begin{bmatrix} \dot{p}_e \\ \omega_e \end{bmatrix}$ $\leftarrow$ linear vel $\leftarrow$ angular vel

Figure 1.4: Differential kinematics problem.

3. Roll-Pitch-Yaw (R-P-Y): (dim = [3x1])
4. Quaternions (dim = [4x1])
5. Angle-Axis (dim = [4x1])

The main difference in representing differently the RF's orientation is their dimension. We should try not to use Euler angles and Roll-Pitch-Yaw because, even though they are the most comfortable in terms of number of values (3, which is the minimum) and to work with (they are easy to understand for humans), they are subject to the so-called 'singularity problem'. This means that there are certain orientations for which there are multiple equivalent definitions of Euler angles, and that creates problems when working with numerical algorithms.

1.3 Differential Kinematics

1.3.1 Geometric Approach

The problem of differential kinematics consists in computing the velocity of the end-effector (both angular and linear, see Fig. 1.4), given the velocities of the joints. It's similar to the forward kinematics problem, but instead of computing a pose we compute a velocity. To do that we use the relative velocity between 2 different bodies in space and we do it iteratively for all the bodies of the robot, starting from the base and towards the E.E. (v_e^0): The velocity of n^{th} link w.r.t. RF_0 is the sum of the relative velocities of each link w.r.t. the previous one.

$$\dot{P}_n^0 = \dot{P}_n^{n-1} + \dot{P}_{n-1}^0 = \dot{P}_n^{n-1} + \dot{P}_{n-1}^{n-2} + \dot{P}_{n-2}^0 = \sum_{i=0}^{n-1} \dot{P}_{i+1}^i \tag{1.1}$$

The expression above of $\dot{P}_n^0$ is simple because it consists of the sum of several simple quantities, each of which depends only on one joint velocity: the velocity of the joint connecting body i to body $i+1$. The translational velocity can be expressed as a function of the joint velocity. If the joint is revolute, we have:

$$\dot{P}_i^{i-1} = [\hat{z}_i \wedge (\vec{P}_i - \vec{P}_{i-1})]\, \dot{q}_i = J_{p_i}\, \dot{q}_i \tag{1.2}$$

If instead the joint is prismatic we have:

$$\dot{P}_i^{i-1} = \hat{z}_i\, \dot{q}_i = J_{o_i}\, \dot{q}_i \tag{1.3}$$

where $\hat{z}_i$ is the axis of translation or rotation of the i^{th} joint, $\dot{q}_i$ is the (scalar) velocity of the i^{th} joint, $r = (\vec{P}_i - \vec{P}_{i-1})$ is the level arm, that is the distance between the frame of body i and the rotation axis. Note that these two expressions are linear in the velocities $\dot{q}_i$ (same for relative angular velocities). Substituting (1.2) into (1.1) we get:

$$\dot{P}_n^0 = \sum_{i=0}^{n-1} J_{P_{i+1}}\, \dot{q}_{i+1} = J_p\, \dot{q} \tag{1.4}$$

with

$$J_p = [J_{p_1}, J_{p_2}, ..., J_{p_n}] \tag{1.5}$$

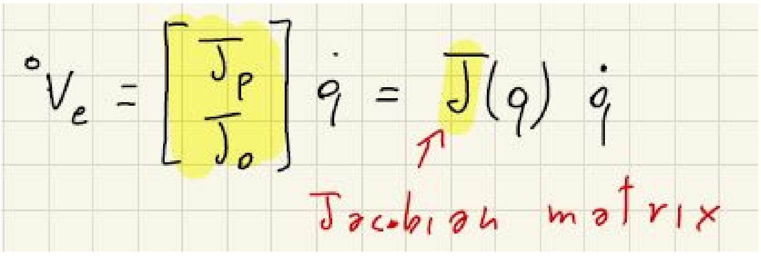
For the angular velocities we have instead:

$$\omega_n^0 = \sum_{i=0}^{n-1} J_{o_i}\, \dot{q}_i = J_o\, \dot{q} \tag{1.6}$$

with

$$J_o = [J_{o_1}, J_{o_2}, ..., J_{o_n}] \tag{1.7}$$

with J_{o_i} and J_{p_i} vectorial components (of the **Jacobian matrices** J_o and J_p) defined differently for the translation/position and orientation. The final expressions (1.4) and (1.6) are linear w.r.t. $\dot{q}$ (joint velocities) and are very similar. We are able to express the velocity of the E.E as a linear function of the joint velocities (see Fig. 1.5).



${}^{0}V_e = \begin{bmatrix} J_p \\ J_o \end{bmatrix} \dot{q} = J(q)\, \dot{q}$
Jacobian matrix

Figure 1.5: Differential Kinematics

1.3.2 Analytic Approach

The analytic approach defines the pose of the E.E as a 6 dimensional vector, where the first 3 elements represent the E.E. position, and the last 3 elements represent the E.E. orientation. Since we have 3 elements for the orientation, we must use one of the 'singularity-prone' representations of the orientation (i.e. R-P-Y or Euler angles).

$$\Phi_n = \begin{bmatrix} x_n \\ y_n \\ z_n \\ \alpha_n \\ \theta_n \\ \gamma_n \end{bmatrix} = \Phi_n(q)$$

is the 6-dim vector function of the joint angles q stated above. Once we have defined the E.E. pose function, the E.E. velocity is simply its time derivative:

$$\dot{\Phi} = \frac{d\Phi}{dt} = \frac{\partial \Phi}{\partial q}\frac{dq}{dt} = \frac{\partial \Phi}{\partial q}\dot{q}$$

where $\frac{\partial \Phi}{\partial q}$ is the analytic Jacobian, which we refer to as J_A. Note that:

$$\begin{bmatrix} J_p \\ J_{A_o} \end{bmatrix} = J_A \neq J = \begin{bmatrix} J_p \\ J_o \end{bmatrix} \tag{1.8}$$

with J_{A_o} is the Jacobian of the analytic orientation. The difference between J_{A_o} and J_o is due to the fact that, before we were trying to find the E.E. angular velocity (for the orientation part), now we want to find **the derivative of 3 angles** (α, β, γ). In fact the $\frac{d\Phi}{dt}$ (time derivative of R-P-Y angles) is different from the angular velocity. So $J_{A_o} \neq J_o$ For the translation instead, the geometric and the analytic Jacobians are equivalent (see Fig. 1.6).

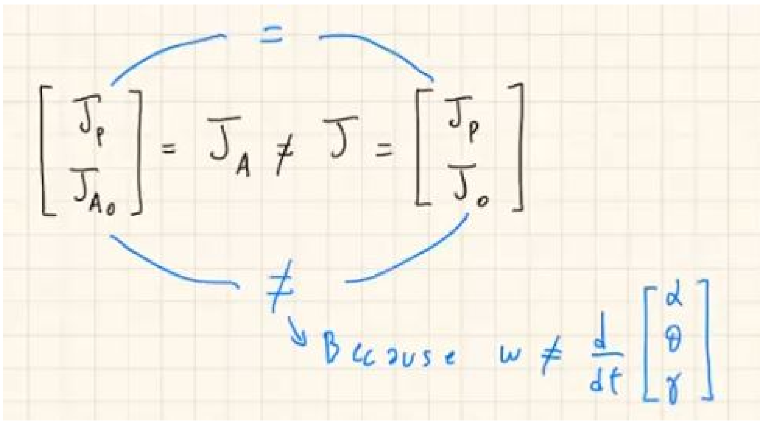

$\begin{bmatrix} J_p \\ J_{A_o} \end{bmatrix} = J_A \neq J = \begin{bmatrix} J_p \\ J_o \end{bmatrix}$
=
≠
Because $\omega \neq \frac{d}{dt}\begin{bmatrix} \alpha \\ \theta \\ \gamma \end{bmatrix}$

Figure 1.6: Difference between Geometric and Analytic Approach

1.4 Statics

Given a kinematic chain (i.e. a robot manipulator), subject to an external wrench (i.e. a 3D force and a 3D moment = 6D force vector w) applied to the E.E., we want to compute the joint forces/torques τ_w generated by this external wrench at each joint.

$$w = \begin{bmatrix} f \\ m \end{bmatrix} \tag{1.9}$$

We can solve this problem by computing the (mechanical) power of the robot at the joints, and the power of the robot at the E.E., and then exploiting the fact that they must be equal because power must be independent of the space in which it is computed. The mechanical power at the joints can be computed as the summation of the products of torque and velocity at each joint:

$$P_\tau = {\tau_w}^T \dot{q} \tag{1.10}$$

Similarly, the power at the E.E. can be computed as:

$$P_{e.e} = f^T \dot{p} + m^T \omega = \begin{bmatrix} f^T & m^T \end{bmatrix} \begin{bmatrix} \dot{p} \\ \omega \end{bmatrix} = w^T v = w^T J \dot{q} \tag{1.11}$$

with v being the E.E. velocity. Equating (1.10) and (1.11), we have:

$$\tau_w{}^T \dot{q} = w^T J \dot{q} \tag{1.12}$$

Since this equality must hold for any joint velocity, this implies that:

$$\tau_w{}^T = w^T J \quad \Rightarrow \quad \tau_w = J^T w \tag{1.13}$$

We conclude that joint torques resulting from the external contact wrenches can be simply computed by multiplying w times J^T. This shows that the Jacobian matrix J has a double role: it maps the joint velocities to the end-effector velocity, and it maps the end-effector wrench to the joint torques.

1.5 Dynamics

The two main problems related to the robot dynamics are:

- Direct Dynamics: given the joint positions, velocities (i.e. the state) and the joint torques, we want to compute the joint accelerations (how the system will move forward in time). This is typically what simulators do.
- Inverse Dynamics: given the state (positions and velocities) and the accelerations, we want to compute the joint torques. This is typically what controllers do: we know how we would like the robot to move (i.e. its accelerations) and we want to compute the torques that have to be applied by the motors to generate the desired accelerations.

Usually while dealing with manipulators (or robot in generals) we are interested in controlling their *dynamics*, so affecting the system's behavior. Let us call $q \in \mathbb{R}^n$ the *Lagrangian coordinates* that are used to describe the system configuration, and denote with $\dot{q} \in \mathbb{R}^n$ their velocity. The dynamic of the system is then described by the following nonlinear differential equation:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = \tau + J(q)^\top w \tag{1.14}$$

where $M \in \mathbb{R}^{n\times n}$ is the symmetric positive-definite *mass-matrix*, $C \in \mathbb{R}^{n\times n}$ is a matrix that takes into account both centrifugal and Coriolis force, while $g \in \mathbb{R}^n$ accounts for gravity; $\tau \in \mathbb{R}^n$ contains the joint torques. If the robot is in contact with the environment (suppose, without loss of generality, that the contact is at the end-effector), the dynamics also features the term $w \in \mathbb{R}^6$, which is the contact *wrench* (composed by a 3D force and a 3D moment), which is projected in joint space using the end-effector Jacobian $J \in \mathbb{R}^{6\times n}$. Often the terms C and g are condensed in the so-called *bias forces* $h \in \mathbb{R}^n$, rewriting (1.14) as:

$$M(q)\ddot{q} + h(q,\dot{q}) = \tau + J(q)^\top w \tag{1.15}$$

Note that, in general, M, h and J are nonlinear functions of the state $(q, \dot{q})$, so linear control theory cannot be applied to control this class of dynamical systems. Nonetheless, (1.15) is linear with respect to $\ddot{q}$ and τ (once $q, \dot{q}$ and w are fixed), which is an important property that can be exploited for its control.

Chapter 2

Reactive Controls

2.1 Joint-Space Motion Control

The trajectory-tracking *control* problem addresses the question of finding the control inputs τ that make the system follow a reference trajectory. Denoted with $q^{ref}(t)$ the *reference joint trajectory*, the control problem can be loosely defined as:

$$\text{find } \tau(t) \qquad \text{such that } q(t) \simeq q^{ref}(t)$$

PID control The simplest strategy that we can use to solve this problem is by means of a *Proportional Integrative Derivative* (*PID*) controller:

$$\tau(t) = K_p e(t) + K_d \dot{e}(t) + K_i \int_0^t e(\zeta)\, d\zeta \tag{2.1}$$

where $e(t) = q^{ref}(t) - q(t)$ is the *error* between the reference trajectory and the real measured coordinates, and $K_p, K_d, K_i \in \mathbb{R}^{n\times n}$ are respectively the *proportional*, *derivative* and *integral gain matrices*, which must be positive-definite to ensure stability.

As a rule of thumb, such feedback gains are tuned by first setting the proportional gain, following the derivative one, and last the integral part. In general the best tracking performance is achieved by increasing the proportional gain K_p: this however leads to a *stiff* system that is usually less safe for human interaction (since high forces are exerted by the robot itself). For this reason more advanced techniques are used to solve the control problem in joint coordinate.

Inverse-dynamics control Assumed that the system is known and deterministic (hypothesis carried out throughout most of the course), then both M and h in (1.15) are known, and can be exploited to improve the control performance. In inverse-dynamics control (also known as *feedback linearization* or *computed torques*), the controller computes a desired acceleration $\ddot{q}^d$ for the joint coordinates, and the torques are then computed by exploiting the system dynamics (1.15):

$$\begin{cases} \tau &= M\ddot{q}^d + h \\ \ddot{q}^d &= \ddot{q}^{ref} + K_d \dot{e} + K_p e \end{cases} \tag{2.2}$$

Using the control law (2.2), the closed-loop dynamics becomes:

$$\begin{aligned}
M\ddot{q} + h &= \tau \\
M\ddot{q} + h &= M\ddot{q}^d + h \\
\ddot{q} &= \ddot{q}^d = \ddot{q}^{ref} + K_d\dot{e} + K_p e \\
0 &= \ddot{e} + K_d\dot{e} + K_p e
\end{aligned}$$

At this point the stability of the system can be analyzed with linear techniques by transforming this differential equation to first order; defining the vector $y = (e, \dot{e}) \in \mathbb{R}^{2n}$, the system reduces to the following linear dynamics:

$$\dot{y} = Ay \qquad \text{with } A = \begin{bmatrix} 0 & I \\ -K_p & -K_d \end{bmatrix}$$

Linear theory provides us ways to tune K_p, K_d in such a way that the overall matrix be stable (by enforcing that all eigenvalues of A have a negative real part).

Controller (2.2) usually leads to less stiff systems (which is typically desirable), but a very accurate model of the systems dynamics is required, i.e. (1.15) must be known.

Integral gain In (2.1) it has been necessary to embed an integral term in the control law to compensate for the effect of gravity (since no system knowledge is required): this effect is nonlinear in the robots configuration and the integral gain aims to compensate such contribution asymptotically.
In (2.2) the integral term is instead missing since all gravitational actions are modeled by the bias force h, thus it is automatically compensated by the computed torques law ($\ddot{q}^d$ must depend just on the tracking error and need not compensate for gravity).

2.2 Cartesian-Space Motion Control

Defining the desired behavior of the robot in joint coordinates is typically hard. A much easier way to define it is in terms of Cartesian space motion of the end-effector. Therefore, it is interesting to have control strategies that work directly in Cartesian space. A simple way to achieve this is to transform the reference Cartesian-space trajectory $x^{ref}(t)$ in a reference joint-space trajectory $q^{ref}(t)$ by solving the so-called *inverse-geometry* problem. This problem consists in finding a valid joint configuration q such that the end-effector position (and possibly orientation) matches a desired value. This solution can theoretically work, but it presents several downsides. First, we have to deal with the high nonlinearity of the geometry function, which makes it impossible to find an analytical solution of the problem, even for simple robots. This is particularly important if the reference trajectory cannot be known in advance (e.g., if the end-effector has to track a moving object), and therefore the inverse-geometry problem must be solved inside the control loop.

We can therefore try to design a control law that works directly with a Cartesian-space reference trajectory $x^{ref}(t)$. We want to compute the joint torques $\tau(t)$ such that the end-effector trajectory $x(t) \in \mathbb{R}^3$ (in Cartesian coordinates) follows $x^{ref}(t)$:

$$x(t) \simeq x^{ref}(t)$$

2.2.1 Operational-Space Control

Let us start by defining the relationship between the end-effector position $x \in \mathbb{R}^3$ and the joint configuration q, which is the so-called *forward geometry* function:

$$x = \text{FG}(q) \tag{2.3}$$

By taking the time derivative of this relationship we get:

$$\dot{x} = \frac{d}{dt}\text{FG}(q) = \frac{d\text{FG}(q)}{dq}\frac{dq}{dt} = J(q)\dot{q} \tag{2.4}$$

Taking the time derivative once again:

$$\ddot{x} = J(q)\ddot{q} + \dot{J}(q,\dot{q})\dot{q} \tag{2.5}$$

This relationship between $\ddot{x}$ and $\ddot{q}$ will be useful in the following derivation. Let us pre-multiplying the systems dynamic (1.15) by JM^{-1}, to map the equation from joint to Cartesian space:

$$\begin{aligned} J\cancel{M^{-1}M}\ddot{q} + JM^{-1}h &= JM^{-1}\tau \\ \ddot{x} - \dot{J}\dot{q} + JM^{-1}h &= \end{aligned} \tag{2.6}$$

where $J\ddot{q}$ has been substituted by $\ddot{x} - \dot{J}\dot{q}$. Assuming that the Jacobian J is full-row rank (i.e. the end-effector is allowed to move in all directions) then $JM^{-1}J^\top \in \mathbb{R}^{3\times 3}$ is invertible; pre-multiplying (2.6) by the matrix $\Lambda \triangleq \left(JM^{-1}J^\top\right)^{-1}$ gives

$$\Lambda\ddot{x} + \underbrace{\Lambda\left(JM^{-1}h - \dot{J}\dot{q}\right)}_{\mu} = \Lambda JM^{-1}\tau$$

As a final step to get our Cartesian-space dynamics, we can assume that the joint torques τ are generated by a virtual force $f_\tau \in \mathbb{R}^3$ applied at the end-effector; in this way we have that $\tau = J^\top f_\tau$, and the previous equation becomes:

$$\begin{aligned} &\Lambda\ddot{x} + \mu = \Lambda JM^{-1}J^\top f_\tau = \left(JM^{-1}J^\top\right)^{-1}JM^{-1}J^\top f_\tau \\ &\Lambda\ddot{x} + \mu = f_\tau \end{aligned} \tag{2.7}$$

Now that we have a Cartesian-space version of the robot dynamics, in the same spirit as the inverse-dynamics control discussed above, we can design a Cartesian-space inverse dynamics controller (a.k.a. Operational-Space Control):

$$\begin{cases} \tau &= J^\top f_\tau \\ f_\tau &= \Lambda\ddot{x}^d + \mu \\ \ddot{x}^d &= \ddot{x}^{ref} + K_d\left(\dot{x}^{ref} - \dot{x}\right) + K_p\left(x^{ref} - x\right) \end{cases} \tag{2.8}$$

A simplified version of this controller can be obtained by compensating for bias forces directly in joint space, rather than in Cartesian space:

$$\begin{cases} \tau &= J^\top f_\tau + h \\ f_\tau &= \Lambda\ddot{x}^d \end{cases} \tag{2.9}$$

To prove the effectiveness of the controller (2.9) we can compute the closed-loop dynamics:

$$\begin{aligned} JM^{-1}\Big(M\ddot{q}+h=\tau=J^\top\Lambda\ddot{x}^d+h\Big) \\ J\ddot{q}+JM^{-1}h = JM^{-1}J^\top\Lambda\,\ddot{x}^d+JM^{-1}h \\ J\ddot{q}=\ddot{x}^d \\ \ddot{x}-\dot{J}\dot{q}=\ddot{x}^d \end{aligned}$$

As long as the systems does not move too fast, the term $\dot{J}\dot{q}$ is negligible and the end-effectors acceleration $\ddot{x}$ matches the desired one $\ddot{x}^d$.

Redundancy resolution

Controller (2.8) aims just to control the 3D motion of the end-effector; however, the number of actuated joints n typically exceeds the 3 degrees of freedom of the end-effector, therefore there are theoretically infinitely many joint configurations that allow to achieve the same end-effector position. The suggested control tracks the desired end-effector position, but it might happen over time that the system reaches *singular configurations* for which J is no longer full-row rank (losing the capability to move in some directions): in this case $JM^{-1}J^\top$ is no longer invertible and the control law cannot be computed.

Redundancy resolution aims to solve this problem: since there are infinitely many configurations that allow to achieve a desired x, the idea is now to find some additional joint torques that do not impact the end-effector motion, but helps avoiding singular configurations.

Observed that $\Lambda JM^{-1}=\left(JM^{-1}J^\top\right)^{-1}JM^{-1}$ is a *left-inverse* of $J^\top$, i.e.

$$\text{let } \left(JM^{-1}J^\top\right)^{-1}JM^{-1}=J^{\top\dagger} \qquad \text{then } J^{\top\dagger}J^\top=I$$

Note that in general $J^\top J^{\top\dagger}\neq I$.

If we now consider any torque τ_0 in the kernel of $J^{\top\dagger}$ it happens that the dynamics of the system in Cartesian space is not affected at all; recalling (2.7) we have that:

$$\Lambda\ddot{x}+\mu=J^{\top\dagger}(\tau+\tau_0)=J^{\top\dagger}\tau \qquad \forall\tau_0\in\ker\{J^{\top\dagger}\}$$

Computing τ_0 can be hard since there is no upfront guarantee that the motion will avoid singular configuration, however we can compute such torques as

$$\tau_0=\left(I-J^\top J^{\top\dagger}\right)\tau_1 \qquad \forall\tau_1\in\mathbb{R}^n$$

Such choice always lies in $\ker\{J^{\top\dagger}\}$, in fact

$$\begin{aligned} J^{\top\dagger}\tau_0 &= J^{\top\dagger}\left(I-J^\top J^{\top\dagger}\right)\tau_1 \\ &= \left(J^{\top\dagger}-J^{\top\dagger}J^\top J^{\top\dagger}\right)\tau_1 \\ &= \left(J^{\top\dagger}-J^{\top\dagger}\right)\tau_1=0 \qquad\qquad \forall\tau_1\in\mathbb{R}^n \end{aligned}$$

With this premise we can improve controller (2.8) by adding an extra term τ_0 that tries to maintain the joint configuration as close as possible to a reference configuration q^{ref}, in order to avoid singular configurations:

$$\begin{cases} \tau &= J^\top f_\tau+\left(I-J^\top J^{\top\dagger}\right)\tau_1 \\ f_\tau &= \Lambda\ddot{x}^d+\mu \\ \ddot{x}^d &= \ddot{x}^{ref}+K_d\left(\dot{x}^{ref}-\dot{x}\right)+K_p\left(x^{ref}-x\right) \\ \tau_1 &= M\big(K_{p1}(q^{ref}-q)-K_{d1}\dot{q}\big)+h \end{cases} \tag{2.10}$$

2.3 Impedance control

Controllers (2.8) and (2.10) allow to control the robot directly in Cartesian coordinates; such techniques require a very accurate knowledge of the system of interest to work properly. Model inaccuracies can be compensated to a certain extent by increasing the controller gains, improving the tracking capability; however, this solution can lead to an unsafe stiff system, since in case of contact the controller might generate high forces, which could brake either the robot or the environment.

To overcome this limitation we can use *interaction control*, particularly useful to model contact of the end-effector with the surrounding environment. The underlying idea is that if we have access to an estimate of the contact force, then we can regulate the motion accordingly. Based on this simple idea, there are two main categories of controllers. *Direct force controllers* mainly focus on controlling the applied contact forces (not covered in this course). *Indirect force controllers* try to enforce a certain dynamic relationship between force and position. This method is also known as *impedance control*.

In impedance control (in Cartesian space) we want the end-effector to behave as a mass-spring-damper subject to an external contact force f:

$$M_d \ddot{x} + B\dot{x} + Kx = f \tag{2.11}$$

where M_d, B, K are matrices defining the desired mass, damping and stiffness.

Once the contact model (2.13) is defined, the controller needs to compute at each time instant the joint torques τ to follow the desired behavior. Starting from (2.7), the dynamics projected in Cartesian space is simply

$$\Lambda\ddot{x} + \mu = J^{\top\dagger}\tau + f$$

Solving explicitly (2.13) for the acceleration provides the desired value

$$\ddot{x}^d = M_d^{-1}\big(f - B\dot{x} - Kx\big),$$

which substituted in the dynamics:

$$\Lambda M_d^{-1}\big(f - B\dot{x} - Kx\big) + \mu = J^{\top\dagger}\tau + f$$

As we did for the Operational-Space Control, assuming the joint torques τ are generated by a virtual end-effector force f_τ, then the previous equation can be solved for f_τ as:

$$\begin{aligned} f_\tau &= \Lambda M_d^{-1}\big(f - B\dot{x} - Kx\big) + \mu - f \\ &= -\Lambda M_d^{-1}\big(B\dot{x} + Kx\big) + \mu + \big(\Lambda M_d^{-1} - I\big)f \end{aligned}$$

At this point in order to compute τ (from f_τ) it's required an estimate of the contact force f that, unfortunately, usually comes from very noisy sensors and with time delays that might lead to unstable systems. A trick that we can use to overcome this limitation is to set $M_d = \Lambda$ in the desired dynamics (2.13): with this choice, f_τ no longer depends on the contact force f (since $\Lambda M_d^{-1} = I$). At the end, the resulting control law simplifies to:

$$\tau = J^\top f_\tau = J^\top\big(-Kx - B\dot{x} + \mu\big) \tag{2.12}$$

Similar derivations can be carried out even for cases where a reference trajectory and a reference contact force have to be tracked, by simply modifying the desired dynamics in this way:

$$M_d(\ddot{x}^{ref} - \ddot{x}) + B(\dot{x}^{ref} - \dot{x}) + K(x^{ref} - x) = f^{ref} - f \tag{2.13}$$

2.4 QP-based reactive control

Up to this point, we have discussed only classic control methods, mainly from the 60's-80's, which do not rely on numerical optimization. In this section we turn our attention to *optimization-based approaches*, which exploit numerical optimization solvers to determine the optimal control inputs. The main advantage of these methods is that they can account for constraints (such as actuator effort, joint position and velocity limits), which classic approaches could not consider. Before delving into this new class of control methods, let us quickly review the kind of convex optimization problems that will be used in this section.

2.4.1 Taxonomy of convex optimization problems

In general not all optimization problems can be solved; moreover, for control purposes, the execution time of numerical algorithms is fundamental since we want to control the system with as little delay as possible. In this section, we are mainly concerned with two kinds of convex optimization problems: i) Least-Squares Programs (LSP's) and ii) Quadratic Programs (QP's) [1]. LSP's are problem with affine equality/inequality constraints and a cost function that is the squared norm of an affine function:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}||Ax-b||^2 \\ \text{subject to} \quad & A_{eq}x = b_{eq} \\ & A_{in}x \leq b_{in} \end{aligned} \tag{2.14}$$

QP's admit the same constraints as LSP's, but a more general kind of cost (i.e. quadratic):

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}x^\top Hx + g^\top x \\ \text{subject to} \quad & A_{eq}x = b_{eq} \\ & A_{in}x \leq b_{in} \end{aligned} \tag{2.15}$$

The QP is convex if and only if the Hessian matrix H is positive semi-definite (i.e. all its eigenvalues have non-negative real part), which implies:

$$x^\top Hx \geq 0 \qquad \forall x \tag{2.16}$$

LSP's are a subclass of QP's, which can be clearly seen by expanding the cost function of (2.14):

$$\frac{1}{2}||Ax-b||^2 = \frac{1}{2}x^\top \underbrace{A^\top A}_{H} x \underbrace{-b^\top A}_{g^\top} x + \frac{1}{2}b^\top b, \tag{2.17}$$

where we have highlighted the connection with the Hessian matrix H and gradient vector g of the QP's cost. The Hessian of an LSP's cost is always positive semi-definite because the quadratic term can be interpreted as a norm of another vector $y \triangleq Ax$, which cannot be negative:

$$x^\top A^\top Ax = y^\top y = ||y||^2 \geq 0 \qquad \forall x \tag{2.18}$$

The reason why the class of LSP's is strict contained in the class of QP's is because of the gradient vector, which cannot be arbitrarely chosen in an LSP, but must be related to the

[1]QP's can be either convex or not, but for the rest of this section we will restrict our discussion to convex QP's.

problem's Hessian through the matrix A: $g = A^\top b$. Indeed, this relationship implies that g must be a linear combination of the rows of A.

Both LSP's and QP's are convex optimization problem that can be solved sufficiently fast to be used inside fast control loops (i.e. with control frequencies between 100 and 1000 Hz). Moreover, since LSP's are a special kind of QP's, QP solvers can be used also to solve LSP's. Indeed, since QP solvers are more common than LSP solvers, often times people use QP solvers to solve LSP's.

Solving Unconstrained LSP's

Since LSP's are convex, then any stationary point $x^\star$ of the cost function $f(x)$ is also a global minimizer of the cost. In case the LSP has no constraints, $x^\star$ can be computed by setting to zero the gradient of the cost f, i.e. solving:

$$\nabla f = A^\top A x - A^\top b = 0$$

Assuming that $A^\top A$ is invertible the explicit solution of the minimizing argument is

$$x^\star = \left(A^\top A\right)^{-1} A^\top b \tag{†}$$

where $\left(A^\top A\right)^{-1} A^\top$ is the *left pseudo-inverse* of A. If $A^\top A$ is not invertible but conversely $AA^\top$ is, then we can use the *right pseudo-inverse* to obtain the solution

$$x^\star = A^\top \left(AA^\top\right)^{-1} b \tag{‡}$$

We can verify that this is indeed a minimizer of f by substituting it in the cost function, and checking that it leads in fact to a zero (minimum) cost:

$$f(x^\star) = \left\|AA^\top \left(AA^\top\right)^{-1} b - b\right\|^2 = \|b - b\|^2 = 0$$

In general, the solution of least-squares minimization problem is determined by means of the *Moore-Penrose pseudo-inverse* $A^\dagger$ as

$$x^\star = \arg\min_x \left\|Ax - b\right\|^2 = A^\dagger b \tag{2.19}$$

where

$$A^\dagger = \begin{cases} A^\top \left(AA^\top\right)^{-1} & \text{if } A \text{ is full-row rank} \\ \left(A^\top A\right)^{-1} A^\top & \text{if } A \text{ is full-column rank} \end{cases} \tag{2.20}$$

We observe now that both solutions have been obtained considering the respective definition of the Moore-Penrose pseudo-inverse based on the condition of the matrix A (leading to the invertibility of $A^\top A$ or $AA^\top$). Even if A is neither full-row and nor full-column rank, its pseudo-inverse $A^\dagger$ can still be computed by means of a matrix decomposition (e.g. Singular Value Decomposition).

2.4.2 Joint-space control

To track a reference trajectory in joint space q^{ref}, we have already discussed the inverse-dynamics controller (2.2), which is:

$$\tau = M\ddot{q}^d + h \qquad \text{with } \ddot{q}^d = \ddot{q}^{ref} + K_d \dot{e} + K_p e$$

The same result can be achieved by solving the following LSP:

$$\begin{aligned} \underset{\tau,\ddot{q}}{\text{minimize}} \quad & \frac{1}{2}\left\|\ddot{q}-\ddot{q}^d\right\|^2 = \frac{1}{2}\left(\ddot{q}-\ddot{q}^d\right)^\top \left(\ddot{q}-\ddot{q}^d\right) \\ \text{subject to} \quad & M\ddot{q}+h=\tau \end{aligned} \tag{2.21}$$

where $\frac{1}{2}\left\|\ddot{q}-\ddot{q}^d\right\|^2$ is the *cost function* that needs to be minimized with respect to the *decision variables* $\ddot{q},\tau$, while satisfying the *constraint* $M\ddot{q}+h=\tau$ (i.e. the system dynamics).

Since in (2.21) the cost is a squared norm, the minimum is obtained when the argument of the norm is zero, i.e. the optimal acceleration is $\ddot{q}^\star = \ddot{q}^d$; this implies that the optimal commanded torques are simply $\tau^\star = M\ddot{q}^d + h$, as in (2.2).

In this case the solution of the minimization problem was trivial, thus applying (2.2) would have been easier, however the formulation (2.21) leaves room for adding other constraints, which until now were impossible to consider.

Actuator effort bounds Real actuators can't exert infinitely high forces, but their actions present a saturation limit that acts as a bound. Considering a symmetric behavior on the forces that can be generated, the actuator effort can be bounded as

$$-\tau^{max} \leq \tau \leq \tau^{max}$$

Furthermore while dealing with electrical actuators, the motor drivers can handle just a limited amount of current, thus limiting the overall torque that can be generated. Since the motor current i is proportional to the motor torque τ, this limitation can be described by the following inequality constraint:

$$-i^{max} \leq \underbrace{K_i\tau}_{i} \leq i^{max}$$

Joint velocity bounds Motors and gears also have limited velocities $\dot{q}^{max}$. We observe that in (2.21) the joint velocity $\dot{q}$ is not a decision variable, but it is a constant because the control inputs cannot affect the current robot velocity. Since we can't act on the current velocity, we can ensure that at the next time step the velocity doesn't exceed its limit. Assuming joint accelerations remain constant during the time step, which is reasonable because time steps are usually very short in reactive control schemes, we have that $\dot{q}(t+\Delta t) = \dot{q}(t) + \Delta t\,\ddot{q}$, thus a constraint that we might add to account for velocity limits is:

$$-\dot{q}^{max} \leq \dot{q} + \Delta t\,\ddot{q} \leq \dot{q}^{max}$$

Joint position bounds Actuators also have bounded joint positions, i.e. q is constrained to lie in a interval $\left[q^{min}, q^{max}\right]$. To satisfy such constraint we might be tempted to exploit the same trick used for the joint velocity limits, i.e. to compute q at the next time-step by integrating $\ddot{q}$ twice, reaching a formulation of the form

$$q^{min} \leq q + \Delta t\,\dot{q} + \frac{1}{2}\Delta t^2\,\ddot{q} \leq q^{max} \tag{∘}$$

This solution would work if we only had joint position and velocity limits. However, since in practice we also have joint acceleration limits (which are a direct consequence of the joint torque limits), this approach can lead to an *unfeasible QP*. This is because in order not to exceed the joint position limits, the controller might be required to produce a "high" (in absolute value) acceleration $\ddot{q}$ that is incompatible with the actuator capabilities.

A heuristic that can be used to mitigate this issue is to use the constraint (∘) with a larger value for the time-step Δt. This has the effect of "looking further into the future", and thus requiring the joint to decelerate more in advance. However, this is not sufficient in general to ensure feasibility. An in-depth discussion of this issue can be found in the paper [1], where an exact solution is suggested, under the assumption of constant joint acceleration bounds. We will come back to this issue when discussing Model Predictive Control methods, which are more suited to account for this kind of state constraints.

QP problem The optimal control problem in (2.21) can be extended in order to embed all the bounds described so far, reaching the formulation:

$$\begin{aligned} \underset{\tau,\ddot{q}}{\text{minimize}} \quad & \frac{1}{2}\|\ddot{q}-\ddot{q}^d\|^2 = \frac{1}{2}\left(\ddot{q}-\ddot{q}^d\right)^\top\left(\ddot{q}-\ddot{q}^d\right) \\ \text{subject to} \quad & M\ddot{q}+h=\tau \\ & -\tau^{max} \leq \tau \leq \tau^{max} \\ & -i^{max} \leq K_i\tau \leq i^{max} \\ & -\dot{q}^{max} \leq \dot{q}+\Delta t\,\ddot{q} \leq \dot{q}^{max} \\ & q^{min} \leq q+\Delta T\,\dot{q}+\frac{1}{2}\Delta T^2\,\ddot{q} \leq q^{max} \end{aligned} \tag{2.22}$$

Such problem is harder to solve than (2.21) because its solution is usually non-trivial. For this reason, *QP solvers* are typically used, which are customized software to solve convex QP problems as (2.21).

2.4.3 Cartesian-Space Control

We can use quadratic programs also to solve control problems in Cartesian space. Recalling the desired Cartesian acceleration $\ddot{x}^d = \ddot{x}^{ref} + K_p\left(x^{ref}-x\right) + K_d\left(\dot{x}^{ref}-\dot{x}\right)$ defined in (2.8), and recalling the relationship between Cartesian-space and joint space accelerations $\ddot{x} = J\ddot{q}+\dot{J}\dot{q}$, we can formulate the problem as:

$$\begin{aligned} \underset{\tau,\ddot{q},\ddot{x}}{\text{minimize}} \quad & \frac{1}{2}\|\ddot{x}-\ddot{x}^d\|^2 \\ \text{subject to} \quad & M\ddot{q}+h=\tau \\ & \ddot{x}=J\ddot{q}+\dot{J}\dot{q} \end{aligned} \tag{2.23}$$

To improve the numerical performance of the algorithm we can remove the last constraint and explicitly substitute it in the cost function:

$$\begin{aligned} \underset{\tau,\ddot{q}}{\text{minimize}} \quad & \frac{1}{2}\|J\ddot{q}+\dot{J}\dot{q}-\ddot{x}^d\|^2 \\ \text{subject to} \quad & M\ddot{q}+h=\tau \end{aligned} \tag{2.24}$$

Exploiting (2.19) we can see that the optimal joint accelerations and torques are computed simply as

$$\ddot{q}^\star = J^\dagger\left(\ddot{x}^d-\dot{J}\dot{q}\right) \qquad \tau^\star = M\ddot{q}^\star + h \tag{2.25}$$

where in this case the coefficient b is equal to $\ddot{x}^d-\dot{J}\dot{q}$ (all these quantities are constant in the problem). Notice that by using a weighted pseudo-inverse[2] in (2.25) with M^{-1} as weight matrix, we would recover the Operational-Space Control law.

[2]A weighted pseudo-inverse of a full-row rank matrix A, with weight matrix W, is defined as $WA^\top(AWA^\top)^{-1}$.

2.4.4 Task-Space Control

In this section we generalize the concept of Cartesian-space control so that we can control not just the end-effector in Cartesian coordinates, but any *task* (a.k.a. a *control objective*) that the robot should achieve.

Each task is usually described by an *task function* $e(x, u, t)$, where $x = (q, \dot{q})$ is the *state* of the system, and $u = \tau$ the inputs (using a more general notation). The task function must be defined by the user so that:

- $e = 0$ implies that the task has been achieved;
- the closer e is to zero, the closer we are to achieving the task;
- e is sufficiently smooth to be differentiable for the required number of times.

Without loss of generality, let us assume that the error function is defined as the difference between a given time-invariant output function $y(x, u)$ and a time-varying reference value $y^{ref}(t)$:

$$e(x, u, t) = y(x, u) - y^{ref}(t) \tag{2.26}$$

Since LSP's can only minimize the norm of an affine function in the decision variable $\ddot{q}$, τ, a transformation is required (as the one to solve the problem in Cartesian space); minimizing directly e is not possible as in general it depends only on q and $\dot{q}$, and not on $\ddot{q}$. To reach this goal we can try to control how e evolves in time, by imposing the following two conditions:

1. $\lim_{t\to\infty} e(t) = 0$;
2. the dynamics of e is affine in the decision variables $\ddot{q}, u$.

To see how we can meet these conditions we analyze separately three cases, depending on whether the output function y depends on q, $\dot{q}$, or τ.

Position task function

Consider a task depending only on the joint positions q, in the form $e(x, u, t) = e(q, t) = y(q) - y^{ref}(t)$. An affine function in $\ddot{q}$ can be obtained by differentiation e in time for two times:

$$\xrightarrow{d/dt} \quad \dot{e} = J\dot{q} - \dot{y}^{ref} \quad \xrightarrow{d/dt} \quad \ddot{e} = J\ddot{q} + \dot{J}\dot{q} - \ddot{y}^{ref} \tag{$\circ$}$$

In this way the second requirement for e is satisfies. Now we need to choose a control policy for $\ddot{e}$ so that e converges to zero. To do that we reduce the dynamic to a first-order differential equation. Introducing the augmented state variable $z \triangleq (e, \dot{e})$, and considering for this case a proportional-derivative controller, the linear system boils down to:

$$\dot{z} = \begin{pmatrix} \dot{e} \\ \ddot{e} \end{pmatrix} = \begin{bmatrix} 0 & I \\ -K_p & -K_d \end{bmatrix} \begin{pmatrix} e \\ \dot{e} \end{pmatrix} = Az$$

In this case matrices $K_p, K_d > 0$ must be chosen in such a way that $\ddot{e} = -K_p e - K_d \dot{e}$ is asymptotically stable (A musth be Hurwitz). Since now also the first condition is met, we can rewrite ($\circ$) as

$$J\ddot{q} + \dot{J}\dot{q} - \ddot{y}^{ref} = -K_p e - K_d \dot{e}$$
$$J\ddot{q} + \dot{J}\dot{q} + K_p e + K_d \dot{e} = 0$$

Linear system theory tells us that such dynamic is exponentially stable, however in this case no constraints have been considered (but they can limit the actual stabilization performance).

Velocity task function

Let us consider a case for which the error function depends just on the joint velocity, so in the form $e(x,u,y) = e(\dot{q},t) = y(\dot{q}) - y^{ref}(t)$; differentiating in time the error function provides

$$\dot{e}(\dot{q},t) = \dot{y} - \dot{y}^{ref} = \frac{\partial y}{\partial \dot{q}}\frac{\mathrm{d}\dot{q}}{\mathrm{d}t} - \dot{y}^{ref} = J\ddot{q} - \dot{y}^{ref}$$

This expression clearly satisfies the second requirement of being affine with respect to $\ddot{q}$; to ensure the second condition we can set $\dot{e}$ as proportional to the current error:

$$\dot{e} = -K_p e$$

If K_p is positive definite the linear dynamics is asymptotically stable, satisfying the second condition. The final cost function that can be used to solve this problem is so

$$\begin{aligned} J\ddot{q} - \dot{y}^{ref} &= -K_p e \qquad \text{with } K_p > 0 \\ J\ddot{q} - \dot{y}^{ref} + K_p e &= 0 \end{aligned} \tag{2.27}$$

Input task function

The only way to build a task function $e(u,t)$ that depends just on the input torques $\tau = u$ is by ensuring that e is actually affine in u (which is a decision variable), i.e. must be of the form

$$e(u,t) = A(t)u - b(t)$$

Task summary

As just presented we might have 3 main kinds of task functions:

- affine functions of the input $\tau = u$;
- nonlinear functions of the state x, for which an affine function of $\ddot{q}$ can be found by means of one/two differentiations in time for velocity and position tasks, respectively.

In general, regardless of which task function we are working with, we can express any task function as an affine function of the decision variables $z = (\ddot{q}, u)$:

$$g(z) = \begin{bmatrix} A_x & A_u \end{bmatrix} \begin{pmatrix} \ddot{q} \\ u \end{pmatrix} - b = Az - b$$

QP-based control *Task-Space Inverse Dynamics* (*TSID*), also referred TO as *QP-based control*, solves optimization problems in the form:

$$\begin{aligned} \underset{z=(\ddot{q},\tau)}{\text{minimize}} \quad & \|Az - b\|^2 \\ \text{subject to} \quad & \begin{bmatrix} M & -I \end{bmatrix} z = -h \\ & \text{other constraints} \end{aligned} \tag{2.28}$$

This is a generalization of the QP joint-space control in (2.22), choosing $A = \begin{bmatrix} I & 0 \end{bmatrix}$ and $b = \ddot{q}^d$, or the Cartesian-space control in (2.24), choosing $A = \begin{bmatrix} J & 0 \end{bmatrix}$ and $b = \ddot{x}^d - \dot{J}\dot{q}$.

2.4.5 Underactuated systems

TSID can easily deal with underactuated systems, which are systems that have less control inputs than degrees of freedom. So far, we have focused on robot manipulators, which are fully actuated because typically each joint is actuated by a motor. However, most mobile robots are underactuated. Let us take, for instance, vehicles. Assuming a vehicle cannot detach from the ground, its configuration can be described with 3 variables: 2 for the position in the plane, and 1 for the orientation. However, a vehicle has typically only 2 actuators: the motor (which is commanded through the accelerator pedal) and the steering wheel. Therefore it is underactuated. The same applies to most aerial vehicles, which have 6 degrees of freedom, and typically 2-to-4 actuators. Underwater robots and legged robots are other examples of typically underactuated systems.

Many underactuated robots can be modeled with a small modification to the dynamics equations of fully-actuated systems. Let us assume that we remove some motors from a robot manipulator, making it underactuated. The effect that this would have on its dynamics is that the torque at the corresponding joints would be alway zero. We can model this by introducing a selection matrix $S \in \mathbb{R}^{n_a \times n}$, which selects the actuated joints of the vector q:

$$q_{actuated} = Sq = S \begin{bmatrix} q_{passive} \\ q_{actuated} \end{bmatrix} \tag{2.29}$$

The same matrix can be also used to modify the dynamics as:

$$M\ddot{q} + h = S^\top \tau = \begin{bmatrix} 0 \\ \tau \end{bmatrix} \tag{2.30}$$

We can see that the dynamics is still affine with respect to $\ddot{q}$ and τ, therefore we can still use it as a constraint in TSID's QP.

2.4.6 Rigid Contacts

Rigid contacts are particularly hard to deal with because they constrain the motion of the robot. Let us suppose that a robot manipulator makes contact with a wall, using its end-effector. After making contact, the end-effector can no longer move in the direction of the wall, and if it tries, it may apply a large contact force that could lead to unpleasant consequences (either for the robot or for the wall). Therefore, we need the controller to be aware of this constraint. This constraint could be modeled as an inequality constraint on the robot configuration, representing the fact that the position of the end-effector (or whatever point the robot is using for making contact) cannot penetrate the wall:

$$c(q) \leq \text{constant} \tag{2.31}$$

However, very often, once the robot has made contact with the environment, we want this contact to be maintained for a certain amount of time. Therefore, for this time period, the contact constraint can be modeled using an equality constraint, representing the fact that the contact point(s) should not move at all:

$$c(q) = \text{constant} \tag{2.32}$$

At this point, we should be familiar with the fact that we cannot include in TSID a constraint that depends only on q. Similarly to what we have already done for the task functions, we need to differentiate this constraint until it becomes a function of $\ddot{q}$:

$$\begin{aligned} J_c\dot{q} &= 0 \\ J_c\ddot{q} + \dot{J}_c\dot{q} &= 0 \end{aligned} \tag{2.33}$$

Besides modeling the motion constraints, rigid contacts also affect the system dynamics by means of the associated contact forces. Therefore, to account for rigid contact in TSID we need to introduce the contact forces λ as decision variables, and account for them in the system dynamics. In conclusion, the QP problem for an underactuated system in rigid contact with the environment is:

$$\begin{aligned} \underset{z=(\ddot{q},\tau,\lambda)}{\text{minimize}} \quad & ||Az - b||^2 \\ \text{subject to} \quad & \begin{bmatrix} M & -S^\top & J_c^\top \\ J_c & 0 & 0 \end{bmatrix} z = \begin{bmatrix} -h \\ -\dot{J}_c\dot{q} \end{bmatrix} \\ & \text{other constraints} \end{aligned} \tag{2.34}$$

2.4.7 Multi-Task Control

Complex robots typically need to carry out multiple tasks at the same time. This is often the case even for robot manipulators because they typically have more degrees of freedom than the dimension of the main task they must perform. For instance, if we want to control the position and orientation of the end-effector of a robot manipulator, then the task has dimension 6. However, many robot manipulators have 7 degrees of freedom, which means that they are *redundant* with respect to the task. In this case, it is recommended to introduce a secondary task in the control framework, to avoid that the uncontrolled degree(s) of freedom lead the robot state to drift and diverge, typically hitting joint limits or reaching undesirable configurations with low dexterity.

Assume a robot is required to perform N tasks, and each task is associated with a cost function g_i defined as:

$$g_i(z) = ||A_i z - b_i||^2, \qquad i = 1, \dots, N \tag{2.35}$$

We can then achieve a multi-task control by formulating TSID's QP problem as:

$$\begin{aligned} \underset{z=(\ddot{q},\tau,\lambda)}{\text{minimize}} \quad & \sum_{i=1}^{N} w_i \, g_i(z) \\ \text{subject to} \quad & \begin{bmatrix} M & -S^\top & J_c^\top \\ J_c & 0 & 0 \end{bmatrix} z = \begin{bmatrix} -h \\ -\dot{J}_c\dot{q} \end{bmatrix} \\ & \text{other constraints,} \end{aligned} \tag{2.36}$$

where $w_i \in \mathbb{R}$ is a user-defined positive weight associated to task i. These weights can be used to define the relative importance between tasks. Higher priority tasks will be assigned higher weights, whereas lower priority tasks will have lower weights. A ratio of 10^3 between the weights of two tasks typically leads in practice to a behavior that is indistinguishable from the one obtained with the null-space projection method used with Operational-Space control (Section 2.2.1).

Chapter 3

Optimal Control

DISCLAIMER: This chapter has not been reviewed yet, so its content may be inaccurate!

Optimal control problems (*OCPs*) are constrained minimization problems in the form

$$\begin{aligned} \min_{x(\cdot),u(\cdot)} \quad & \int_0^{\top} \ell\big(x(t),u(t),t\big)\,\mathrm{d}t + \ell_f\big(x(T)\big) & & \\ \text{sub. to:} \quad & \dot{x}(t) = f\big(x(t),u(t),t\big) & \forall t & \\ & g\big(x(t),u(t),t\big) \le 0 & \forall t & \\ & x(0) = x_0 & & \end{aligned} \tag{3.1}$$

where $\ell(\cdot)$ is the *running cost*, $\ell_f(\cdot)$ is the *terminal* (final) *cost*, f is the *system's dynamic*, g are the *path constraints* and x_0 is the *initial condition*. Moreover x are the so called *states* (that for manipulator are simply joint coordinates and velocities) while u are the *inputs* (joint torques).
For ease of notation the time-dependence of both x and u will now be dropped.

Optimal control problems from a first look are very similar to quadratic programs like (2.22) at 18, however the main difference in this case is that both $x(t)$ and $u(t)$ are *trajectories* that needs to be identified. In QPs we assumed in fact that a desired motion $y^{ref}(t)$ was given, while OCP solves a problem with theoretically infinitely many degrees of freedom that build the optimal trajectory y^{ref} itself.

Example: simple pendulum Let's consider the classical example of a simple pendulum that we want to "swing up"; choosing q in such a way that for $q = 0$ the pendulum is in the upright configuration, then the simplest OCP that we can formulate for this problem is

$$\begin{aligned} \min_{x,\tau} \quad & q(T) = 0 \\ \text{sub. to:} \quad & I\ddot{q} = \tau + mg\sin(q) \end{aligned}$$

where the lonely constraint is just the system's dynamic. In this case we used just a terminal cost to enforce to reach the desired motion at the end of the time horizon. Even if this problem lead to the desired configuration, no "rules" on how the system's behavior are set, implying that there's no penalization for a fast convergence or high velocities.

One way to reach the desired configuration in less time possible is to consider an integral cost that adds when we are out of target, considering the following OCP:

$$\begin{aligned} \min_{x,\tau} \quad & \int_0^{\top} q^2 \, \mathrm{d}t \\ \text{sub. to:} \quad & \begin{pmatrix} \dot{q} \\ \ddot{q} \end{pmatrix} = \begin{pmatrix} \frac{\mathrm{d}q}{\mathrm{d}t} \\ I^{-1}(\tau + mg\sin q) \end{pmatrix} \end{aligned}$$

where in this case the dynamic has been explicitly written in state-space after a transformation to first order given the state $x = (q, \dot{q})$.
Solving this problem would lead to the generation of infinitely high torques and joint velocities in order to reach the target configuration in little time as possible to reduce the integral value.

To avoid this problem we might add as integral costs also the square of the joint velocities and the torque, so the solver needs to find the perfect trade-off to reach the desired configuration in low time but without generating high velocities or torques:

$$\begin{aligned} \min_{x,\tau} \quad & \int_0^{\top} \left(q^2 + \dot{q}^2 + \tau^2\right) \mathrm{d}t + 10^3 q(T) \\ \text{sub. to:} \quad & I\ddot{q} = \tau + mg\sin(q) \end{aligned}$$

In this case it has been added also a heavy-weighted terminal cost to ensure that the system actually reach the desired configuration (since the trade-off in the integral might lead to a non-requested solution).

Using integral cost to bound velocities and torques is an easy way to solve the problem, but similar results can be obtained by setting hard constraints on some variables, i.e.

$$\begin{aligned} \min_{x,\tau} \quad & \int_0^{\top} q^2 \, \mathrm{d}t \\ \text{sub. to:} \quad & I\ddot{q} = \tau + mg\sin(q) \\ & |\tau| \leq \tau^{max} \\ & |\dot{q}| \leq \dot{q}^{max} \end{aligned}$$

Optimal control method families Tables/ocpfamilies.tex Once an optimal control problem (3.1) has been states, different families of solvers, described in table **??**, can be used to determine the optimal trajectories. In particular there are two "orthogonal" categorizations:

- *continuous-time* solvers are searching for the continuous-time solution of the problem itself, while *discrete-time* methods rely on a discretized (approximated) version of the same problem;
- *global* solvers are searching the global optimum for the problem, while *local* are starting from a guess and reach the closes minima point; the latter of course is faster at runtime.

For robotic applications mainly we use *direct method* solvers; still we start introducing the other ones as they present critical aspect that are useful in understanding direct methods and the next chapter on *Reinforcement Learning*.

3.1 Dynamic programming

Dynamic programming (*DP*) methods are based on the *Bellman's optimality principle* for which

« *given the optimal control starting from A at time $t = 0$ and reaching B at $t = T$, then given any condition $\overline{x}$ in the optimal trajectory for $\overline{t} \in (0, T)$ then the the optimal solution from $\overline{x}$ to B is still the same.* »

This is a simple, yet powerful, concept after which all minimization algorithms have been developed on. In particular for what concerns dynamic programming problem (3.1) is discretized in N time-steps and rewritten as

$$\begin{aligned} \min_{X,U} \quad & \sum_{i=0}^{N-1} \ell(x_i, u_i) \\ \text{sub. to:} \quad & x_{i+1} = f\big(x_i, u_i\big) \qquad\qquad \forall i = 0, \dots, N-1 \end{aligned} \tag{3.2}$$

where[1] $X \in \mathbb{R}^{n\times N}$ and $U \in \mathbb{R}^{m\times N}$ are the matrices containing all the states and inputs at the discrete-time steps, i.e.

$$X = \begin{bmatrix} x_1 & \dots & x_N \end{bmatrix} \qquad\qquad U = \begin{bmatrix} u_0 & \dots & u_{N-1} \end{bmatrix} \tag{3.3}$$

We point out that in (3.2) the terminal cost has been removed just to simplify the calculation of the following proof, but can be added at any time.
Furthermore we state that dynamic programming can be used only to solve *unconstrained minimization problems* (or, better, the lonely constraint allowed is the one of the systems dynamic).

Solution with QP After discretization of (3.1), (3.2) can be seen as a "simple" optimization problem in the variables X, U, similarly to (2.21) at page 17.

QP solvers can so be used to obtain the global optimum of the problem, however (3.2) suffers of dimensionality: increasing the number of states/inputs or improving the discretization N heavily affects the computational load. Problems might become too large to be handled just with very simple scenarios, so solving (3.2) with QPs is not a feasible solution.

After discretization OCP (3.2) reduces to a simple optimization problem such (2.21) at page 17; the main issue of using QP solvers for this problem is the high dimensionality of the problem itself that scales badly with both the number of states/inputs and the number of discretization steps.

Solution with the Bellman's principle To numerically solve (3.2), DP solvers exploit the Bellman's optimality principle; considering in fact $M \in \mathbb{N}$ such that $0 < M < N$, then the overall problem can be casted to the following unconstrained minimization:

$$\min_{X,U} \left(\underbrace{\sum_{i=0}^{M-1} \Big(\ell(x_i, u_i) + \mathcal{I}\big(x_{i+1} - f(x_i, u_i)\big)\Big)}_{=C_0(X_{1:M}, U_{0:M-1})} + \underbrace{\sum_{i=M}^{N-1} \Big(\ell(x_i, u_i) + \mathcal{I}\big(x_{i+1} - f(x_i, u_i)\big)\Big)}_{=C_M(x_M, X_{M+1:N}, U_{M:N-1})} \right) \tag{†}$$

In this expression it has been introduced the *indicator function* $\mathcal{I}$, a mathematical "trick" used to convert the dynamic constraint into a cost; to ensure the condition, the system is infinitely penalized when such constraint isn't satisfied, so by considering

$$\mathcal{I}(x) = \begin{cases} 0 & \text{if } x = 0 \\ \infty & \text{otherwise} \end{cases} \tag{3.4}$$

[1]in this case we assume that the state x has n dimensions while inputs u have dimension m.

Algorithm 1 pseudo-code for solving a discretized optimal control problem using dynamic programming.

```
initialize $V_n(x_N) = \ell_f(x_N)$                    ▷ look-up table that needs to be defined $\forall x_N$

for $i$ from $N-1$ to $0$ do
    $V_i(z) = \min_u \left(\ell(z,u) + V_{i+1}(f(z,u))\right)$
end for

for $i$ from $0$ to $N-1$ do                          ▷ at runtime
    $u_i^\star(x) = \min_u \left(\ell(x,u) + V_{i+1}(f(z,u))\right)$
end for
```

At this point we can rewrite (†) as the sum of two independent minimization problems as follows:

$$\min_{X_{1:M},U_{0:M-1}} C_0\left(X_{1:M}, U_{0:M-1}\right) + \underbrace{\min_{X_{M+1:N},U_{M:N-1}} C_M\left(x_M, X_{M+1:N}, U_{M:N-1}\right)}_{=V_M(x_M)} \tag{3.5}$$

Here we observe that the first is a "canonical" minimization, while te second one is parametric in x_M, i.e. the value of the second minimization is affected by the state x_M reached in the optimization of C_0.
Such second term is usually referred as the *value function* $V_M(x_M)$ and evaluates the optimal cost that needs to be payed from time $t = M$ starting from a (generic) initial condition x_M, or, in other words, the "optimal cost-to-go given x_M as initial condition. Exploiting the definition of the value function we have been able to split the initial problem (†) into two (almost) independent minimization.

In this way (3.5) can be concisely rewritten as

$$V_0(x_0) = \min_{X_{1:M},U_{0:M-1}} C_0\left(X_{1:M}, U_{0:M-1}\right) + V_M(x_m) \tag{3.6}$$

i.e. "the cost that we pay starting from $t = 0$ with x_0 initial condition and behaving optimally".
Choosing now $M = 1$ allow us to rewrite the value function with a *recursive formulation* as follows:

$$V_i(x_i) = \min_{u_i} \Big(\ell(x_i, u_i) + V_{i+1}\left(f(x_i, u_i)\right)\Big) \tag{3.7}$$

Here we see that the minimization is independent from the current state x_i that is the parameter of the value function, but what needs to be determined is the value of u_i that allows to reduce the overall cost of the current time-step combined with the cost-to-go.

DP algorithm Dynamic programming exploits the recursive definition of the value function (3.7) to solve the optimization problem; as presented in algorithm 1, the main idea is to regard V as a look-up table where for each current state, the optimal cost-to-go is given. Such table is built backward in time, i.e. we start from the terminal cost (that's known given the parametric final state x_N) and, going backward in time, we compute each value in the table in order to minimize the overall cost-to-go given the current configuration.
Finally at run-time the optimal controls $u^\star$ are chosen in such a way that they minimize the cost-to-go toward the final target.

The main advantage of this algorithm is that it doesn't provide an optimal trajectory, but rather a *feedback policy*: once the look-up table is filled, then for each state the system

reach at run-time the optimal behavior can be automatically determined.
This comes however with a big issue: the parametric minimization itself. This method to work requires the generation of a look-up table that suffers of dimensionality based on the discretization used to describe the states and the inputs as well as on time.

In the simplified case where the systems dynamic is linear and the cost is simply a quadratic function in x and u, then the value function V is convex and a simpler closed-form solution can be obtained, leading to the sub-class problems of the so called *(discrete-time) linear quadratic regulators* (*LQR*), described in more depth in section 3.5 (page 33).

3.2 Hamilton-Jacobi-Bellman

As the name suggest, the *Hamilton-Jacobi-Bellman* (*HJB*) method is based on the Bellman's optimality principle and solves the optimal trajectory directly in continuous-time (while in dynamic programming a discretization was required). The underlying ides of HJB is to take DP and push the discretization step toward an infinitesimal value.

Considering the value function (3.7) as been obtained from a δ time-step discretization, then $V(\cdot)$ depends just on the current time t_i and the parametric state

$$V(t_i, x) = \min_u \ell_d(x,u) + V\big(t_{i+1}, f(x,u)\big)$$

where $\ell_d = \ell(x,u)\delta$ is the discretized cost. Pushing δ towards 0, then the value function can be approximated by it's first Taylor series expansion as

$$\begin{aligned} V(t_i, x) &= \min_u \ell_d(x,u) + V(t_i, x) + \frac{\partial V}{\partial t}\delta + \frac{\partial V}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t}\delta \\ 0 &= \min_u \ell(x,u)\delta + \dot{V}\delta + \nabla_x V(t_i,x) f(x,u)\delta \end{aligned}$$

that divided by the common term δ allows us to reach the *Hamilton-Jacobi-Bellman equation*

$$-\dot{V} = \min_u \ell(x,u) + \nabla_x V\, f(x,u) \tag{3.8}$$

This expression models the Bellman's optimality principle in a continuous-time setting; the main issue is that (3.8) is not a ordinary differential equation (ODE), but a partial differential equation (PDE) since it depends on the differentiation of the unknown function V with respect to both time t and states x.

Still if the dynamic is linear and the cost is quadratic, the problem simplifies to the solution of a continuous-time LQR (see sec. 3.5).

HJB solvers still exploits the backward integration from the final cost $V(T, x_T) = \ell_f(x_T)$ to determine the optimal feedback policy, but the problem is now harder to numerically integrate.

3.3 Pontryagin minimum principle

Methods based on the *Pontryagin minimum principle* starts from the HJB equation (3.8) and is based on the observation that the optimal control $u^\star$ depends just on the gradient $\nabla_x V$ of the value function, and not by $V(\cdot)$ itself. Introducing the *adjoint variable* $\lambda(t)$ defined as $\Big(\nabla_x V\big(t, x(t)\big)\Big)^\top$, then the optimal control can be computed as

$$u^\star(t, x, \lambda) = \underbrace{\min_u \ell(x,u) + \lambda^\top f(x,u)} \tag{3.9}$$

TODO: Rivedere bene il PMP e capire come funziona e come l'equazione è derivata

3.4 Direct methods

Direct methods (*DM*) is a family of optimal control solvers highly used in robotic applications; they mainly search for local optima trajectories of a discretized optimal control problem from the continuous-time one.

As seen so far, the main issue associated to the solution of optimal control problems is related to the high dimensionality (due to the time-continuity of the dynamics). The main idea in direct methods is to discretize the time axis and define a parametrization (such a piecewise constant function) for the inputs u inside such interval.
After discretization the OCP becomes an optimization problem that can be solved by means of *non-linear programming* (*NLP*) *solvers*. Direct methods works as follows:

- once the time-axis is discretized, we parametrize the trajectories of the decision variables of the problem (inputs and/or states); for this purpose we can use piece-wise constant functions or polynomials (or any other function);
- we enforce the constraints (due to both dynamic and path constraints) on the defined time-grid of times $t_0 < t_1 < \cdots < t_N$.

Intuitively this is the simplest algorithm that we might come up with (compared also with the solution previously discussed), however we have to deal with issue of *over-discretization* (leading to a numerical cost growth) or *under-discretization* (smaller dimensionality but at runtime some constraints might be violated since they are enforced only in few discrete time-steps).
With respect to all other presented solution, direct methods are overall faster and for this very reason are used in robotic also in feedback loop; for this last point it's crucial to develop algorithms that are able to solve OCPs fast enough to keep up with the robot's dynamic.

There are mainly 3 algorithms for solving OCP problems with direct methods:

- *single shooting* (or *sequential*): the discretization is performed on the controls trajectory $u(t)$ that's so is the lonely decision variable; the state trajectory $x(t)$ is instead obtained by integration of the dynamics (that so can be removed from the constraints since the condition is always satisfied by construction);
- *collocation* (or *simultaneous*): the discretization is performed on both inputs u and states x; in this case the dynamic is used to enforce the constraint between such trajectories;
- *multiple shooting*: this can be regarded as a combination of both single shooting and collocation since the states x are discretized on a coarser time-grid (with respect to the inputs u). Intermediate values for the states are compute by integration. Such method can be regarded as a sequence of single shootings.

The choice of the method that should be used heavily depends on the NLP solver chosen; collocation leads to a problem with bigger matrices that are however sparser, so some software might exploit such aspect; conversely, single shooting have a lower system's dimensionality but require a good integrator to accurately compute the state trajectories.

3.4.1 Single shooting

In *single shooting* we discretize only the control $u(t)$ on a fixed time-grid $0 = t_0 < t_1 < \cdots < t_N = t_f$; for sake of simplicity u is usually parametrized as piece-wise within the integration time-step. Called y the decision variable (that describes the parametrization of u) and once the system can be integrated given such parameters, then the NLP problem that needs to be solved is of the form

$$\begin{aligned} \min_{y} \quad & \int_0^{t_f} \ell\big(x(t_i, y), u(t_i, y), t\big)\, \mathrm{d}t + \ell_f\big(x(t_f, y)\big) \\ \text{sub. to:} \quad & g\big(x(t_i, y), u(t_i, y), t_i\big) \qquad \forall i \in [0, N-1] \end{aligned} \tag{3.10}$$

where in this case g is the *discretized path constraints*. The dynamic is not included since it's knowledge has been used to compute the state trajectory $x(t_i, y)$. Regarding the running cost as a time-dependent function of the form

$$c(t) = \int_0^{\top} \ell(\cdot)\, \mathrm{d}t$$

then the related computation can be reduced to the integration of the following ordinary differential equation:

$$\begin{cases} \dot{c}(t) = \ell(\cdot) \\ c(0) = 0 \end{cases}$$

Calling $\overline{x} = (x, c)$ the *augmented state vector*, then the overall ordinary differential equation that must be integrated to solve the OCP (3.10) is

$$\dot{\overline{x}} = \begin{pmatrix} f(x, u, t) \\ \ell(x, u) \end{pmatrix} \tag{3.11}$$

Once this expression is integrated by means of an integrator and its derivative with respect to the decision variable u is computed, then NLP solvers might be able to converge to the local minima by exploiting a gradient descent.

3.4.2 Numerical integration

Given a ordinary differential equation in the form $\dot{x} = f(x, t)$ subjected to an initial condition $x(0) = x_0$, then numerical integrators deals with the computation of $x(t)$ for all times $t \in [0, T]$; f in this case is not written as dependent from the input since when we evaluate the dynamic u has a fixed and known value, i.e. integration is not performed parametrically.

As long as f is *Lipshitz continuous* (a stronger condition then requiring a continuous first derivative of f), then it is proven that the ODE has a unique solution in the neighborhood of the initial point. Based on this assumption we can numerically integrate by using different algorithms, such the Euler method.

Euler method Recalling the formal definition of the incremental ratio

$$\dot{x} = f(x, t) = \lim_{h \to 0} \frac{x(t+h) - x(t)}{h}$$

the Euler method uses the approximation for a "sufficiently small" h to compute the next state $x(t+h)$ given the initial condition $x(t)$ simply by reverting the equation:

$$x(t+h) \approx x(t) + h\, f(x, t) \tag{3.12}$$

This method to be accurate requires h to be "very small": accuracy is so inversely proportional to the numerical time complexity (to be more accurate we need lower h, increasing the number of required evaluation to cover the same time span). As we'll see better later, this is called a "1^{st} order method", so with a minimum consistency; in general higher the order, more favorable is the trade-off between time complexity and accuracy.

Midpoint method The midpoint method is another integration algorithm that falls in the category of the 2^{nd} order. Once we compute

$$K_1 = f\big(x(t), t\big)$$

and

$$K_2 = f\left(x(t) + \frac{h}{2}K_1, t + \frac{h}{2}\right)$$

then the state at the next time-step is approximated by the formula

$$x(t+h) \approx x(t) + hK_2 \tag{3.13}$$

The idea behind this formula is that first we compute the slope K_1 and use half of a Euler step to approximate a "mean" slope K_2 inside the discretization time-step. Given the same value of h, the midpoint method is usually more accurate then Euler, but it also requires a double value of function f evaluation (2 vs 1).

Properties of numerical integration Given an ODE with exact trajectory solution $x(t)$ and an integrator whose output is described by $\hat{x}\big(t, t_0, x(t_0)\big)$ where $t_0, x(t_0)$ are respectively the time and initial condition for starting the integration, then we define the *local error* $e(\cdot)$ as

$$e(t) = x(t) - \hat{x}\big(t, t-h, x(t-h)\big) \tag{3.14}$$

while the *global error* $E(\cdot)$ is simply

$$E(t) = x(t) - \hat{x}\big(t, t_0, x(t_0)\big) \tag{3.15}$$

As the name suggests, the global error evaluates the "distance" between the truth value and the integrated given the same initial condition, while the local error measures the distance considering as initial true condition just one time-step ahead.

With this definitions any "good" numerical integration scheme must satisfy the following 3 conditions:

i) *convergence*, i.e. $\lim_{h\to 0} E(t) = 0$;

ii) *consistency order*: $\lim_{h\to 0} e(t) = \mathcal{O}(h^{p+1})$ where $p > 0$ is the *order* of the numerical integration; such parameter is key since it rules the asymptotic convergence of the system;

iii) *stability*. This concept is now well defined, but intuitively we can say that the global error E must be bounded for $t \to \infty$; this requirement is important especially for so called "stiff" ODEs.

Consistency order of the explicit Euler method Rewriting the local error (3.14) in a more readable index notation $e(t) = x_{n+1} - \hat{x}_{n+1}(x_n) = e_{n+1}$, the definition of $\hat{x}_{n+1}$ given by the Euler method is simply $x_n - hf(x_n, t_n)$. Considering now the Taylor series expansion of the true solution around x_n that's

$$x_{n+1} = x_n + h\dot{x}_n + \mathcal{O}\big(h^2\big)$$

then we can show that

$$\begin{aligned} e_{n+1} &= x_{n+1} - \hat{x}_{n+1} = x_n + h\dot{x}_n + \mathcal{O}\big(h^2\big) - x_n - h\,f(x_n, t_n) \\ &= \mathcal{O}\big(h^2\big) \end{aligned}$$

proving that the consistency order is actually $p = 1$.

Runge-Kutta methods Both Euler and midpoint methods can be regarded as "instantiations" of the more general definition of the *Runge-Kutta* (*RK*) *methods*, one-step integration algorithms of the form

$$x_{n+1} = x_n + h \sum_{i=0}^{q} b_i K_i \tag{3.16}$$

with

$$K_i = f\left(x_n + h \sum_{i=0}^{q} a_{ij} K_j, t_n + c_i h \right)$$

where a_{ij}, b_i, c_i are all tabled coefficients; q is the so called *order* of the Runge-Kutta method that in general might differ from the consistency order p. Usually we call *RK4* a Runge-Kutta method of order $q = 4$.
It's true that $p \leq q$, and in particular for $q \geq 5$ the consistency order is always lower than the method's one.

As example, the explicit Euler method is a RK1 integration scheme with $b_1 = 1$, $a_{11} = 1$ and $c_1 = 0$; the midpoint integrator is instead a RK2 scheme.

Every-time it happens that $a_{ij} = 0$ for all $j \geq i$, then the *method* is said *explicit* and the integration can be performed by simple evaluations of the function f (while for *implicit methods* it is required to solve an implicit non-linear problem in x_{t+1}, increasing the cost complexity).

Runge-Kutta methods are unequivocally described by the so called *Butcher tableaus* storing the parameters for the integration in a form:

$$\begin{array}{c|ccc} c_1 & a_{11} & \dots & a_{1q} \\ \vdots & \vdots & \ddots & \\ c_q & a_{q1} & & a_{qq} \\ \hline & b_1 & \dots & b_q \end{array} \tag{3.17}$$

The best trade-off between accuracy and numerical complexity is favorable up to order $q = 4$ (to ge a consistency order $p = 5$ it is required at a RK6 method); one of the most widely used Buther tableau is the RK4 explicit method described as

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

3.4.3 Computation of sensitivities

Optimal control problems, as already mentioned, are usually solved exploiting gradient-based search thus the computation of the derivative is a key concept. Any OCP (3.1) has a cost that needs to be minimized: this value usually depends on the state trajectory $x(t)$ that is computed by integration of the dynamics considering the input $u(t)$.
In this section we will deal with the computation of the *sensitivities*, so the derivative of the result of an integration.

Given the sequence of piecewise constant inputs $u(t) = y_i$ (for any $t \in [t_i, t_{i+1}]$), the states are computed by integration of $\dot{x} = f(x, y_i, t)$ for all $t \in [t_i, t_{i+1}]$. Exploiting the Euler discretization, the cost function can be approximated as

$$c(y) = \int_0^{\top} \ell(x(t), u(t), t)\, \mathrm{d}t + \ell_f(x(T)) \approx \sum_{i=0}^{N-1} \ell(x_i, y_i) h + \ell_f(x_N)$$

In order to apply numerical optimization we can compute the gradient of $c(\cdot)$ with respect to the optimization variable y as

$$\begin{aligned}\frac{\mathrm{d}c(y)}{\mathrm{d}y} &= \sum_{i=0}^{N-1} \frac{\mathrm{d}\ell(x_i, y_i)}{\mathrm{d}y} h + \frac{\mathrm{d}\ell_f(x_N)}{\mathrm{d}y} \\ &= \sum_{i=0}^{N-1} \left(\frac{\partial \ell}{\partial x_i} \frac{\mathrm{d}x_i}{\mathrm{d}y_i} + \frac{\partial \ell}{\partial y_i} \right) h + \frac{\partial \ell_f}{\partial x_N} \frac{\mathrm{d}x_N}{\mathrm{d}y_i}\end{aligned} \tag{3.18}$$

Typically the cost function ℓ is relatively simple and it's partial derivative can be easily obtained analytically (since we can design ℓ); the main issue in (3.18) is that the derivative $\mathrm{d}x_i/\mathrm{d}y_i$ depends on the numerical integration of the state.
Since Runge-Kutta methods are single-step integration methods, they can be described using a so called *integration function* ϕ that determines the next step given just the current state and the applied input:

$$x_{i+1} = \phi_i(x_i, y_i)$$

whose derivative is simply

$$\frac{\mathrm{d}x_{i+1}}{\mathrm{d}y} = \frac{\partial \phi_i}{\partial x_i} \frac{\mathrm{d}x_i}{\mathrm{d}y} + \frac{\partial \phi_i}{\partial y} \tag{3.19}$$

In this equation we might observe that the derivative of x_{i+1} depends on it's previous state x_i; in this case if we firstly fix $\mathrm{d}x_0/\mathrm{d}y = 0$ (since the initial state is known and do not depend on the applied input), then each next derivative can be computed integrating forward in time. Furthermore we can see that the matrix

$$\frac{\partial \phi_i}{\partial y} = \begin{bmatrix} \frac{\partial \phi_i}{\partial y_0} & \frac{\partial \phi_i}{\partial y_1} & \cdots & \frac{\partial \phi_i}{\partial y_{N-1}} \end{bmatrix}$$

is sparse, since the next state depends just on the currently applied input (not the others), implying

$$\frac{\partial \phi_i}{\partial y_j} = 0 \qquad \forall i \neq j$$

The only element that still needs to be covered in (3.19) is the derivative $\partial \phi_i/\partial x_i$ whose value depends just on the integration scheme used.

Sensitivities of the explicit Euler method Let's consider the Euler method (3.12) defined by the integrator function

$$\phi(x, u) = x + h\, f(x, u)$$

In this case the sensitivities can be easily computed from the analytical definition as

$$\frac{\partial \phi}{\partial x} = I + h \frac{\partial f}{\partial x} \qquad\qquad \frac{\partial \phi}{\partial u} = h \frac{\partial f}{\partial u}$$

3.4.4 Collocation

Collocation is another category of direct methods that discretizes also the states $x(t)$ (in this case still with piecewise constant functions for simplicity). Calling

$$x(t) = s_i \qquad \forall t \in [t_i, t_{i+1}]$$

the discretized values for the states, then the dynamic $\dot{x} = f(x,u)$ can be approximated using the Euler method as

$$\underbrace{\overbrace{\frac{s_{i+1}-s_i}{t_{i+1}-t_i}}^{\approx \dot{x}} - f\Big(\overbrace{\frac{s_{i+1}+s_i}{2}}^{\approx x}, y_i\Big)}_{=c_i(s_i,s_{i+1},y_i)} = 0 \qquad \forall i \tag{3.20}$$

Considering this discretized problem, the cost integral for a given time-step can be regarded as

$$\int_{t_i}^{t_{i+1}} \ell\big(x(t),u(t)\big)\,\mathrm{d}t \approx \ell\left(\frac{s_i+s_{i+1}}{2}, y_i\right)\big(t_{i+1}-t_i\big) = \ell_i\big(s_i,s_{i+1},y_i\big)$$

Based on the presented discretization of the OCP (note that this is just one of the infinitely many that we might come up with), then an approximation of the problem that can be solved using NLP software is

$$\begin{aligned}
\min_{s,y} \quad & \sum_{i=0}^{N-1} \ell_i\big(s_i,s_{i+1},y_i\big) + \ell_f\big(s_N\big) & \\
\text{sub. to:} \quad & s_0 - x_0 = 0 & \text{: initial condition} \\
& c_i\big(s_i,s_{i+1},y_i\big) = 0 & \text{: discretized dynamics} \\
& g_i\big(s_i,y_i,t_i\big) = 0 & \text{: discretized constraints}
\end{aligned} \tag{3.21}$$

This problem has an higher number of decision variables with respect to single shooting (since we have to add the cardinality of y with the one of s), but comes with the advantage of having a *sparser problem*: each constraint in fact depends just con the current position and, at worst, a neighbor state, i.e.

$$\frac{\partial^2 c_i}{\partial s_i \partial s_{i\pm j}} = 0 \qquad \forall j > 1$$

3.4.5 Multiple shooting

Multiple shooting is a direct method that can be regarded as a combination of single shooing and collocation; from the latter it inherits the fact that the states are discretized, but in this case on a coarser time-grid with respect to the one of the controls. Inside the "bigger" state time-chunk, the dynamic is integrated to have a smooth trajectory.
By doing so no dynamic constraint should be considered in the OCP problem, but we have to ensure the state continuity $x_i(t_i) = s_i$ at each boundary of the state time-axis.

TODO: migliorare spiegazione e aggiungere

3.5 Linear quadratic regulator

Given a linear discrete-time system described as

$$x_{t+1} = Ax_t + Bu_t \tag{3.22}$$

we want to design a set of controls u_i for which:

- the resulting state trajectory $x(t)$ is "small", staying close to zero leading to a good regulation (subtly implying that $x(t)$ is modeling the error from a reference target);

- the control sequence u_i is also "small" in order to reduce the required actuator effort.

Intuitively such conditions are in contrast since one can't have a good regulation performance without asking a minimum amount of effort, so a trade-off between the two requirements is necessary. *Linear quadratic regulators* (*LQR*) deals with this by solving the following unconstrained minimization problem with costs in quadratic form:

$$\begin{aligned}\min_{U,X} \quad & \mathcal{J}(U,X) = \sum_{i=0}^{N-1}\left(x_i^\top Q x_i + u_i^\top R u_i\right) + x_N^\top Q_f x_N \\ \text{sub. to:} \quad & x_{t+1} = Ax_t + Bu_t\end{aligned} \tag{3.23}$$

where $Q, Q_f \geq 0$ and $R > 0$ are all symmetric matrices weighting respectively the cost of the states and the controls. Choosing Q "big" (with respect to R) tells that is more important having a good system regulation, while if R is dominant we want to rather bound the actuator effort instead of the regulation performance.

We can observe that (3.23) has a quadratic cost and is subjected to a linear constraint, so it's a *convex QP problem* as seen in sec. 2.4.1 (page 15). Describing the whole state dynamic in a vector $X = (x_0, \ldots, x_N)$ it can be shown that

$$\underbrace{\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix}}_{X} = \underbrace{\begin{bmatrix} 0 & & & & \\ B & 0 & & & \\ AB & B & 0 & & \\ \vdots & \vdots & & \ddots & \\ A^{N-1}B & A^{N-2}B & & B & 0 \end{bmatrix}}_{G} \underbrace{\begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-1} \end{pmatrix}}_{U} + \underbrace{\begin{bmatrix} I \\ A \\ A^2 \\ \vdots \\ A^N \end{bmatrix}}_{H} x_0 \tag{†}$$

where the matrix G has been obtained recursively applying (3.22) and collecting all elements in a matrix form. With this notation (3.23) can be compactly rewritten as

$$\begin{aligned}\min_{U,X} \quad & \mathcal{J}(U,X) = \|\overline{Q}X\|^2 + \|\overline{R}U\|^2 \\ \text{sub. to:} \quad & x_{t+1} = Ax_t + Bu_t\end{aligned} \tag{3.24}$$

where

$$\overline{Q} = \text{diag}\{Q^{1/2}, \ldots, Q^{1/2}, Q_f^{1/2}\} \qquad \overline{R} = \text{diag}\{R^{1/2}, \ldots, R^{1/2}\}$$

Moreover we can substitute the definition of the dynamics (†) and restate the problem as the following unconstrained minimization:

$$\min_{U} \mathcal{J}(U) = \|\overline{Q}(GU + Hx_0)\|^2 + \|\overline{R}U\|^2 = \left\| \begin{bmatrix} \overline{Q}G \\ \overline{R} \end{bmatrix} U + \begin{bmatrix} \overline{Q}Hx_0 \\ 0 \end{bmatrix} \right\|^2 \tag{3.25}$$

Recalling (2.19), page 16, the solution of this problem can be obtained by means of the Moore-Penrose pseudo-inverse; since the matrix that needs to be inverted has size $N(n+m) \times Nm$, the overall numerical complexity for QP solvers is $\mathcal{O}(N^3 nm^3)$.

Solving such problem with QP solvers is highly inefficient, so for this reason dynamic programming (see sec. 3.1, page 24) is usually used to reduce the computational cost to $\mathcal{O}(Nn^3)$. The idea is that if we are able, given the value function $V_{t+1}(z)$, to compute $V_t(z)$, then we can start from V_N and go backward in time to compute the optimal trajectory.
We start off defining the value function as

$$\begin{aligned}V_t(z) = \min_{w, U_{t+1}} \quad & z^\top Qz + w^\top Rw + \sum_{k=t+1}^{N-1}\left(x_k^\top Q x_k + u_k^\top R u_k\right) + x_N^\top Q_f x_N \\ \text{sub. to:} \quad & x_{k+1} = Ax_k + Bu_k \\ & u_t = w\end{aligned}$$

and we observe the recursive definition

$$V_t(z) = \min_w \underbrace{z^\top Qz + w^\top Rw}_{\ell(z,u)} + V_{t+1}(\underbrace{Az + Bw}_{f(z,w)}) \tag{3.26}$$

Observing the similarity with (3.5), ℓ is the cost that we pay at the current time considering the input $u_t = w$, while the second term is the cost-to-go given the state that we land in. The main advantage now in LQR (with respect to the general case of dynamic programming) is the linear dynamic that will help us reach an easier solution.

By construction the value function V_{t+1} can always be regarded as quadratic and simplified to the notation

$$V_{t+1}(z) = z^\top P_{t+1} z \qquad \text{with } P_{t+1} = P_{t+1}^\top \geq 0,\ P_N = Q_f$$

Observed that this definition surely holds for the last time-step, we can elaborate (3.26) as

$$\begin{aligned}
V_t(z) &= \min_w \left(z^\top Qz + w^\top Rw + V_{t+1}(Az + Bw)\right) \\
&= z^\top Qz + \min_w \Big(w^\top Rw + \left(Az + Bw\right)^\top P_{t+1}\left(Az + Bw\right)\Big) \\
&= z^\top Qz + \min_w \left(w^\top (R + B^\top P_{t+1} B)w + 2z^\top A^\top P_{t+1} Bw\right) + z^\top A^\top P_{t+1} Az \\
&= z^\top \left(Q + A^\top P_{t+1} A\right) z + \min_w \big(\underbrace{w^\top Hw + 2g^\top w}_{f(w)}\big)
\end{aligned}$$

where to simplify the notation it has been used $H = R + B^\top P_{t+1} B$ and $g^\top = z^\top A^\top P_{t+1} B$. From the last equality we clearly see that the value function always have a fixed cost based on the parameter z, while the second term can actually be minimized in the control input w. Being f a convex quadratic function we can exploit 2.19 to determine the optimal control input as

$$w^\star = -H^{-1} g = -\left(R + B^\top P_{t+1} B\right)^{-1} B^\top P_{t+1} Az = K_t z \tag{3.27}$$

Plugging back the optimal solution inside the definition of the value function provides

$$\begin{aligned}
V_t(z) &= z^\top \left(Q + A^\top P_{t+1} A\right) z + g^\top H^{-1} H H^{-1} - 2g^\top H^{-1} g \\
&= z^\top \underbrace{\left(QA^\top P_{t+1} A - A^\top P_{t+1} B H^{-1} B^\top P_{t+1} A\right)}_{P_t = P_t^T \geq 0} z
\end{aligned} \tag{3.28}$$

With this we proved that if V_{t+1} is a quadratic function, then also V_t must be so; furthermore it happens that P_t is always symmetric and semi-positive definite.

The main advantage of solving an LQR problem with dynamic programming is that the *optimal control input u* turns out to be not an *open-loop trajectory*, but a *linear feedback policy* that depends on a time-varying matrix K_t as shown in (3.27).

Algorithm 2 shows the pseudo-code to solve the LQR problem, summarizing the procedure yet described.

Time-varying system The definition of the LQR algorithm works fine also with time-varying systems $x_{t+1} = A_t x_t + B_t u_t$; the lonely drawback is in this case is that no steady-state regulation exists.

Algorithm 2 pseudo-code for solving the linear quadratic regulator problem.

```
initialize $P_N = Q_f$

for $t$ from $N$ to 1 do                                        ▷ backward pass
    $P_{t-1} = Q + A^\top P_t A - A^\top P_t B\big(RB^\top P_t B\big)^{-1} B^\top P_t A$
end for

for $t$ from 0 to $N-1$ do                                       ▷ forward pass
    $K_t = -\big(R + B^\top P_{t+1} B\big)^{-1} B^\top P_{t+1} A$
    $u_t^\star = K_t x_t$                                        ▷ at runtime
end for
```

3.5.1 Steady-state regulation

While running the algorithm we usually observe that for N "large", the majority of the time-steps shows a constant feedback matrix K_t that converges to zero only toward the end of the OCP time horizon. This initial constant solution is called *steady-state regulator* that's associated to the solution of the *discrete-time algebraic Riccati equation* defined as

$$P_{ss} = Q + A^\top P_{ss} A - A^\top P_{ss} B\big(RB^\top P_{ss} B\big)^{-1} B^\top P_{ss} A \tag{3.29}$$

The solution of P_{ss} can be obtained by means of direct or recursive method and allows to compute the constant feedback

$$K_{ss} = -\big(R + B^\top P_{ss} B\big)^{-1} B^\top P_{ss} A$$

that can be use as long as we are not close to the end of the time horizon.

3.5.2 Inhomogeneous systems and tracking problems

The linear quadratic regulator can be solved also to the extend problem of *inhomogeneous systems*, so when at both cost and dynamic we add a fixed term, so considering the following NLP:

$$\begin{aligned} \min_{U,X} \quad &= \sum_{t=0}^{N-1} \begin{pmatrix} x_t^\top & u_t^\top & 1 \end{pmatrix} \begin{bmatrix} Q_t & S_t & q_t \\ S_t^\top & R_t & s_t \\ q_t^\top & s_t^\top & 0 \end{bmatrix} \begin{pmatrix} x_t \\ u_t \\ 1 \end{pmatrix} + \begin{pmatrix} x_N^\top & 1 \end{pmatrix} \begin{bmatrix} Q_f & q_N \\ q_N^\top & 0 \end{bmatrix} \begin{pmatrix} x_N \\ 1 \end{pmatrix} \\ \text{sub. to:} \quad & x_{i+1} = Ax_i + Bu_i + c_i \\ & x_0 = x^{\text{init}} \end{aligned} \tag{3.30}$$

The solution of such regulator is achieved by applying a proper transformation that turns the problem into the yet studied formulation in (3.23). Defining the "augmented states" $\overline{x} = (x, 1)$, then the discrete-time dynamic can be rewritten as

$$\overline{x}_{t+1} = \begin{pmatrix} x_{t+1} \\ 1 \end{pmatrix} = \begin{bmatrix} A_t & c_t \\ 0 & 1 \end{bmatrix} \begin{pmatrix} x_t \\ 1 \end{pmatrix} + \begin{bmatrix} B_t \\ 0 \end{bmatrix} u_t = \overline{A}_t \overline{x}_t + \overline{B}_t u_t$$

while the cost becomes

$$\mathcal{J} = \sum_{t=0}^{N-1} \begin{pmatrix} \overline{x}_t^\top & u_t^\top \end{pmatrix} \begin{bmatrix} \overline{Q}_t & \overline{S}_t \\ \overline{S}_t^\top & R_t \end{bmatrix} \begin{pmatrix} \overline{x}^\top \\ u_t \end{pmatrix} + \overline{x}_N^\top \overline{Q}_f \overline{x}_N$$

with

$$\overline{Q}_t = \begin{bmatrix} Q_t & q_t \\ q_t^\top & 0 \end{bmatrix} \qquad \overline{S}_t = \begin{bmatrix} S_t \\ s_t^\top \end{bmatrix}$$

The lonely difference with respect to the "standard" LQR case is that now in the cost there's also a mixed cost term $x^\top S u$ (previously not present).

Tracking problem Let's assume that there are two reference trajectories $x^{\text{ref}}, u^{\text{ref}}$ for both states and inputs; the inhomogeneous formulation (3.30) of the LQR problem can be also used to optimize the tracking of such references. Considering the cost

$$\mathcal{J} = \sum_{t=0}^{N-1} \left(x_t - x_t^{\text{ref}}\right)^\top Q_t \left(x_t - x_t^{\text{ref}}\right) + \left(u_t - u_t^{\text{ref}}\right)^\top R_t \left(u_t - u_t^{\text{ref}}\right)$$

we can reduce to the inhomogeneous formulation by simply expanding the quadratic terms:

$$\begin{aligned} \mathcal{J} &= \sum_{t=0}^{N-1} x_t^\top Q_t x_t + u^\top R u - 2x_t^{\text{ref}} Q x - 2{u_t^{\text{ref}}}^\top R u \\ &= \sum_{t=0}^{N-1} \begin{pmatrix} x^\top & u^\top & 1 \end{pmatrix} \begin{bmatrix} Q & 0 & Qx_t^{\text{ref}} \\ 0 & R & Ru_t^{\text{ref}} \\ {x_t^{\text{ref}}}^\top Q & {u_t^{\text{ref}}}^\top R & 0 \end{bmatrix} \begin{pmatrix} x \\ u \\ 1 \end{pmatrix} \end{aligned}$$

3.6 Differential dynamic programming

Differential dynamic programming (*DDP*) can be regarded as an extension of the linear quadratic regulator to non-linear dynamics and cost. Solutions are obtained by linearization of the problem, and for this reason convergence is not ensured, but the underlying step ideas of the algorithms are as follows:

1. linearize the cost around the current trajectory;
2. solve the LQR problem to get the variation of the controls u;
3. perform a line-search to ensure convergence.

Denoting with U_i the set of control inputs $(u_i, \dots, U_{N-1})$, we generally define the value function simply as

$$V(z, i) = \min_{U_i} \mathcal{J}_i(z, U_i)$$

and exploiting the Bellman's optimality principle it leads to the following recursive definition:

$$V(z, i) = \min_{w} \underbrace{\Big(\ell_i(z, w) + V\big(f(z, w), i+1\big)\Big)}_{Q(z,w)}$$

The dependency on w and z of Q is usually non-linear, however given a reference trajectory $(\overline{x}, \overline{u})$ it's possible to perform a linearization up to the second order (in order to get a quadratic cost) exploiting the Taylor series expansion:

$$Q(\overline{x} + z, \overline{u} + w) \approx Q(\overline{x}, \overline{u}) + \begin{bmatrix} Q_x^\top & Q_u^\top \end{bmatrix} \begin{pmatrix} z \\ w \end{pmatrix} + \frac{1}{2} \begin{pmatrix} z^\top & w^\top \end{pmatrix} \begin{bmatrix} Q_{xx} & Q_{xu} \\ Q_{xu}^\top & Q_{uu} \end{bmatrix} \begin{pmatrix} z \\ w \end{pmatrix} \tag{3.31}$$

where with the subscripts[2] we intend the differentiation operation defined as

$$Q_i = \frac{\partial Q}{\partial i} \qquad\qquad Q_{ij} = \frac{\partial^2 Q}{\partial i\, \partial j}$$

with Q_i vector and Q_{ij} a matrix. Further denoting with V' the value function $V(\cdot, i+1)$ evaluated at the next time step, we can explicitly compute the different terms of expansion of Q in terms of the cost ℓ and dynamic f:

$$\begin{aligned} Q_x &= \ell_x + f_x^\top V'_x \\ Q_u &= \ell_u + f_u^\top V'_x \\ Q_{xx} &= \ell_{xx} + f_x^\top V'_{xx} f_x + \cancel{V'_x f_{xx}} \\ Q_{uu} &= \ell_{uu} + f_u^\top V'_{xx} f_u + \cancel{V'_x f_{uu}} \\ Q_{xu} &= \ell_{xu} + f_x^\top V'_{xx} f_u + \cancel{V'_x f_{xu}} \end{aligned} \tag{3.32}$$

In this expression some terms are neglected since the evaluation of the derivatives f_{ij} will lead to *tensors* (3D matrices) that are hard to deal with; furthermore it has been observed the contribution of such elements is in practice negligible.

The optimal cost of the linearized quadratic expression (3.31) is found as

$$w^\star = \min_w Q(z, w) = -Q_{uu}^{-1}\big(Q_u + Q_{ux} z\big) = \overline{w} + Kz \tag{3.33}$$

Substituting back this optimal result in 3.31 provides us

$$\begin{aligned} V(z,i) &= \overline{Q} + Q_x^\top z + Q_u w^\star + \frac{1}{2} z^\top Q_{xx} z + \frac{1}{2} w^{\star\top} Q_{uu} w^\star + \frac{1}{2} z^\top Q_{xu} w^\star = \dots \\ &= \Delta V + V_x z + \frac{1}{2} z^\top V_{xx} z \end{aligned} \tag{3.34}$$

with

$$\begin{aligned} \Delta V &= \overline{Q} - \frac{1}{2} Q_u^\top Q_{uu}^{-1} Q_u \\ V_x &= Q_x - Q_{xu} Q_{uu}^{-1} Q_u \\ V_{xx} &= Q_{xx} - Q_{xu} Q_{uu}^{-1} Q_{ux} \\ \overline{Q} &= Q(\overline{x}, \overline{u}) + V'(\overline{x}, \overline{u}) = \overline{\ell} + \overline{V}' \end{aligned} \tag{3.35}$$

Regularization Looking at the terms in (3.35) it can be easily observe that their computation relies on the inversion of the matrix Q_{uu} that, unluckily, can't be guaranteed a-priori. In general to avoid the possibility of having singular matrices that needs to be inverted, we can perform a *regularization* by adding a rescaled identity matrix:

$$\overline{Q}_{uu} = Q_{uu} + \mu I \tag{3.36}$$

with μ "small". This do not provide any mathematical guarantee that $\overline{Q}_{uu}$ is actually invertible, but numerically it becomes more likely. Intuitively one can regard such regularization as adding a new term in the linearized cost proportional to the input itself, i.e.

$$\overline{\ell}(\overline{x} + z, \overline{u} + w) = \ell(\overline{x} + z, \overline{u} + w) + \frac{1}{2}\mu \|w\|^2$$

[2]notation that will be also used later while defining the value function V_i, V_{ij} and the costs ℓ_{ij}.

Based on the regularization, terms in (3.35) are reconsidered as

$$\begin{aligned}\Delta V &= \overline{Q} + Q_u^\top \overline{w} + \frac{1}{2}\overline{w}^\top Q_{uu}\overline{w} \\ V_x &= Q_x^\top + Q_u^\top K + \overline{w}^\top Q_{uu}K + \overline{w}^\top Q_{xu}^\top \\ V_{xx} &= Q_{xx} + K^\top Q_{uu}K + Q_{xu}K + K^T Q_{xu}^T\end{aligned} \tag{3.37}$$

with

$$\overline{w} = -\overline{Q}_{uu}^{-1} Q_u \qquad K = -\overline{Q}_{uu}^{-1} Q_{ux} \tag{3.38}$$

that are based on the optimal control solution (3.33) exploiting the regularized matrix.

3.6.1 Iterative LQR

One way to solve differential dynamic programming is by means of the *iterative linear quadratic regulator* (*iLQR*) algorithm, summarized as follows:

1. given an initial set of control inputs U and the dynamic f of the system, we simulate the sequence fo states X;
2. we perform a *backward pass*: exploiting the definition of the value function, we start from the end and compute the updated values of the inputs and the corresponding gains;
3. wer perform a *forward pass* with a line-search to find the optimal update step.

Steps 2 and 3 are iterated until convergence. Algorithm 3 presents the whole procedure of the iLQR.

Line-search Once the backward pass is done we have a set of variational inputs $\overline{w}$ that can be applied to the yet applied inputs $\overline{U}$; the goal now is to estimate "how much times" (α) of such quantity we should add to obtain the best cost reduction.

To do so we firstly need to compute the cost given by the value function at time 0 considering a state variation $x - \overline{x} = z = 0$ that evaluates to

$$V_0(0) = \Delta V_0 =$$

TODO: finire di spiegare l'algoritmo e aggiornare lo pseudo-codice per il costo e regolarizzazione dinamica

3.7 Model predictive control

In *model predictive control* (*MPC*) we use the resulting optimal control for controlling a system/robot inside a control loop. The main idea is to use the current state as initial condition for solving a finite horizon OCP and apply just the first torque computed (not the whole trajectory); at the next time-step the controller will solve the same problem but with a new initial condition.

Algorithm 3 pseudo-code for solving the differential dynamic programming problem using the iterative LQR algorithm.

```
Require: $U$ initial set of inputs, $f$ dynamic of the system, $x_0$ initial state
  $X$        $\leftarrow f(U, x_0)$
  $\mu$        $\leftarrow \mu_0$

  while is not converged do
    $\overline{U}$        $\leftarrow U$                                   ▷ store reference input trajectory
    $\overline{X}$        $\leftarrow X$                                   ▷ store reference state trajectory
    $c_0$        $\leftarrow \ell(\overline{X}, \overline{U})$                            ▷ cost with current trajectories

    $V_x(N)$   $\leftarrow \nabla_x \ell_N$                                       ▷ backward pass steps
    $V_{xx}(N)$  $\leftarrow \nabla_{xx} \ell_N$
    for $i$ from $N-1$ to 0 do
      compute $Q_x, Q_u, Q_{xx}, Q_{uu}, Q_{xu}$ as in (3.32)
      $\overline{Q}_{uu}$     $\leftarrow Q_{uu} + \mu I$                               ▷ regularization
      $\overline{w}$       $\leftarrow -\overline{Q}_{uu}^{-1} Q_u$                                 ▷ eq. (3.38)
      $K$       $\leftarrow -\overline{Q}_{uu}^{-1} Q_{ux}$
    end for

    $\alpha$        $\leftarrow 1$                                        ▷ forward pass steps
    for $k$ from 1 to $k_{max}$ do                                ▷ perform a line-search
      $X, U$     $\leftarrow$ simulate system with input $\overline{U} + \alpha \overline{w} + K(x - \overline{x})$
      $c$       $\leftarrow \ell(X, U)$
      $\Delta \ell(\alpha)$   $\leftarrow \Delta V_0 - \ell(\overline{U})$
      if $c < \gamma c_0$ then
        exit the for loop
      else
        $\alpha$       $\leftarrow k_\alpha \alpha$                                  ▷ $k_\alpha \in (0, 1)$
      end if
    end for

    if $k = k_{max}$ then
      algorithm has converged
      return $U, K$                       ▷ open loop control and feedback gains
    end if
  end while
```

Considering an already-discretized OCP problem, MPC deals with the solution of the following problem at each time-step:

$$
\begin{aligned}
X^\star, U^\star = \arg\min_{X,U} \quad & \sum_{k=0}^{N-1} \ell\big(x_k, u_k\big) \\
\text{sub. to:} \quad & x_{k+1} = f\big(x_k, u_k, k\big) \quad k = 0, \dots, N-1 \\
& x_{k+1} \in \mathcal{X}, u_k \in \mathcal{U} \quad k = 0, \dots, N-1 \\
& x_0 = x^{\text{meas}}
\end{aligned}
\tag{3.39}
$$

At each time-step the applied control is simply $\tau = u_0^\star$, while the rest of both $U^\star, X^\star$ is unused (numerically to improve convergence speed, we might use the yest computed $U^\star$ as initial guess for the next OCP).

Challenges in model predictive control This control technique seems promising as it solves at each iteration the optimal control possible, however we have to tackle few challenges that are not that irrelevant, in particular

i) *feasibility* deals with the fact that OCPs might be not always feasible, so that we might reach states where not all constraints can be satisfied. To avoid this issue we have to ensure *recursive feasibility;*

ii) *stability* deals with the stabilization of the system itself. In fact at each iteration the OCP looks on a different and "increased" time-horizon after which the problem is solved. It might happen in fact that in order to "stabilize in the future", the system tends to initially diverge from the desired configuration: if this process is iterated we might go toward a system instability (and not stability);

iii) *computation time*: solving OCP problem requires time, and if we think to re-compute everything at each iteration we need to do it fast.

3.7.1 Feasibility

Infinite horizon MPC Let's consider a case where at each iteration the time horizon is set to infinite ($N = \infty$): this means that each time the OCP sees always the "same amount of time". Assuming that there are no disturbances, predicted and actual trajectories computed each time are the same, since they follow from the Bellman's optimality principle.

If we now consider a cost $\ell(x, u) \geq \alpha \|x\|$ for any combination x, u with $\alpha > 0$, then having a finite cost implies that the system is stable (since eventually x will set to zero, thus no cost is added).
Moreover, since predicted and actual trajectories are equal, it turns out that we have *recursive feasibility* along a closed-loop trajectory.

The main idea now is to add a terminal cost and a constraint to a finite horizon OCP in order to "mimic" an infinite horizon problem, reminiscing the concept of the value function.

Terminal cost and constraint With the idea just said, the problem that we want to solve in order to improve feasibility becomes

$$\begin{aligned} V_N^\star(x_0) = \arg\min_{X,U} \quad & \sum_{k=0}^{N-1} \ell(x_k, u_k) + \ell_f(x_N) \\ \text{sub. to:} \quad & x_{k+1} = f(x_k, u_k, k) && k = 0, \dots, N-1 \\ & x_{k+1} \in \mathcal{X}, u_k \in \mathcal{U} && k = 0, \dots, N-1 \\ & x_0 = x^{\text{meas}} \\ & x_N \in \mathcal{X}_f && \text{terminal constraint} \end{aligned} \tag{3.40}$$

The main objective now is concerning the generation of both $\ell_f(\cdot)$ and $\mathcal{X}_f$.

Feasibility If an OCP is subjected to only input constraints, then it happens that it's always feasible; on the other hand, if the problem is subjected to hard state constraint, if $N < \infty$ there's no guarantee that the OCP remains feasible, even while staying in the nominal case.

The *maximum output admissible theory* tells us that in general a limited amount of horizon time-steps $N < \infty$ is enough to guarantee the problem feasibility; in particular it's necessary to use a *maximal control invariant set* as a terminal constraint to ensure that the closed-loop convergence. Such theory is based on the following theorem:

« *If $\mathcal{X}_f$ is a control invariant set, then the model predictive control problem is recursively feasible.* »

In particular

« *a set $\mathcal{S}$ is control invariant if $\forall x \in \mathcal{S}$ there exists an input $u \in \mathcal{U}$ such that $f(x, u) \in \mathcal{S}$.* »

i.e. when we start in the set $\mathcal{S}$ it's always possible to stay there.

In practice computing control invariant sets is generally hard, specially for non-linear systems. For linear system with dynamic $x^+ = Ax$ and output $y = Cx$ with $y \in \mathcal{Y} = \{y \in \mathbb{R}^p \ : \ h_i(y) \leq 0, i = 1, \dots, s\}$ then the maximal output admissible set is found as

$$\mathcal{O}_\infty = \{x \in \mathbb{R}^n \ : \ h_i(CA^+x) \leq 0, i = 1, \dots, s, t = 0, \dots, \infty\}$$

TODO: finire questa parte

3.7.2 Stability

In order to address the stability issue of MPC, we have to firstly define what's a positive invariant set and an exponential Lyapunov function.

« *A set $\mathcal{S}$ is positive invariant set if $\forall x \in \mathcal{S}$ it holds that $f(x) \in \mathcal{S}$* »

« *Suppose that $\mathcal{X}$ is a positive invariant set, a function $V : \mathbb{R}^n \to \mathbb{R}^+$ is an exponential Lyapunov function if $\exists \alpha_1, \alpha_2, \alpha_3 > 0$ such that*

$$\begin{aligned} V(x) &\geq \alpha_1 \|x\| \\ V(x) &\leq \alpha_2 \|x\| \\ V(f(x)) - V(x) &\leq -\alpha_3 \|x\| \end{aligned}$$

for all $x \in \mathcal{X}$. »

With this premise

> « *If there exists a Lyapunov function in the set $\mathcal{X}$, then the origin is exponentially stable in $\mathcal{X}$; furthermore, if $\mathcal{X} = \mathbb{R}^n$ the origin is globally exponentially stable.* »

Since the value function $V(\cdot)$ represents the cost-to-go that we expect to decrease over time, then we can regard it as a Lyapunov function. Given the optimal value function $V_0^\star(x_0)$ and the value function(not necessarily optimal) at the sequent time-step $V_1(x_1)$ as

$$V_0^\star(x_0) = \sum_{i=0}^{N-1} \ell(x_i, u_i) + \ell_f(x_N) \qquad V_1(x_1) = \sum_{i=1}^{N} \ell(x_i, u_i) + \ell_f(x_{N+1})$$

then their difference evaluates to

$$V_1(x_1) - V_0^\star(x_0) = \ell(x_N, u_N) + \ell_f(x_{N+1}) - \ell_f(x_N) - \ell(x_0, u_0) \leq -\alpha_3 \|x_0\| \qquad \forall x_N \in \mathcal{X}_f$$

To have $V(\cdot)$ as a Lyapunov function we need to assume that

- $f(0,0) = 0$, $\ell(0,0) = 0$, $\ell_f(0) = 0$;
- $\mathcal{Z}$ is closed and $\mathcal{X}_f$ is compact (i.e. closed and bounded);
- for any state $x \in \mathcal{X}_f$ exists a control $u \in \mathcal{U}$ such that

  $$f(x,u) \in \mathcal{X}_f$$

  so $\mathcal{X}_f$ is a control invariant set, and that the terminal cost decreases, i.e.

  $$\ell_f(f(x,u)) - \ell_f(x) \leq -\ell(x,u)$$

  Furthermore must exists two constants $\alpha_1, \alpha_f > 0$ that allows to lower bound the running cost and upper bound the terminal cost, i.e.

  $$\begin{aligned} \ell(x,u) &\geq \alpha_1 \|x\| && \forall x \in \mathcal{X}_N,\ \forall u \in \mathcal{U} \\ \ell_f(x) &\leq \alpha_f \|x\| && \forall x \in \mathcal{X}_f \end{aligned}$$

  where $\mathcal{X}_N$ is the set of states which the OCP has a solution.

If all this axioms are satisfied, then $V_N^\star(\cdot)$ is a Lyapunov function.
TODO: aggiungere considerazioni finali con sistema lineare

3.7.3 Computation time

To improve the convergence time of the numerical algorithms to solve OCPs, we can use a *warm start* that exploits the previous solution of the same problem as initial guess for the current OCP. This is done by shifting back by 1 time-step the optimal trajectory, i.e. $u_k^{\text{guess}} = u_{k+1}^\star$; the last element is instead initialized to zero ($u_{N-1}^{\text{guess}} = 0$). To bound also the computation time, we don't iterate until a full convergence, but we just do a fixed amount of Newton iteration (1 is usually fine).

Chapter 4

Reinforcement Learning

DISCLAIMER: This chapter has not been reviewed yet, so its content may be inaccurate!

Reinforcement learning (*RL*) is a method originally developed within the data-science community that, in the last years, has also been extended to robotic and control applications. It has a lot of similarities with optimal control, however key differences are that:

- reinforcement learning tries to find the *global optimal policy* for arbitrarily complex problems;
- assumes that the system's *dynamic* is *unknown* and assumed to be *stochastic*[1];
- initially it was developed to solve problem with discrete sets for both states and control, however in the recent days it has also been extended to continuous-time systems[2];
- typically solves an infinite horizon problem.

Terminology Even if concepts are very similar, reinforcement learning uses slightly different terminology and notation, as can be seen in table 4.1. The main difference is in the attitude toward the problem: in optimal control we want to minimize a cost, while in reinforcement learning we want to maximize the reward.

[1]in this chapter we will mainly consider deterministic dynamic as subset of the more general formulation.

[2]in genera the discretization of the continuous-time dynamics scales badly with the growth of the problem size

Table 4.1: terminology and notation comparison between optimal control and reinforcement learning.

optimal control	**reinforcement learning**
state x	state s
control u	action a
plant/dynamic	environment
cost	reward
cost-to-go	return
cost-to-go of a policy	value function
value function	optimal value function

For sake of simplicity, in this chapter we will mainly stick with the optimal control notation and consider the problem as a minimization (not a maximization of the reward); only exception is the subtle difference between *value function* and *optimal value function* that will be taken from the reinforcement learning vocabulary.

4.1 Markov decision process

A *Markov decision process* (*MDP*) is used to describe the environment of a reinforcement learning problem; a MDP is characterized by being fully observable and satisfying the *Markov property* for which "the future is independent from the past, given the present", mathematically speaking

$$\Pr\{x_{t+1}|x_t\} = \Pr\{x_{t+1}|x_t, x_{t-1}, \dots, x_1\} \tag{4.1}$$

where $\Pr\{X|Y\}$ is the *conditional probability* of X happening given that Y has already happened. The probability of switching from one state to another is described by the *state transition matrix* $\mathcal{P}_{xx'} = \Pr\{x_{t+1} = x'|x_t\}$ defined as

$$\mathcal{P}_{xx'} = \begin{bmatrix} P_{11} & \dots & P_{1n} \\ \vdots & \ddots & \\ P_{n1} & & P_{nn} \end{bmatrix} \qquad P_{ij} \in [0, 1] \tag{4.2}$$

Each state transition matrix, to be valid, must have that the sum of all the elements of each row adds up to 1 (since each state must evolve surely at each time-step). Furthermore in case of deterministic dynamic each element is either 0 (transition not possible) or 1 (only transition admissible).

Markov process

A *Markov process* (*MP*) is described by the tuple

$$\langle \mathcal{X}, \mathcal{P} \rangle \tag{4.3}$$

where $\mathcal{X}$ is the (finite) set of the allowed states and $\mathcal{P}$ is the state transition matrix ruling the probability of going from one state to the other; usually a Markov process is also referred as a *Markov chain*.

Properties of the state transition matrix Since $\mathcal{P}$ has all non-negative entries and is irreducible (because it's associated to a fully connected graph), by the *Person-Frobenius theorem* it holds that

- the largest (in norm) eigenvalue r of $\mathcal{P}$ must satisfy

  $$\min_i \sum_j P_{ij} \leq r \leq \max_i \sum_j P_{ij}$$

  Since each row sums up exactly to 1, then it follows that the maximum eigenvalue of $\mathcal{P}$ is $r = 1$;

- all other eigenvalue λ of $\mathcal{P}$ are smaller then r, i.e.

  $$\lambda_i\{\mathcal{P}\} < 1 \qquad i = 2, 3, \dots$$

A Markov chains sets up a probabilistic framework to describe a system's dynamic, considering $x^+ = \Pr\{x^+|x\}$; optimal control instead relies on the deterministic approach $x^+ = f(x, u)$ that we will try to stick with.

Markov reward process

A *Markov reward process* (*MRP*) is described by a tuple

$$\langle \mathcal{X}, \mathcal{P}, C, \gamma \rangle \tag{4.4}$$

where C is the *cost function* and $\gamma \in (0,1)$ is the *discount factor*. The first estimates the expected cost at the next time-step given the current state, i.e.

$$c_x = \mathbb{E}\left\{\ell_{t+1} | x = x_t\right\}$$

while the discount factor, as name suggests, is a measure of "how relevant a future cost is with respect to the present". The cost-to-go (*return* in RL jargon) is in fact the *total discounted cost* from t to infinity, i.e.

$$\mathcal{J}_t = \ell_t + \gamma \ell_{t+1} + \gamma^2 \ell_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k \ell_{t+k} \tag{4.5}$$

Here it's clear that the discount factor can be chosen to favor a reduction of the cost "in short time" ($\gamma \to 0$: myopic evaluation) or in the "long run" ($\gamma \to 1$: far-sighted evaluation).

Value function In reinforcement learning the value function $V(\cdot)$ represents the cost that we pay starting from a given state x, i.e.

$$V(x) = \mathcal{J}_t(x = x_t)$$

In light of the Bellman's optimality principle and (4.5) it turns out that

$$V(x) = \ell(x) + \gamma\, V\big(f(x)\big) \tag{4.6}$$

Considering in general the elements (4.4) of the Markovian reward process, then it directly follows

$$V = C + \gamma \mathcal{P} V$$

$$\begin{pmatrix} V(1) \\ \vdots \\ V(n) \end{pmatrix} = \begin{pmatrix} \ell(1) \\ \vdots \\ \ell(n) \end{pmatrix} + \gamma \begin{bmatrix} P_{11} & \dots & P_{1n} \\ \vdots & \ddots & \\ P_{n1} & & P_{nn} \end{bmatrix} \begin{pmatrix} V(1) \\ \vdots \\ V(n) \end{pmatrix}$$

It turns out that this is a linear equation in the unknown V, so by simply reversing the terms

$$V = (I - \gamma \mathcal{P})^{-1} C \tag{4.7}$$

This formulation is simple, but is applicable only to Markov chains of small size; when the number of states become larger, the numerical inversion of $I - \gamma\mathcal{P}$ becomes numerically ill conditioned, thus iterative methods to evaluate V are preferred.

Markov decision process

A *Markov decision process* (*MDP*) is described by the tuple

$$\langle \mathcal{X}, \mathcal{U}, \mathcal{P}, C, \gamma \rangle \tag{4.8}$$

where $\mathcal{U}$ is the (finite) set of control inputs. Denoting

$$P^u_{xx'} = \Pr\left\{x' = x_{t+1} | x = x_t, u = u_t\right\} \qquad C^u_x = \ell_t\big(x = x_t, u = u_t\big)$$

we define the *policy* π the probability distribution of the control inputs u given the states x, i.e.

$$\pi\{u|x\} = \Pr\left\{u = u_t | x = x_t\right\} \tag{4.9}$$

In case of deterministic policies the policy reduces simply to a function $u = \pi\{x\}$.

Action-value function We denote with $Q^{\pi}(x,u)$ the *action-value function* that's representing the cost that we pay starting from a state x and applying inputs u based on the control policy π:

$$Q^{\pi}(x,u) = \mathcal{J}_t\big(x = x_t, u = u_t, u_{k>t} \simeq \pi\big) \tag{4.10}$$

and considering the Bellman's theorem

$$\begin{aligned} Q^{\pi}(x,u) &= \ell(x,u) + \gamma Q^{\pi}\Big(f(x,u), \pi\{f(x,u)\}\Big) \\ &= \ell(x,u) + \gamma V^{\pi}\big(f(x,u)\big) \end{aligned} \tag{†}$$

where to simplify the notation we write $Q^{\pi}(x,\pi\{x\})$ simply as $V^{\pi}(x)$

Optimal value function and policy The *optimal value function* is the one associated with a policy the minimize it's value, i.e.

$$V^{\star}(x) = \min_{\pi} V^{\pi}(x) \tag{4.11}$$

thus the *optimal action-value function* is

$$Q^{\star}(x,u) = \min_{\pi} Q^{\pi}(x,u) \tag{4.12}$$

The optimal policy $\pi^{\star}$ is the one that satisfies $\pi^{\star} \leq \pi$ for any other policy π, where $\pi \leq \pi'$ requires that $V^{\pi}(x) \leq V^{\pi'}(x)$ for any x. This is achieved by minimizing the optimal action-value function among all possible inputs, i.e.

$$\pi^{\star}(x) = \arg\min_{u \in \mathcal{U}} Q^{\star}(x,u) \tag{4.13}$$

The main idea is that if we have access to the optimal action-value function, then it's possible to generate the optimal control policy.

Bellman optimality equality Given that the optimal value function is

$$V^{\star}(x) = \min_{u} Q^{\star}(x,u)$$

and recalling (†) allows us to state the following condition that the optimal solution must obey to:

$$\begin{cases} V^{\star}(x) &= \min_u \ell(x,u) + \gamma V^{\star}\big(f(x,u)\big) \\ Q^{\star}(x,u) &= \ell(x,u) + \gamma \min_{u'} Q^{\star}\big(f(x,u), u'\big) \end{cases} \tag{4.14}$$

Even in this case of deterministic system, there's no close-form solution for the nonlinear minimization so iterative algorithms are necessary.

4.2 Dynamic programming

Dynamic programming (*DP*) is a set of methods that are relying on the full knowledge of the Markovian decision process in order to solve two kinds of problems:

- *prediction*, so given $\langle \mathcal{X}, \mathcal{U}, \mathcal{P}, C, \gamma \rangle$ and a policy π, it computes the value function V^{π} associated to such policy;
- *control*, so given the MDP it outputs both the optimal policy $\pi^{\star}$ as well as the related optimal value function $V^{\star}$.

The latter is of course more interesting in control application as it provides us the best way to act on a system given the cost it's subjected to.

Finite and infinite horizon Dynamic programming was already introduced in sec. 3.1, page 24, in order to find the global optimum of a discretized optimal control problem; in that context we dealt with finite horizon problems for which theory is simpler and algorithms can compute solutions in a finite time.

For what concerns reinforcement learning we usually deal with *infinite horizon* problem for which theory is more complex (but elegant), assuming that both policy and value functions are time independent. Such theory is also a good approximation of problems with "long" (but finite) time horizons.

Bellman's operators Before continuing with dynamic programming, let's first define the *Bellman (expectation backup) operator*

$$T^{\pi}(V) = C^{\pi} + \gamma \mathcal{P}^{\pi} V \tag{4.15}$$

and the *Bellman optimality backup operator*

$$T(V) = \min_{u \in \mathcal{U}} C^{\pi} + \gamma \mathcal{P}^{\pi} V \quad \Leftrightarrow \quad (TV)(x) = \min_{u \in \mathcal{U}} \ell(x,u) + \gamma V\big(f(x,u)\big) \tag{4.16}$$

4.2.1 Prediction: iterative policy evaluation

Let's consider a prediction problem where it's requested to evaluate the value function V given the policy π. The solution to such problem is performed by iteratively applying the Bellman expectation operator:

$$V_{k+1} = T^{\pi}(V_k) \tag{4.17}$$

where k is the iteration index; expanded the value function is iteratively updated as

$$V_{k+1}(x) = \ell\big(x, \pi\{x\}\big) + \gamma V_x\big(f(x, \pi\{x\})\big) \qquad \forall x \in \mathcal{X}$$

So in practice we can chose an arbitrary initial value function V_0 since the *iterative policy evaluation* algorithm is guarantee to asymptotically converge, i.e.

$$\lim_{k \to \infty} V_k = V^{\pi}$$

Proof of convergence In order to prove the convergence of the iterative policy evaluation we firstly need to show that T^{π} is contracting. Given two value function V, Z we see that

$$\big\|T^{\pi}(V) - T^{\pi}(Z)\big\|_{\infty} = \big\|C^{\pi} + \gamma \mathcal{P}^{\pi} V - C^{\pi} - \gamma \mathcal{P}^{\pi} Z\big\|_{\infty} = \gamma \big\|\mathcal{P}^{\pi}(V - Z)\big\|_{\infty}$$

Due to the Person-Frobenius theorem it follows that

$$\gamma \big\|\mathcal{P}^{\pi}(V - Z)\big\|_{\infty} \leq \gamma \|V - Z\|_{\infty} \qquad \Rightarrow \qquad \big\|T^{\pi}(V) - T^{\pi}(Z)\big\|_{\infty} \leq \gamma \|V - Z\|_{\infty}$$

so given any two value function, the relative step obtained by applying the Bellman's expectation operator is upper bounded by the norm difference $V - Z$.

We can now show that given V_0 and V_{π}, then $V_k \to V_{\pi}$ for $k \to \infty$. Considering

$$\big\|V_k - V_{\pi}\big\|_{\infty} = \big\|T^{\pi}(V_{k-1}) - T^{\pi}(V_{\pi})\big\|_{\infty} \leq \gamma \big\|V_{k-1} - V_{\pi}\big\|_{\infty} \leq \gamma^2 \big\|V_{k-2} - V_{\pi}\big\|_{\infty}$$

so generalizing

$$\big\|V_k - V_{\pi}\big\|_{\infty} \leq \gamma^k \big\|V_0 - V_{\pi}\big\|_{\infty}$$

Since $\gamma \in (0,1)$, then V_k converges to V_{π} at a geometric rate.

Lastly we show that V_π is a unique fixed point for the operator T^π; assuming that V and Z are both fixed point for T^π, i.e. $T^\pi(V) = V$ and $T^\pi(Z) = Z$, then we can show that V must equate Z. Because T^π is contracting, then

$$\left\| T^\pi(V) - T^\pi(Z) \right\|_\infty \leq \gamma \|V - Z\|_\infty$$

but since V, Z are fixed points

$$\left\| T^\pi(V) - T^\pi(Z) \right\|_\infty = \|V - Z\|_\infty$$

This condition must hold at the same time, so

$$\gamma \|V - Z\|_\infty \geq \|V - Z\|_\infty$$

bug since $\gamma < 1$, then it must have $\|V - Z\|_\infty = 0$, implying that $V = Z$.

4.2.2 Control: policy and value iteration

Policy iteration In this case we deal with a more complex problem where, given a Markovian decision process, we want to find the optimal policy $\pi^\star$. The idea is that we start from an arbitrary policy π_0 and we perform until convergence the following two steps:

1. we evaluate the policy π_k, so we compute the value function V^{π_k};
2. we improve the policy by acting greedily with respect to the value function V^{π_k}, so

$$\pi_{k+1}(x) = \arg\min_{u \in \mathcal{U}} \ell(x,u) + \gamma V^{\pi_k}\big(f(x,u)\big) \tag{4.18}$$

Such algorithm always converges to the optimum policy:

$$\lim_{k \to \infty} \pi_k = \pi^\star$$

Proof of convergence Let's consider the simplified mathematical notation $\pi' \leq \pi$ to represent $V^{\pi'} \leq V^\pi$, we want to show now that $\pi_{k+1} \leq \pi_k$ for any iteration k, implying that we are always going closer to the optimal solution. Calling $\pi = \pi_k$ and $\pi' = \pi_{k+1}$ we see that

$$\begin{aligned}
\pi'(x) &= \arg\min_u \Big(\ell(x,u) + \gamma V^\pi\big(f(x,u)\big) \Big) \\
&= \underbrace{\min_u \Big(\ell(x,u) + \gamma V^\pi\big(f(x,u)\big) \Big)}_{i)} \leq \underbrace{\ell\big(x, \pi(x)\big) + \gamma V^\pi\Big(f\big(x, \pi(x)\big)\Big)}_{ii)}
\end{aligned}$$

where $i)$ is the cost that we get by following π' for the first step and then following π, while $ii)$ is the one obtained by simply following π. Considering now that

$$Q^\pi(x, \pi') = \min_u Q^\pi(x,u) \leq Q^\pi\big(x, \pi(x)\big) = V^\pi(x)$$

since by acting greedily always leads to a cost improvement (or at worst the cost remains equal), then we can see that

$$\begin{aligned}
V^\pi(x) &\geq \min_u \ell(x,u) + \gamma V^\pi\big(f(x,u)\big) = \ell(x,\pi') + \gamma V^\pi\big(f(x,\pi')\big) \\
&\geq \ell(x,\pi') + \gamma Q^\pi(x', \pi') = \ell(x,\pi') + \gamma \ell(x',\pi') + \gamma^2 V^\pi(x'') \\
&\geq \ell(x,\pi') + \gamma \ell(x',\pi') + \gamma^2 Q^\pi(x'', \pi') \\
&\geq \sum_{i=0}^{\infty} \gamma^i \ell\big(x^{(i)}, \pi'\big) = V^{\pi'}(x) \\
&\geq V^{\pi'}(x)
\end{aligned}$$

thus showing that $\pi' \leq \pi$, so at each iteration we can't have a policy worsening.

Modified policy iteration The main issue in policy iteration is that evaluating exactly the policy can take many iterations, so the idea is just to compute an approximation of V^{π_k} using just m_k policy evaluation iterations. Under mild assumptions this eventually converges to V^{π_k}.

With this idea taking $m_k = \infty$ leads to the yet described policy iteration method, while with $m_k = 1$ we get *value iteration*; in the latter case every-time we update V we use a policy that's greedy with respect to the value function itself.

Value iteration The algorithm of *value iteration* simply arts with an arbitrary guess V_0 for the value function and at each iteration

$$V_{k+1}(x) = \min_{u\in\mathcal{U}} \ell(x,u) + \gamma V_k\big(f(x,u)\big) \qquad \forall x \in \mathcal{X} \qquad \Rightarrow V_{k+1} = T(V_k)$$

The value function eventually converges to the optimal value

$$\lim_{k\to\infty} V_k = V^\star$$

and the optimal policy $\pi^\star$ is computed at the end from $V^\star$ as

$$\pi^\star = \arg\min_{u\in\mathcal{U}} \ell + \gamma V^\star$$

TODO: proof

4.3 Model-free prediction

Model-free prediction deals with the problem of finding the value function $V(\cdot)$ of an unknown Markovian decision process by simply acting on the environment and observing the response.

4.3.1 Monte Carlo policy evaluation

Monte Carlo (*MC*) methods can be used to solve model-free prediction problem and it's based on the simple idea of having the value function that's the mean return. This method come with the caveat that can be applied only to *episodic* Markov decision processes, so each episode must have an end.

The goal of this method is to learn V^π from episodes of experience under a policy π, so once it has been collected

$$x_1, u_1, \ell_1, x_2, u_2, \ell_2, \dots, x_k \backsim \pi$$

Once experience has been collected we can use the equation of the total discounted cost to compute the overall cost-to-go as

$$\mathcal{J}_t = \ell_t + \gamma\ell_{t+1} + \cdots + \gamma^{T-1}\ell_{T-1}$$

The value function then is the expected cost-to-go under a policy π; this can be modeled including also the stochastic part (due to the policy, the MDP and the cost) considering

$$V^\pi(x) = \mathbb{E}_\pi\{\mathcal{J}_t | x_t = x\} \tag{4.19}$$

Monte Carlo policy evaluation estimates V^π as the average cost-to-go; for what concerns the algorithm the first time a state is visited in an episode we should:

- increment a counter $N(x) = N(x) + 1$;
- increment the total cost of such state $C(x) = C(x) + \mathcal{J}(x)$;
- estimate the value function as $V(x) = V(x)/N(x)$.

By law of large number the estimate $V(x)$ tends to $V^\pi(x)$ as $N(x) \to \infty$. The idea is that by getting experience from more and more episodes, the stochastic contribution on the value function estimation gets negligible.

It also exists an *every-visit Monte Carlo* variant for which counter and costs are incremented every-time a state is visited (even inside the same episode).

Incremental Monte Carlo update To avoid storing the total cost $C(x)$ as well as the value function estimate $V(x)$ that are highly correlated, we can compute the mean incrementally, in fact

$$\begin{aligned} V_N &= \frac{1}{N}\sum_{i=0}^{N}\mathcal{J}_i = \frac{\mathcal{J}_N + \sum_{i=0}^{N-1}\mathcal{J}_i}{N} = \frac{\mathcal{J}_N + (N-1)V_{N-1}}{N} \\ &= V_{N-1} + \frac{\mathcal{J}_N - V_{N-1}}{N} \end{aligned}$$

This definition can be easily extended for *non-stationary* problems in order to consider a running mean that allow to embed a *forget factor* α to discount information from older episodes:

$$V(x) = V(x) + \alpha\big(\mathcal{J} - V(x)\big) \tag{4.20}$$

4.3.2 Temporal difference learning

Temporal difference learning (*TD0*) is an alternative to Monte Carlo evaluation for model-free prediction and is based on the cost-to-go estimation based on 1 step look-ahead to update the value function:

$$V(x_t) = V(x_t) + \alpha_t\big(\underbrace{\overbrace{\ell_t + \gamma V(x_{t+1})}^{\text{TD target}} - V(x_t)}_{\text{TD error}}\big) \tag{4.21}$$

where the TD target is the estimated return; here we can find an high similarity with the Monte Carlo incremental update (4.20). Moreover choosing $\alpha_t = 1$ and sampling all state uniformly, then TD0 is exactly the policy evaluation algorithm presented in sec. 4.2.1, page 48.

Properties The TD0 algorithm is a stochastic approximation algorithm; since the Bellman operator 4.15 has a unique fixed point, then if TD0 converges and all states are sampled infinitely many times than the estimate must converge to V^π.

Converges is guaranteed is the step-size sequence α_t satisfies the *Robinson-Monroe condition* for which

$$\sum_{t=0}^{\infty}\alpha_t = \infty \qquad \text{and} \qquad \sum_{t=0}^{\infty}\alpha_t^2 < \infty$$

A plausible sequence that satisfies this condition is $\alpha_t = \overline{\alpha}/t$, however in practice people use constant step size since the algorithm works anyway.

Table 4.2: comparison between Monte Carlo and temporal difference methods for solving a model-free prediction problem.

Monte Carlo	**temporal difference**
must wait until the end of an episode to know the cost	can learn after every step
only works with episodic (terminating) environments	works also in non-terminating environment
the cost-to-go $\mathcal{J}_t$ is an unbiased estimate of $V^{\pi}(x_t)$	target is a biased estimate of $V^{\pi}(x_t)$

Comparison with Monte Carlo Table 4.2 mainly compares Monte Carlo and temporal difference methods, showing the pros and the cons of each one.

In general Monte Carlo has an high variance but a zero bias; it's characterized by good convergence properties (even with function approximators) and is not very sensitive to the initialization of the problem. It's also simple and intuitive to understand.

Temporal difference instead has a lower variance but embeds some bias; it's usually more efficient then Monte Carlo and TD0 converges to $V^{\pi}(x)$ (but not always with function approximators); furthermore is more sensitive to the initialization of the estimate.

n-step return The concept of temporal difference can also be extended to a n-step return; TD0 in fact can be regarded as a 1-step return, but we can have a general formulation

$$\begin{aligned} n = 2 \qquad & \mathcal{J}_t^{(2)} = \ell_t + \gamma \ell_{t+1} + \gamma^2 V(x_{t+2}) \\ n \qquad & \mathcal{J}_t^{(n)} = \ell_t + \gamma \ell_{t+1} + \cdots + \gamma^{n-1} \ell_{t+n-1} + \gamma^n V(x_{t+n}) \end{aligned}$$

Note that by choosing $n = \infty$, the evaluation describes the Monte Carlo method. In general a TD$(n-1)$ method is updated with the following function:

$$V(x_t) = V(x_t) + \alpha_t \big(\mathcal{J}_t^{(n)} - V(x_t)\big) \tag{$\circ$}$$

Forward view TDλ The λ-cost-to-go $\mathcal{J}_t^{\lambda}$ combines all n-step cost-to-go $\mathcal{J}_t^{(n)}$ using as weights the sequence $(1-\lambda)\lambda^{n-1}$, i.e.

$$\mathcal{J}_t^{\lambda} = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} \mathcal{J}_t^{(n)} \tag{4.22}$$

As for Monte-Carlo, such temporal difference can be computed only from complete episodes (since the time horizon n theoretically should go to infinity), for this reason the method is referred as *forward view TDλ*; once $\mathcal{J}_t^{\lambda}$ is computed, we can use ($\circ$) to update the value function.

Backward view TDλ The *backward view TDλ* allows an online update of the value function at every step starting from incomplete sequences. Defining the *eligibility traces* $e_t(\cdot)$ as

$$\begin{aligned} & e_0(x) = 0 \\ & e_t(x) = \gamma \lambda e_{t-1}(x) + \mathbb{1}(x_t = x) \qquad \gamma < 0 \end{aligned} \tag{4.23}$$

Table 4.3: classification of control learning methods based on their interaction and optimization performance.

optimization performance	**environment interaction**	
	yes	**no**
offline	active learning	non-interactive learning
online	online learning	

Table 4.4: control learning categories based on the quantities that should be learned and the knowledge on the respective Markov decision process.

Markov decision process	**learned quantities**	
	$V(x)/Q(x,u)$	$V(x)/Q(x,u)$ and $\pi(x)$
known	value iteration	policy iteration
unknown	direct methods	actor-critic methods

then $e(x)$ increase every-time I visit x; otherwise it decays exponentially. Such function $e(\cdot)$ is a measure of how often and how recently I've visited x; recalling the TD error $\delta_t = \ell_{t+1} + \gamma V(x_{t+1}) - V(x_t)$, then the update of the value function is given by

$$V(x) = V(x) + \alpha\delta_t e_t(x) \tag{4.24}$$

4.4 Control learning methods

While dealing with *control learning* problems, we have different methods based on the interactivity (if the learned can interact actively with the environment, influencing the future) and the on/off-line performance (if the cost-to-go is compute while learning or after learning), as it can be seen in table 4.3.

As can be seen in table 4.4, control learning problems are further categorized based on the quantities that need to be learned, as well as on the knowledge of the underlying Markov decision process of the environment.
It has to be noted that when the MDP in known, that it's required to learn the $V(x)$, while if the MDP is unknown we usually estimate the action-value function $Q(x,u)$ (instead of V) since it allows to compute the policy

$$\pi(x) = \arg\min_u Q(x,u) = \arg\min_u \ell(x,u) + \gamma V(x')$$

As last classification, we have *on-policy learning* when we learn about a given policy π using experience sampled from π itself, while in *off-policy learning* we learn π using experience sampled using another policy $\tilde{\pi}$.

4.4.1 Q learning

Q learning is a direct method[3] that iteratively updates an estimate $Q_k(x,u)$ (thus the name of the method) of the optimal action-value function $Q^\star(x,u)$. After observing a transition (x,u,ℓ,x'), the algorithm update considering a TD error δ as follows:

$$\begin{aligned}\delta(Q_k) &= \ell + \gamma \min_{u'\in\mathcal{U}} Q(x',u') - Q(x,u) \\ Q_{k+1}(x,u) &= Q_k(x,u) + \alpha_k\,\delta(Q_k)\end{aligned} \tag{4.25}$$

[3]so it do not require the a-priori knowledge on the Markov decision process

At stochastic equilibrium, for each pair (x, u) visited infinitely often it must hold

$$\mathbb{E}\{\delta(Q)\} = 0 \qquad \Rightarrow \qquad TQ - Q = 0 \qquad \Rightarrow \qquad Q = Q^\star$$

The requirement that (x, u) must be visited infinitely often set a need for exploration, i.e. the learner must be able to learn from any possible condition. Q_k then converges to $Q^\star$ when appropriate learning rates α_k are used.

Control strategy While solving an online learning problem, key is choosing the input u that needs to be applied to the system. A strategy that we might use is to be greedy with respect to the action-value function, i.e.

$$u(x) = \arg\min_w Q_k(x, w) \tag{4.26}$$

This choice allows us to always chose the control that leads to an estimated lower cost in the future, however this might not be optimal since exploration is not ensure and could lead to a large loss over time. A good learner in fact must choose also suboptimal controls in order to explore.

For this reason we might use the so called *ε-greedy strategy* for which

- we choose a random control with probability ε;
- we choose a greedy control (4.26) with probability $1 - \varepsilon$.

This simple technique ensures exploration, and in practice we can have a value of ε that changes over time (typically decreasing over time toward zero).

Another technique that we might use is the *Boltzmann exploration* for which the control is chosen with a probability Π proportional to it's mean, considering so

$$\Pi(u) = \frac{\exp\big(-\beta Q_t(u)\big)}{\sum_{u' \in \mathcal{U}} \exp\big(-\beta Q_t(u')\big)} \tag{4.27}$$

where $Q_t(u)$ is the mean cost obtained applying the control u; in this way we choose more often controls that are assumed to lead to a lower cost. Furthermore choosing $\beta \to \infty$, this strategy reduces to the greedy one.

One last technique that we might use is *optimism in the face of uncertainty*, a method that weights the greedy strategy by preferring inputs that have been explored fewer times. We chose in fact the control with the *lowest confidence bound* defined as

$$\text{LCB}(u) = Q_t(u) \tag{4.28}$$

where the uncertainty on $Q_t(u)$ is

$$Q_t(u) - L\sqrt{\frac{2\log(t)}{n_t(u)}}$$

where $L = \max\{|\ell(u)|\}$ and $n_t(u)$ is the number of times that u was selected.

4.4.2 Actor-critic

The *actor-critic methods* can be regarded as a generalized policy iteration (sec. 4.2.2) extended to unknown Markov decision process. In that case we alternated a policy evaluation with a policy improvement in order to converge to the optimal policy $\pi^\star$.

The extension to unknown MDPs is carried out using Monte Carlo (sec. 4.3.1) or temporal difference (sec. 4.3.2) to evaluate the policy. Since the exact evaluation of V^π can take infinitely many samples, the idea is to improve the policy π based on an approximation of the value function (so the MC/TD doesn't run until convergence, but for a given amount of time).

The value function is called *critic* because it evaluates the policy (so measures "how good" the learner perform), while in turn the *actor* is the policy itself.
In general the policy used to generate transitions is typically not the same that is evaluated and improved: the *behavior policy* mixes a small amount of exploration into a *target policy*. We note that the generalized policy iteration can generate policies that are worse then the previous ones.

SARSA: critic In the *SARSA method* we use TD0 to learn the *critic* (so the action-value function Q) from on-policy samples. After each transition (x, u, ℓ, x', u')[4] the update of Q is performed as

$$\begin{aligned} \delta(Q_k) &= \ell(x,u) + \gamma Q(x',u') - Q(x,u) \\ Q_{k+1}(x,u) &= Q_k(x,u) + \alpha_k \, \delta(Q_k) \end{aligned} \tag{4.29}$$

We note the similarity with Q learning (4.25), but in this case we use $Q(x', u')$ instead of $\min_z Q(x, z)$. Note that if the policy π is fixed, then SARSA and TD0 are exactly the same thing.

As for temporal difference, we can extend this concept to multi-step evaluation: using TDλ gives the method SARSA(λ).

From TD, SARSA inherits the possibility of diverging in off-policy settings.

SARSA: actor The goal of an actor-critic method is also to reach the optimal policy, a goal of the *actor* part. The simplest idea to improve the policy is by acting greedily with respect to the action-value function Q (that can be computed on the fly).

To ensure exploration we might use ε-greedy strategies, but in order to let SARSA converge to the optimal action-value function $Q^\star(x, u)$ we need to ensure that

1. we are *Greedy in the Limit with Infinite Exploration* (*GLIE*) policies, so all state-action (x, u) are visited infinitely many times and the policy converges to the greedy one ($\varepsilon \xrightarrow{x\to\infty} 0$);
2. the sequence of step sizes α_k satisfies the Robbins-Monroe conditions.

[4]using reinforcement learning naming it becomes (s, a, r, s', a'), thus the name *SARSA*.

Bibliography

[1] Andrea Del Prete. Joint Position and Velocity Bounds in Discrete-Time Acceleration / Torque Control of Robot Manipulators. *IEEE Robotics and Automation Letters*, 3(1), 2018.