Advanced Optimization-based Robot Control

Marco Peressutti
230403
marco.peressutti@studenti.unitn.it

August 9, 2025

Contents

1 Reactive Control **4**
- 1.1 Robot Dynamic Modeling . . . 4
  - 1.1.1 Homogeneous Transformation Matrix . . . 5
  - 1.1.2 Representation of orientation . . . 5
  - 1.1.3 Differential Kinematics . . . 6
  - 1.1.4 Statics . . . 7
  - 1.1.5 Dynamics . . . 7
- 1.2 Joint Motion control . . . 8
  - 1.2.1 Open-Loop control . . . 8
  - 1.2.2 PID control . . . 8
  - 1.2.3 PD + gravity compensation . . . 9

2 Optimal Control **10**
- 2.1 Introduction . . . 10
  - 2.1.1 History . . . 10
  - 2.1.2 Optimal control problem (O.C.P.) . . . 11
  - 2.1.3 Optimal control method families . . . 12
- 2.2 Dynamic Programming (DP) . . . 12
  - 2.2.1 Principle of optimality (Bellman) . . . 12
  - 2.2.2 Dynamic Programming approach . . . 13
- 2.3 Direct methods . . . 15
  - 2.3.1 Introduction . . . 15
  - 2.3.2 Single Shooting - Sequential Approach . . . 16
  - 2.3.3 Collocation - Simultaneous Approach . . . 20
  - 2.3.4 Multiple shooting . . . 21
  - 2.3.5 Comparison of Direct methods . . . 22
- 2.4 Linear Quadratic Regulator (LQR) . . . 22
  - 2.4.1 LQR via Dynamic Programming . . . 24
  - 2.4.2 Steady State Regulator . . . 26
  - 2.4.3 Time Varying System . . . 27
  - 2.4.4 Inhomogeneous system and costs . . . 27
  - 2.4.5 Tracking problems . . . 27
  - 2.4.6 Differential Dynamic Programming (DDP) . . . 28
- 2.5 Model Predictive Control (MPC) . . . 32
  - 2.5.1 Infinite-Horizon MPC . . . 33
  - 2.5.2 Feasibility . . . 34
  - 2.5.3 Stability . . . 37
  - 2.5.4 Computation time . . . 41
  - 2.5.5 Uncertainties in MPC . . . 42

3 Reinforcement Learning **43**
- 3.1 Introduction . . . 43
- 3.2 Terminology comparison . . . 43
- 3.3 Framework: Markov Decision Processes (MDP) . . . 44
  - 3.3.1 Markov Process or Markov Chain . . . 44

3.3.2 Bellman equation in the context of RL . . . 46
3.3.3 Markov Decision Process MDP . . . 46
3.3.4 Action-Value Function (Q function) . . . 47
3.3.5 Optimal Value Function and Optimal Action-Value Function . . . 47
3.3.6 Bellman Optimality equation . . . 48
3.4 Model Based Control . . . 48
3.4.1 Dynamic Programming . . . 48
3.4.2 Prediction . . . 50
3.4.3 Control . . . 51
3.5 Model Free Prediction . . . 53
3.5.1 Monte Carlo Policy Evaluation . . . 53
3.5.2 Temporal Difference Learning - TD0 . . . 54
3.5.3 Comparison betweeen MC and TD0 . . . 55
3.5.4 Bootstrapping and Sampling . . . 56
3.5.5 n-Step return . . . 56
3.5.6 TD(λ) . . . 57
3.6 Model Free Control . . . 59
3.6.1 Types of control learning problems . . . 59
3.6.2 Control Learning Methods . . . 59
3.6.3 On-policy vs Off-policy learning . . . 60
3.6.4 Direct Methods: Q learning . . . 61
3.6.5 Actor-Critic methods: Generalized policy iteration . . . 63
3.7 Value function Approximator . . . 64
3.7.1 Types of value function approximation . . . 65
3.7.2 Gradient descent . . . 66
3.7.3 Incremental prediction algorithms . . . 66
3.8 Incremental Control with function approximation . . . 67
3.8.1 Convergence of Control Algorithm . . . 67
3.8.2 Batch Learning . . . 67
3.8.3 Deep Q-Network . . . 68
3.8.4 Implementation TIps . . . 68

A Numerical Integration **69**
A.1 Numerical Integration Methods . . . 69
A.1.1 Explicit Euler . . . 69
A.1.2 Mid-Point Method . . . 69
A.1.3 Runge-Kutta Methods . . . 70
A.2 Properties of Integration Schemes . . . 71

B Lyapunov Stability **74**
B.1 Exponential Lyapunov functions . . . 74

Index **75**

Bibliography **80**

Chapter 1

Reactive Control

1.1 Robot Dynamic Modeling

In this section we are going to review Multibody dynamics modeling. A few things that we will define today are:

Multibody dynamics modeling

- Kinematics model (forward)
- Statics
- Dynamics

and in doing so we are going to see:

- Homogeneous matrix
- concept of Jacobian
- ...
- $M\,\ddot{q} + C\,\dot{q} + g(q) = \tau + J^T\,w$

A Robot Manipulator is a set/collection of rigid bodies (links) connected together by joints (which can be either revolute or prismatic).

Robot Manipulator

Tipically joints are not passive i.e. they are connected to an actuation system (e.g. electric motor with a gear box that can actuate the joints by providing torque, this torque is then translated into an acceleration which is gonna create some motion).
The reason for the combination motor + transmission is because electric motor provides small torque and high velocity, but what we need in the actuation of the robotic arm is high torque and low velocity.
Therefore the presence of a trasmission trades off the velocity for torque.

links

joints

What we are going to be discussing is what the relationship between the torque that is provided by the motors and the resulting acceleration that is generated on the joints and how this affect the motion of the whole robot, especially the end-effector, where the gripper is generally placed.

end-effector

By q_i we denote to the i-th joint coordinate and by q a common vector containing all the joint coordinates (configuration vector).

configuration vector

$$q = \begin{bmatrix} q_1 \\ q_2 \\ \dots \\ q_n \end{bmatrix}$$

To represent the configuration of the robot in space the only thing that we need to know are the angles of the joints (that are measured w.r.t. a reference angle considered to be zero).

When working with rigid bodies in 3D space, a quantity that is very useful to know is the Homogeneous Transformation matrix, that is used to define the relative pose of two rigid bodies in space.

Homogeneous Transformation matrix

1.1.1 Homogeneous Transformation Matrix

Let us consider 2 rigid bodies, each with a reference frame attached to it.

Then we can define a Homogeneous matrix in the form:

$$H_1^0 = \left[\begin{array}{ccc|c} & R_1^0 & & p \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

where the rotation matrix containes the axes of frame 1 in the reference frame 0

$$R_1^0 = \begin{bmatrix} x_1^0 & y_1^0 & z_1^0 \end{bmatrix}$$

The rotation matrix has the property that its inverse is equal to its transpose (orthonormal matrix)

$$R_1^0 = (R_0^1)^{-1} = (R_1^0)^T$$

Now that we have defined Homogeneous matrix we can use them to define the kinematics, i.e. geometry of the manipulator. We can in fact define a reference frame attached to each link and a homogeneous matrix to represent the relative pose of each frame w.r.t. the previous one.

Once we have this chain of homogeneous transformation matrices, we can put them together by matters of matrix multiplication and obtain the global homogeneous transformation that defines the pose of the end effector w.r.t. frame 0 (atteched to the ground). So this gives us the pose of the end-effector in the world.

Given a sequence of frames that have different positions and orientations terminating with an end effector frame. We can define for each link a transformation matrix. If we want to compute the global transformation from 0 to n when n is the frame attached to the end effector, we can just multiply all the matrices together.

Once we apply this method to a serial manipulator what we get is that the homogeneous transformation from the frame i-1 to i will depend only on the i-th joint coordinate/angle.

Each joint angle changes one homogeneous transformation but the global one will depend on all the joint coordinates

$$H_n^0(q_1, q_2, \ldots, q_n) = H_n^0(q)$$

1.1.2 Representation of orientation

A quick notes on how to represent orientation. The problem with transformation matrices is that they are very redundant so you need a 3×3 matrix, 9 numbers to represent an orientation. Actually an orientation is 3 dimensional quantity, so in general it is not really practical to use a rotation matrix which contains 9 number to represent something that is a 3 dimensional information.

Theer are other ways to represent a 3D orientation:

- Rotation matrix — 3×3, redundant representation
- Euler angles — 3, minimal representation
- Roll-Pitch-Yaw — 3, minimal representation
- Angle-axis — 4, redundant representation
- Quaternions — 4, redundant representation

Minimal representation is the minimum number of parameters to define a 3D quantity. The problem with minimal representation is that they have singularities i.e. points in the orientation space where a very small change in the orientation of the body results in a very large change in Euler angles and RPY. This is the reason why we usually do not use them, even though they have an underling geometric meaning.

1.1.3 Differential Kinematics

By differentail kinematics, we refer to the relationship between the velocity of the joints of the robot and the velocity of all end-effector (in general the velocity of all the links).

We are interested mainly in the geometrical approach to differential kinematics. What we want to compute is the velocity of the end-effector ${}^0v_e$ w.r.t. frame 0, intertial frame.

$$ {}^0v_e = \begin{bmatrix} \dot{p_e} \\ \omega_e \end{bmatrix} $$

that contains the linear velocities and the angular velocities.

In order to compute the velocity of the end-effector we need to know the velocities of the joints of the robot and that is tipically measured by sensors and the relationship between the velocity of the joints and the velocity of the end-effector. This relationship is defined by a matrix that is called Jacobian of the kinematic chain.

We can actually derive the elements of the jacobian by writing down the relationship between the relative velocity of each link of the robot w.r.t. the previous links' frame. Similarly to what we have done with homogeneous matrices, we chain them alltogether by matter of matrix multiplication.

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

$$ {}^{i-1}\dot{p}_i = \begin{cases} [\hat{z}_i \times (p_i - p_{i-1})]\,\dot{q}_i = J_{pi}\,\dot{q}_i & \text{Revolute} \\ \hat{z}_i\,\dot{q}_i = J_{oi}\,\dot{q}_i & \text{Prismatic} \end{cases} \tag{1.2} $$

$$ {}^0\dot{p}_n = \sum_{i=0}^{n-1} J_{pi+1}\,\dot{q}_{i+1} = J_p\,\dot{q} \tag{1.3} $$

$$ {}^0\omega_n = \sum_{i=0}^{n-1} J_{oi+1}\,\dot{q}_{i+1} = J_o\,\dot{q} \tag{1.4} $$

Therefore we can represent the relationship between the joint coordinate velocity and the end effector velocity in the following compact form:

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

There exists an analitical approach, where we see the pose of the end-effector as a function of the joint angles.

$$ \phi_n = \begin{bmatrix} x_n \\ y_n \\ z_n \\ \zeta_n \\ \vartheta_n \\ \psi_n \end{bmatrix} = \phi_n(q) $$

Having this representation, we can derive it with respect to time and exploit the chain rule to compute the relationship of the end-effector velocity and the joint coordinates velocity.

$$ \dot{\phi} = \frac{\partial \phi}{\partial q}\frac{dq}{dt} = \frac{\partial \phi}{\partial q}\dot{q} $$

The matrix of partial derivatives of the pose with respect to the joint coordinates is the analitical Jacobian.

Note that the analitical jacobian is different from the geometrical jacobian: in particular the translational part is the same but the rotational part is different

$$ \begin{bmatrix} J_p \\ J_{Ao} \end{bmatrix} = J_A \neq J = \begin{bmatrix} J_P \\ J_o \end{bmatrix} $$

This is not a surprise because in the analitacal approach we used the derivative of Euler angles whereas in the geometrical approach we used the angular velocity, and these two quantities are different.

1.1.4 Statics

Problem formulation: given wrench (force + moment) applied at the end-effector, we want to compute the joint torques generated by this wrench. wrench

The way we can compute this relationship between the wrench adn the joint torques is by equating the total power of the system on the end-effector space and the power that is computed in the joint space.

These two quantities must be equal indipendently of where it compute it.

1. Power at the joints:
$$P_\tau = \tau^T \dot{q}$$
2. Power at the end-effector:
$$P_e = f^T \dot{p} + m^T \omega = \begin{bmatrix} f^T & m^T \end{bmatrix} \begin{bmatrix} \dot{p} \\ \omega \end{bmatrix} = w^T v = w^T J \dot{q}$$

Equating the two expressions:
$$\tau^T \dot{q} = w^T J \dot{q} \quad \forall \dot{q}$$
and infere that:
$$\tau^T = w^T J$$
$$\tau = J^T w$$

1.1.5 Dynamics

Inside the field of dynamics there are two main problems: forward/direct dynamics and inverse dynamics. The difference between these two problems is what you want to compute and what are your inputs.

So let us define them:

- Direct dynamics: $q, \dot{q}, \tau$ are known and we want to compute $\ddot{q}$. Direct dynamics
  Usually this is the problem to solve during simulations.
- Inverse dynamics: $q, \dot{q}, \ddot{q}$ are known and we want to compute τ. Inverse dynamics
  Usually this is the problem solved in controllers.

But they are both based on the same equation.

The final equation for the multibody dynamics is:
$$M(q)\,\ddot{q} + C(q,\dot{q})\,\dot{q} + g(q) = \tau + J(q)^T w$$
where:

- $M(q)$ $\in \mathbb{R}^{n\times n}$ is the mass matrix, it is always positive-definite and symmetric
- $C(q,\dot{q})$ $\in \mathbb{R}^{n\times n}$ accounts for centrifugal and Coriolis effect
- $g(q)$ $\in \mathbb{R}^n$ accounts for gravity torques

The mathematical formulation of this expression is non linear, because C, M and g depends on q and $\dot{q}$.

What is good about this equation is that if you know q and $\dot{q}$ at a specific instant then M, C and g becomes constant and the relationship becomes linear in $\ddot{q}, \dot{q}$ and w.

Most of the time we will refer to:
$$C(q,\dot{q})\,\dot{q} + g(q) = h(q,\dot{q})$$

1.2 Joint Motion control

We have a given motion that we would like the robot to track. This motion is specified in terms of a reference joint trajectory, and we want to find a controller that is able to follow this trajectory as accurately as possible. When we say that we want to find a controller, what we mean is that we want to find a mathematical law that is able to give us the joint torques (control inputs of the system) as a function of the joint angles and joint velocity (state of the system).

reference joint trajectory

Formally: given a reference joint trajectory $q^{ref}(t)$, compute the joint torques $\tau(t)$ such that $q(t) \approx q^{ref}(t)$. Assumption: the system dynamics is known, i.e. $M\,\ddot{q} + h = \tau$, and there is no contact with the environment (i.e. $w = 0$).

1.2.1 Open-Loop control

A very simple idea is to compute the joint torques is simply to compute $\ddot{q}^{ref}(t)$ and $\dot{q}^{ref}(t)$ of the reference dynamics and obtain the control input torques ($\tau(t)$) by substituting these quantities in the system dynamics formulation.

$$\tau(t) = M(q^{ref}(t))\,\ddot{q}^{ref}(t) + h(\dot{q}^{ref}(t), q^{ref}(t))$$

This controller approach is openloop because it does not use the current state of the system in the control law.

However, the problem with open-loop control works only if:

- there is no perturbation acting on the system
- the initial state of the robot needs to match the initial state of the reference trajectory.

$$q(0) = q^{ref}(0) \qquad ; \qquad \dot{q}(0) = \dot{q}^{ref}(0)$$

- you have perfect knowledge of the model.

1.2.2 PID control

The idea is to compute the joint torques in a way that is proportional to the tracking error, its derivative and its integral.

$$\tau(t) = k_p(q^r(t) - q(t)) + k_d\,(\dot{q}^r(t) - \dot{q}(t)) + k_i \int_0^t (q^r(s) - q(s))\,ds$$

where k_p, k_d, k_i are positive-definite, diagonal gain matrices.

The controller is not open loop anymore because we are using the feedback on the state of the system (joint position and velocities).

In order to implement this controller, sensors are needed that measure at least the joint angles.

This works quite well and many robots in the industry are actually controlled with PID, because it is very simple and you need zero knowledge about the dynamics of the system.

If we have $\dot{q}^{ref} = 0$ (i.e. static configuration) then we have a guarantee of convergence i.e. $q(t) \to q^{ref}(t)$. proof:

$$M\,\ddot{q} + C\,\dot{q} + g = \tau = k_p\,e + k_d\dot{e} + k_i \int_0^t e\,ds$$

At steady state ($\dot{q} = \ddot{q} = 0$) we obtain:

$$g = k_p e + k_i \int_0^t e\,ds$$

taking the derivative of this equation:

$$k_p\,\dot{e} = -k_i e$$

which is linear dynamical system which is stable as well because k_p and k_i are positive-definite and diagonal. This means that the error dynamics is in the form:

$$\dot{e} = -k_p^{-1}\,k_i\,e$$

which eigenvalues are negative and therefore $e \to 0$

1.2.3 PD + gravity compensation

We can make use of the model dynamics in order to improve the tracking performance.

We are going to use only a part of the model dynamics (the gravity torque), in order to compensate for the torque due to gravity, but for the rest we are not going to use our knowledge of the model, but instead we are going to utilize a PD controller.

$$\tau(t) = k_p(q^r(t) - q(t)) + k_d(\dot{q}^r(t) - \dot{q}(t)) + g(q(t))$$

If we imagine to eliminate the k_p and k_d terms, then we obtain a controller that will mantain the robot still and it acts as if there is no gravity.

This controller works better than PID alone.

The steady state analysis yields:

$$\not{g} = k_p\, e + \not{g}$$

which implies that

$$k_p e = 0$$

If the controller converges and reaches a steady state, than it must be a steady state where the error is equal zero (only possible solution).

Of course in this controller we are using some knowledge of the system dynamics (i.e. the gravity term), therefore we need to haev an estimate of the gravity compensation and a way to compute it sufficiently fast to be used inside our controller (which tipically is not a problem).

in general, PID controller is preferred (not better) because it works indipendently of where it is applied, but with PD+gravity compensation we have a gain less to tune.

Chapter 2

Optimal Control

2.1 Introduction

Optimal control is quite different type of control theory w.r.t. reactive control, but the advantage of the control methods to be discussed is that they can be utilized even to do planning (what trajectory you are going to execute before you actually execute it).

2.1.1 History

Optimal control was born in the XVII century and was created by Bernoulli (1696). The first control problem that was solved can be described as follows:

Imagine that you have a ball in a certain position A and you would like that ball to reach a target position B. Under this condition Bernoulli asked himself, what is the shape of the ground so

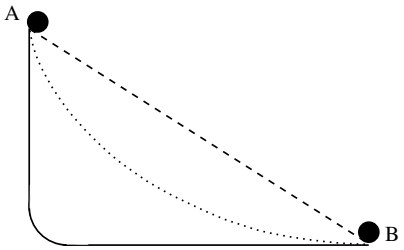


Figure 2.1: Comparison of three different paths: straight line (dashed), maximum initial acceleration path (continuous) and brachistocrone (dotted)

that the ball in A reaches B in minimum time?

Several possibilities can be considered:

1. a straight line
2. something that goes down sharply and then reaches the target point with a straight line.

Turned out that was neither of those: they are both slower than minimum time, which materializes with a brachistochrone curve.

So Optimal control is a framework that helps us answer questions such as: what is the trajectory that allows me to go from this place to another

- in minimum time
- consuming the minimum amount of energy
- reaching the maximum velocity possible at the end of the trajectory.

These kind of question where we want to find the best motion for doing something, and by best we mean that it maximizes a certain critiria.

Before we dive deep into what optimal control is all about let us clarify what is the difference between Optimal control and Reactive control:

1. One of the key feature of reactive control which we have seen so far is that it always assume that you have a reference trajectory that you want to track. But we have never answer the question how do we actually compute these reference trajectories?

   There are some cases where those reference trajectories are basically given by the problem (e.g. robot that cuts a material with a laser, knowing the profile of the cut you automatically know the trajectory that the end-effector should follow), but there are many cases where it is not obvious what trajectory the robot manipulator should follow.

   In all of those circumstances where there is no clear trajectory to follow Reactive control is not enough because it does not allow to choose a specific solution for this problem, on this aspect Optimal control is the superior tool to leverage.

2. In reactive control the performance of the system are measured using quantities such as rise time, overshoot and settling time. In Optimal control, instead, we can use many more criteria in order to optimize the quality of the motion (minimize the energy, maximize the speed, minimize the time taken to do the motion)

3. In optimal control since it is based in optimized it is really easy to handle constraints. Inequality constraints are commonly used to represent the limits of the system (current limits, friction cones, joint limits, torque limits).

Reactive control	**Optimal control**
need reference trajectory	no need for reference trajectory
rise time	objective clearly specified throguh cost function
overshoot	
settling time	
hard to handle constraints (e.g. joint position limits)	can handle constraints

2.1.2 Optimal control problem (O.C.P.)

An optimal control problem takes the form of a minimization problem, but it is not really a minimization problem:

$$\underset{x(\cdot),u(\cdot)}{\text{minimize}} \int_0^T l(x(t),u(t),t)\,dt + l_f(x(T))$$

subject to:

- $\dot{x}(t) = f(x(t),u(t),t) \quad \forall t \in [0,T]$ (dynamics) — dynamics
- $x(0) = x_0$ (initial conditions) — initial conditions
- $g(x(t),u(t),t) \leq 0 \quad \forall t \in [0,T]$ (path constraints), e.g. actuator limits — path constraints

where:

- $\int_0^T l(x(t),u(t),t)\,dt$ is the running cost — running cost
- $l_f(x(T))$ is the terminal cost — terminal cost

As said before this is not a standard minimization/optimization problem, but an optimal control problem because

- $x(\cdot), u(\cdot)$ are trajectories, i.e. infinite-dimensional objects.
- the constraints of the problem are infinitely many, given that they must be valid for all instances in the range $[0,T]$.

Example of OCP: Pendulum

We can imagine a pendulum robot and we want him to reach the vertical position (i.e. $q = 0$), where q is the angle between the pose of the robot and the desired pose.

$$\underset{x(\cdot),u(\cdot)}{\text{minimize}} \int_0^T q(t)^2\,dt + q(T)^2$$

subject to:

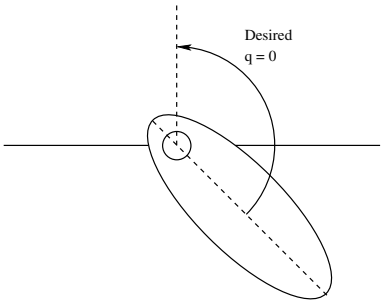
- $I\,\ddot{q} = \tau + m\,g\,sin(q) \Rightarrow \begin{bmatrix}\dot{q}\\ \ddot{q}\end{bmatrix} = \begin{bmatrix}\dot{q}\\ I^{-1}(\tau + m\,g\,sin(q))\end{bmatrix}$
- $q(0) = q_0$
- $\dot{q}(0) = \dot{q}_0$
- $q^{min} \leq q(t) \leq q^{max} \qquad \forall t \in [0, T]$
- $|\tau(t)| \leq \tau^{max} \qquad \forall t \in [0, T]$

where

- $x \triangleq (q, \dot{q})$
- $u \triangleq \tau$

However the formulation of the running cost will make the robot manipulator reach $q = 0°$ as fast as possible, and maybe it is not something that you want. For this reason, additional terms are also included in the running cost. e.g,

$$\int_0^T (q(t)^2 + u(t)^2 + \dot{q}(t)^2)\,dt$$



2.1.3 Optimal control method families

There are many different approaches to solve a control problem. Some of them works in Continous Time, other in Discrete time, some of them are global and other are local.

	optimize ⇒ discretize Continuous time	discretize⇒optimize Discrete time
Global	Hamilton-Jacobi-Bellman (HJB)	Dynamic Programming (DP)
Local	Pontryagin Maximum Principle (PMP) Calculus of variations Indirect methods	Direct methods

2.2 Dynamic Programming (DP)

2.2.1 Principle of optimality (Bellman)

Dynamics Programming is based on the principle of optimality, aka the Bellman principle.

principle of optimality

Bellman principle

Let us suppose we have an optimal trajectory to go from one point to another.

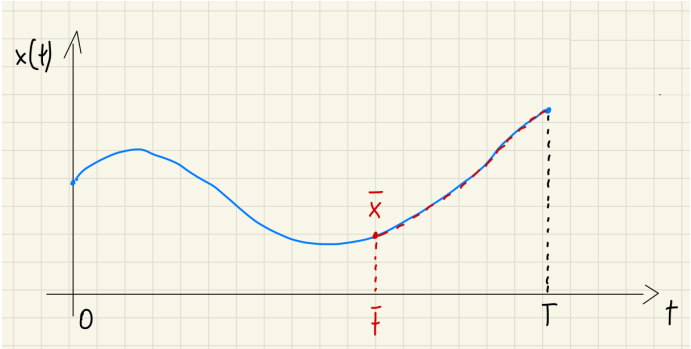


Figure 2.2: Bellman optimality principle visual proof: if the continuous line is an optimal trajectory than the dotted subtrajectory is also optimal

What the principle of optimality says is that if you have an optimal trajectory then each subarc is in itself an optimal arc.

In other words, if the trajectory is the optimal trajectory in order to go from point A to point B, then no matter which initial point of the curve you start from the arc connecting such initial condition to the final point is the optimal trajectory.

This is rather trivial because if there is another arc that is the optimal trajectory to reach the final point than I could replace such arc to the original optimal trajectory, but since the original one is optimal by assumption that cannot be the case.

The principle of optimality can be used to simplify the solution of a discrete time optimal control problem.

2.2.2 Dynamic Programming approach

In dynamic programming the optimal control problem is assumed to be already in discrete time; that might be bacause we already have a discrete time problem (e.g. games, chess) or might be a continuous time problem that you discretized in time.

Therefore, the trajectories $x(\cdot)$ and $u(\cdot)$ are not in continuous time, but they are trajectories in discrete time (basically matrices).

The integral is replaced by a summation (equivalent of the integral).

$$\underset{X,U}{\text{minimize}} \sum_{i=0}^{N-1} l(x_i, u_i)$$

The dynamics that we have as a constraints are a discrete time dynamics. subject to:

- $x_{i+1} = f(x_i, u_i) \qquad i = 0 \dots N-1$
- No terminal cost for sake of simplicity

To apply the principle of optimality to dynamic programming we need to split this problem into 2 parts.

Take M s.t. $0 \leq M < N$. Then we split the problem cost:

$$\underset{X,U}{\text{minimize}} \left[\sum_{i=0}^{M-1} (l(x_i, u_i) + I(x_{i+1} - f(x_i, u_i))) + \sum_{i=M}^{N-1} (l(x_i, u_i) + I(x_{i+1} - f(x_i, u_i))) \right] \tag{2.1}$$

$$\underset{X,U}{\text{minimize}} \left[c_0(X_{1:M}, U_{0:M-1}) + c_M(x_M, X_{M+1:N}, U_{M:N-1}) \right] \tag{2.2}$$

In this formulation that constraint has been integrated in the cost function through the Indicator function. Such operator is zero when the input is zero and it is infinity otherwise.

Indicator function

Our constraints can now be seen as a high cost term if it is violated.

Now instead of doing the minimization of the sum of the two terms, since they depend on the different part of the trajectory (even though they both depend on x_M) I can treat them as decoupled and find the minimum of c_M, find the value of x_M and lastly minimize c_0.

Formally:

$$\underset{X_{1:M},U_{0:M-1}}{\text{minimize}} \left[c_0(X_{1:M}, U_{0:-1}) + \underset{X_{M+1:N},U_{M:N-1}}{\text{minimize}} \; c_M(x_M, X_{M+1:N}, U_{M:N-1}) \right]$$

$$V_0(x_0) = \underset{X_{1:M},U_{0:M-1}}{\text{minimize}} \; c_0(X_{1:M}, U_{0:M-1}) + V_M(x_M)$$

We have still an optimal control problem, which parameters are still hidden inside c_0, but the big difference is that now we are optimizing over a smaller trajectory thanks to a separation of the problem formulation into two smaller minimization problems.

And since M can be chosen to be every value we want, we can repeat the same process as many times as we want and make the smaller problems to be optimizing over one value of the trajectory (i.e. one time step).

We therefore solve a sequence of problems: one for each time step, which simplifies drastically the original problem.

By iterating over we can get the optimal solution of the form:

$$V_i(x_i) = \underset{u_i}{\text{minimize}} \left[l(x_i, u_i) + V_{i+1}(f(x_i, u_i)) \right]$$

$V_M(x_M)$ can be interpreted as the optimal cost that we have to pay if we start from state x_M at time M onward and behaving optimally. This function is called Value function or Optimal cost-to-go.

Value function

Optimal cost-to-go

In conclusion:

Given a discrete-time finite-horizon OCP, we can initialize the dynamic programming algorithm writing the value function for the last time step

$$V_N(x_N) = l_f(x_N)$$

And then you use the recursive optimality principle (backward in time):

$$V_i(z) = \underset{u}{\text{minimize}} \, l(z, u) + V_{i+1}(f(z, u)) \qquad i = N-1, ..., 0$$

Once you have $V_i \; \forall i \in [0, N]$ you can compute the optimal control as:

$$\boxed{u_i^*(x) = \underset{u}{\text{argmin}} \, l(x, u) + V_{i+1}(f(x, u))}$$

Observations:

- u_i^* is not only an optimal control trajectory, but an optimal feedback control policy, because u can be computed as function of x so depending where we are in the state we get a different value of u.
- The function to minimize is often called Q function.
- the formulation of $V_i(z)$ is a parametric optimization, which is different from numerical optimization, because the result is not a value, but a function.

optimal feedback control policy

Q function

parametric optimization

numerical optimization

In summary:

1. Dynamic Programming is a global method, so it tries to get the global solution for a discrete time OCP.
2. The solution is not an open loop trajectory but it is a feedback control policy. So at every state and every time it tells you what the optimal solution is.
3. The main problem with dynamic programming is that it is applicable only in specific cases.

(a) discrete state and control space. So the number of states and control inputs is bounded. (tipically not the case in robotics) In this scenario parametric optimization is replaced by numerical optimization applied to all possible states.
The results are stored in a table called lookup table.
The main disadvantage is the so called Curse of dimensionality. In fact, one may think, if we have a continuos robotic problem, can we discretize the state and the control space with a sufficiently fine grid. Yes, but the only problem is that if you want to discretize with a fine grid than you will get a very large state and control space (even if it is discrete).
So suppose that you have a robot manipulator with 6 joints and you want to discretize the range of each joint both in position and velocity in 10 values. In this case you are going to have 10^{12} possible states which is a big number.
The table that you are going to store in memory is of the same size, it has 10^{12} columns.
So Dynamic programing is only applicable up to 2 to 4 states-control.
However, in Dynamic programming it is easy to handle integer variables.

lookup table

Curse of dimensionality

(b) Linear dynamics and quadratic cost function. In this case the problem that you have to solve is a Linear Quadratic Regulator (LQR).
The problem that we try to solve becomes a quadratic program that has a closed solution.

Linear Quadratic Regulator (LQR)

4. There are variants of Dynamic programming that go under the name of Approximate Dynamic Programming or Neural Dynamic Programming and those are basically Reinforcement Learning algorithms.

Approximate Dynamic Programming

Neural Dynamic Programming

2.3 Direct methods

Direct methods is a family of methods where there are different classes of methods, and the main difference w.r.t. Dynamic Programming is that they are local: i.e. they do not try to solve the problem for every possible state at every possible time and find the global optimum, but they just try to find a solution that is locally optimal.

Direct methods

It means that if you try to perturbe the solution you can only do worst.

2.3.1 Introduction

The key idea of Direct methods is really simple and intuitive.

Considering the following optimal control problem, which looks a lot like an optimization problem, you proceed to discretize the parameters of the problem. So you take the trajectory and you parametrize such trajectory (e.g. with a polynomial) and obtain a finite number of parameters to represent this trajectory (e.g. coefficients of the polynomial).

In this way the number of decision variables is now finite.

The second problem related to the constraints (which are infinitly many) and you discretize em in time: i.e. you check the constraint at each time step (e.g. every 10ms).

The problem now becomes an optimization problem, so you can apply Newton Method to try to find the local solution.

However, the discretization process is not as simple as it sounds.

In summary:

- Discretize the Optimal control Problem and use a non linear programming solver (NLP) to find a local optimum.

non linear programming solver (NLP)

- The Discretization takes two steps:
  1. Parametrize state and control trajectories (e.g. with polynomials).
  2. Enforce constraints (both dynamics and path inequalities) on a grid.

$$t_0 < t_1 < t_2 < ... < t_N$$

Tons of theory/methods to choose trajectory parametrization and time grid so that NLP is a good approximation of OCP and well conditioned (a matrix is ill-conditioned if it has both very large and very small elements, in this conditioned it may occur loss of precision).

well conditioned

ill-conditioned

- There are three main families of Direct methods:
  1. Single Shooting aka Sequential approach.

     Single Shooting

     Sequential approach

     - Discretize only the control trajectory ($u(t)$).
     - Comput the state $x(t)$ integrating the dynamics.
     - No need to have dynamic constraints (you can omit the state as a control variable of the problem, because you can indirectly compute it from $u(t)$ and from the knowledge of the dynamics).

  2. Collocation aka Simultaneous approach.

     Collocation

     Simultaneous approach

     - Discretize both $x(t)$ and $u(t)$
     - Enforce the dynamics on a time grid. In this case you have to use the dynamics as a constraint to make sure since you have both $x(t)$ and $u(t)$ as decision variables, the solver chooses the state in a coherent way.

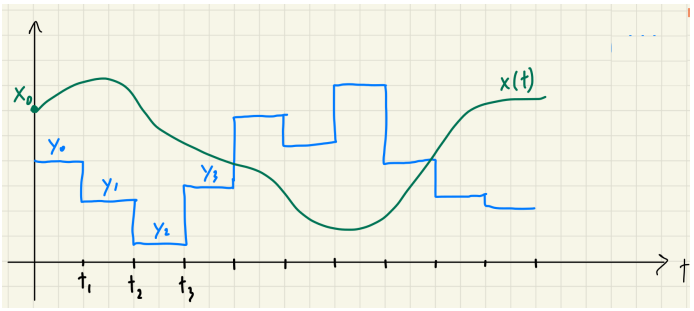
  3. Multiple Shooting

     Multiple Shooting

     - Discretize $u(t)$
     - Discretize $x(t)$ only at few points in time
     - Compute intermediate values of $x(t)$ by matter of integration

2.3.2 Single Shooting - Sequential Approach

In single shooting you discretize the control $u(t)$ on a fixed grid $0 = t_0 < t_1 < ... < t_N = t_f$



A very classical option for the discretization of the control is to do a piece-wise constant trajectory. So the control can only change at certain time and it remains constant.

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

You then proceed to compute $x(t)$ from $u(t)$ by integrating the dynamics

$$\begin{aligned} \dot{x} &= f(x, u, t) \\ x(0) &= x_0 \end{aligned}$$

Of course the control can be discretize in other ways, but in practice this is what happends most of the time (e.g. piece-wise polynomial).

The non linear program that you get by using single shooting is:

$$\underset{y}{\text{minimize}} \int_0^{t_f} l(x(t;y), u(t;y))\, dt \; + \; l_f(x(t_f;y))$$

subject to:

- $g(x(t_i;y), u(t_i;y), t_i) \leq 0 \qquad i = 0...N$ are the Discretized Path constraints

  Discretized Path constraints

The Running cost integral tipically is computed when integrating dynamics, and you do it by using an augmented state augmented state

$$c(t) = \int_0^t l(\cdot,\cdot)\, dt$$

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

Given that we can define the augmented state $\bar{x}$ as:

$$\bar{x} = (x, c)$$

$$\dot{\bar{x}} = \begin{bmatrix} f(x,u,t) \\ l(x,u) \end{bmatrix}$$

What we still need to do in order to implement a single shooting optimal control method are the computation of the sensitivities.

Computing the sensitivities

The sensitivities are the derivatives of the integration scheme that we are using to derive the state from the control. sensitivities

Why do we need to differentiate between integration schemes? Because we are using an optimal optimization solver that requires the gradient of the cost function to work. So our cost function depends both on the state and the control, but actually the state depends itself on the control and this dependency is decided by the integration scheme that we are using.

So when we are computing the gradient of the cost function we need to take into account the gradient of the integration scheme, because to compute the cost function we need to evaluate the integration scheme.

This is not very difficult, it boils down to apply the chain rule in order to compute the derivatives of a function that it is itself a function of a variable, which is not the variable we want to differentiate, but an indirect dependency. (in broad terms: we want to differentiate the cost function which depends on the state which once again depends on u)

There is an efficient way to conduct this computation which is tipically used to implement single shooting methods.

Let us assume to have:

- Cost function and constraints of a single shooting NLP depend on x and u
- $x(t)$ us computed from $u(t)$ by matter of integration
- $u(t)$ is discretized in a piecewise matter or alternative way
  This means that given the generic dynamic equation:

$$\dot{x} = f(x(t), u(t), t)$$

Assuming a piece wise constant discretization of the control $u(t)$ of the form:

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

we can reformulate the differential equation with the notion of the control variable discretization:

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

Then we have a cost function (which is a function of y)

$$c(y) = \int_0^T l(x(t), u(t))\, dt + l_f(x(T)) \approx \sum_{i=0}^{N-1} l(x_i, y_i) \cdot h + l_f(x_N)$$

where the running cost integral is approximated with the Euler Method instead of using the augmented state. Tipically we are not required to be accurate in the computation of the running cost, but we have to be accurate during the computation of the dynamics.

We obtained our cost that we would like to differentiate, i.e. we want to compute:

$$\frac{dc}{dy} = \sum_{i=0}^{N-1} \frac{dl(x_i, y_i)}{dy} \cdot h + \frac{dl_f(x_n)}{dy}$$

In order to do so we need to compute the two unique differentiation in the above formulation

- l depends on y directly (because is a function of y) but it also depends on y indirectly (because is a function of x that you obtain by integrating the dynamics using y as a control input). Therefore we need to take care of both dependencies

$$\frac{dl}{dy} = \frac{\partial l}{\partial x_i}\frac{dx_i}{dy} + \frac{\partial l}{\partial y}$$

- For the second term we only have the indirect dependency because the final cost has no direct dependency from the control.

$$\frac{dl_f}{dy} = \frac{\partial l_f}{\partial x_N}\frac{dx_N}{dy}$$

From an analysis of the two expressions we can conclude that the partial derivatives of l and l_f are usually easy to compute, because they are just the derivatives of these functions (which is the running cost, that we decide as a designer of the optimal control problem) and it is tipically a simple function (e.g. quadratic function that penalizes the deviations of the state or the control from a given reference).

The total derivatives of the two expressions are less trivial to compute and they depend on the integration scheme. Their meaning is: how much a certain state is gonna change if I modify my control input (considering the whole trajectory of control inputs, not a single control input).

From this observations we can conclude that the partial derivatives are vectors whereas the total derivative are matrices.

We will now look into how to compute the $\frac{dx_i}{dy}$ terms.

Let us introduce the integrator function Φ, which is the discretized dynamics function, such that:

integrator function

discretized dynamics function

$$x_{i+1} = \Phi(x_i, y_i) \qquad \Rightarrow \qquad \frac{dx_{i+1}}{dy} = \frac{\partial \Phi_i}{\partial x_i}\frac{dx_i}{dy} + \frac{\partial \Phi_i}{\partial y}$$

This is basically what our integration scheme computes. We have derived a recoursive relationship that allows us to compute the term $\frac{dx_i}{dy}$ for time $i+1$ if we know the value of that term at the previous time step.

However, we need to know the other two derivatives of the integrator function that will depend on our choice of integration scheme.

Note that:

$$\frac{\partial \Phi}{\partial y} = \left(\frac{\partial \Phi_i}{\partial y_0}, ..., \frac{\partial \Phi_i}{\partial t_{N-1}}\right)$$

where

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

Two questions remain unanswered:

1. How do we initialize the computation? (i.e. what is the value of $\frac{dx_0}{dy}$)

   We can initialize the computation with

   $$\frac{dx_0}{dy} = 0$$

   Because the input only affect the future not the present, the initial condition does not depend on the choice of control input.

2. How do we compute the other two terms in the above expression?

   This depends on the integration scheme that we are using. Let us review all the method that we have reviewed in AppendixA.

   - **Explicit Euler** Suppose Φ is explicit Euler (RK1), than:

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

     where f is the continuous time dynamic of the system.

     $$\frac{\partial \Phi}{\partial x} = I + h\, \frac{\partial f}{\partial x}$$
     $$\frac{\partial \Phi}{\partial u} = h\, \frac{\partial f}{\partial u}$$

     Of course we need the derivatives of the continuous time dynamics.
   - **RK4** Suppose that Φ is RK4 integrator and $f(\cdot)$ is time independent.

     $$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + \frac{1}{2}hk_1, y_i) \\ k_3 &= f(x_i + \frac{1}{2}hk_2, y_i) \\ k_4 &= f(x_i + hk_3, y_i) \\ x_{i+1} &= x_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \triangleq \Phi_i(x_i, y_i) \end{aligned}$$

     The sensitivity w.r.t. x is:

     $$\frac{\partial \Phi_i}{\partial x_i} = I + \frac{h}{6}\left(\frac{\partial k_1}{\partial x_i} + 2\frac{\partial k_2}{\partial x_i} + 2\frac{\partial k_3}{\partial x_i} + \frac{\partial k_4}{\partial x_i}\right)$$

     where:

     $$\begin{aligned} \frac{\partial k_1}{\partial x_i} &= \left.\frac{\partial f}{\partial x}\right|_{x_i} \\ \frac{\partial k_2}{\partial x_i} &= \left.\frac{\partial f}{\partial x}\right|_{x_{i2}} \left(I + \frac{1}{2}h\frac{\partial k_1}{\partial x_i}\right) && x_{i2} = x_i + \frac{1}{2}hk_1 \\ \frac{\partial k_3}{\partial x_i} &= \left.\frac{\partial f}{\partial x}\right|_{x_{i3}} \left(I + \frac{1}{2}h\frac{\partial k_2}{\partial x_i}\right) && x_{i3} = x_i + \frac{1}{2}hk_2 \\ \frac{\partial k_4}{\partial x_i} &= \left.\frac{\partial f}{\partial x}\right|_{x_{i4}} \left(I + \frac{1}{2}h\frac{\partial k_3}{\partial x_i}\right) && x_{i4} = x_i + \frac{1}{2}hk_3 \end{aligned}$$

     whereas the sensitivity w.r.t y is:

     $$\frac{\partial \Phi_i}{\partial y_i} = \frac{h}{6}\left(\frac{\partial k_1}{\partial y_i} + 2\frac{\partial k_2}{\partial y_i} + 2\frac{\partial k_3}{\partial y_i} + \frac{\partial k_4}{\partial y_i}\right)$$

$$
\begin{aligned}
\frac{\partial k_1}{\partial y_i} &= \left.\frac{\partial f}{\partial y}\right|_{xi} \\
\frac{\partial k_2}{\partial y_i} &= \left.\frac{\partial f}{\partial y}\right|_{x_{i2}} + \left.\frac{\partial f}{\partial x}\right|_{x_{i2}} \left(\frac{1}{2} h \frac{\partial k_1}{\partial y_i}\right) \\
\frac{\partial k_3}{\partial y_i} &= \left.\frac{\partial f}{\partial y}\right|_{x_{i3}} + \left.\frac{\partial f}{\partial x}\right|_{x_{i3}} \left(\frac{1}{2} h \frac{\partial k_2}{\partial y_i}\right) \\
\frac{\partial k_4}{\partial y_i} &= \left.\frac{\partial f}{\partial y}\right|_{x_{i4}} + \left.\frac{\partial f}{\partial x}\right|_{x_{i4}} h \frac{\partial k_3}{\partial y_i}
\end{aligned}
$$

In summary:

- Remove x from problem variables
- Discretize $u(t) = y_i \qquad \forall t \in [t_i, t_{i+1}]$
- Compute x by integrating dynamics $\dot{x} = f(x, u)$
- Since x is computed by integrating the dynamics we do not need to have the dynamics as a constraint in the problem. i.e. remove dynamics from constraints of the problem
- Tipically in single shooting you use a High-order integration scheme because it is more efficient
- In any case, when you use an integration scheme you need to differentiate the integration scheme itself in order to compute the gradient of the cost and of the constraints. (Hardest part in the implementation)

2.3.3 Collocation - Simultaneous Approach

In collocation, we not only discretize the control input $u(t)$, but we also discretize the state $x(t)$ trajectory.

We are going to have many more decision variables in our problem, we are also gonna have many more constraints given that in this case we need to represent the dynamics of the system as a constraint in the problem. However, this is not necessarily slower to solve even though we have so many variables and constraints.

Formally:

- We discretize $x(t)$, tipically with polynomials (e.g. order k=0, piece-wise constant) on fine grid (i.e. the time between two successive steps is going to be small) which defines how accurate the representation of the dynamics is.

  $$x(t) = s_i \qquad \forall t \in [t_i, t_{i+1}] \qquad \text{One polynomial for each time interval}$$

  This was not the case for single shooting where you could take a large step size and use an high-order integration scheme to retrieve an accurate integration of the dynamics.
- Replace the dynamics $\dot{x} = f(x, u)$ with:

  $$\frac{s_{i+1} - s_i}{t_{i+1} - t_i} - f\left(\frac{s_{i+1} + s_i}{2}, y_i\right) = 0 \qquad\qquad i = 0, ..., N-1$$

  The first fraction is an approximation of $\dot{x}$, whereas the $\dfrac{s_{i+1} + s_i}{2}$ is an approximation of x. This formulation can be interpreted as a constraint that relates s_{i+1} with s_i and y_i, i.e.:

  $$c(s_{i+1}, s_i, y_i) = 0$$

- Approximate the cost integral:

  $$\int_{t_i}^{t_{i+1}} l(x(t), u(t))\, dt \approx l\left(\frac{s_i + s_{i+1}}{2}, y_i\right)(t_{i+1} - t_i) \triangleq l_i(s_i, s_{i+1}, y_i)$$

  It is not exactly an Euler integration scheme, but it is quite similar.

We can now write down the complete problem to see how it looks like.

$$\underset{s,y}{\text{minimize}} \sum_{i=0}^{N-1} l_i(s_i, s_{i+1}, y_i) + l_f(s_N)$$

subject to:

- $s_0 - x_0 = 0$
- $c_i(s_i, s_{i+1}, y_i) = 0 \qquad i = 0, ..., N-1$
- $g_i(s_i, y_i, t_i) \leq 0 \qquad i = 0, ..., N$

This problem has a very important feature which is sparsity, (e.g. a matrix is sparse if it has a lot of zeros inside), i.e. the gradient, the jacobian, the hessian of the problem (so the derivatives of cost function and constraints) are sparse matrices and vectors. `sparsity`

It is important because if you have a sparse problem and you use a solver that it is able to exploit the sparcity then it can be solved much more efficiently than if you just neglet the sparcity.

Why can we say that this problem formulation is sparse? What is the feature that gives away the sparsity? For instance from the constraint: $c_i(s_i, s_{i+1}, y_i) = 0$, we can deduct that the Jacobian will be sparse

The sparsity is given to this term by the dependency of some limited variables, in other terms the constraint does not depend on all the variables. In this case when we are going to compute the derivative of the constraint w.r.t. all the variables, many of them will be zero.

The same concept applies to the cost function (dependency on a subset of all the variables). For this term the hessian will be sparse (however the gradient will not be sparse).

2.3.4 Multiple shooting

Multiple shooting it is a middle ground between collocation and single shooting.

- Discretize the control $u(t)$ on a coarse grid $t_0 < t_1 < ... < t_n$ `coarse grid`

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

- Integrate numerically ODE in each interval $[t_i, t_{i+1}]$, starting from a state variable s_i:

$$\begin{aligned} \dot{x}(t) &= f(x_i(t), y_i) \qquad\qquad t \in [t_i, t_{i+1}] \\ x_i(t_i) &= s_i \end{aligned}$$

- Obtain $x_i(t)$ and Numerically compute the integrals:

$$l_i(s_i, y_i) = \int_{t_i}^{t_{i+1}} l(x_i(t), y_i)\, dt$$

So in multiple shooting you have the control input discretized as in single shooting, but you also have some state variables and this is the main difference between the two approaches.

Contrary to Collocation where you have the state variable on a fine grid, here you have the state variables on a coarse grid, that guarantees very few state variables.

The remaining values of the states inbetween this state variables will be computed by matter of numerical integration.

What can happen in multiple shooting is that the result of the integration does not match with the value of the state variable that you have at the end, so extra contraints needs to be formulated to guarantee continuity of the state trajectory. The problem formulation then becomes:

$$\underset{s,y}{\text{minimize}} \sum_{i=0}^{N-1} l_i(s_i, y_i) \; + \; l_f(x_N)$$

subject to:

- $s_0 - x_0 = 0$

- $s_{i+1} - x_i(t_{i+1}, s_i, y_i) = 0$ where $x_i(\cdot)$ is the result of numerical integration starting from s_i. And these type of constraints are called continuity contraints.
- $g(s_i, y_i) \leq 0$

continuity contraints

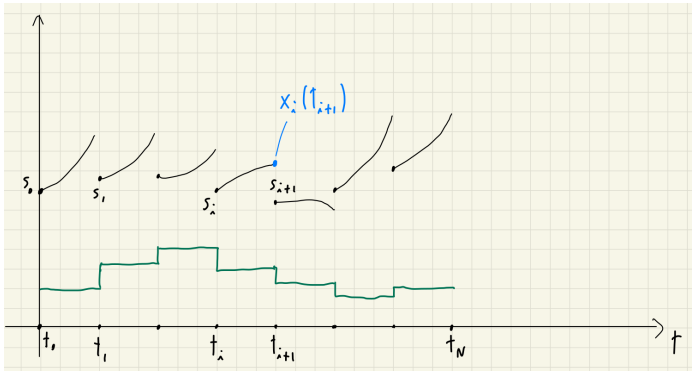


Figure 2.3: Multiple shooting sketch

During the optimization problem the solver is allowed to have partial solution that have discontinuities and then the gaps will be collapsed to zero little by little. At the end of the solution problem we will obtain a continuous state trajectory.

This may seem like a bad property of the problem, but actually is what allows the solver to find better solutions most of the time.

In practical cases, Collocation and Multiple Shooting can find better solutions than Single Shooting.

2.3.5 Comparison of Direct methods

Single shooting	**Multiple shooting**	**Collocation**
⊕ Small problem	⊜ Medium problem	⊖ Large problem
⊕ Need only initial guess for $u(t)$	⊕ Sparser than single shooting	⊖ Cannot adapt time grid
⊕ Can use off-the shelp ODE solver	⊖ Less sparse than collocation	⊕ Sparse problem
⊖ Cannot exploit knowledge of x in initialization	⊕ Unstable Ok	⊕ Work well with unstable systems
⊖ ODE solution can depend very nonlinearly on y	⊕ Initialize $x(t)$	⊕ Can initialize x
⊖ Numerical issues for unstable systems		

2.4 Linear Quadratic Regulator (LQR)

Linear Quadratic Regulator LQR is a method to solve a very specific kind of optimal control problem. What is special about this optimal control problem is that is particularly simple because the dynamics is represented by a discrete-time linear system:

Linear Quadratic Regulator LQR

discrete-time linear system

$$x_{t+1} = A\,x_t + B\,u_t$$

And the objetive is to choose the control inputs $u_0, u_1, \ldots$ so that:

- x is "small" $\Rightarrow$ good regulation/performance

- u is "small" $\Rightarrow$ small effort/energy consumption

These are usually competing objectives (you cannot have both at the same time) because if you keep u very small than x does whatever it wants, and tipically if you want to keep x small you need some control effort.

So we need to find a trade-off between these two objectives, and that is what LQR does.

More precisely the LQR control problem takes the following form:

$$\underset{U}{\text{minimize}}\, J(U) = \sum_{i=0}^{N-1} = (x_i^T\, Q\, x_i + u_i^T\, R\, u_i) + x_N^T\, Q_f\, x_N$$

subject to:

- $x_{i+1} = A\, x_i + B\, u_i \qquad i = 0, ..., N-1$

With a quadratic cost function, where$Q$ defines the quadratic form of the state, R is the matrix that defines the quadratic form of the control and Q_f defines the quadratic form of the final cost.

All the quadratics form are strictly positive definites (strinctly positive eigenvalues) or semi-positive definites (eigenvalues can be zero), i.e.

$$Q = Q^T \geq 0 \qquad ; \qquad Q_f = Q_f^T \geq 0 \qquad ; \qquad R = R^T > 0$$

As a consequence, any time that you have a state or a control that is not zero than you are increasing the cost. So idealy you would like to have all the variables at zero to achieve zero cost which is the minimum of the cost function (however that is not going to be possible).

The two knobs to tune the problem to achieve the behaviour of the system that you want are Q and R, in particular:

- Q large $\Rightarrow$ x small is more important
- R large $\Rightarrow$ u small is more important

The LQR optimal control problem can be rewritten in the following form

$$\underset{X,U}{\text{minimize}}\, J(X, U) = \|\bar{Q}\, X\|^2 + \|\bar{R}\, U\|^2$$

subject to:

- $X = G\, U + H\, x_0$
- with block diagonal matrices:

$$\bar{Q} \triangleq diag(Q^{1/2}, ..., Q^{1/2}, Q_f^{1/2}) \qquad \text{and} \qquad \bar{R} \triangleq diag(R^{1/2}, ..., R^{1/2})$$

where we introduced the variables X and U containing the whole trajectory of the state and the control (discretized).

Also the dynamics can be written in linear form:

$$\begin{aligned}
x_1 &= A\, x_0 \,+\, B\, u_0 \\
x_2 &= A\, x_1 \,+\, B\, u_1 \\
&\quad \text{substituting the first expression into the second} \\
&= A^2\, x_0 \,+\, AB\, u_0 + B\, u \\
x_3 &= A^3\, x_0 + A^2 B u_0 + AB u_1 + B\, u_2
\end{aligned}$$

Iterating through each state we can represent the linear dynamics system in the matrix form:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} A \\ A^2 \\ \vdots \\ \vdots \\ \vdots \\ A^N \end{bmatrix} x_0 + \begin{bmatrix} B & & & \\ AB & & & \\ \vdots & & & \\ \vdots & & & \\ \vdots & & & \\ A^{N-1}B & A^{N-2}B & \cdots & B \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ \vdots \\ \vdots \\ u_{N-1} \end{bmatrix}$$

So we can see that the state trajectory takes the following compact form:

$$X = H\,x_0 + G\,U$$

If we eliminate X from the problem variables given the linear dynamic constraints, obtaining the new problem without constraints:

$$\begin{aligned}\underset{U}{\text{minimize}}\ J(U) &= \|\bar{Q}(G\,U + H\,x_0)\|^2 + \|\bar{R}\,U\|^2 \\ &= \left\| \begin{bmatrix} \bar{Q}\,G \\ \bar{R} \end{bmatrix} U + \begin{bmatrix} \bar{Q}\,H\,x_0 \\ 0 \end{bmatrix} \right\|^2\end{aligned}$$

Now we can clearly see that the LQR optimal control problem is nothing more that a big least-squares problem, with dimension $N(n+m) \times Nm$ and cost $O(N^3nm^2)$ using Quadratic regulator.

least-squares problem

where:

- N is the length of the horizon (tipically quite large)
- n is the size of the state
- m is the size of the control

horizon

As a consequence, LQR gives us a more efficient way of computing the solution of the least square program by exploiting the structure of the program.

2.4.1 LQR via Dynamic Programming

LQR can be derived from the principle of Dynamic Programming.

Dynamic Programming is in general very difficult to apply to a real problem because you need to solve a parametric optimization problem. And LQR is one of those rare cases in which you can actually use Dynamic Programming, because the parametric optimization problem is a quadratic program (can be solved analytically).

We therefore proceed to write down the LQR problem defininig the value function (optimal cost starting at a certain state, at a certain time):

$$V_t(z) = \underset{u_t=(u_t,\dots u_{N-1})}{\text{minimize}} \sum_{k=1}^{N-1} (x_k^T\,Q\,x_k + u_k^T\,R\,u_k)\ +\ x_N^T\,Q_f\,x_N$$

subject to:

- $x_t = z$
- $x_{k+1} = A\,x_k\ +\ B\,u_k \qquad \forall k = t, ..., N-1$

Therefore the value function $V_t(z)$ represent the optimal cost starting in z at time t.

Of course the value function at time 0 starting at state x_0 (i.e. $V_0(x_0)$) is nothing more than the optimal cost of the original problem.

The objective of Dynamic Programming is recursively minimize the value function starting at the last time step N backwards. The reason why we do so is because for the last time step we already know the value function, i.e. the terminal cost of the LQR problem:

$$V_N(z) = z^T\,Q_f\,z$$

From this point we can leverage the Bellman optimality equation that defines the value function at time i as a function of the value function at time $i+1$ and apply this recursively going backward in time.

$$V_t(z) = \underset{w,U_{t+1}}{\text{minimize}}\ z^TQz + w^TRw\ +\ \sum_{k=t+1}^{N-1} (x_k^TQx_k + u_k^TRu_k) + x_N^TQ_fx_N$$

subject to:

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

- $u_t = w$

Note that the value function above was split in the cost at time t (left) and the cost starting from the t going onwards until N.

The second part of the problem is nothing more than the value function at time $t+1$, i.e. $V_{t+1}(Az + Bw)$. So I can replace that part obtaining:

$$v_t(z) = \underset{w}{\text{minimize}}\, z^T Q z + w^T R w \; + \; V_{t+1}(Az + Bw)$$

NOTE: $Az + Bw$ is the next state i.e. x_{t+1}.

Now that we obtained the recursive optimality equation we can initialize it with the quadratic form for time N and compute the value function recursively backward in time to see if it mantains this quadratic form or if it takes a more complex form.

Assuming the v_{t+1} is quadratic (that is true for $t+1 = N$), we can define the following quantities:

$$V_{t+1}(z) = z^T P_{t+1} z \qquad P_{t+1} = P_{t+1}^T \geq 0 \qquad P_n = Q_f$$

And expand the minimization problem formulation with this new quantities

$$\begin{aligned} V_t(z) &= \min_w (z^T Q z + w^T R w + V_{t+1}(Az + Bw)) \\ &= z^T Q z \; + \min_w (w^T R w \; + (Az + Bw)^T P_{t+1} (Az + Bw)) \\ &= z^T Q z \; + \min_w (w^T (R + B^T P_{t+1} B) w + 2 z^T A^T P_{t+1} B w) + z^T A^T P_{t+1} A z \\ &= z^T (Q + A^T P_{t+1} A) z + \min_w (\, (w^T H w + 2 g^T w) \triangleq f(w) \,) \end{aligned}$$

where H is the Hessian of the quadratic form and g is the gradient of the linear form.

Hessian of the quadratic form

gradient of the linear form

The minimization process is achieved by computing the gradient of the objective function to minimize (i.e. $f(w)$) and set it to zero:

$$\nabla f = 2Hw + 2g = 0$$

which has solution:

$$\begin{aligned} w^* &= -H^{-1} g = -(R + B^T P_{t+1} B)^{-1} B^T P_{t+1} A z \\ &= K_t z \end{aligned}$$

Note that the matrix H is always invertible by assumption: in fact, R and P_{t+1} are positive definite and their sum will always be a positive definite matrix, which is always invertible.

At this point we can compute $V_t(z)$ with the optimal value of the decision variable $w = w^*$.

$$\begin{aligned} V_t(z) &= z^T (Q + A^T P_{t+1} A) z + g^T H^{-1} \cancel{H H^{-1}} g - 2 g^T H^{-1} g \\ &= z^T (Q + A^T P_{t+1} A) z - g^T H^{-1} g \\ &= z^T (Q + A^T P_{t+1} A - A^T P_{t+1} B H^{-1} B^T P_{t+1} A) z \end{aligned}$$

The result is a purely quadratic function in z and the matrix inside is what we called P_t. By doing so we demonstrated that if the value function

$$V_{t+1}(z) = z^T P_{t+1} z$$

is purely quadratic at time $t+1$ than also the value function at time t is purely quadratic, i.e.

$$V_t(z) = z^T P_t z$$

Therefore the value function is quadratic for all t and we have a formulation of the value function at time t starting from the value function starting at time $t+1$.

We can also prove that if the value function is positive semi-definite at time $t+1$ then also the value function at time t will be positive semi-definite.

Proof. Given $P_{t+1} \geq 0, Q \geq 0, R > 0$ then $P_t \geq 0$, which translates into proving that :

$$P_t = Q + ATPA - A^T PB(R + B^T PB)^{-1} B^T PA \geq 0$$

$$P - PB(R + B^T PB)^{-1} B^T P \geq 0$$
$$S(I - S^T B(R + B^T SS^T B)^{-1} B^T S)S^T \geq 0$$
$$I - S^T B \left(\begin{bmatrix} L & B^T S \end{bmatrix} \begin{bmatrix} L^T \\ S^T B \end{bmatrix} \right)^{-1} B^T S \geq 0$$
$$I - \begin{bmatrix} 0 & 1 \end{bmatrix} V^T (VV^T)^{-1} V \begin{bmatrix} 0 \\ 1 \end{bmatrix} \geq 0$$
$$I - \begin{bmatrix} 0 & 1 \end{bmatrix} V^\dagger V \begin{bmatrix} 0 \\ 1 \end{bmatrix} \geq 0$$
$$\begin{bmatrix} 0 & 1 \end{bmatrix} (I - V^\dagger V) \begin{bmatrix} 0 \\ 1 \end{bmatrix} \geq 0$$
$$I - V^\dagger V \geq 0$$

$$S \triangleq P^{1/2} \Rightarrow P = SS^T$$
$$L \triangleq R^{1/2} \Rightarrow R = LL^T$$
$$B^T S = \begin{bmatrix} L & B^T S \end{bmatrix} \begin{bmatrix} O_m \\ I_n \end{bmatrix} = V \begin{bmatrix} O_m \\ I_n \end{bmatrix}$$
$$I = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Where the left hand side of the inequality is the Projection matrix (into nullspace of V) which eigenvalues are either 0 or 1 which implies that the inequality is always satisfied. □

Projection matrix

Summary of the DP approach to solve the LQR problem:

1. Set $P_N = Q_f$ initialization of value function at time N with the terminal cost.
2. For $t = N...1$, compute
$$P_{t-1} = Q + A^T P_t A - A^T P_t B(R + B^T P_t B)^{-1} B^T P_t A$$
3. For $t = 0...N-1$ compute the gain matrix
$$K_t = -(R + B^T P_{t+1} B)^{-1} B^T P_{t+1} A$$
4. For $t = 0...N-1$, compute the control input
$$u_t = K_t x_t$$

You may notice that for this specific problem we actually not only recover the trajectory of optimal control inputs but we also have an optimal control policy given that u_t is a linear function of the state. This gives us something more than an open-loop trajectory in contrast with the Direct methods (especially in Single Shooting).

The obtained optimal linear feedback policy can be used online on the real system: we can compute u based on the current state, which might be different from what we would expect from the nominal plant (more useful in the presence of perturbations).

optimal linear feedback policy

2.4.2 Steady State Regulator

Usually P_t converges as t decreases below N.

Steady-state value satisfies:

$$P_s = Q + A^T P_s A - A^T P_s B(R + B^T P_s B)^{-1} B^T P_s A$$

Discrete-time algebraic Riccati Equation (DT-ARE) Can be solved by direct method or iterating RIccati recursion.

Discrete-time algebraic Riccati Equation (DT-ARE)

Therefore, for t not close to N we have constant feedback gain:

$$u = -(R + B^T P_s B)^{-1} B^T P_s A$$

2.4.3 Time Varying System

LQE is easily extended to Time Varying systems:

$$x_{t+1} = A_t x_t + B_t u_t$$

and Time Varying cost matrices:

$$\sum_{t=0}^{N-1} (x_t^T Q_t x_t + u_t^T R_t u_t) + x_N^T Q_f x_N$$

In this case there need not to be a steady-state solution though.

2.4.4 Inhomogeneous system and costs

$$\text{minimize} \sum_{t=0}^{N-1} \begin{bmatrix} x_t^T & u_t^T & 1 \end{bmatrix} \begin{bmatrix} Q_t & S_t & q_t \\ S_t^T & R_t & s_t \\ q_t^T & s_t^T & 0 \end{bmatrix} \begin{bmatrix} x_t \\ u_t \\ 1 \end{bmatrix} + \begin{bmatrix} x_N^T & 1 \end{bmatrix} \begin{bmatrix} Q_f & q_N \\ q_N^T & 0 \end{bmatrix} \begin{bmatrix} x_N \\ 1 \end{bmatrix}$$

subject to:

- $x_{t+1} = A_t x_t + B_t u_t + c_t$
- $x_0 = x^{init}$

Define augmented state $\bar{x} = \begin{bmatrix} x & 1 \end{bmatrix}$

$$\begin{bmatrix} x_{t+1} \\ 1 \end{bmatrix} = \begin{bmatrix} A_t & c_t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ 1 \end{bmatrix} + \begin{bmatrix} B_t \\ 0 \end{bmatrix} u_t$$

$$\bar{x}_{t+1} = \bar{A}_t \bar{x}_t + \bar{B}_t u_t \quad \Leftarrow \text{Homogeneous}$$

$$J = \sum \begin{bmatrix} \bar{x}_t^T & u_t^T \end{bmatrix} \begin{bmatrix} \bar{Q}_t & \bar{S}_t \\ \bar{S}_t^T & R_t \end{bmatrix} \begin{bmatrix} \bar{x}_t \\ u_t \end{bmatrix} + \bar{x}_N^T \bar{Q}_f \bar{x}_N$$

with:

$$\bar{Q}_t \triangleq \begin{bmatrix} Q_t & q_t \\ q_t^T & 0 \end{bmatrix} \qquad \bar{S}_t \triangleq \begin{bmatrix} S_t \\ s_t^t \end{bmatrix}$$

The only difference is cross term $x^T S u$ in cost function. DP solution easily extended.
Fin doptimal fedback policy for augmented system:

$$u = \bar{K}\bar{x} = \begin{bmatrix} K & k \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix} = Kx + k$$

Why inhomogeneous systems and costs:

- Tracking problems
- Linearization of nonlinear systems
- Filtering problems (e.g. Kalman)

2.4.5 Tracking problems

$$J = \sum_{t=0}^{N-1} (x_t - \bar{x}_t)^T Q (x_t - \bar{x}_t) + (u_t - \bar{u}_t)^T R (u_t - \bar{u}_t)$$

$\bar{x}, \bar{u}$ = reference trajectories to track.
Can be rewriteen as inhomogeneous cost:

inhomogeneous cost

$$J = \sum x^T Q x + u^T R u - 2\bar{x}^T Q x - 2\bar{u}^T R u$$
$$= \sum \begin{bmatrix} x^T & u^T & 1 \end{bmatrix} \begin{bmatrix} Q & 0 & Q\bar{x} \\ 0 & R & R\bar{u} \\ \bar{x}^T Q & \bar{u}^T R & 0 \end{bmatrix} \begin{bmatrix} x \\ u \\ 1 \end{bmatrix}$$

2.4.6 Differential Dynamic Programming (DDP)

Differential Dynamic Programming (DDP) is in some sense an extension of LQR, that handles non-linear dynamical systems and also non-linear cost functions. This is a method that can be applied to a much more generic kind of optimal control problems, which takes the form:

Differential Dynamic Programming (DDP)

non-linear dynamical system

non-linear cost function

$$\text{minimize} \sum_{t=0}^{N-1} l_t(x_t, u_t) + l_N(x_N)$$

subject to:

- $x_{t+1} = f(x_t, u_t)$
- $x_0 = x^{init}$

It is not completely a generic control problem because we do not have inequality constraints or any other type of constraints beside the dynamics. But surely more generic than Direct Methods, given that we can handle non linearities.

The idea of DDP vaguely resemble the approach of Newton Method and Numerical Optimization methods.

Start with a guess for the decision variable of the problem U and alternate between the following three steps:

1. Linearize around current trajectory
2. Solve LQR to get variation of U
3. Line search to ensure convergence

This is just an euristic, there is no guarantee of convergence, sometimes it will converge to the global optimum and sometimes it will converge to a local optimum.

The idea it remains the same, i.e. apply dynamic programming, but since you cannot really apply DP to the original system, you apply it to a linearization (aka local approximation) of the system

$$V(z, i) = \underset{U_i}{\text{minimize}}\, J_i(z, U_i) \qquad \text{where:} \quad U_i = (u_i, ..., u_{N-1})$$

We can subsequently apply Bellman optimality principle in order to split the cost in two parts:

$$V(z, i) = \underset{w}{\text{minimize}} \left[l_i(z, w) + V(f(z, w), i + 1) \right] = \min_w Q(z, w)$$
$$Q(\bar{x} + z, \bar{u} + w) \approx Q(\bar{x}, \bar{u}) + \begin{bmatrix} Q_x^T & Q_u^T \end{bmatrix} \begin{bmatrix} z \\ w \end{bmatrix} + \frac{1}{2} \begin{bmatrix} z^T & w^T \end{bmatrix} \begin{bmatrix} Q_{xx} & Q_{xu} \\ Q_{xu}^T & Q_{uu} \end{bmatrix} \begin{bmatrix} z \\ w \end{bmatrix}$$

Since we do not really know how to minimize a non linear function we can take a quadratic approximation (via Taylor expansion) and minimize it.

where:

$$Q_x \triangleq \frac{\partial Q}{\partial x} \qquad Q_u \triangleq \frac{\partial Q}{\partial u} \qquad Q_{xx} \triangleq \frac{\partial^2 Q}{\partial x^2} \qquad Q_{uu} \triangleq \frac{\partial^2 Q}{\partial u^2} \qquad Q_{xu} \triangleq \frac{\partial Q_u}{\partial x}$$

Once defined the derivative of the value function as follows:

$$V' \triangleq V(\cdot, i + 1)$$

We can expand the partial derivatives, using the chain rule:

$$
\begin{aligned}
Q_x &= l_x + f_x^T V'_x \\
Q_u &= l_u + f_u^T V'_x \\
Q_{xx} &= l_{xx} + f_x^T V'_{xx} f_x + V'_x f_{xx} \\
Q_{uu} &= l_{uu} + f_u^T V'_{xx} f_u + V'_x f_{uu} \\
Q_{xu} &= l_{xu} + f_x^T V'_{xx} f_u + V'_x f_{xu}
\end{aligned}
$$

Moreover the quadratic cost function takes the following compact form:

$$
\begin{aligned}
Q(\bar{x}, \bar{u}) &= l(\bar{x}, \bar{u}) + V'(f(\bar{x}, \bar{u})) \\
\bar{Q} &= \bar{l} + \bar{V}'
\end{aligned}
$$

The DDP problem is reduce to minimizing the local quadratic model of Q. In doing so we get:

$$
w^* = \underset{w}{\operatorname{argmin}}\, Q(z, w) = -Q_{uu}^{-}1(Q_u + Q_{ux}\, z)
$$

This expression is a bit more complicated than what we had before with LQR because it is not a purely linear function of the state, but it is an affine function of the state (Bias/constant term Q_u and linear term $Q_{ux}z$). The reason why it is not purely linear is because the cost is not purely quadratic but you have also the affine term (i.e. Linear terms).

Substituting w^* inside the definition of the local approximation of the Q function Q to recover the value function at the previous time step V, i.e.:

$$
\begin{aligned}
V &= \bar{Q} + Q_x^T z + Q_u^T w^* + \frac{1}{2} z^T Q_{xx} z + \frac{1}{2} {w^*}^T Q_{uu} w^* + z^T Q_{xu} w^* \\
&= \bar{Q} + (Q_x^T - Q_u^T Q_{uu}^{-1} Q_{ux}) z + \frac{1}{2} z^T (Q_{xx} + \cancel{Q_{xu} Q_{uu}^{-1} Q_{ux}} - 2 Q_{xu} Q_{uu}^{-1} Q_{ux}) z + \\
&\qquad \cancel{Q_u^T Q_{uu}^{-1} Q_{ux} z} - \cancel{z^T Q_{xu} Q_{uu}^{-1} Q_u} + \dots \\
&= \Delta V + V_x z + \frac{1}{2} z^T V_{xx} z
\end{aligned}
$$

where:

- $V_x \triangleq Q_x - Q_{xu} Q_{uu}^{-1} Q_u$
- $V_{xx} \triangleq Q_{xx} - Q_{xu} Q_{uu}^{-1} Q_{ux}$
- $\Delta V = \bar{Q} - \frac{1}{2} Q_u^T Q_{uu}^{-1} Q_u$

Once again what we just observed is that the value function starting from a quadratic approximation of the value function, if we propagate it backward in time using Bellman optimality equation, we still get a quadratic approximation of the value function for time $t-1$.

By applying this principle iteratively we always maintain a quadratic form of the value function.

There is one small detail: when we are working with a non linear system (that you do not need to worry about in the LQR problem given the initial assumption). The matrix Q_{uu}, which we need to invert in order to compute the optimal control inputs, might not be invertible.

The way this is tipically handled is to introduce regularization: regularization

$$
\bar{Q}_{uu} = Q_{uu} + \mu\, I
$$

where μ is the regularization value. If this value is sufficiently large that you can be sure that the new regularized matrix will always be invertible. regularization value

You can then replace the inverse of Q_{uu} with the regularized version:

$$
w^* = -\bar{Q}_{uu}^{-1}(Q_u + Q_{ux} z) = \bar{w} + K z \qquad \bar{w} \triangleq -\bar{Q}_{uu}^{-1} Q_u \qquad K \triangleq -\bar{Q}_{uu}^{-1} Q_{ux}
$$

Substituting w^* inside Q to get the value function, you get:

$$
\begin{aligned}
V &= \bar{Q} + Q_x^T z + Q_u^T w^* + \frac{1}{2} z^T Q_{xx} z + \frac{1}{2} {w^*}^T Q_{uu} w^* + z^T Q_{xu} w^* \\
&= \bar{Q} + Q_u^T \bar{w} + (Q_x^T + Q_u^T K) z + \frac{1}{2} z^T (Q_{xx} + K^T Q_{uu} k + 2 Q_{xu} K) z + \bar{w}^T Q_{uu} K z \\
&\quad + \frac{1}{2} \bar{w}^T Q_{uu} \bar{w} + z^T Q_{xu} \bar{w} \\
&= \quad \bar{Q} + Q_u^T \bar{w} + \frac{1}{2} \bar{w}^T Q_{uu} \bar{w} && \Leftarrow \Delta V \\
&\quad + (Q_x^T + Q_u^T K + \bar{w}^T Q_{uu} K + \bar{w}^T Q_{xu}^T) z && \Leftarrow V_x \\
&\quad + \frac{1}{2} z^T (Q_{xx} + K^T Q_{uu} K + 2 Q_{xu} K) z && \Leftarrow V_{xx}
\end{aligned}
$$

Summary of the DDP algorithm:

1. Given an initial guess for the optimal control trajectory $\bar{U}$, compute the state trajectory $\bar{X}$ by matter of forward simulation in time $\dot{x} = f(x, u)$
2. Backward Pass (it is called this way because we have a loop backward in time):

   (a) Initialize the derivatives of the value function with the derivatives of the terminal cost, ie.e
   $$V_x(N) = \nabla_x l_N \qquad V_{xx}(N) = \nabla_{xx} l_N$$
   (b) Loop backward in time ($i = N-1, ..., 0$)

   - Compute the derivatives of the Q function, i.e. $Q_x, Q_u, Q_{xx}, Q_{uu}, Q_{xu}$
   - Compute the optimal control input
   $$w^* = -\bar{Q}_{uu}^{-1}(Q_u + Q_{ux} z) = \bar{w} + Kz$$
   - Compute the value function at time i, i.e. $V_x(i), V_{xx}(i)$
3. Forward Pass, since you do not the state yet, we cannot compute the control input. Basically we need to do a line search.

   (a) set $\alpha = 1$. Which translate into taking a full step.

   (b) Simulate with
   $$
   \begin{aligned}
   u &= \bar{u} + w \\
   &= \bar{u} + \alpha\, \bar{w} + K(z) \\
   &= \bar{u} + \alpha\, \bar{w} + K(x - \bar{x})
   \end{aligned}
   $$

   (c) If cost has not decreased enough, you decrease α and repeat the previous step

   (d) Assign $\bar{x} = x$ and $\bar{u} = u$

   (e) If the forward pass has not converged repeat the Backward Pass, around the new trajectory $\bar{U}$.

In the end you obtain:

1. Optimal open-loop control $\bar{u}$
2. Locally optimal linear feedback gains K

Which can be summarized in the feedback control policy (locally optimal)

$$u = \bar{u} + K(x - \bar{x})$$

How much should the cost decrease?

It depends on α, because if α is very small we cannot expect a big improvement in the cost because we are changing the control input by a small amount, whereas if α is large we can expect a larger improvement. The idea is the same that we presented with Newton Method in line search, i.e. you compute how much you expect the cost to improve if your approximation of the function that you are minimzing is exact. And this will give you a baseline to compare against how much the cost is actually decreasing.

To compute how much the cost should decrease you need to take a look at the variation of the value function starting from the value function at time 0 for $z = 0$:

$$\begin{aligned} V_0(0) = \Delta V_0 &= \alpha \sum_{i=0}^{N-1} \bar{w}_i^T Q_{u,i} + \frac{\alpha^2}{2} \sum_{i=0}^{N-1} \bar{w}_i^T Q_{uu,i} \bar{w}_i + \sum_{i=0}^{N} \bar{l}_i \\ &= \alpha \sum_{i=0}^{N-1} \bar{w}_i^T Q_{u,i} + \frac{\alpha^2}{2} \sum_{i=0}^{N-1} \bar{w}_i^T Q_{uu,i} \bar{w}_i + J(\bar{U}) \\ \Delta V_i &= \alpha \bar{w}_i^T Q_{u,i} + \frac{\alpha^2}{2} \bar{w}_i^T Q_{uu,i} \bar{w}_i + \Delta V_{i+1} + \bar{l}_i \end{aligned}$$

Here you can see that α appears both linearly and quadratically.

Hence the expected cost improvement takes the form:

$$\Delta J(\alpha) = \Delta V_0 - J(\bar{U}) = \alpha d_1 + \frac{\alpha^2}{2} d_2$$

That means that every time I change α in the forward pass I do not need to recompute d_1 and d_2, but I only need to recompute $\Delta J(\alpha)$.

What is tipically done is:

$$\frac{J(U) - J(\bar{U})}{\Delta J(\alpha)} > c_1$$

You compute the real improvement (numerator) and the expected improvement (denominator) and finally accept this step if the ratio between the two is greater than a certain value (smaller than 1).

The parameter c_1 is an hyperparameter of the problem that you have to tune by hand.

hyperparameter

Regularization

How do we tune the regularization value μ.

In fact if this parameter is too small it will happen that:

- $\bar{Q}_{uu}$ may be singular
- $\bar{w}$ may be too large, which results in very large control inputs. In this situation the local quadratic model may no longer be accurate with resulting small cost improvements

On the other hand if this parameter is too large then $\bar{w}$ may be too small and the convergence will be too slow.

Therefore the Regularization Scheme that we are going to follow is:

Regularization Scheme

- If $\bar{Q}_{uu}$ is not invertible or $\dfrac{J(\bar{U}) - J(U)}{\Delta J(\alpha)} < c_2$, then we will increse the regularization factor
- otherwise we will decrease the regularization factor, in order to speed up the convergence.

In Summary:

- DP gives us an efficient algorithm to solve LQR
- The solution of LQR yields a globally optimal feedback policy
- Extension to non linear systems (DDP)
  - efficient recursive algorithm
  - no guarantees of convergence

globally optimal feedback policy

- can only find local optimum
- feedforward + feedback policy
- many hyper-parameters for regularization/line search c_1, c_2, μ, k_μ, K

2.5 Model Predictive Control (MPC)

Model Predictive Control (MPC) is a very popular technique used in many robotic systems and other fields outside robotics. Model Predictive Control (MPC)

Moreover, this particular control method connects what we have seen in reactive control and optimal control, because what we have seen so far in optimal control is just a mean of computing a trajectory and then using reactive control we can track the reference trajectory.

With MPC we are unifying these 2 problems and solutions, because the idea of MPC is to use optimal control not only for computing the reference trajectory, but also to design the real time control loop (aka stabilizing the system).

We took a glimps of this type of approach in DDP, because DDP does not only yield the control trajectory but also the feedback gains: this allows us to the result of DDP as a feedback controller. However this is really specific to DDP since it is the only method that gives us the feedback gains (i.e. something to stabilize the systems).

All the other methods, such as Direct methods, only yield a control trajectory not the feedback gains, so in order for them to work they need to be coupled with a reactive controller.

Why do we need something else to do the stabilization? Mainly because we will always have uncertainty in the system, so even if you plan for an optimal trajectory, you cannot hope that you apply this control and the system would behave exactly as planned, you always need a feedback to stabilize the system.

In MPC we are trying to get rid completely of the reactive controller and use the optimal controller also to stabilize the system. The principle behind this method goes as follow:

Solve a finite-horizon OCP using the current state (that you measure or estimate) as initial state. finite-horizon

Under this condition the optimal control problem takes the form:

$$X^*, U^* = \underset{X,U}{\operatorname{argmin}} \sum_{k=0}^{N-1} l(x_k, u_k)$$

subject to

- $x_{k+1} = f(x_k, u_k, k) \qquad k = 0, ..., N-1$
- $x_{k+1} \in \mathcal{X}, \quad u_k \in \mathcal{U} \qquad k = 0, ..., N-1$
- $x_0 = x^{meas}$

The process to follow is:

1. Solve the optimal control problem
2. Obtain the optimal state trajectory and optimal control trajectory
3. Take the first value of the optimal control trajectory u_0, and apply it
4. In the next iteration of the control loop, repeat the same process we the value of the optimal control input obtained in the previous step.

   Of course you consider the initial state as the state measured or estimated in the current iteration.

By solving the optimal control problem at each iteration you are always behaving in a optimal way because you are re-optimizing the trajectory based on the real state of the system not based on where you thought your system is according to your model.

The main problem of MPC comes from the difference between the trajectory that you predict whe you solve the OCP and the trajectory that you actually going to follow with your real system (this point does not take into account the presence of disturbances). Even if you do not have any disturbance, the trajecory that you are going to predict with the optimal control problem and the one that you are going to get with the MPC are going to be different.

The reason is because we are re-optimizing the trajectory at each step with an unseen point w.r.t. the previous optimization.

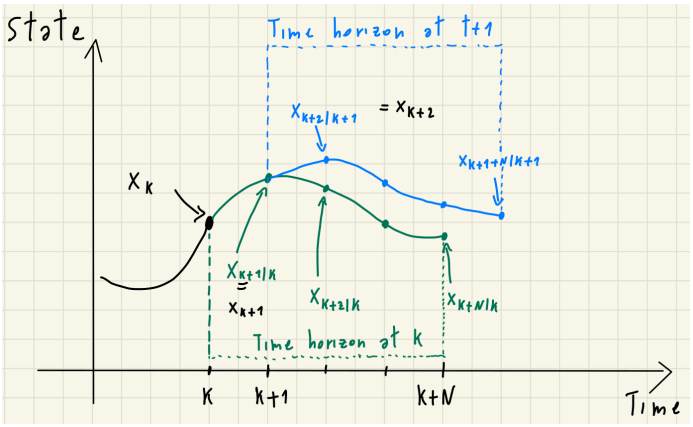


Figure 2.4: First two iteration of a generic MPC: the real trajectory differs from the computed ones due to the fact that the algorithm re-optimize the trajectory on a different set of states

In other terms

- the OCP at time k and $k+1$ optimize over different horizons so it can result in different trajectories.
- Even without disturbances, predicted and actual trajectories can be different
- What if at time $k+1$ I shorten the horizon to $N-1$?
  In this case the trajectory would be the same due to Bellman optimality principle.

The three main challenges when dealing with MPC are:

1. feasibility. feasibility

   Can I ensure that the OCP is always feasible?

   What can I do if it is not feasible?

   This is a problem given that if the problem is not feasible than the solver will not be able to yield any solution and so you lack a control input to apply to the system.

2. stability. stability

   Can I ensure that MPC stabilizes the system?

   Because even if you are computing the trajectory that all converge to zero, maybe the real trajectory that system is performing is diverging.

3. computation time computation time

   Can I solve the OCP sufficiently fast?

2.5.1 Infinite-Horizon MPC

What would happen if instead of having a finite horizon MPC, we had an horizon that is infinite?

If the horizon is infinite than the horizon that is seen by each OCP that you solve is the same since they all go to infinity. So you no longer have the problem that we have discussed before (i.e. the trajectory that you get when optimizing is different from the trajectory that you actually get on the real system), because at each iteration the OCP uses the same horizon and the trajectory will be the same at each optimization.

In short: in this case predicted and actual trajectories are the same, assuming no distrurbances. This follows from Bellman's principle of optimality.

Having $N = \infty$, if cost $l(x, u) \geq \alpha \|x\| \ \forall x, u$ for some $\alpha > 0$ then

1. Having a finite cost implies stability. From the condition of the cost we can infer that the the cost can only be finite only if the state (or its norm) goes to zero.

2. Since predicted and actual trajectories are equal, I have recursive feasibility along the closed-loop trajectory.

recursive feasibility

Even though this scenario is purely theoretical (in practice we cannot impose an infinite horizon), it helps us get the key idea of stability and recursive feasibility of MPC which is: try to mimic an infinite horizon problem by including a terminal cost and terminal constraint that mimic the effect that I would get with the infinite horizon (much like what we did with DP with the value function).

terminal cost

terminal constraint

In other terms the terminal cost should be an approximation of the tail of my infinite horizon running cost and the terminal constraints should be an approximation of all the extra constraints I would get with an infinte horizon.

infinite horizon running cost

If we take the standard finite horizon OCP:

$$V_N^*(\bar{x}) = \underset{U}{\text{minimize}} \sum_{k=0}^{N-1} l(x_k, u_k)$$

subject to:

- $x_{k+1} = f(x_k, u_k) \qquad k = 0, ..., N-1$
- $x_k \in \mathcal{X}, \quad u_k \in \mathcal{U} \qquad k = 0, ..., N-1$
- $x_0 = \bar{x}$

In order to ensure fisibility and stability we can extend the formulation using a terminal cost and terminal constraints:

$$V_N^*(\bar{x}) = \underset{U}{\text{minimize}} \sum_{k=0}^{N-1} l(x_k, u_k) + l_f(x_N)$$

subject to:

- $x_{k+1} = f(x_k, u_k) \qquad k = 0, ..., N-1$
- $x_k \in \mathcal{X}, \quad u_k \in \mathcal{U} \qquad k = 0, ..., N-1$
- $x_0 = \bar{x}$
- $x_N \in \mathcal{X}_{\{}$

The question remains: how can we choose $l_f(\cdot)$ and $\mathcal{X}$?

2.5.2 Feasibility

When talking about feasibility we need to distinguish between two cases, because they are very different from each other:

1. Input constraints only

   Input constraints only

   This case is always feasible, because u is our decision variable so if your control set is not empty, you can always choose a point inside this set and always obtain a solution.

2. Hard state constraints

   Hard state constraints

   If $N < \infty$ there is no guarantee that OCP remains feasible, even in nominal case.

   $N = \infty$ ensures feasibility, but OCP has infinitely many constraints.

The Maximum Output Admissible Set theory states that $N < \infty$ is enough to enforce recursive feasibility, but it does not specify how long the horizon should be in order that this is guaranteed.

Maximum Output Admissible Set theory

A better way to deal with this problem is to use the Maximum Control Invariant Set which is a set that can be used as a terminal constraint and this ensures closed-loop convergence, but it can reduce the domain of feasibility, i.e. starting from a state for your original problem would be feasible, by using the Maximum Control Invariant Set as a constraint, your problem might become not feasible (especially if the system is perturbed).

Maximum Control Invariant Set

closed-loop convergence

domain of feasibility

In practice, this is almost the best thing that you could do (if you can do it).

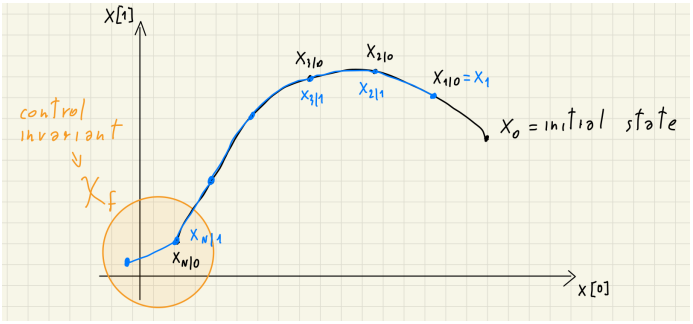
Control Invariant Set

A set S is Control Invariant if once you are inside the set you can stay inside the set. Control Invariant

In mathematical term: A set S is control invariant if $\forall x \in S, \exists u \in \mathcal{U}$ s.t. the next state $f(x,u) \in S$.

Theorem 1. *If $\mathcal{X}_f$ is control invariant then MPC will be recursively feasible.*

Visual proof:



Example: For a manipulator, set of zero velocity is control invariant (with zero velocity the manipulator remains in the same position with zero velocity i.e. you remain in the same state) if

$$|g(q)| \leq \tau^{max} \qquad \forall q \in [q^{min}, q^{max}]$$
$$S = \{(q,0) | q^{min} \leq q \leq q^{max}, q \in \mathbb{R}^n\}$$

Which means that if the motor are sufficiently strong to compensate for gravity, then the set of possible states where the velocity is zero S is control invariant.

How to compute the control invariant sets in general?

- Hard problem for nonlinear systems (e.g. robot manipulator)
- "Maximal Output Admissible Set" theory for linear systems. The same theory gives you some numerical algorithm to compute control invariant sets, but they only apply to linear systems. Let us suppose that we have a system with linear dynamics:

  $$x^+ = Ax \qquad y = Cx$$

  And we have some constraints on the output (which are allowed to be nonlinear, and they can include state constraints and input constraints)

  $$y \in \mathcal{Y} = \{y \in \mathbb{R}^p : h_i(y) \leq 0, i = 1, ..., s\}$$

  The objective of the algorithm is to find a specific control invariant set, that is the maximum output admissible set:

  $$O_\infty = \{x \in \mathbb{R}^n : h_i(CA^t x) \leq 0, i = 1, ..., s, t = 0, ..., \infty\}$$

  which is the set of all initial states x, s.t. if you start in x the constraints on the output are always satisfied.

Maximal Output Admissible Sets

Theorem 2. *If A is Lyapunov stable, h_i are continuous and they contain the origin (i.e. $h_i(0) \leq 0$) then:* origin

$$O_\infty = O_{t^*} = \{x \in \mathbb{R}^n : h_i(CA^t x) \leq 0, i = 1, ..., s, t \in \{0, ..., t^*\}\}$$

With the only difference that we can compute the set with a finite number of computation instead of infinity.

One possible way to compute t^*, even if it is not the best way, is to start with $t = 0$ and compute O_t until $O_{t+1} = O_t$.

This is theoretically fine but actually comparing O_{t+1} to O_t could be quite hard, so there is another way to do it:

1. Sove this problem for $i = 1, ..., s$:

$$\max_x J_i(x) = h_i(CA^{t+1}x)$$
$$\text{s.t. } h_j(CA^k x) \leq 0 \qquad j = 1, ..., s, k = 0, ..., t$$

2. If $J_i^* < 0 \quad \forall i = 1, ..., s$ then assign $t^* = t$.

   Otherwise increment t and repeat.

What you are trying to do is similar to what we were trying to do with the first method, but the computation that you need to at each iteration is just s-optimization problems, where s is the number of inequalities that define your constraints set, and what you are trying to do is to see if it is possible to violate one of the constraints at the time step $t + 1$ assuming that you satisfied the constraints at all the previous time steps.

If it is possible to do so, then you need to increase t and keep going, whenever it is not possible to do so, it means that all constraints are satisfied and at that point you obtain t^* and the algorithm stops.

What about control inputs?

Fix $u = Kx$, which implies $x^+ = (A + BK)x$. Ultimately obtaining :

$$y = \begin{bmatrix} I \\ K \end{bmatrix} x \qquad \mathcal{Y} = \mathcal{X} \times \mathcal{U}$$

Theorem 3. *If $\mathcal{X}_f$ is control invariant then the MPC is persistently feasible*

persistently feasible

Proof plan:

1. Given X_f compute backward reachable sets X_i, i.e. set from which X_f can be reached
2. Show that if X_f is control invariant, then X_i is control invariant $\forall i < N$
3. Show that if X_1 is control invariant, then the MPC is recursively feasible

Proof. 1. Given $x_n \in \mathcal{X}_f = \mathcal{X}_n$ define backward reachable sets recursively:

backward reachable sets

$$\mathcal{X}_i = \{x \in \mathcal{X} | \exists u \in \mathcal{U}, \text{s.t. } f(x, u) \in \mathcal{X}_{i+1}\}$$

By definition, if $x_N \in \mathcal{X}_N$ then $x_i \in \mathcal{X}_i \qquad \forall i \in [0, N-1]$

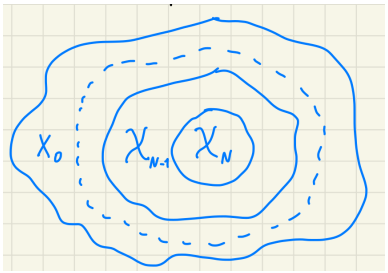
2. If $X_f = \mathcal{X}_N$ is control invariant, then $\mathcal{X}_i$ is control invariant $\forall i$.

   Assume $\mathcal{X}_{i+1}$ is control invariant. This is true if and only if $\forall x \in \mathcal{X}_{i+1}$, which implies that $\exists u$ s.t. $f(x, u) \in \mathcal{X}_{i+1}$.

   So from $\mathcal{X}_{i+1}$ you can stay in $\mathcal{X}_{i+1}$.
   We know that $\mathcal{X}_i$ contains all states from which you can reach $\mathcal{X}_{i+1}$ so we infer:

$$\mathcal{X}_{\rangle + \infty} \subseteq \mathcal{X}_i$$

   Since from $\mathcal{X}_i$ you can reach $\mathcal{X}_{i+1}$, which is contained in $\mathcal{X}_i$, we infer that from $\mathcal{X}_i$ you can stay in $\mathcal{X}_i$ which implies that $\mathcal{X}_i$ is control invariant.



   By induction, starting from $\mathcal{X}_N$ we can show that $\mathcal{X}_i$ is control invariant $\forall i < N$.

3. We can show that

$$\mathcal{X}_i \subseteq \mathcal{X}_0$$

   Since the next state $x_1 \in \mathcal{X}_1$ then it is true that $x_1 \in \mathcal{X}_0$. However this is true if and only if the next MPC problem is feasible because $\mathcal{X}_0$ is the set of all states from which MPC is feasible (by definition of $\mathcal{X}_0$)

□

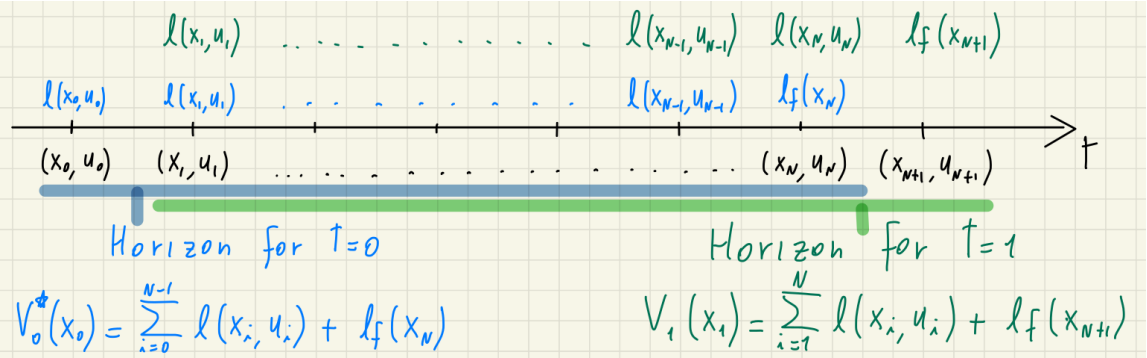
2.5.3 Stability

We can utilize Lyapunov Stability theorems to verify stability conditions of the MPC. Under this conditions, the key idea is to use the Value function (aka Optimal cost-to-go) as the exponential Lyapunov function.

Value function

exponential Lyapunov function

The reason why it makes sense to use the Value function as a Lyapunov function is displayed in the following figure: Along the time horizon over which we are optimizing and associated to each



time step we have a cost which depends on the state and control at that time step.

So when we are at $t = 0$ the horizon that we see, goes until time step N, whereas at $t = 1$ the horizon that we see goes until time step $N + 1$.

Imagine the cost that we will get in each of these two control problems that we solve:

- When we solve the first optimal control problem the cost will be:

$$V_0^*(x_0) = \sum_{i=0}^{N-1} l(x_i, u_i) + l_f(x_N)$$

  with a summation of the running costs and the terminal cost
- The same applies to the second optimal control problem starting at $t = 1$:

$$V_1(x_1) = \sum_{i=1}^{N} l(x_i, u_i) + l_f(x_{N+1})$$

  With the only difference being on the limit of the summation and the argument of the terminal cost.

We can notice that the intermediate running costs in the range $t \in [1, N-1]$ are equal: the only difference is in the terms:

$$l(x_0, u_0) \qquad l_f(x_N) \qquad l(x_N, u_N) \qquad l_f(x_{N+1})$$

Hence, we can write down the difference between the cost V_1 and the cost V_0^* (the first two conditions of the Lyapunov function are easy to fullfill):

$$\begin{aligned} V_1(x_1) - V_0^*(x_0) &= \sum_{i=1}^{N} l(x_i, u_i) + l_f(x_{N+1}) - \sum_{i=0}^{N-1} l(x_i, u_i) - l_f(x_N) \\ &= \cancel{\sum_{i=1}^{N-1} l(x_i, u_i)} + l(x_N, u_N) + l_f(x_{N+1}) - l(x_0, u_0) - \cancel{\sum_{i=1}^{N-1} l(x_i, u_i)} - l_f(x_N) \\ &= l(x_N, u_N) + l_f(x_{N+1}) - l_f(x_N) - l(x_0, u_0) \end{aligned}$$

if i am able to prove that:

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

then the value function is an exponential Lyapunov function and I can use it to prove the stability of the origin for my system when it is controlled by the MPC.

One thing to notice is that V_0^* is the optimal value function (i.e. the states and controls are optimal) , whereas forV_1 we assumed that we have used the values of the optimal state and control until x_N and u_N computed at the previous time step.

optimal value function

However, they may not be the optimal ones given that we are optimizing on a different horizon.

So by considering a non optimal value function we are taking a conservative approach: we are assuming that the trajectory will stay the same until the second to last time step.

This means that the condition that I want is a sufficient condition, but it is not necessary. So if it is satisfied I am sure that the system controlled by the MPC is stable.

sufficient condition

Stability Assumption

Ass. #1: The origin is an equilibrium point, the runnning cost at the origin is 0, the terminal cost at the origin is 0.

$$f(0,0) = 0 \qquad l(0,0) = 0 \qquad l_f(0) = 0$$

Note that assuming $f(0,0)$ is not restictive.

Proof. If we want to stabilize $x^* \neq 0$ s.t. $f(x^*, u^*) = x^*$, i.e. x^* must be an equilibrium.

To achieve so we can define a new state $\bar{x}$, a new control $\bar{u}$ and new dynamics as follows:

$$\begin{aligned} \bar{x} &= x - x^* \\ \bar{u} &= u - u^* \end{aligned}$$

$$\begin{cases} x^+ = f(x,u) \\ x^* = f(x^*, u^*) \end{cases} \quad \Rightarrow \quad x^+ - x^* = f(\bar{x} + x^*, \bar{u} + u^*) - x^*$$

$$\bar{x}^+ = \bar{f}(\bar{x}, \bar{u})$$

So that $\bar{f}(0,0) = f(0 + x^*, 0 + u^*) - x^* = 0$ □

Ass. #2: The union of the state space and control space is a closed set (set that contains the boundaries):

closed set

$$\mathcal{X} \times \mathcal{U} \text{ is closed}$$

The terminal set is a compact set (≈ closed + bounded set)

compact set

$$\mathcal{X}_f \text{ is compact}$$

Ass. #3: For every state in the terminal constraint set there must exists a value of the control a collection of conditions must be respected.

Formally:

$$\forall x \in \mathcal{X}_f \ \exists u \in \mathcal{U} \text{ s.t.}$$

(a) $\mathcal{X}_f$ is control invariant

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

and the terminal cost decreases

$$l_f(f(x,u)) - l_f(x) \leq -l(x,u)$$

(b) There must exists 2 positive scalar, such that the running cost is lower bounded and the terminal cost is upper bounded, i.e. $\exists \alpha_1, \alpha_f > 0$ s.t.

$$\begin{aligned} l(x,u) &\geq \alpha_1 \|x\| \qquad && \forall x \in \mathcal{X}_N, \quad \forall u \in \mathcal{U} \\ l_f(x) &\leq \alpha_f \|x\| && \forall x \in \mathcal{X} \end{aligned}$$

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

If these three assumptions are satisfied then $V_N^*(\cdot)$ is an exponetial Lyapunov function and therefore the origin is exponentially stable.

Proof.

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

where:

- $l(x_N, u_N) + l_f(x_{N+1}) - l_f(x_N) \leq 0$
  by Ass #3(a)
- $-l(x_0, u_0) \leq -\alpha_1\|x_0\|$
  by Ass #3(b)

So we can be sure that the summation of these two will always be less that some negative function of $\|x_0\|$ □

Example: Linear System

Let us consider a linear system with dynamics:

$$x^+ = Ax + Bu$$

with some control constraints and state constraints:

$$u \in \mathcal{U} \qquad x \in \mathcal{X}$$

Whenever we have control and state constraints it is hard to find a global Control Lyapunov Function (CLF). For this reason we aim to find a local CLF inside a given region, called invariant region (given that it has the control invariant property).

Control Lyapunov Function (CLF)

invariant region

Let us define the running cost the following quadratic function:

$$l(x, u) = \frac{1}{2}(x^T Q x + u^T R u) \qquad \text{with } Q > 0\,,\ R > 0$$

and then let us assume that the couple (A, B) is stabilizable (i.e. it can be stabilized). Under these conditions the Value function of unconstrained infinite horizon problem $\mathbb{P}_\infty^{un}$ is known and defined as:

stabilizable

$$V_\infty^{un}(x) = \frac{1}{2}x^T P x$$

where

$$P = A_k^T P A_k + Q_k$$

with

$$A_k \triangleq A + BK \qquad Q_k = Q + K^T R K \qquad u = Kx \qquad K = -(B^T P B + R)^{-1} B^T P A^T$$

In practice you can compute P with an infinite summation of matrices

$$P = \sum_{i=0}^{\infty} (A_k^T)^i Q_k A_k^i$$

Of course this is not really practical (because you are doing an infinite summation), but since you are taking the power of a matrix that is stable (i.e. A), this matrix is going to zero as you keep summing. That means that at a certain point you can sto the summation and the error will be substantially small.

Knowing that this is the solution to the problem without the constraints what is very often done is, we try to utilize it as a terminal cost in the problem with the constraint:

$$l_f(x) = V_\infty^{un}(x)$$

This makes sense given that we are looking for a local Control Lyapunov Function that is valid only inside a certain region, we can hope that if we are sufficiently close to the origin then the constraint won't play any role (in other words we expect that if we are sufficiently close to the origin the optimal choice won't be affected by the constraints) and that is what happens in practical application.

However, we still need to prove that using this particular choice of final cost, the stability assumption that we formulate for the Lyapunov Function are satisfied.

- **Terminal cost decrease**:

$$l_f(A_k x) + \frac{1}{2}x^T Q_k x - l_f(x) \leq 0 \qquad \forall x \in \mathbb{R}^n$$

where:

$$\begin{aligned} l_f(x^+) &= l_f(Ax + Bu) && u = Kx \\ &= l_f(Ax + BKx) \\ &= l_f((A + BK)x) \\ &= l_f(A_k x) \end{aligned}$$

$$\begin{aligned} l(x,u) &= \frac{1}{2}(x^T Q x + u^T R u) && u = Kx \\ &= \frac{1}{2}x^T(Q + K^T R K)x \\ &= \frac{1}{2}x^T Q_k x \end{aligned}$$

To verify that the assumption is verified, we substitute the uncontraint value function as the final cost, i.e.

$$\begin{aligned} l_f(A_k x) + \frac{1}{2}x^T Q_k x - l_f(x) &\leq 0 \\ x^T A_k^T P A_k x + x^T Q_k x - x^T P x &\leq 0 \end{aligned}$$

We obtained a summation of three pluriquadratic functions of the state, so this summation will always be negative if the matrix defining the quadratic form is negative semidefinitive

$$A_k^T P A_k + Q_k - P \leq 0$$

which reminds us of the implicit definition of P, which is equal to zero.
We obtained that the terminal cost defined as the unconstrained value function can decrease and we also know that it decreases for the specific choice of the control input:

$$u = Kx$$

Note: K could takes an arbitrary value, as long as the assumption are satisfied

- **Terminal constraints**
  Choose $\mathcal{X}_f$ so that $u = Kx \in \mathcal{U} \forall x \in \mathcal{X}_f$ and $u = Kx$ makes $\mathcal{X}_f$ will be positive invariant.
  We can choose $\mathcal{X}_f$ to be maximal invariant constrait admissible set for $x^+ = (A + BK)x$.

In summary:

1. We can pick $\mathcal{X}_f$ and $l_f(\cdot)$ such that:
   - $\mathcal{X}_f$ is control invariant
   - $\forall x \in \mathcal{X}_f \, \exists u \in \mathcal{U}$ s.t. $l_f(f(x,u)) - l_f(x) \leq -l(x,u)$
2. We can pick $\mathcal{X}_f = \{0\}$, i.e. picking the origin are your terminal constraint set.

Both conditions of the previous point are satisfied: the first one by definition given that the origin is control invariant, the second one is satisfied given that the terminal cost and the running cost at the origin are zero.

However this method results in a small basin of attraction, because you are constraining the terminal state to be in a set with just one point.

small basin of attraction

3. We can pick N "large enough" and set $l_f(x) = 0, \mathcal{X}_f = \mathcal{X}$. With nonlinear systems such as robots, that is what many people do (sadly).

Stability-Extensions

1. **What if the system is time varying?**

   So far we talked about stability of a point, i.e. the origin.

   That is a very specific task for a robot to achieve, quite useless actually, because it means that you want the robot manipulator to stay still.

   Tipically you want to do trajectory tracking and with the theory that we have looked at so far, we cannot really ensure stability for trajectory tracking because the running cost is not always the same but it depends on time.

   In this case we can cast the trajectory tracking as regulation of time-varying system.

   So the good news is that the theory can be extended to a time varying system:

   Stability and recursive feasibility can still be ensured with time-varying $\mathcal{X}_f$ and $l_f(\cdot)$.

2. **What if the cost function measures some sort of energy consumption?**

   Another situation arises when the cost function measures some sort of energy consumption (quite common). In that case one of the assumptions that we took is not satisfied, i.e.

   $$l(x, u) \geq \alpha \|x\|$$

   Because the running cost is not lower bounded anymore by the norm of the state, because the energy consumption does not depend on the norm of the state but tipically depends on both state and control.

   In this situation, people rely on Economic MPC theory to prove and ensure stability of the system.

   Economic MPC theory

2.5.4 Computation time

In robotics, a lot of attention has been given to this issue: how do you make compilation faster?

One of the key idea that you can use to speed up computation is warm start, that goes as follow:

warm start

When you are solving the MPC optimal control problem for a given horizon from 0 to N, and then at the next iteration you would have to solve another optimal control problem that looks quite a lot like the one we just solved (but we are shifting the horizon one step forward).

As a consequence, it seems really reasonable that the optimal solution won't change very much, because the problem and the solution are very similar. So what we can do is:

Take the optimal solution that you compute it at time 0 and use it as an initial guess for the problem that you solve at the next time step. Since most of the time the optimal solution will be really similar then you have a good initial guess and the solver needs to take just a few iteration to converge to the optimal solution.

Formally:

- Shift trajectory back by 1 time step:

$$u_k^{guess} = u_{k+1}^*$$

- Use zero as initial guess for last time step:

$$u_{N-1}^{guess} = 0$$

Another simple trick is that instead of iterating until convergence to a very small convergence threshold, do not iterate until convergence: at each iteration of the control loop just do one optimization iteration.

convergence threshold

In this way you keep computation time to a minimum and you are show that you are always doing computation on the most recent information on the state of your robot (just do 1 Newton iteration).

There are many other ideas used to improve computation time, but these are the most general ones.

2.5.5 Uncertainties in MPC

Until now we did not really took into account uncertainty. We assumed that when we are at state x we apply control u and the next state will be $f(x, u)$.

uncertainty

In practice, what happens is that when we are at state x, we apply control u and the new state will be $f(x, u) + noise$, because we do not have perfect knowledge of the dynamics, the actuators cannot really deliver the control input as you asked and you do not really know that your current state is x because you have some uncertainty in your estimation due to sensor noise.

sensor noise

There are two main approaches to deal with uncertainties.

Robust MPC

In the Robust MPC approach, basically you consider the next state x^+ as a function of the state, the control input and the noise (with the noise bounded in a certain set), i.e.

Robust MPC approach

$$x^+ = f(x, u, w) \qquad w \in \mathcal{W}$$

As a consequence the generic constraint of the problem needs to be satisfied for every uncertainty that my happens

$$g(x, u, w) \leq 0 \qquad \forall w \in W$$

In this way, i will always maintain a safety margin w.r.t. the constraint boundary and this safety margin is what allows us to ensure that the constraints will be satisfied for any possible uncertainty realization.

Stochastic MPC

Another approach is to define the uncertainty not simply bounded but it is a stocastic uncertainty, very often model with a Gaussian distribution with zero mean:

stocastic uncertainty

Gaussian distribution

$$w\ \mathcal{N}(0, \Sigma)$$

In this case, since the uncertainty is Gaussian you cannot hope that the constraints will be satisfied for every possible value, given that it has a small probability of being infinitely large. So tipically, you impose that the probability of the constraint to be satisfied is greater than some value:

$$Pr[g(x, u, w) \leq 0] \geq 0.95$$

In general the Robust approach is easier to implement (mathematically speaking), but even though the stochastic approach is harder to implement it is less conservative.

less conservative

Both these problem are well understood with Linear dynamics, i.e. there exists methods to find the solution for both approaches. with Nonlinear dynamics, some methods exist, but it is still ongoing research (as of now the methods either requires high computation effort or are extremely conservative).

Linear dynamics

Nonlinear dynamics

Chapter 3

Reinforcement Learning

3.1 Introduction

Reinforcement Learning (RL), much like Optimal control, is a mathematical framework that allows us to solve optimal control problems.

Reinforcement Learning (RL)

In some sense is very similar to Optimal control, but with a very big difference: in Optimal control, you tipically assume to know the analytical/mathematical model of the system that you want to control, whereas in Reinforcement learning you don't.

Either you do not know the mathematical relation between input and state or you may assume that you have a simulator, but you do not have access to code.

Of course, since we have less information it is a much more difficult problem than optimal control, but they share the same structure and some of the key concepts (e.g value function).

A few differene between Reinforcement learning and Optimal control are:

- Optimal control was born in the control community, whereas reinforcement learning was born in the Computer Science community.
- RL tries to find the gloablly optimal policy
- RL assumes that the dynamics are unknown
- Initally the community of RL was focused on problems in which the state and the control spaces were finite
- RL uses different terminology than optimal control, with different names and symbols, even though they represent the same concept
- tipically RL assumes the dynamics of the system to be stochastic (randomness on the state of the system)
  However we will not take this assumption in this module (not a big deal), for sake of simplicity.
- tipically RL literature focuses on optimizing over an infinite horizon

gloablly optimal policy

3.2 Terminology comparison

Reinforcement Learning	**Optimal control**
State s	State x
Action a	Control u
Environment	Plant
Reward	Cost function
Return	Cost-to-go
Maximize	Minimize
Value function	Cost to go of a policy
Optimal Value function	Value function (Cost to go of the optimal policy)

3.3 Framework: Markov Decision Processes (MDP)

Markov Decision Processes (MDP) are used to describe the environment in Reinforcement Learning problems, which is assumed to be a fully observable environment and it has Markov property.

Markov Decision Processes (MDP)

fully observable environment

Markov property

Markov property means that if you know the current state of the system than knowing the past states does not add any information. The state tells you everything you know about the system.

Or in other terms: the future is independent on the past given the present.

$$\mathbb{P}(x_{t+1}|x_t) = \mathbb{P}(x_{t+1}|x_1, ..., x_t)$$

It can be immediately seen that the system is considered stochastic.

The motion of the state is defined by the State transition probability which the probability of going from one state x to another state x'

State transition probability

$$\mathcal{P}_{xx'} = \mathbb{P}(x_{t+1} = x'|x_t = x)$$

Since in classical Reinforcement Learning you assume that the size of the state space is finite, then you can collect together all the state transition probabilities in a big matrix, called State Transition matrix.

State Transition matrix

$$\mathcal{P} = \begin{bmatrix} \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \cdots & \mathcal{P}_{nn} \end{bmatrix}$$

The sum of each row of the matrix will be 1. In deterministic systems $\mathcal{P}$ contains only 0 and 1.

3.3.1 Markov Process or Markov Chain

A Markov Process is defined by a tuple $< \mathcal{X}, \mathcal{P} >$, where:

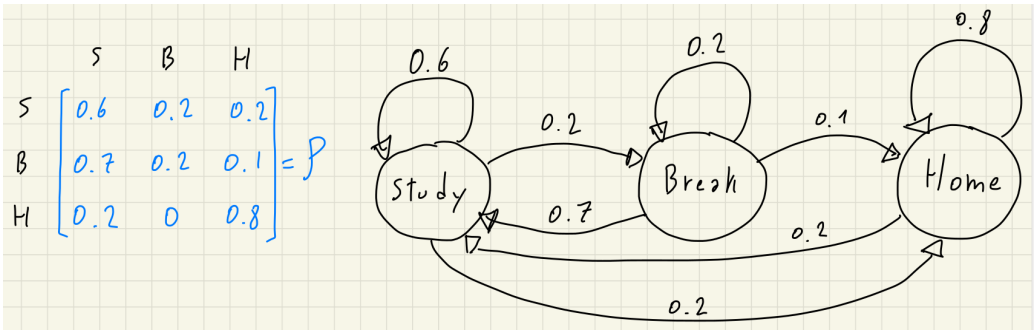
Markov Process

- $\mathcal{X}$ is a finite set of states
- $\mathcal{P}$ is the state transition probability matrix

state transition probability matrix

A Markov Process is also known as a Markov Chain

Markov Chain



In particular, since $\mathcal{P}$ has all nonegative entries, and it is also irreducible (associated to a strongly connected graph), by using Perron-Frobenius theorem we know that:

irreducible

strongly connected graph

Perron-Frobenius theorem

- The largest (in nom) eigenvalue r of $\mathcal{P}$ satisfies the following inequalities:

$$\min_i \sum_j \mathcal{P}_{ij} \leq r \leq \max_i \sum_j \mathcal{P}_{ij} \quad \Rightarrow \quad r = 1$$

  which means that the highest eigenvalue of the matrix is 1
- All eigenvalues of $\mathcal{P}$ are smaller than 1.

$$\lambda_i(\mathcal{P}) \leq 1 \qquad \forall i$$

We now have all the tools to try to connect what we have just seen for Markov processes to what we have seen so far for dynamical systems:

- In Optimal control, since we mostly dealt with deterministic systems, the dynamic was encoded as:

  $$x^+ = f(x)$$

  In which, we represented the next state as a function of the state (and additionally the control inputs).
- In Markov processes, we can represent the state evolution (aka dynamics) using probability density functions:

  $$\mathbb{P}(x^+|x)$$

deterministic systems

probability density functions

We therefore can conclude that:

If a system is deterministic, both approaches can be used (however we will try to use the first approach whenever possible).

Markov reward process

A Markov Reward Process is completely defined by a tuple $< \mathcal{X}, \mathcal{P}, C, \gamma$, where:

Markov Reward Process

- $\mathcal{X}$ is the state space
- $\mathcal{P}$ is the probability transition matrix
- C is the reward (aka cost function)

  $$C_x = \mathbb{E}[l_{t+1}|x_t = x]$$

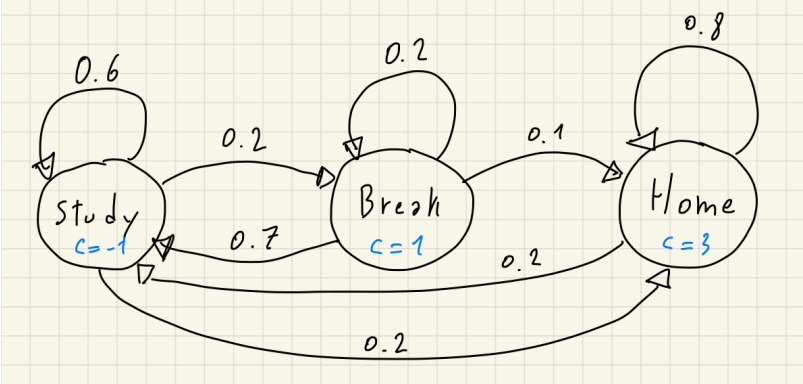
  We consider the expected value because we assume that the system is stochastic.
- γ is the discount factor

  $$\gamma \in [0, 1]$$

  which discounts the cost into the future

discount factor

If we consider the example of Markov Process described before



We can translate it into a Markov Reward Process, by simply adding a cost associated with each state. In this situation the lower the cost, the better, since the agent is rewarded for studying and punish for doing breaks and going home.

Regarding the discount factor, the parameter enters into the computation of the cost-to-go (or return), which is nothing more than the total discounted cost from time t to ∞:

$$J_t = l_t + \gamma l_{t+1} + \gamma^2 l_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k l_{k+t}$$

Following the above definition, we can infer that γ represents the preference for later costs over immediate costs (how much you value the future over the present):

later costs

immediate costs

- a value of γ close to 0, we obtain a myopic evaluation (the future has small impact on the decision)
- a value of γ close to 1, we obtain a far-sight evaluation ($\sim$ optimal control)

myopic evaluation

far-sight evaluation

But why do we need to consider the discount factor?

- In order to avoid an infinite cost, since an infinite cost cannot be minimized.
- There is only a certain degree of uncertainty about the future prediction (reason to favour immediate reward instead of delayed reward).
- If the cost is associated with some monetary budget, interest rate and inflation can be modeled using the discount factor.

monetary budget

3.3.2 Bellman equation in the context of RL

Let us define the cost starting from state x, as:

$$V(x) = J_t(x_t = x)$$

The Bellman Equation can be obtained by decomposition of the cost in two parts (assumption: no control input):

Bellman Equation

$$V(x) = l(x) + \gamma V(f(x))$$

Which can also be represented in matrix form, since RL considers a finite amount of states and control inputs and therefore the Value function and other quantities can be stored in a vector/matrix:

$$\begin{bmatrix} V(1) \\ \vdots \\ V(n) \end{bmatrix} = \begin{bmatrix} l(1) \\ \vdots \\ l(n) \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \cdots & \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} V(1) \\ \vdots \\ V(n) \end{bmatrix}$$

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

The reason why the matrix form and the analytical form of Bellman's equation are equal is because the system is assumed to be deterministic: this means that each row of the transition probability matrix $\mathcal{P}$ will be zero except for the element (which is 1) associated with the next state.

In this way the matrix product between the transition probability matrix with the Value function vector will yield the Value function at the next state, i.e.

$$V(f(x)) = \mathcal{P} V$$

However, if the system is considered stochastic the matrix formulation still holds whereas the analitical formulation does not make sense since $f(x)$ becomes a probability distribution and therefore cannot be utilized to evaluate the value function.

Moreover, the matrix form is simply a linear system of equation in V and therefore we can ensure that we can explicitly compute it:

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

However the Direct solution is only possible for small MRPs. Otherwise, in the case of a large Markov Reward Process with many possible states, then iterative methods can be utilized.

3.3.3 Markov Decision Process MDP

A Markov Decision Process MDP is uniquely defined by a tuple $< \mathcal{X}, \mathcal{U}, \mathcal{P}, C, \gamma >$ where:

Markov Decision Process MDP

- $\mathcal{X}$ is the state space
- $\mathcal{U}$ is the control space: (finite) set of control inputs
- $\mathcal{P}$ is the transition probability matrix

$$\mathcal{P}^u_{xx'} = \mathbb{P}(x_{t+1} = x' | x_T = x, u_t = u)$$

- C is the cost

$$C^u_x = l_t(x_t = x, u_t = u)$$

- γ is the discount factor

Now that we have introduced the control inputs in the formulation of the system, we can define the notion of policy: distribution of the control inputs over the states (i.e. a stochastic function). policy

Given a certain state, you obtain a distribution over all possible control inputs to apply.

$$\pi(u|x) = \mathbb{P}(u_t = u|x_t = x)$$

From now on we will assume deterministic policies,i.e.: deterministic policies

$$u = \pi(x)$$

3.3.4 Action-Value Function (Q function)

The Action-Value Function is something very similar to the Value function. The Value function is a function of the state and yields the return (cost-to-go) that you will encounter starting from that state and follow a certain policy π. Action-Value Function

Whereas the Action-Value function is an extension of the Value function that takes as input the state and the control input, and yields the cost that you will encounter starting from a state x applying the control u and, from that point onward, following the policy π.

$$Q^{\pi}(x, u) = J_t(x_t = x, u_t = u, u_{k>t} \sim \pi)$$

We can further decompose the Q-function into two separate terms:

$$\begin{aligned} Q^{\pi}(x, u) &= l(x, u) + \gamma Q^{\pi}(f(x, u), \pi(f(x, u))) \\ &= l(x, u) + \gamma V^{\pi}(f(x, u)) \end{aligned}$$

where

$$V^{\pi}(x) = Q^{pi}(x, \pi(x))$$

3.3.5 Optimal Value Function and Optimal Action-Value Function

The Optimal Value Function is the minimum Value function over all possible policies Optimal Value Function

$$V^*(x) = \min_{\pi} V^{\pi}(x)$$

SImilarly we can define the Optimal Action-Value Function: Optimal Action-Value Function

$$Q^*(x, u) = \min_{\pi} Q^{\pi}(x, u)$$

And as a result the Optimal policy is defined as: Optimal policy

$$\pi^* \leq \pi \qquad \forall \pi$$

where

$$\pi \leq \pi' \qquad \text{if } V^{\pi}(x) \leq V^{\pi'}(x) \quad \forall x \in \mathcal{X}$$

One possible way of computing the optimal policy, which is often used in reinforcement learning, is to first:

- Minimize the Action-Value Function Q^* over u:

$$\begin{aligned} \pi^*(x) &= \operatorname*{argmin}_{z \in \mathcal{U}} Q^*(x, z) \\ &= \operatorname*{argmin}_{z \in \mathcal{U}} l(x, u) + \gamma V^*(f(x, u)) \end{aligned}$$

But by assumption we do not know the dynamics $f(x, u)$, that is why we utilize the Q function.

Therefore, if we know $Q^*(x, u)$, then we immediatly have the optimal policy.

3.3.6 Bellman Optimality equation

The Bellman Optimality equation is the optimal version of the Bellman equation:

Bellman Optimality equation

$$\begin{aligned} V^*(x) &= \min_u Q^*(x,u) \\ &= \min_u l(x,u) + \gamma V^*(f(x,u)) \end{aligned}$$

which implies that:

$$\begin{aligned} Q^*(x,u) &= l(x,u) + \gamma V^*(f(x,u)) \\ &= l(x,u) + \gamma \min_{u'} Q^*(f(x,u),u') \end{aligned}$$

We obtained:

- A nonlinear systems (because a minimization is non linear)
- No closed-form solution
- Iterative algorithm are utilized to find the solution

3.4 Model Based Control

3.4.1 Dynamic Programming

Before diving deep into Reinforcement Learning algorithms we need to understand the algorithmic foundation of RL, which is Dynamic Programming.

Dynamic Programming

We have already discussed Dynamic Programming in the context of Optimal Control, but as we will see in this section RL rely much more on Dynamic Programming than Optimal control.

However, note that Dynamic Programming is not yet RL since, in DP, you assume that you have full knowledge of the model (in our case of the MDP).

In the context of Dynamic Programming there are two classes of problems that we can deal with:

1. Prediction

   Prediction

   In a prediction problem you already have a MDP and a policy (function that tells you the control input to apply in each state), and you want to evaluate how good that policy is (i.e. the value function associated with that policy).

   Formally:

   - Input: MDP $< \mathcal{X}, \mathcal{U}, \mathcal{P}, C, \gamma >$ and policy π
   - Output: Value function V^π (not the optimal one)

2. Control

   Control

   In a control problem you already have a MDP and you want to find the optimal control policy (and tipically the optimal value function associated to it).

   optimal control policy

   optimal value function

   Formally:

   - Input: MDP
   - Output: optimal value function V^* and optimal policy π^*

Difference between Finite and Infinite Horizon

What we have discussed when talking about DP is the case of Finite Horizon, so we assumed an horizon over which we want to optimize was finite. What we are gonna see instead is the application of DP with Infinite Horizon.

Finite Horizon

Infinite Horizon

The main reason of this change in approach is due to the fact that in Optimal control you usually deal with a Finite Horizon problem, whereas in RL most of the literature deals with Infinite Horizon problems; as a consequence, we need to base our algorithms on different versions of DP (we will see that this changes the way the algorithm behaves).

- Finite horizon
  - Theory is simpler (you start fro the last time step and you already now the Value function associated with it)
  - mostly used in Optimal Control
  - the policy and Value function depends on time, because depending on how much time you have left, the optimal thing to do might be different.

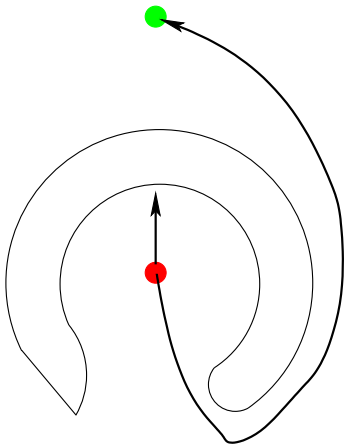


Figure 3.1: With a finite horizon (if not much time is left to reach the target) the best thing the algorithm can do is get as close as possible to the goal point, with an infinite horizon the algorithm has enough time to find the path that reaches the goal position

- Infinite horizon
  - theory is more complex, but it is elegant (even though the theory is complex, this kind of complexity in the end disappears resulting in a beautiful and simple algorithm, sometimes simpler than the version with finite horizon)
  - mostly used in RL
  - the policy and Value function are stationary (independent of time)
  - good approximation of long finite horizon problems

Bellman operators

Before diving deep into the DP algorithms with infinite horizon we need to introduce the notion that we will use in defining them:

- Bellman (expectation backup) operator T^π.

  Bellman (expectation backup) operator

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

  The input is the Value function (or vector of Value functions) and the output is another Value function.
- Bellman optimality (backup) operator T.

  Bellman optimality (backup) operator

$$T(V) = \min_{u \in \mathcal{U}} C^u + \gamma \mathcal{P}^u V$$

  In this case you do not have a policy to follow but you minimize over all possible control inputs (for each possible state):

$$(TV)(x) = \min_{u \in \mathcal{U}} l(x,u) + \gamma V(f(x,u)) \qquad \forall x \in \mathcal{X}$$

The expression are basically the Bellman optimality principle equation:

$$V_k(x) = \min_u l(x,u) + V_{k+1}(f(x,u))$$

3.4.2 Prediction

Iterative Policy Evaluation

The Iterative Policy Evaluation algorithm can be defined as follow: `Iterative Policy Evaluation`

- Problem: Given a MDP and a given policy π. Evaluate the given policy.
- Solution: Apply iteratively Bellman expectation operator:

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

This means that we compute a sequence of approximation of the Value function, where each approximation is computed from the previous approximation.
The recursive application of the Bellman operator can be represented in a shorter notation:

$$V_{k+1} = T^\pi V_k$$

In this case k refers to the iteration of the algorithm not the time step since it does not depend on time

The algorithm goes as follow:

1. Start with an arbitrary Value function V_0
2. Iterate using
$$V_{k+1} = T^\pi V_k$$
3. Guarantee to converge to V_π, since
$$\lim_{k \to \infty} V_k = V_\pi$$
In practice you will converge before infinity.

Proof. Starting from:

$$V_{k+1} = T^\pi V_k$$

1. We first need to prove that T^π is contracting, i.e. the distance (measured using the L_∞ norm) between two value functions V and Z decreases if you apply the Bellman operator. `contracting`

$$\begin{aligned} \|T^\pi V - T^\pi Z\|_\infty &= \|\cancel{\mathcal{C}^\pi} + \gamma \mathcal{P}^\pi V - \cancel{\mathcal{C}^\pi} - \gamma \mathcal{P}^\pi Z\|_\infty \\ &= \gamma \|\mathcal{P}^\pi (V - Z)\|_\infty \\ &\leq \gamma \|V - Z\|_\infty \end{aligned}$$

Because each row of $\mathcal{P}^\pi$ sums to 1.

In the specific case of a deterministic problem, by applying the transition probability matrix only changes the order of the difference of Value functions not the norm.

2. Secondly we would like to prove that given an arbitrary Value functin V_0 and the value associated with the policy V_π, $V_k \to V_\pi$ as $k \to \infty$

$$\begin{aligned} \|V_k - V_\pi\|_\infty &= \|T^\pi V_{k-1} - T^\pi V_\pi\|_\infty \\ &\leq \gamma \|V_{k-1} - V_\pi\|_\infty \\ &\leq \gamma^2 \|V_{k-2} - V_\pi\|_\infty \\ &\leq \dots \\ &\leq \gamma^k \|V_0 - V_\pi\|_\infty \end{aligned}$$

Which states that V_k converges to V_π at geometric rate, and the geometric rate is defined by the discount factor $\gamma \in [0, 1]$ `geometric rate`

Note that V_π is invariant, therefore

$$V_\pi = T^\pi V_\pi$$

3. Prove V_π is unique fixed point of T^π.

   Assume V and Z are both fixed points of T^π, i.e.

$$T^\pi V = V \qquad T^\pi Z = Z$$

   By doing so we are going to prove that the only way that could be two of them is if they are equal $V = Z$

$$\begin{aligned} \|T^\pi V - T^\pi Z\|_\infty &\leq \gamma \|V - Z\|_\infty && \text{Because } T^\pi \text{ is contracting} \\ \|T^\pi V - T^\pi Z\|_\infty &= \|V - Z\|_\infty && \text{Because } V, Z \text{ are fixed points} \\ \gamma \|V - Z\|_\infty \geq \|V - Z\|_\infty \quad &\Rightarrow \quad \|V - Z\|_\infty && \text{Because } \gamma < 1 \end{aligned}$$

   Therefore the only way the inequality holds is if and only if $V = Z$, which means that the only way these two points are both fixed is if they are the same point.

□

3.4.3 Control

Policy Iteration

The Policy Iteration algorithm can be defined as follow: `Policy Iteration`

- Problem: Given an MDP, find the optimal policy π^*

The algorithm goes as follow:

1. Start with an arbitrary policy π_0
2. Iterate for $k = 0...\infty$

   - Evaluate the policy $\pi_k \Rightarrow V^{\pi_k}$ (bottleneck)
   - Improve the policy acting greedily w.r.t. V^{π_k}:

$$\pi_{k+1}(x) = \underset{u}{\operatorname{argmin}}\, l(x,u) + \gamma V^{\pi_k}(f(x,u)) \qquad \forall x \in \mathcal{X}$$

     The minimization is computed by evaluating the argument for all possible u, you store the results in a vector and then you compute the minimum of the vector (minimization by enumeration). `minimization by enumeration`

This algorithm always converges to the optimal policy π^*

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

We can show that this convergence works:

Proof. The way you show that policy iteration converges to the optimal policy is by showing that at each iteration of the algorithm, you obtain a policy that is better than the previous one, i.e,:

$$\pi_{k+1} \leq \pi_k \quad \forall k \qquad \Longleftrightarrow \qquad \pi' \leq \pi$$

Here the inequality applied to the policies means that the value function associated to a policy is always less than or equal to the value function associated to another policy.

$$\begin{aligned} \pi'(x) = &\underset{u}{\operatorname{argmin}}[l(x,u) + \gamma V^\pi(f(x,u))] \\ &\min_u [l(x,u) + \gamma V^\pi(f(x,u))] \leq l(x,\pi(x)) + \gamma V^\pi(f(x,\pi(x))) \end{aligned}$$

Where :

- the lhs of the inequality is the cost we get by following π' for first step and then following π (because we are using V^π to estimate the tail of the cost)
- the rhs of the inequality is the cost following π

Then I need to find a way to iterate in order to show that following π' is better than to follow π, and the way we are going to do it is by using the Q function:

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

$$\begin{aligned}
V^\pi(x) &\geq \min_u [l(x,u) + \gamma V^\pi(f(x,u))] && V^\pi(x) \geq Q^\pi(x, \pi') \\
&\geq l(x, \pi') + \gamma Q^\pi(x', \pi') && x' \triangleq f(x, \pi') \\
&= l(x, \pi') + \gamma l(x', \pi') + \gamma^2 V^\pi(x'') && \text{Cost following } \pi' \text{for 2 steps and then } \pi \\
&\geq l(x, \pi') + \gamma l(x', \pi') + \gamma^2 Q^\pi(x'', \pi') && \pi' \text{ for 3 steps} \\
&\geq \sum_{i=0}^{\infty} \gamma^i l(x^{(i)}, \pi') = V^{\pi'}(x)
\end{aligned}$$

$$V^\pi \geq V^{\pi'} \iff \pi' \leq \pi$$

□

Modified policy iteration

Since most of the computation time of the policy iteration algorithm is spent evaluating the policy, we can introduce a variation of the algorithm called Modified policy iteration algorithm, based on the idea that maybe we do not need to evaluate the policy exactly, but instead compute an approximation of its value function.

Modified policy iteration

- Use m_k policy evaluation iterations to compute V^{π_k}.
  Under some mild assumptions even this algorithm converges to V^*.
- Of course if we take $m_k = \infty$ we recover the policy iteration algorithm.
- If we take $m_k = 1$ we obtain the Value Iteration algorithm.
  Every time you update V you use a policy that is greedy w.r.t. V.

Value Iteration

Value iteration

In the Value iteration algorithm you do not keep track of the policy but instead you store in memory the Value function.

Value iteration

So you try to compute the optimal value function and then, only at the end, when you have obtained the optimal value function, you compute the policy.

1. Start with an arbitrary estimation of the Value function V_0
2. Iterate for $k = 0, ..., \infty$
   - For all the states $x \in \mathcal{X}$
     You update the Value function:
     $$V_{k+1}(x) = \min_u l(x, u) + \gamma V_k(f(x, u))$$
     This for loop can be simply rewritten as:
     $$V_{k+1} = TV_k$$
3. V_k is guaranteed to converge to the optimal value function V^*
   $$\lim_{k \to \infty} V_k = V^*$$
4. Once you have obtained the optimal value you can compute the optimal policy π^*
   $$\pi^* = \operatorname*{argmin}_u l + \gamma V^*$$

3.5 Model Free Prediction

In this section we dive deep into the decision process of Reinforcement Learning: in fact, so far, we assumed that the MDP is given and well defined (as well as the dynamics of the system) and we were able to exploit it in order to find the optimal policy or the value function given a fixed policy.

From now on, we will start relaxing this assumption, i.e. we do not know the MDP and we can only choose the control inputs and observe state and cost.
However, in this section we will focus on prediction not control, which will be our foundation for doing control in the next sections.

With prediction, we mean that the policy is assumed to be given (fixed policy) and we just want to find the value function associated with that policy.

fixed policy

This vaguely reseamble the DDP approach policy evaluation, which turned out to be the basis for doing policy iteration (first algorithm that we look at in order to find the optimal policy).

policy evaluation

Formally: Problem: Find the value function of unkown MDP by acting (choosing the action/control input) and observing.

In order to solve the problem there are two main approaches, which are the extreme of a more general approach.

1. Monte Carlo (MC)

   Monte Carlo (MC)

   The key idea of this method is really simple: if I would like to estimate the value of a state

   - I start from that state
   - I run my policy
   - Collect all the costs until the end of the episode (all the episodes must end)
   - I sum all the costs and that is going to give me an estimate of the value of that state.

   Tipically in RL we assume that the MDP and/or the policies and/or the costs are stochastic, so I would have to repeat this process several times. By taking the average over the different summations, we would obtain the mean/estimate.

   In summary: the value of a state can be estimate from the mean of the cost-to-go, however it can only be applied to episodic MDPs (i.e. all episodes must end).

   episodic MDPs

2. Temporal Difference Learning TD0

   Temporal Difference Learning TD0

3.5.1 Monte Carlo Policy Evaluation

The goal of Monte Carlo Policy Evaluation is to learn the value of a given policy V_π from episodes of experience under policy π.

Monte Carlo Policy Evaluation

By episode we mean one execution of the policy from a starting state to a terminal state.

episode

As a consequence, each episode will be a list of state, control and cost for each step:

starting state

$$x_1, u_1, l_1, ..., x_k \sim \pi$$

terminal state

As mentioned before we can compute the Cost-to-go, i.e. the total discounted cost:

$$J_t = l_t + \gamma l_{t+1} + ... + \gamma^{T-1} l_{t+T-1}$$

and then once we have the cost-to-go for each time we can use it to estimate the value function, by means of expected cost-to-go under policy π:

expected cost-to-go

$$V^\pi(x) = \mathbb{E}[J_t | x_t = x]$$

Since we are using the expected value operator we are implicitly assuming some stochasticity in MDP or in the policy or in the cost.

In case of deterministic processes, you only need one sample (no need to apply the expected value operator).

Summary: Monte Carlo policy evaluation estimates V^π as the average cost-to-go.

The algorithm implementation has two versions, depending on when to start accumulating the cost: if you encounter a state multiple times you are gonna get different values depending wheter you start accumulating the cost the first time you meet that state or every time you meet the state.

- First-visit MC policy evaluation
  The first time that a state x is visited in an episode:
  1. Increment a counter (how many times have you met that state)
  $$N(x) \leftarrow N(x) + 1$$
  2. Increment the total cost (in order to compute the cost-to-go)
  $$C(x) \leftarrow C(x) + J(x)$$
  3. Estimate value as the average cost-to-go
  $$V(x) \leftarrow \frac{C(x)}{N(x)}$$

  These three steps are repeated only the first time you encounter a state (no increment happens if you encouter the same state).
  This method works by exploiting the law of large numbers
  $$V(x) \to V^{\pi}(x) \qquad \text{as } N(x) \to \infty$$

- Every visit MC policy evaluation
  the counter and total cost are incremented every time a state is visited.

Since computing the mean each time is annoying what is very often done is Incremental MC updates: update the mean incrementally.

$$\begin{aligned} V_N &= \frac{1}{N}\sum_{j=1}^{N} J_i && \text{decompose summation} \\ &= \frac{J_N + \sum_{j=1}^{N-1} J_j}{N} && V_{N-1} = \frac{1}{N-1}\sum_{j=1}^{N-1} J_j \\ &= \frac{J_N + (N-1)V_{N-1}}{N} \\ &= V_{N-1} + \frac{1}{N}(J_N - V_{N-1}) \end{aligned}$$

In this way we can compute the new mean starting from the previous one. However as N becomes large the influence of the increment tends toward zero. However, this is an undesirable effects especially in non-stationary environments than new samples will have a lower effect on the estimate of the value.

That is why, very often, instead of dividing for N a fixed value is used (it an be useful to track a running mean, i.e. forget old episodes):

$$V(x) \leftarrow V(x) + \alpha(J - V(x))$$

In this view α is how fast we are going to update based on the most recent sample.

3.5.2 Temporal Difference Learning - TD0

Temporal Difference Learning (TD0) is an alternative method to MC, and it estimates the cost-to-go by using the 1-step look ahead, i.e.:

$$V(x_t) \leftarrow V(x_t) + \alpha_t(l_t + \gamma V(x_{t+1}) - V(x_t))$$

where:

- $l_t + \gamma V(x_{t+1}) - V(x_t)$ is the TD error
- $l_t + \gamma V(x_{t+1})$ is the TD target, which is out estimated cost-to-go

It is now very useful to compare it to MC using the inremental update:

$$V(x_t) \leftarrow V(x_t) + \alpha_t(J_t - V(x_t))$$

where

$$J_t \approx l_t + \gamma V(x_{t+1})$$

where rhs reminds us of Bellman equation. The key intution is instead of collecting all the costs starting from the state till the end of the episode, I only see 1 cost and in order to estimate the tail of the cost-to-go I use the estimate of the value function. Whereas MC target is based on what we observe from the environment.

Policy evaluation is TD(0) with $\alpha_t = 1$ and sampling all states uniformly.

The properties of TD(0) are:

- TD(0) is a stochastic approximation algorithm. stochastic approximation algorithm
- Since Bellman operator has a unique fixed point, then if TD(0) converges (and all states are sampled infinitely often) then it must converge to V^π.
- Convergence is guaranteed if step-size sequence satisfies Robbins-Monro (RM) conditions: Robbins-Monro (RM) conditions

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

  which means that cannot tend to zero and how much you update should tend towards zero (you need to update by smaller and smaller amounts to ensure convergence), e.g.

$$\alpha_t = \frac{\bar{\alpha}}{t}$$

- In practice, I constant α is sufficient in most of the time: the algorithm might converge anyway even if we do not have any guarantee of convergence.

3.5.3 Comparison betweeen MC and TD0

MC	TD
• Must wait until end of episode to know cost	• can learn after every step
• only works for episodic environments	• works in non-terminating environments
• MC target (cost-to-go) is unbiased estimate of $V_\pi(x_t)$, given that it is an observation	• TD target is a biased estimate of $V_\pi(x_t)$, given that it is based on the current estimation of the value (which can be arbitrary bad)
• Higher variance estimate of $V_\pi(x_t)$, given that J_t contains the cost of each transition and at each transition some stochasticity exists. Therefore more variance is added to the equation at each transition	• Lower variance estimate of $V_\pi(x_t)$, given that it has only one transition (the estimate of the tail is not stochastic)

In practice, it depends which one perform better:

- MC has high variance, zero bias:
  - very good convergence properties (given by the fact that has zero bias)
  - works well even with function approximation.
    At a certain point in order to scale up this algorithm to be able to work with real system in continuous time, you cannot accept anymore working with arrays, given that the size will grow too much. What you need to do, as a consequence, to make it work with large systems is to use function approximation e.g. neural networks (the value functino will be a NN: instead of changing values inside the vector, we are going to change the weights of the NN).

  - Not very sensitive to initialization.
  - Simple to understand and use

- TD has low variance, some bias:
  - Usually is more efficient than MC
  - converges to $V_\pi(x_t)$ faster
  - but convergence is not guaranteed if you are using functino approximation
  - more sensitive to the initial guess/value

3.5.4 Bootstrapping and Sampling

We can describe the algorithms that we have seen so far with a table.

Sampl. \ Bootstr.	No	Yes
No	ExhaustiveSearch	DP
Yes	MC	TD

where:

- Bootstrapping: updates of the quantity that you are trying to estimate involve an estimate (TD0).
  The TD target is computed based on the current estimate of the value function. `Bootstrapping`
- Sampling: updates sample an expectation (MC).
  Informally we are collecting samples of a quantity and computing the mean: we are approximating an expectation using the mean over a few samples. `Sampling`

3.5.5 n-Step return

Let us know take a step further and see how we can generalize MC and TD0 to a more general class of algorithms which actually include MC and TD0 as special cases.

We can do that by introducing the idea of n-Step return, which is instead of collecting just one transition (i.e. one cost) and summing to this cost the discount times the estimate of the value at the next state, we can generalize this concept to several steps: `n-Step return`

$$
\begin{aligned}
n = 1 \quad & J_t^{(1)} = l_t + \gamma V(x_{t+1}) && TD0 \\
n = 2 \quad & J_t^{(2)} = l_t + \gamma l_{t+1} + \gamma^2 V(x_{t+2}) && \\
\vdots \quad & \vdots && \\
n \quad & J_t^{(n)} = l_t + \gamma l_{t+1} + ... + \gamma^{n-1} l_{t+n-1} + \gamma^n V(x_{t+n}) && \\
\vdots \quad & \vdots && \\
n = \infty \quad & J_t^{(\infty)} = l_t + \gamma l_{t+1} + ... + \gamma^{T-1} l_{t+T-1} && MC
\end{aligned}
$$

And the update equation will be simply (n-step TD learning): `n-step TD learning`

$$V(x_t) \leftarrow V(x_t) + \alpha_t (J_t^{(n)} - V(x_t))$$

In this way we introduced a new kind of algorithm to do prediction and estimating the value of a given policy where instead of using the 1-step cost-to-go (TD0) or the infinite-step cost-to-go

(MC), we can use a more general n-step estimation of the cost-to-go which is going to give us some algorithms which are an inbetween MC and TD0.

So, of course, an algorithm which is an hybrid between these two extreme approaches, will give us something inbetween the properties of MC and TD0: e.g. in terms of bias-variance trade-off we can imagine that as we utilize a higher value of n we are going to get more and more variance in the estimate and lower and lower bias.

In other terms we can make TD0 tends towards the property of MC by tuning the n parameter to the values that we want. In fact the best performance are achieved neither with MC nor with TD0 but for a value of n inbetween.

3.5.6 TD(λ)

We can do something a bit more convoluted: since we have a family of algorithms (one for each value of n) and we do not know which one to use, what if we use all of them?
So we use all the estimates of the cost to go by putting them together with some weights, and that is exactly what TD(λ) does.

TD(λ)

So $TD(\lambda)$ is a new algorithm for doing prediction where you combine all the n-step costs-to-go $J_t^{(n)}$ together by using weights of the form:

$$(1-\lambda)\lambda^{n-1}$$

As a consequence, the TD(λ) target is given by the expression:

TD(λ) target

$$J_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}J_t^{(n)}$$

This means that the algorithm is going to use an higher weight for the first estimate of the cost to go ($n=1$) and a smaller and smaller weight for the remaining n-step costs-to-go estimation.

The reason of the $(1-\lambda)$ premultiplication is just for normalizing the weights, given that the summation of all the weights must be equal to 1 (otherwise you are not taking an average anymore)

$$\sum_{n=1}^{\infty}\lambda^n = \frac{1}{1-\lambda} \qquad \text{with: } 0<\lambda<1$$

The TD(λ) algorithm can be seen in two different fashions: forward view and backward view.

forward view

backward view

- the forward view is much easier and more intuitive to think of and analyze in terms of theoretical properties but is very tricky to implement
- the backward view is much more convoluted and harder to analyze but this is what makes the implementation of this algorithm easier.

Forward view

With the forward view, the value is updated as usual:

$$V(x_t) \leftarrow V(x_t) + \alpha_t(J_t^{\lambda} - V(x_t))$$

The problem with the implementation of this algorithm is that we can only learn from complete episodes (as in MC).

complete episodes

That is the reason why people have derived the backward view of TD(λ).

Backward view

The Backward view of TD(λ) is a little more criptic and hard to understand how it works, but actually, given some assumption, it will be equivalent to the Forward view of TD(λ).

However, this view of the algorithm allows us to updates the estimate at each transition without waiting till the very end.

In order to implement the Backward view of TD(λ) we need to introduce new quantities called Eligibility traces, defined by the following two equations:

Eligibility traces

$e_0(x) = 0$ Initialize Eligibility trace of each state to 0

$e_t(x) = \gamma\lambda e_{t-1}(x) + 1(x_t = x)$ if you meet state x the eligibility trace increase by 1

Once the state is met, then the eligibility trace will decrease.

These quantities measure how often and how far in the past you have visited the state x:

- If you have visited state x very recently, then the associated Eligibility trace will be a value very close to one.
- If you have never visited state x, then the associated Eligibility trace is going to be zero.
- If you have visited state x a long time ago, then the associated Eligibility trace is going to be very close to zero.

In summary: $e(x)$ increases every time I visit x, otherwise it decreases exponentially and it measures how often and recently x has been visited.

Recalling the definition of the TD error:

$$\delta_t = l_t + \gamma V(x_{t+1}) - V(x_t)$$

we can now update the estimate of the value for every state we have visited according to this formulation:

$$V(x) \leftarrow V(x) + \alpha\delta_t e_t(x) \qquad \forall x$$

Proof of equivalency between forward and backward view

- When $\lambda = 0$ only the current state is updated (TD(0)), i.e.:

$$e_t(x) = \mathbf{1}(x_t = x)$$

$$V(x) \leftarrow V(x) + \alpha\delta_t e_t(x) = \begin{cases} V(x) + \alpha\delta_t & \text{if } x = x_t \\ V(x) & \text{otherwise} \end{cases}$$

- When $\lambda = 1$ we get MC.
  Assume x is visited once at $t = k$:

$$e_t(x) = \begin{cases} 0 & \text{if } t < k \\ \gamma^{t-k} & \text{if } t \geq k \end{cases}$$

  The accumulated updates of the value are:

$$\begin{aligned} \sum_{t=1}^{T-1} \alpha\delta_t e_t(x) &= \alpha \sum_{t=k}^{T-1} \gamma^{t-k}(l_{t+1} + \gamma V(x_{t+1}) - V(x_t)) \\ &= \alpha[(l_{k+1} + \cancel{\gamma V(x_{k+1})} - V(x_k) \\ &\quad + (\gamma l_{k+2} + \cancel{\gamma^2 V(x_{k+2})} - \cancel{\gamma V(x_{k+1})}) + \cancel{\dots} \\ &\quad + (\gamma^{T-1-k} l_T - \cancel{\gamma^{T-1-k} V(x_{T-1})}) \qquad \text{Because } V(x_T) = 0 \\ &= \alpha \left[\sum_{t=0}^{T-1-k} \gamma^t l_{k+t+1} - V(x_k) \right] \\ &= \alpha(J_k - V(x_k)) \qquad \Leftarrow \text{Same as MC} \end{aligned}$$

Consequntly:

1. If Value function is only updated offline at the end of the episode then the total update of backward TD(1) is the same of MC
2. Using a similar derivation we could show that backward TD(λ) results in same updates of forward TD(λ) $\forall\lambda$
3. For online updates backward and forward TD(λ) are slightly different.

3.6 Model Free Control

In this section we will cover the application of RL to control. We first encountered Model Based Control with the Dynamic Programming algorithms, but in this instance we will dive deep into the Model Free Control (i.e. given a MDP how can we find a globally optimal policy to behave inside this process?).

Model Based Control

Model Free Control

First of all, when we want to do control we need to be a bit more specific about what our objectives are and what our assumptions are since we might end up with quite different algorithms depending on what we care about.

3.6.1 Types of control learning problems

First and foremost, we need to distinguish between cases in which we assume that we can choose how to interact with the environment versus cases in which how we interact with the environment is already predefined (we can only observe the data that is collected with a given policy or maybe data that was collected in the past).

Furthermore if we assume that there is interaction with the environment, we need to distinguish between two cases:

1. Offline performance optimization: in which we optimize the performance of the system after the learning process has finish.

   In this scenario, we have a phase in which we want to learn the policy, but we do not care how it performs and then a second phase in which we care about performing very well (without the learning).

2. Online performance optimization: in which what we really want to optimize (miximize/minimize) is the cost during training. At the same time we would like to learn and perform well in the task. (more challenging)

optimize performance		Interaction	
		Yes	No
	Offline	Active Learning	Non-Interactive Learning
	Online	Online Learning	

Formally:

- by Interactive we refer to the case in which the learner can actively influence observations
- by Online perfomance we mean the Cost-to-go while learning
- by Offline performance we mean the Cost-to-go after learning

Interactive

Online perfomance

Offline performance

The most common scenario that is treated in RL is Online Learning, and that is what we are going to focus on. From an objective point of view, this class of algorithms is more challenging than Active Learning, but easier than Non-Interactive Learning.

Online Learning

The main challenge of Online Learning is finding a tradeoff between performing well and learning because tipically the action that need to be taken in order to learn are not necessarily the same actions that you need to take in order to perform well (e.g. In order to learn better you need to make some mistakes).

Active Learning

Non-Interactive Learning

3.6.2 Control Learning Methods

When it comes to the specific of how to do the learning phase, then we need to distinguish between two categories:

1. cases in which we assume that the MDP is known vs unknown
2. cases in which we learn the Value function or the Q function vs Value function or the Q function and the policy

We already covered the cases of learning with a known MDP, namely Value iteration and Policy iteration.

Whereas in case of unknown MDP, when we want to learn only the Q function the class is called Direct Methods, instead when we want to learn both the Q function and the policy, the class is called Actor-Critic Methods (because the policy is thought as an actor and the Q function is thought as a critic that evaluates the behaviour of the policy/actor).

Direct Methods

Actor-Critic Methods

		Learned Quantities	
		$V(x)/Q(x,u)$	$V(x)/Q(x,u)$ + π
MDP	Known	Value Iteration	Policy Iteration
	Unknown	Direct Methods	Actor-Critic Methods

Please note that when we know the MDP we learn the Value function whereas when the MDP is unknown we learn the Q function. The reason lays on the fact that when we want to compute the policy from the value, we need to compute the following minimization

$$\pi(x) = \underset{u}{\operatorname{argmin}}\, l(x,u) + \gamma V(x')$$

However, with an unknown MDP we cannot compute the next state x', therefore the best thing we can do is minimize the equivalent Q function:

$$\pi(x) = \underset{u}{\operatorname{argmin}}\, l(x,u) + \gamma V(x') = \underset{u}{\operatorname{argmin}}\, Q(x,u)$$

An alternative would be to learn separately the Value function and the dynamics, then the first expression would be enough, but if we only know the value function that specific minimization is not enough.

3.6.3 On-policy vs Off-policy learning

There is yet another distinction that we need to make in order to discuss about RL alogithms:

- On-policy
  We want to learn a certain policy π while we follow that policy π. So the control inputs are decided by this policy and we want to learn about this policy.
  - *learn on the job*
  - learn about policy π from experience sampled from π.

On-policy

- Off-policy
  The policy utilized to control the system is not the same one we want to learn.
  - *look over someone's shoulder*
  - learn about policy π from experience sampled from μ (i.e. another policy).

Off-policy

This is interesting especially under the perspective of Safety: in RL the first step is to explore in order to gather data given that, in the beginning, it knows nothing about the system is controlling so it going to do random things. This is tipically not really safe.

Safety

So when we work with safety critical systems is quite common that the policy that you want to use to control the system is not the same that you want to learn (e.g. control the robot manipulator with a classical and reliable algorithm such as PID, but as you collect the data you want to learn about the fancy optimal policy which eventually will be used to control the system).

3.6.4 Direct Methods: Q learning

Direct methods are a class of methods in which you only learn the Q function not the policy, since you will figure out the policy later by minimizing the Q function. Direct methods

The idea is to work iteratively update an estimate of the Q function $Q_k(x, u)$ and improve it to the optimal Q function $Q^*(x, u)$.
Here we are still working on a discrete space setting, so the state and control space are finite and the Q function is a table/matrix with $|\mathcal{X}| \times |\mathcal{U}|$ dimension (i.e. Q function associated with a specific state and control).

You therefore control the system with some policy and at each transition you keep track of

$$(x, u, l(x, u), x')$$

and you use them to improve the estimate of Q with the update rule based on the temporal difference learning:

$$\delta(Q) = l + \gamma \min_{u'} Q(x', u') - Q(x, u) \qquad \forall (x, u)$$

Given this TD error, which measure the difference between the expected Q $Q(x, u)$ and the estimation of Q based on the current measure of the cost $l + \gamma \min_{u'} Q(x', u')$. Based on that error you can update the estimate of Q as:

$$Q_{k+1}(x, u) = Q_k(x, u) + \alpha_k \delta(Q_k)$$

We can notice that it is quite similar to TD0, but the main difference is that the TD error is not computed based on the value function, since we are doing control, and the presence of a minimization.

When we reach stochastic equilibrium for each pair state and control which need to be visited infinitely often we obtain: visited infinitely often

$$\mathbb{E}[\delta(Q)] = TQ - Q = 0$$

We know that there is only one Q function that satisfies this expression which is a unique fixed point of the Bellman optimality operator. This imples that Q at stochastic equilibrium converges to the optimal Q function: unique fixed point

$$Q = Q^*$$

Of course Q_k converges when appropriate learning rates α_k are used.

Q learning is a off-policy method.

A key point in this convergence proof is that all pair of state and control needs to be visited infinitely often: this means that the algorithm needs to explore the entire state and control spaces (i.e. need exploration). exploration

Online learnign in "bandits"

How do you choose the control/action u?

Probably the most intuitive things to do is say: we have an estimate of the Q function, which tells us how good a certain control is when we are in a given state, can't we simply choose the control input that minimizes Q for the current state (i.e. being greedy)?

Of course the answer is no: if you just behave greedly then you are not going to explore everything and you are not gonna meet the convergence requirements.

Example. Imagine to have 2 doors in front of you and you can choose to open either of those with a certain cost associated with them. You goal is to minimize the cost and the system that assigns the cost is stochastic (i.e. you are not always gonna get the same cost when opening the same door).

You objective if you want to minimize the cost-to-go with a infinite horizon is to find out which of the 2 doors has the lower expected value, because if you sample the door with the lower expected value infinitely many times the cost that you are going to get, on average, will be the expected value associated with that door.

The problem is as follows:

•Left door	$\Rightarrow$	$l = 0$	$\Rightarrow$	$V(left) = 0$
•Right door	$\Rightarrow$	$l = -1$	$\Rightarrow$	$V(right) = -1$
•Right door	$\Rightarrow$	$l = -3$	$\Rightarrow$	$V(right) = -2$

With a greedy approach you are going to sample the door that yielded the lower cost in the first sample. However, we have no guarantee that the right door is better than the left door, since we will continue sampling the right door. □

This is a problem well studied in RL called Exploration-Exploitation trade-off because you have two conflicting objectives when doing online learning: explore and learn more and improve your policy, but at the same time you want to minimize your cost-to-go that is exploit the knowledge that you already collected.

Exploration-Exploitation trade-off

This is studied in a very simple MDP with a single state, i.e. no evolution of the state, (so called "bandit problem").

bandit problem

- Greedy strategy:

  Greedy strategy

  - Always choose control with best estimated cost
  - It can fail to find the best action/control (large loss over time)
  - Good learner must choose sub-optimal control to explore

- ε-Greedy strategy

  ε-Greedy strategy

  - Choose random control with probability ε
  - Choose greedy control with probability $1 - \varepsilon$
  - Simple way to ensure exploration
  - ε can change over time (tipically decreasing to 0), given that in the beginning you would like to explore a lot and not minimize Q, whereas as soon as you collect more data your estimate becomes better and better, so it makes more sense to be greedy.

- Boltzmann Exploration

  Boltzmann Exploration

  Not choose always the greedy action as the best action but assign probability to each control proportional to the value associated with that control.

  - Choose control with probability proportional to its mean

$$\pi(u) = \frac{exp(-\beta Q_t(u))}{\sum_{u' \in \mathcal{U}} exp(-\beta Q_t(u'))} \qquad Q_t(u) = \text{mean cost obtained applying } u$$

  - For $\beta \to \infty$, we retrieve greedy strategy.
  - Choose more often controls that should lead to low cost.
  - The potential problem is if you are unlucky in the beginning with a given action/control, you might get a very high cost, then you are going to assign a very low probability to that control. So it might take a very long time to correct the estimation of the value associated with that control.

- Optimism in the face of uncertainty

  Optimism in the face of uncertainty

  You keep track of how much uncertaint you are about the value of each control input and how uncertaint you are depends on how many times you have sampled that control input. Since when you have a lot of uncertainty you try to be optimistic, you favour the control input with the highest uncertainty.

  - Choose control with Lowest Confidence Bound (LCB)

    Lowest Confidence Bound (LCB)

  -

$$LCB(u) = Q(u) - \text{uncertainty on } Q(u) = Q_t(u) - L\sqrt{\frac{2\,\log(t)}{n_t(u)}}$$

–

$$L = \max\left(|l(t)|\right) \quad , \quad n_t(u) = \#\text{ times } u \text{ was selected}$$

Note that the uncertainty associated with a control is going to decrease as the number of times that control was selected increase.

3.6.5 Actor-Critic methods: Generalized policy iteration

So inside the Actor-Critic methods the largest family of methods are the so called Generalized policy iteration because they are a generalization of the Dynamic Programming algorithm Policy iteration.

Actor-Critic methods

Generalized policy iteration

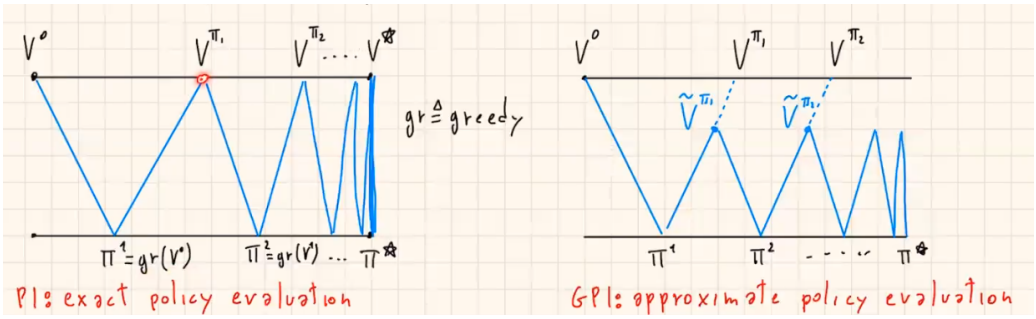
In particular:

- Extension of Policy Iteration to unknow MDP
- The key idea of Policy Iteration is to alternate between two phases, given a policy and the Q function.
  You alternate between evaluating the current policy to update the value/Q function and then in the second phase, given the value function that you estimated improve the policy by acting greedily w.r.t. that estimation of the value.
  In short: alternate between policy evaluation and policy improvement

policy evaluation

policy improvement

- The generalization of policy evaluation phase can be done with Model Free prediction methods (MC, TD0, TDλ).
  However this is time consuming. So generally you are not going to wait for an accurate evaluation of the policy, but you are just going to evaluate it roughly and then try to improve the policy based on that rough evaluation of the previous policy.
- Exact policy evaluation can take infinitely many samples
- Solution: improve policy based on approximate Value function



This class of methods is called Actor-Critic because the value function is called critic given that it evaluates the policy, which is called actor, in order to improve it.

critic

actor

Moreover, this class of methods is on-policy, therefore the policy that you are using to act/choose the control inputs must be the same policy that you want to evaluate. However, this is not completely true since there is always the need for exploration, so you cannot just act greedily w.r.t. the value, there must be some kind of exploration mechanism.

For this reason, 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 the tarket policy

behavior policy

tarket policy

The key difference between GPI and PI is that in GPI there is no guarantee that at each iteration is going to be a bit better then the previous one (GPI can generate policies that are worse than previous ones).

SARSA: policy evaluation

Let us now investigate a specific algorithm that falls into this class, called SARSA.

SARSA

The name comes from the notation that it is used into reinforcement learning:

- the state x is called state s
- the control u is called action a
- the cost l is called reward r

Since at each iteration we observe of keep track of the tuple:

$$(x, u, l, x', u')$$
$$(s, a, r, s, a)$$

The key idea of SARSA is to exploit TD(0) in order to learn Q from on-policy samples (since we are estimating the value of the policy who uses u').

Tipically an ε-greedy policy is utilized to act and at each transition you collect a tuple with the initial state and control, the cost and the next state and control. You use this information in order to update the estimate of the Q function according to the following expression:

$$\delta(Q) = l(x, u) + \gamma Q(x', u') - Q(x, u)$$
$$Q_{k+1}(x, u) = Q_k(x, u) + \alpha_k \delta(Q_k)$$

- It is very similar to Q learning, but it uses $Q(x', u')$ instead of $\min_z Q(x', z)$.
- if π was fixed you would obtain the same equation as temporal difference learning.
- we are constrained to use TD(0), we can extend the algorithm to TD(N) or TD(λ).
- as TD, SARSA can diverge in off-policy situations

SARSA: policy improvement

We have different options available:

1. The simplest one is to take the policy that is greedy w.r.t. Q, since a greedy policy can be computed on the fly, because for each state that we need to choose a control input for we can just look at the Q for that state, look at all the value of Q for all the control inputs and choose the one that minimizes Q.
2. Use $\varepsilon-$ greedy as behavior policy to ensure exploration.

Even for SARSA we have some guarantee of convergence to $Q^*(x, u)$ if:

- Greedy in the Limit with Infinite Exploration (GLIE).
  This means that the algorithm is allowed not to be greedy in the limit, but as time goes towards infinity the policy needs to tend towards a greedy policy and we need to have infinite exploration.
  Formally:
  - All state-control pairs are visited infinitely many times
  - Policy converges to a greedy policy, e.g.:

$$\varepsilon_k = \frac{1}{k}$$

Greedy in the Limit with Infinite Exploration (GLIE)

- Robbins-Monro (RM) conditions sequence of step sizes:

$$\sum_{k=1}^{\infty} \alpha_k = \infty \quad ; \quad \sum_{k=1}^{\infty} \alpha_k^2 < \infty$$

Robbins-Monro (RM) conditions

Summary:

3.7 Value function Approximator

The problem with the algorithms of Model Free Control is that we assumed the MDP to be discrete (i.e. state and control spaces are discrete) which makes this spaces very very large if you want to apply them to real problems.

Discrete-space algorithms do not scale to large systems

DP	TD
Policy Evaluation $V(x) = l + \gamma V(x')$	TD Learning $V(x) \overset{\alpha}{\leftarrow} l + \gamma V(x')$
Policy Iteration $V(x) = l + \gamma V(x')$ $\pi(x) = \text{argmin}_u\, l + \gamma V(x'(u))$	Sarsa $Q(x,u) \overset{\alpha}{\leftarrow} l + \gamma Q(x',u')$ $\pi(x) = \text{argmin}_u\, Q(x,u)$
Value Iteration $V(x) = \min_u l + \gamma V(x'(u))$	Q-Learning $Q(x,u) \overset{\alpha}{\leftarrow} l + \gamma \min_{u'} Q(x',u')$

- Backgammon: 10^{20} states
- Go: 10^{170} states
- Robot: continuous state space

In all these cases there are too many states/controls to store and learn.

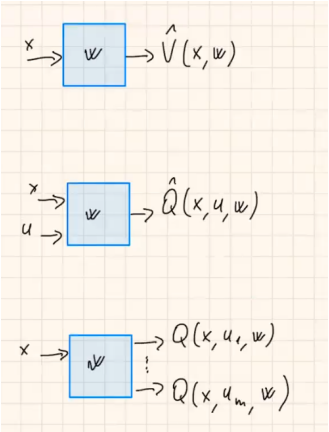
In order to solve this problem: instead of representing the value or Q function with a table or an array we can use function approximation techniques in order to approximate these functions

$$\hat{V}(x,w) \approx V_\pi(x) \qquad , \qquad \hat{Q}(x,u,w) \approx Q_\pi(x,u)$$

where w are the parameters of the function.

This function approximation are nothing more than parametric functions, so the learning process aims to learn the function parameters such that the function approximator reseambles the actual functions and, as a consequence, reduce the size of the vector that you work with.

3.7.1 Types of value function approximation



First of all both Q and/or V can be learnt:

1. You can learn the Value function by tuning the parameters of the function approximator, by providing as input the state x
2. You can learn the Q function by tuning the parameters of the function approximator, by providing as input both state x and control u
3. You can parametrize the Q function, in the case of discrete control space, by providing as input only the state x, obtaining several output each of which corresponds to each possible value of the control input $u_1, ..., u_m$.

Very often, in the context of RL, the kind of value function approximators that are used are Neural Networks.

Neural Networks

However any differential approximator can be used:

- linear combination of features (arbitrary functions of the state)
- Fourier basis
- Polynomials

The main problem with these approach is that we are trying to approximate a non-stationary target (i.e. V) and the data is not independent normally distributed i.i.d..

non-stationary target

independent normally distributed i.i.d.

3.7.2 Gradient descent

In order to fit the value function via function approximators, the most common approach that is used is the Gradient Descent, which consists of taking a small step in the direction of gradient of the function that you want to minimize, i.e.:

Gradient Descent

$$\min_w J(w) \qquad \Rightarrow \qquad w = w - \frac{1}{2}\alpha \nabla_w J(w)$$
$$\alpha = \text{ step-size parameter}$$

If the step is sufficiently small then the algorithm will converge to a local optimum.

In the context of RL we do not use gradient descent but a slight variation called Stochastic Gradient Descent (SGD). The reason it is stochastic is that the quantity that we are trying to minimize is an expectation of the square difference between the real value function and the approximated value function.

Stochastic Gradient Descent (SGD)

It is an expectation because the value of this depends on the policy π, hence we want to take the expectation of u following the distribution of the policy π.

$$J_w = \mathbb{E}_\pi[(V_\pi(x) - \hat{V}(x,w))^2]$$
$$\Delta w = -\frac{1}{2}\alpha \nabla_w J = \alpha[(\mathbb{V}_\pi - \hat{\mathbb{V}})\nabla_{\lessapprox} \hat{\mathbb{V}}]$$

Since we do not have any tool to actually compute the argument of the expectation, one workaround is to sample such quantity for different value of the state. By doing so, the update rule of the coefficient takes the form:

$$\Delta w = \alpha\,(V_\pi(x) - \hat{V}(x,w))\nabla_w \hat{V}(x,w)$$

Similarly to what we did with MC or TD learning: we run a simulation following a policy π that we want to evaluate, encountering different states (i.e. our actual samples).

The interesting things about this approach is that there is a whole theory behind this stochastic gradient descent method, which tells us that if we perform the update in such a way, (by sampling the gradient that we want to minimize), then the expected update is equal to the full gradient update that we would get by computing the gradient of the expectation.

Of course, the stochasticity of the system will cause the gradient update to be noisy, but on average the noisy is going to cancel out and ultimately the algorithm will perform a step in the right direction.

Now the next question is: how do we get the target $V(x)$?

3.7.3 Incremental prediction algorithms

In order to perform SGD we need a target for our value function, an oracle that can tell us the value function that the function approximator should fit.

We have already seen how to get it when we looked at the prediction problem (how to evaluate a policy and compute the value of that policy). We have also seen that there are two main classes of approaches to get the value:

- For MC, the target is obtained by computing the cost-to-go $J_t(x)$.
  As a consequence, a simulation is run, the costs are collected for each time step and once one reaches the end, the cost-to-go is computed for each state by doing a backward computation.
  In the end, the update rule of the coefficient will take the form:

  $$\Delta w = \alpha\,(J_t - \hat{V}(x_t,w))\,\nabla_w \hat{V}(x_t,w)$$

  Note: the gradient of the value function approximation depends on the method use to approximate the function.

MC converges to local optimum even with nonlinear approximators; in the case we are using a linear approximators then the problem will be a convex/quadratic problem and the local optimum will be also a global optimum.

- For $TD(0)$, the target given by the TD target:

$$\Delta w = \alpha(l_t + \gamma \hat{V}(x_{t+1}, w) - \hat{V}(x_t, w))\nabla_w \hat{V}(x_t, w)$$

TD does not have the same properties as MC, but it converges (close) to global optimum with linear approximators. In case of nonlinear approximators this approach is not guaranteed to converge given that there is bias in the estimator.

With these Incremental Predition algorithms, we can do prediction using function approximation, and we can extend this approach to control.

Incremental Predition algorithms

3.8 Incremental Control with function approximation

When we switch from prediction to control, we need to switch from value function to action value function Q, since we do not know the dynamics.

The idea is to Generalized policy iteration with an approximate action-value function, i.e.:

$$J(w) = \min_w \mathbb{E}_\pi[(\hat{Q}(x, u, w) - Q_\pi(x, u))^2]$$

Once again SGD will be used to find the local minimum, which translates to:

- For MC, target is cost-to-go $J_t(x_t, u_t)$
- For TD(0), the target is $l_t(x_t, u_t) + \gamma \hat{Q}(x_{t+1}, u_{t+1}, w)$

And then, as we did before, we can use an ε-greedy polity to ensure exploration.

How can we get a greedy behaviour with a continuous space? For the moment we are going to focus on the case in which the space is continuous but the control space is discrete. Because if the control space is discrete then it is really easy to implement the greedy control policy by minimizing the estimation of the Q function because it requires just to compute the Q function for all possible control inputs.

If instead, the control space was continuous then this problem of being greedy w.r.t. Q becomes much more complex since it requires to define a new optimization problem in continuous space in order to be greedy. However, in this circumstance you cannot ensure that you are going to be globally greedy.

3.8.1 Convergence of Control Algorithm

	Table Lookup	Linear	Nonlinear
MC control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-learning	✓	✗	✗

where ✓indicates that the algorithm is guaranteed to converge, (✓) indicates that the algorithm chatter around the optimum and ✗indicates that the algorithm is not guaranteed to converge.

3.8.2 Batch Learning

Instead of learning incrementally, learn after having collected a batch of traning data, i.e. collect all the data by running the policy for some number of episodes and then use all this batch of data points to perform an update of the weights of the neural network.

The function that you want to minimize in that case is the least-squares prediction, i.e.:

$$\min_w \sum_{t=1}^{T} (V_t - \hat{V}(x_t, w))^2$$

The result of doing batch learning can be achieved by using a so called Experience replay buffer, which is nothing more than a list of all the data points that were observed in the past:

Experience replay buffer

- Each time you take a simulation step you store the state and the value associated to that state $< x_t, V_t >$ in the replay buffer
- When the buffer is full, you sample from buffer
- Apply SGD to update the weights w (what is different is that we are not updating w at every step, and we are not using the latest data point to update the weights, but instead we are using data points at different point in time and states).
- It is guaranteed to LS solution

The reason we are not using Batch Learning is that the batch would be too large.

3.8.3 Deep Q-Network

In order to make this really work for continuous state space we need to introduce another trick presented in the algorithm called Deep Q-Network (2015, MNIH et al.).

Deep Q-Network

It is an extension of Q learning to continuous state space, whereas the control space is still discrete.

It uses neural network to represent the Q function $Q(x, u, w)$. The policy that was used is an $\varepsilon-$greedy policy with ε decreasing over time.

Uses experience replay buffer:

- Store transitions (x_t, u_t, l_t, x_{t+1}) in replay buffer
- Every few steps (e.g. 4) sample randomly from replay buffer a mini-batch of transitions (e.g. size 32)
- Use stochastic gradient descent to optimize w

What is new is to use fixed Q targets. In fact, since you are using Temporal difference to compute the targets for fitting the Q function, instead of using just one Q network you are going to use a different network to compute the targets and one network to update the weights.

fixed Q targets

- Compute Q targets using old-fixed parameters w^- (old values of the parameter w)
- Optimize MSE between Q-network and Q-learning targets

$$\underset{w}{\text{minimize}}\, \mathbb{E}[l + \gamma \min_{u'} Q(x', u', w^-) - Q(x, u, w)]$$

- Every C steps update the target weights:

$$w^- \leftarrow w$$

  That is the purpose of stabilizing the algorithm since if you do not fix the parameters of your target network, instability can be observed: the target that you are trying to reach is constantly moving since it depends on w and w is constantly changing.
  By keeping a target fixed, we give enough time to the SGD to converge to that time before you update the target.
- Store w resulting in best performance as performance can get worse during training.

3.8.4 Implementation TIps

- Implement new Pendulum environment with continuous state and discrete control
- Install tensorflow
- Read papers in folder orc/03_assignment
- Read code in "DQN template.py"
- Use online documentation of tensorflow and keras
- If you manage to make it work for simple pendulum you can then try it with a double pendulum or ur5 robot

Appendix A

Numerical Integration

We are going to review Numerical Integration indipendently from Optimal Control.

Given an Ordinary Differential Equation (ODE)

$$\dot{x} = f(x, t)$$

with

$$x(0) = x_0$$

Compute $x(t)\ \forall t \in [0, T]$.

If f is Lipshitz continuous (i.e. the first derivative of the function are bounded) than this problem has an unique solution. Lipshitz continuous

Moreover in f has been omitted the dependency on $u(t)$ given that it is a function of time t and/or x.

A.1 Numerical Integration Methods

A.1.1 Explicit Euler

Explicit Euler is the simplest numerical integration scheme based on the definition of the derivative as: Explicit Euler

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

with h "small" we can approximate $\dot{x}$ with the value on the rhs without the limit and write:

$$h\,\dot{x} \approx x(t+h) - x(t)$$

which means that

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

So the key assumption of the Explicit Euler is to take h sufficiently small in order for it to work but the methods does not specify exactly how small it must be the step. However taking h small means that it will be computationally expensive.

Moreover it is not very accurate (first order method).

A.1.2 Mid-Point Method

The Mid-Point Method is slightly better than the Explicit Euler. Mid-Point Method

You take a first step to reach the middle of the time step, you look at the value of $\dot{x}$ in the middle and this is the value that you use in the whole time step.

So you go back and you integrate the time step using the value of $\dot{x}$ that you computed in the middle of the step:

$$\boxed{\begin{aligned} k_1 &= f(x_0, t_0) \\ k_2 &= f\left(x_0 + \frac{h}{2}k_1, t_0 + \frac{h}{2}\right) \\ x_1 &= x_0 + h\,k_2 \end{aligned}}$$

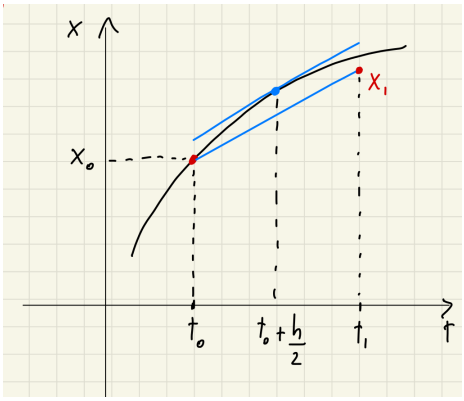


Figure A.1: Mid-Point method: find the midpoint of the interval, compute derivative, compute area starting from the initial point of the interval using the slope of the midpoint

1. Use the slope in the middle of the step to integrate forward
2. Estimate $x\left(t_0 + \frac{h}{2}\right)$ with the slop at t_0
3. Second order method (order 2)
4. Need 2 function evaluation for each step

The theory that allows us to state that almost always the midpoint method is better than explicit Euler is the theory of numerical integration scheme.

A.1.3 Runge-Kutta Methods

The Euler method and midpoint method are both specific cases of Runge-Kutta method. Runge-Kutta method

The RK methods are based on the following equations:

$$\boxed{\begin{aligned} x_{n+1} &= x_n + h \sum_{i=1}^{q} b_i\, k_i \\ k_i &= f\left(x_n + h \sum_{j=1}^{q} a_{ij}\, k_j, t_n + c_i\, h\right) \end{aligned}}$$

where:

- q is the order of the method (not to be confused with the consistency order of the method)
- For $q = 1$ we recover Euler by setting $(b_1 = 1, a_{11} = 0, c_1 = 0)$
- The Mid-point method is an RK method of order $q = 2$
- If $a_{ij} = 0\ \forall j \geq i$ the method is called explicit method. explicit method
  Otherwise the method is called implicit method (need to solve system of equations because to compute k_i you need to know the value of other k that are all interdependent). implicit method
- For $q > 1$ there exist many versions of the same order.
- the term $\sum_{i=1}^{q} b_i\, k_i$ is an approximation of $\dot{x}$ with an weighted average of k_i, with weights b_i. Where
$$\sum b_i = 1 \qquad b_i \geq 0$$

Therefore to define a RK method you need all the values of parameter a, b and c. Tipically these parameters are stored in a so called Butcher Tableau. Butcher Tableau

$$\begin{array}{c|c} c_1 & a_{11} \cdots\cdots a_{1q} \\ \vdots & \vdots \quad \ddots \quad \vdots \\ c_q & a_{q1} \cdots\cdots a_{qq} \\ \hline & b_1 \cdots\cdots b_q \end{array}$$

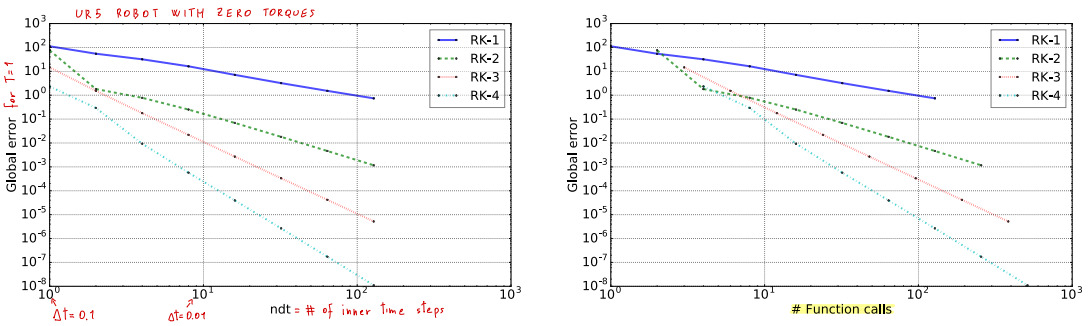
- Once you have this tableau you have uniquely defined the RK method
- Most common explicit RK4 method and takes the form

$$\begin{aligned} k_1 &= f(x_n,\, t_n) \\ k_2 &= f\left(x_n + \frac{1}{2}h\,k_1, t_n + \frac{1}{2}h\right) \\ k_3 &= f\left(x_n + \frac{1}{2}h\,k_2, t_n + \frac{1}{2}h\right) \\ k_4 &= f(x_n + h\,k_3, t_n\, +\, h) \\ x_{n+1} &= x_n + \frac{h}{6}(k_1 + 2\,k_2 + 2\,k_3 + k_4) \end{aligned}$$

and it can be uniquely defined by the Butcher Tableau:

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

The main reason this method is so common comes from the fact that it draws the limit for which you get a method that has the same consistency order of the order of the RK scheme (i.e. for $p \geq 5 \nexists$method with $q = p$).



A.2 Properties of Integration Schemes

Definitions:

- Integrator output: $\hat{x}(t, t_0, x(t_0))$
- Exact trajectory: $x(t)$
- Local integration error: $e(t) = x(t) - \hat{x}(t, t-h, x(t-h))$
- Global integration error: $E(t) = x(t) - \hat{x}(t, t_0, x(t_0))$

Now that we have defined these two error we can talk about the properties of integration schemes which are:

1. Convergence. Convergence

   Convergence is the basic properties that ensures that as the time step that you use for integration goes to zero the global error should go to zero as well:

   $$\lim_{h \to 0} E = 0$$

   It is not a very difficult property to achieve. Basically all the integration methods have this property.

2. Consistency order p. Consistency order

   Tells you how quickly the error goes to zero. In particular it is formulated using the local error. In particular the limit of the local error should go to zero as a polynomial of the time step:

   $$\lim_{h \to 0} e = O(h^{p+1}) \qquad p > 0$$

   So when we refer to a method order 1 or 2 we refer to the value of p on the above definition.

3. Stability. Stability

   There is no clear definition of stability. But we can approximately define it as the global error remains bounded as we keep integrate ($t \to \infty$)

From this definitions we can now quantify the performance of each method through the notion of consistency order.

- The local error for Explicit Euler takes the form:

  $$e(t) \triangleq x(t+h) - \hat{x}(t+h, t, x(t)) = x_{n+1} - \hat{x}_{n+1}(x_n) = e_{n+1}$$

  where

  $$x_n \triangleq x(t_n)$$

  So we can reformulate the expression as:

  $$\hat{x}_{n+1} = x_n + h\, f(x_n, t_n)$$

  Let us express x_n with a Taylor series around x_n:

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

  Whatever remains is something in the order of h^2. And Finally we can compute the error with the expression above:

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

  So this means that the method is of order 1 (i.e. $p = 1$)
- The local error for Mid-point method takes the form:

  $$e(t) = x(t+h) - \hat{x}(t+h, t, x(t)) = x_{n+1} - \hat{x}_{n+1}(x_n)$$

  where

  $$\begin{aligned} k_1 &\triangleq f(x_n, t_n) = \dot{x}_n \\ \hat{x}_{n+1} &= x_n + h\, f\left(x_n + \frac{h}{2} k_1, t_n + \frac{h}{2}\right) \\ f &= \dot{x} \\ \dot{f} &= \left(\frac{\partial f_n}{\partial x_n}\frac{dx_n}{dt_n} + \frac{\partial f_n}{\partial t_n}\right) = \ddot{x} \end{aligned}$$

Let us express x_{n+1} with the Taylor series around x_n:

$$x_{n+1} = x_n + h\,\dot{x}_n + \frac{h^2}{2}\ddot{x}_n + O(h^3)$$

And let us express $f(x_n + \frac{h}{2}\dot{x}_n, t_n + \frac{h}{2})$ with the Taylor series around (x_n, t_n):

$$\begin{aligned} f\left(x_n + \frac{h}{2}\dot{x}_n, t_n + \frac{h}{2}\right) &= f(x_n, t_n) + \frac{\partial f_n}{\partial x_n}\frac{h}{2}\dot{x}_n + \frac{\partial f_n}{\partial t_n}\frac{h}{2} + O(h^2) \\ &= \dot{x}_n + \frac{h}{2}\ddot{x}_n + O(h^2) \end{aligned}$$

We can finally write the error using the expressions above:

$$\begin{aligned} e_{n+1} &= x_{n+1} - \hat{x}_{n+1} \\ &= \cancel{x_n} + \cancel{h\dot{x}_n} + \frac{h^2}{2}\ddot{x}_n + O(h^3) - \left(\cancel{x_n} + h\left(\cancel{\dot{x}_n} + \frac{h}{2}\ddot{x}_n + O(h^2)\right)\right) \\ &= O(h^3) \end{aligned}$$

So this means that the method is of order 2 (i.e. $p = 2$)

Appendix B

Lyapunov Stability

B.1 Exponential Lyapunov functions

Stability of a method can be proved using Lyapunov functions, which are one of the main tools to prove stability of nonlinear systems. Lyapunov functions

Def: A set S is positive invariant if $\forall x \in S$ the next state is also inside the set, i.e. $f(x) \in S$. In other terms if x starts in S then it stays inside S. positive invariant

Def: Suppose $\mathfrak{X}$ is positive invariant, a value function $V : \mathbb{R}^n \to \mathbb{R}^+$ is an exponential Lyapunov function if $\exists \alpha_1, \alpha_2, \alpha_3 > 0$ (three positive constants) s.t. $\forall x \in \mathfrak{X}$, the following conditions are satisfied: exponential Lyapunov function

$$V(x) \geq \alpha_1 \|x\|$$
$$V(x) \leq \alpha_2 \|x\|$$
$$V(f(x)) - V(x) \leq -\alpha_3 \|x\|$$

Which means that the value function needs to be

- lower bounded
- upper bounded
- decreasing exponentially along the trajectory of x.

The following theorem specifies when the system is exponentially stable, i.e.: exponentially stable

$$\|x_k\| \leq c\,\gamma^k \,\|x_0\| \quad \text{for some } c > 0 \text{ and } \gamma \in [0, 1]$$

Theorem 4. *If $\exists$ a Lyapunov function then the origin (state $x = 0$) of the system is exponentially stable in the set $\mathfrak{X}$.* origin

If $\mathfrak{X} = \mathbb{R}^n$ then the origin is globally exponentially stable.

Even though this theorem can be applied to stabilize the origin, it can be extended to an arbitrary state.

Index

Multibody dynamics modeling . . . 4
Robot Manipulator . . . 4
links . . . 4
joints . . . 4
end-effector . . . 4
configuration vector . . . 4
Homogeneous Transformation matrix . . . 4
wrench . . . 7
Direct dynamics . . . 7
Inverse dynamics . . . 7
reference joint trajectory . . . 8
dynamics . . . 11
initial conditions . . . 11
path constraints . . . 11
running cost . . . 11
terminal cost . . . 11
principle of optimality . . . 12
Bellman principle . . . 12
Indicator function . . . 13
Value function . . . 14
Optimal cost-to-go . . . 14
optimal feedback control policy . . . 14
Q function . . . 14
parametric optimization . . . 14
numerical optimization . . . 14
lookup table . . . 15
Curse of dimensionality . . . 15
Linear Quadratic Regulator (LQR) . . . 15
Approximate Dynamic Programming . . . 15
Neural Dynamic Programming . . . 15
Direct methods . . . 15
non linear programming solver (NLP) . . . 15
well conditioned . . . 15
ill-conditioned . . . 15
Single Shooting . . . 16
Sequential approach . . . 16
Collocation . . . 16
Simultaneous approach . . . 16
Multiple Shooting . . . 16
Discretized Path constraints . . . 16
augmented state . . . 17
sensitivities . . . 17
integrator function . . . 18
discretized dynamics function . . . 18
sparsity . . . 21
coarse grid . . . 21

continuity contraints . . . 22
Linear Quadratic Regulator LQR . . . 22
discrete-time linear system . . . 22
least-squares problem . . . 24
horizon . . . 24
Hessian of the quadratic form . . . 25
gradient of the linear form . . . 25
Projection matrix . . . 26
optimal linear feedback policy . . . 26
Discrete-time algebraic Riccati Equation (DT-ARE) . . . 26
inhomogeneous cost . . . 27
Differential Dynamic Programming (DDP) . . . 28
non-linear dynamical system . . . 28
non-linear cost function . . . 28
regularization . . . 29
regularization value . . . 29
Backward Pass . . . 30
Forward Pass . . . 30
hyperparameter . . . 31
Regularization Scheme . . . 31
globally optimal feedback policy . . . 31
Model Predictive Control (MPC) . . . 32
finite-horizon . . . 32
feasibility . . . 33
stability . . . 33
computation time . . . 33
recursive feasibility . . . 34
terminal cost . . . 34
terminal constraint . . . 34
infinite horizon running cost . . . 34
Input constraints only . . . 34
Hard state constraints . . . 34
Maximum Output Admissible Set theory . . . 34
Maximum Control Invariant Set . . . 34
closed-loop convergence . . . 34
domain of feasibility . . . 34
Control Invariant . . . 35
origin . . . 35
persistently feasible . . . 36
backward reachable sets . . . 36
Value function . . . 37
exponential Lyapunov function . . . 37
optimal value function . . . 38
sufficient condition . . . 38
closed set . . . 38
compact set . . . 38
Control Lyapunov Function (CLF) . . . 39
invariant region . . . 39
stabilizable . . . 39
small basin of attraction . . . 41
Economic MPC theory . . . 41
warm start . . . 41
convergence threshold . . . 42
uncertainty . . . 42
sensor noise . . . 42
Robust MPC approach . . . 42
stocastic uncertainty . . . 42

Gaussian distribution . . . 42
less conservative . . . 42
Linear dynamics . . . 42
Nonlinear dynamics . . . 42
Reinforcement Learning (RL) . . . 43
gloablly optimal policy . . . 43
Markov Decision Processes (MDP) . . . 44
fully observable environment . . . 44
Markov property . . . 44
State transition probability . . . 44
State Transition matrix . . . 44
Markov Process . . . 44
state transition probability matrix . . . 44
Markov Chain . . . 44
irreducible . . . 44
strongly connected graph . . . 44
Perron-Frobenius theorem . . . 44
deterministic systems . . . 45
probability density functions . . . 45
Markov Reward Process . . . 45
discount factor . . . 45
later costs . . . 45
immediate costs . . . 45
myopic evaluation . . . 45
far-sight evaluation . . . 45
monetary budget . . . 46
Bellman Equation . . . 46
Markov Decision Process MDP . . . 46
policy . . . 47
deterministic policies . . . 47
Action-Value Function . . . 47
Optimal Value Function . . . 47
Optimal Action-Value Function . . . 47
Optimal policy . . . 47
Bellman Optimality equation . . . 48
Dynamic Programming . . . 48
Prediction . . . 48
Control . . . 48
optimal control policy . . . 48
optimal value function . . . 48
Finite Horizon . . . 48
Infinite Horizon . . . 48
Bellman (expectation backup) operator . . . 49
Bellman optimality (backup) operator . . . 49
Iterative Policy Evaluation . . . 50
contracting . . . 50
geometric rate . . . 50
Policy Iteration . . . 51
minimization by enumeration . . . 51
Modified policy iteration . . . 52
Value Iteration . . . 52
Value iteration . . . 52
fixed policy . . . 53
policy evaluation . . . 53
Monte Carlo (MC) . . . 53
episodic MDPs . . . 53
Temporal Difference Learning TD0 . . . 53

Monte Carlo Policy Evaluation . . . 53
episode . . . 53
starting state . . . 53
terminal state . . . 53
expected cost-to-go . . . 53
First-visit MC policy evaluation . . . 54
law of large numbers . . . 54
Every visit MC policy evaluation . . . 54
Incremental MC updates . . . 54
Temporal Difference Learning (TD0) . . . 54
1-step look ahead . . . 54
TD error . . . 54
TD target . . . 54
stochastic approximation algorithm . . . 55
Robbins-Monro (RM) conditions . . . 55
Bootstrapping . . . 56
Sampling . . . 56
n-Step return . . . 56
n-step TD learning . . . 56
TD(λ) . . . 57
TD(λ) target . . . 57
forward view . . . 57
backward view . . . 57
complete episodes . . . 57
Eligibility traces . . . 57
Model Based Control . . . 59
Model Free Control . . . 59
Interactive . . . 59
Online perfomance . . . 59
Offline performance . . . 59
Online Learning . . . 59
Active Learning . . . 59
Non-Interactive Learning . . . 59
Direct Methods . . . 60
Actor-Critic Methods . . . 60
On-policy . . . 60
Off-policy . . . 60
Safety . . . 60
Direct methods . . . 61
visited infinitely often . . . 61
unique fixed point . . . 61
exploration . . . 61
Exploration-Exploitation trade-off . . . 62
bandit problem . . . 62
Greedy strategy . . . 62
ε-Greedy strategy . . . 62
Boltzmann Exploration . . . 62
Optimism in the face of uncertainty . . . 62
Lowest Confidence Bound (LCB) . . . 62
Actor-Critic methods . . . 63
Generalized policy iteration . . . 63
policy evaluation . . . 63
policy improvement . . . 63
critic . . . 63
actor . . . 63
behavior policy . . . 63
tarket policy . . . 63

SARSA . . . 63
Greedy in the Limit with Infinite Exploration (GLIE) . . . 64
Robbins-Monro (RM) conditions . . . 64
Neural Networks . . . 65
non-stationary target . . . 66
independent normally distributed i.i.d. . . . 66
Gradient Descent . . . 66
Stochastic Gradient Descent (SGD) . . . 66
Incremental Predition algorithms . . . 67
Experience replay buffer . . . 68
Deep Q-Network . . . 68
fixed Q targets . . . 68
Lipshitz continuous . . . 69
Explicit Euler . . . 69
Mid-Point Method . . . 69
Runge-Kutta method . . . 70
explicit method . . . 70
implicit method . . . 70
Butcher Tableau . . . 70
Convergence . . . 72
Consistency order . . . 72
Stability . . . 72
Lyapunov functions . . . 74
positive invariant . . . 74
exponential Lyapunov function . . . 74
exponentially stable . . . 74
origin . . . 74

Bibliography