Introduction to Robotics

Julien Carpentier, Stéphane Caron, Yann de Mont-Marin

Class notes by Antoine Groudiev

Last modified 9th November 2024

Contents

Abstract

This document is Antoine Groudiev's class notes while following the class *Motion planning in robotics and graphical animation* (Planification de mouvement en robotique et en animation graphique) at the Computer Science Department of ENS Ulm. It is freely inspired by the lectures of Justin Carpentier, Stéphane Caron, and Yann de Mont-Marin.

1 Introduction

2 Position and Orientation

2.1 Introduction

Kinematics studies the *movement* of an object – in our case of a robot – without taking into acount the *forces* generating it. Instead, it only handles aspects such as position, orientation, speed and momentum of bodies in movement.

Consider for instance a robotic arm. We can design a simplified scheme of the robot and its environment, to create a kinematic pipeline and reference frames associated to each of these objects.

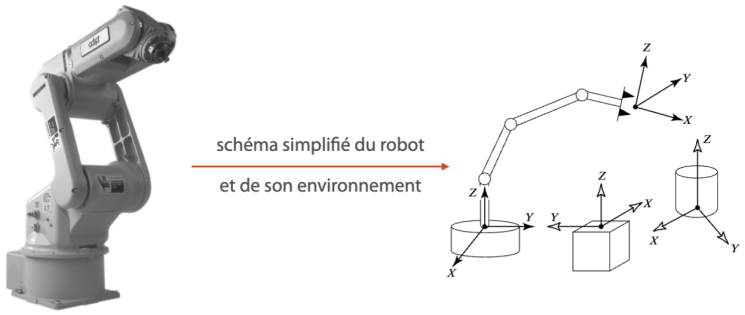
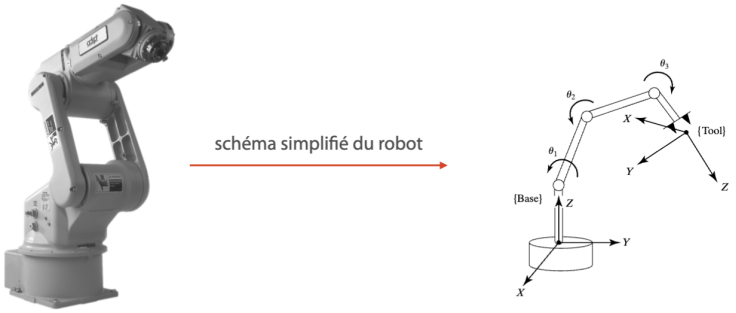


Figure 2.1: Simplified scheme of the robot, the kinematic pipeline and reference frames.

Direct kinematics allows to compute the position and orientation of the terminal organ given, for instance, the angles of the articulations.



Invert kinematics answers the question the other way around: given the position and orientation of a boyd, how can we compute the values of the articulations angles. Invert kinematics is used for instance for trajectory tracking: given a reference trajectory, how can we compute the speed of the articulations?

2.2 Points, frames and transformations

2.2.1 Position of a point in space

Once that a reference frame $\{A\}$ is defined, we can localize any point of the universe given a *position vector*:

$$ {}^{A}P = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} $$

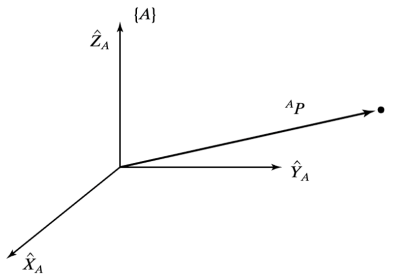


Figure 2.2: Vector and position of the point established in the frame $\{A\}$.

2.2.2 Position and orientation of a body in space

To define the orientation of a body in space, we need to define a reference frame $\{B\}$ attached to this body. The orientation is therefore defined as the expression of this coordinate system in the reference frame $\{A\}$.

$$ {}^{A}_{B}R = \begin{bmatrix} {}^{A}\hat{X}_B & {}^{A}\hat{Y}_B & {}^{A}\hat{Z}_B \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} $$

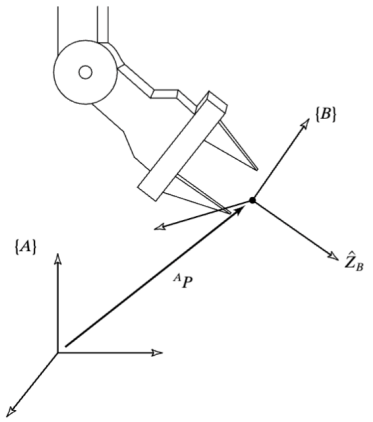


Figure 2.3: Expression of the coordinate system $\{B\}$ in the reference frame $\{A\}$, and the associated position and orientation of the body.

Each element of the matrix ${}^{A}_{B}R$ is the scalar product between the vectors of the two coordinate systems $\{B\}$ and $\{A\}$:

$$ {}^{A}_{B}R = \begin{bmatrix} {}^{A}\hat{X}_B & {}^{A}\hat{Y}_B & {}^{A}\hat{Z}_B \end{bmatrix} = \begin{bmatrix} \hat{X}_B \cdot \hat{X}_A & \hat{Y}_B \cdot \hat{X}_A & \hat{Z}_B \cdot \hat{X}_A \\ \hat{X}_B \cdot \hat{Y}_A & \hat{Y}_B \cdot \hat{Y}_A & \hat{Z}_B \cdot \hat{Y}_A \\ \hat{X}_B \cdot \hat{Z}_A & \hat{Y}_B \cdot \hat{Z}_A & \hat{Z}_B \cdot \hat{Z}_A \end{bmatrix} $$

The lines correspond to the axes of the frame $\{A\}$ expressed in the frame $\{B\}$. Note that the invert of a rotation matrix is its transpose:

$$ {}^{A}_{B}R^T = {}^{A}_{B}R^{-1} = {}^{B}_{A}R $$

2.2.3 Rotation matrices

We showed that the orientation of the body in space could be expressed as a 3-dimensional rotation matrix. The group of 3-dimensional rotation matrices is denoted SO(3), and is composed of all the matrices that are orthonormal, that is orthogonal and with a determinant equal to 1:

$$\mathrm{SO}(3) = \left\{ R \in \mathcal{M}_3(\mathbb{R}) \,\middle|\, RR^T = I_3 \text{ and } \det(R) = +1 \right\} \tag{2.2.1}$$

If we write:

$$R = \begin{bmatrix} \hat{X} & \hat{Y} & \hat{Z} \end{bmatrix}$$

Then $R \in \mathrm{SO}(3)$ if and only if:

$$\begin{cases} \hat{X} \cdot \hat{Y} = 0 \\ \hat{Y} \cdot \hat{Z} = 0 \\ \hat{Z} \cdot \hat{X} = 0 \end{cases} \quad \text{and} \quad \begin{cases} \hat{X} \cdot \hat{X} = 1 \\ \hat{Y} \cdot \hat{Y} = 1 \\ \hat{Z} \cdot \hat{Z} = 1 \end{cases}$$

Note that we have 9 degrees of freedom and 6 independent constraints, so the group SO(3) is 3-dimensional.

2.3 Rotation representations

There are several ways to represent a rotation matrix, each with its own advantages and drawbacks. The most common representations are:

- Orthonormal 3 by 3 matrices — 9 components
- Euler angles — 3 components
- Axis-angle representation — 3 components
- Quaternions — 4 components

The number of components used to be an important factor when computers were slow and memory was expensive. Nowadays, the choice of representation is more about the ease of use and the properties of the representation.

2.3.1 Euler angles

Each rotation can be represented by the composition of three elementary rotations around the fixed axes of the frame $\{A\}$.

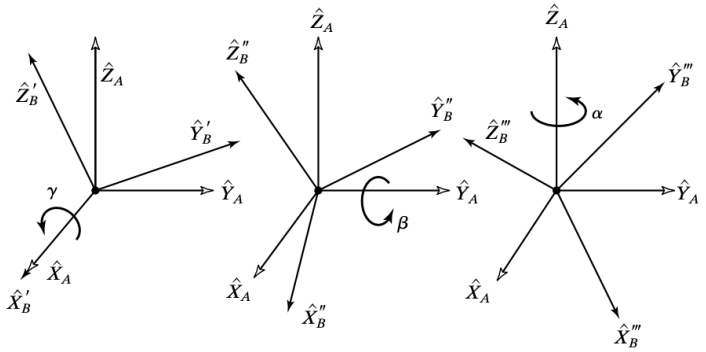


Figure 2.4: Euler angles representation of a rotation.

In terms of rotation matrices, any rotation matrix can be written as ${}^A_B R(\gamma, \beta, \alpha)$, defined by:

$$\begin{aligned} {}^A_B R(\gamma, \beta, \alpha) &= R_Z(\alpha) R_Y(\beta) R_X(\gamma) \\ &= \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix} \end{aligned}$$

The Euler angles are not unique, as the same rotation can be represented by different sets of Euler angles. This is called gimbal lock, and is a major drawback of the Euler angles representation.

We can also express the Euler angles given the rotation matrix:

$${}^A_B R(\gamma, \beta, \alpha) = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \implies \begin{cases} \beta = \text{atan2}(-r_{31}, \sqrt{r_{11}^2 + r_{21}^2}) \\ \alpha = \text{atan2}(r_{21}/\cos\beta, r_{11}/\cos\beta) \\ \gamma = \text{atan2}(r_{32}/\cos\beta, r_{33}/\cos\beta) \end{cases}$$

2.3.2 Axis-angle and quaternions

We are given a vector $\vec{k}$ and an angle θ; the rotation represented by these two elements is the rotation of angle θ around the axis $\vec{k}$. We can define:

$$\begin{cases} \varepsilon_1 = k_x \sin\frac{\theta}{2} \\ \varepsilon_2 = k_y \sin\frac{\theta}{2} \\ \varepsilon_3 = k_z \sin\frac{\theta}{2} \\ \varepsilon_4 = \cos\frac{\theta}{2} \end{cases}$$

we have $\varepsilon_1^2 + \varepsilon_2^2 + \varepsilon_3^2 + \varepsilon_4^2 = 1$, creating a unit quaternion. The rotation matrix associated to the quaternion is:

$$R_\varepsilon = \begin{bmatrix} 1 - 2\varepsilon_2^2 - 2\varepsilon_3^2 & 2(\varepsilon_1\varepsilon_2 - \varepsilon_3\varepsilon_4) & 2(\varepsilon_1\varepsilon_3 + \varepsilon_2\varepsilon_4) \\ 2(\varepsilon_1\varepsilon_2 + \varepsilon_3\varepsilon_4) & 1 - 2\varepsilon_1^2 - 2\varepsilon_3^2 & 2(\varepsilon_2\varepsilon_3 - \varepsilon_1\varepsilon_4) \\ 2(\varepsilon_1\varepsilon_3 - \varepsilon_2\varepsilon_4) & 2(\varepsilon_2\varepsilon_3 + \varepsilon_1\varepsilon_4) & 1 - 2\varepsilon_1^2 - 2\varepsilon_2^2 \end{bmatrix}$$

The invert operation is also simple to compute:

$${}^A_B R(\gamma, \beta, \alpha) = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \implies \begin{cases} \varepsilon_1 = \frac{r_{32} - r_{23}}{4\varepsilon_4} \\ \varepsilon_2 = \frac{r_{13} - r_{31}}{4\varepsilon_4} \\ \varepsilon_3 = \frac{r_{21} - r_{12}}{4\varepsilon_4} \\ \varepsilon_4 = \frac{1}{2}\sqrt{1 + r_{11} + r_{22} + r_{33}} \end{cases}$$

2.4 Angular velocity

The angular velocity of a body is a vector that describes the rotation of the body in space. It is defined as the derivative of the rotation matrix with respect to time. Angular velocity matrices are skew-symmetric matrices, i.e. matrices of the tangent space of SO(3), denoted so(3):

$$\text{so}(3) = \left\{ W \in \mathcal{M}_3(\mathbb{R}) \,\middle|\, W^\top = -W \right\} = \left\{ \begin{bmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{bmatrix} \,\middle|\, w_x, w_y, w_z \in \mathbb{R} \right\}$$

Indeed, consider the rotation matrix $R(t) \in \mathrm{SO}(3)$, and denote $\dot{R}(t) = \frac{\partial R(t)}{\partial t}$ the derivative of $R(t)$ with respect to time. We have:

$$\begin{aligned} R(t)R(t)^\top &= I_3 \\ \dot{R}R^\top + R\dot{R}^\top &= 0_3 \\ \dot{R}R^\top &= -R\dot{R}^\top \\ \dot{R}R^\top &= -\left(\dot{R}R^\top\right)^\top \end{aligned}$$

Hence, by defining $W := \dot{R}R^\top$, we have $W \in \mathrm{so}(3)$.

Note that $R(t)$ is the solution of the differential equation $\dot{R} = WR$, with $R(0) = R_0 \in \mathrm{SO}(3)$. The solution is:

$$R(t) = R_0 \exp(Wt) \quad \text{where} \quad \exp(Wt) = \sum_{n=0}^{+\infty} \frac{(Wt)^n}{n!}$$

This gives us a new parameterization of the group SO(3), using the angular velocity matrices. Recall that a matrix $W \in \mathrm{so}(3)$ is associated to a vector $w = (w_x, w_y, w_z) \in \mathbb{R}^3$; we denote $\hat{w} = W$. The main question associated to this representation is to compute the infinite sum of the exponential function:

$$\exp(tW) = \sum_{n=0}^{+\infty} \frac{(tW)^n}{n!} \tag{2.4.1}$$

We can use the fact that any skew-symmetric matrix $W \in \mathrm{so}(3)$ is nilpotent, i.e. there exists an integer $n \in \mathbb{N}$ such that $W^n = 0_3$. This allows us to compute the exponential function:

$$\hat{w} = W = \begin{bmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{bmatrix} = w \times \cdot$$

Therefore, $\hat{w}^2 = ww^\top - I_3$, and $\hat{w}^3 = -\hat{w}$. We can then regroup the different terms of the exponential function, by transforming the exponents greater than 3:

$$\exp(tW) = I_3 + \sin(t)\hat{w} + \frac{1 - \cos(t)}{t}\hat{w}^2 \tag{2.4.2}$$

2.5 Exponential and logarithm map

We saw that the exponential map is an application from the tangent space of SO(3), so(3) to the group SO(3), defined by:

$$\begin{aligned} \exp : \mathrm{so}(3) &\longrightarrow \mathrm{SO}(3) \\ tW &\longmapsto \exp(tW) = I_3 + \sin(t)\hat{w} + \frac{1 - \cos(t)}{t}\hat{w}^2 \end{aligned}$$

This function is surjective and 2π-periodic.

We can define a reciprocal function, the logarithm map, that is the inverse of the exponential map:

$$\log : R \in \mathrm{SO}(3) \longrightarrow w \in \mathrm{so}(3)$$

Note that

$$\|w\| = \cos^{-1}\left(\frac{\mathrm{Tr}(R) - 1}{2}\right)$$

and therefore:

$$w = \frac{\|w\|}{2\sin(\|w\|)} \begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix}$$

This gives us a way to compute the distance between two rotations:

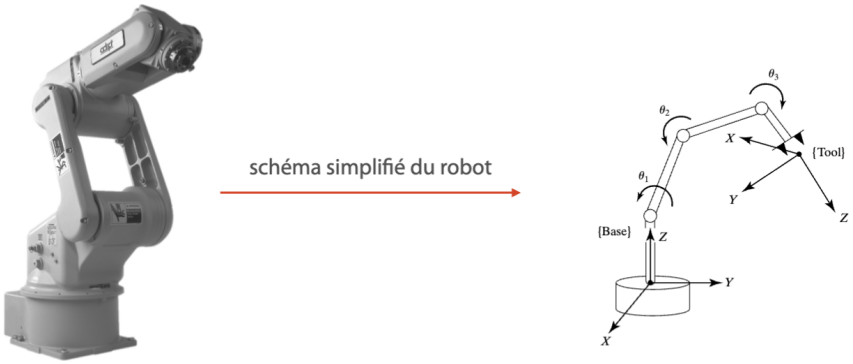
$$d(R_1, R_2) = \|\log(R_1^\top R_2)\|$$

Such a construction comes handy when we want to minimize the distance between two rotations, for instance in the context of trajectory tracking.

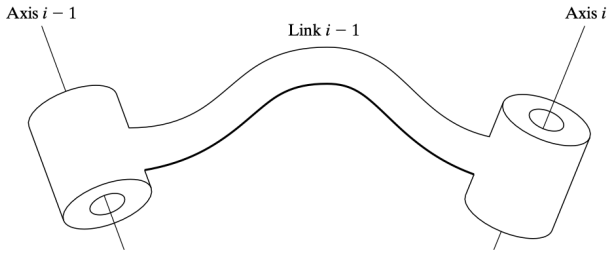
2.6 Rigid Body Transformations

3 Forward Kinematics

Forward kinematics allows to compute the position and orientation of the kinematic chain given the joint parameters (e.g. angles, lengths, etc.). For instance, we can compute the position and orientation of the terminal organ given the angles of the articulations.



The kinematic chain is composed of rigid bodies interconnect by joints. The joins define the degrees of freedom of the cinematic chain, which are the parameters that we can control.



3.1 The various possible articulations

4 Inverse Kinematics

5 Motion Planning

Motion planning is the problem of finding a sequence of valid actions that will transform the robot from its initial configuration to a desired configuration. Such actions can be constrained: we might ask the robot to avoid obstacles, to respect its kinematic limits, or to minimize energy consumption.

We will first introduce a mathematical framework to describe the robot's configuration space, obstacles, and collision, then we will present some algorithms to solve the motion planning problem.

5.1 Configuration space

5.1.1 Definitions

The ambient space, also called the *workspace*, is in most cases the Euclian manifold $\mathbb{E}^3$. In this ambient space, a physical system can be described as a set of compacts subsets called *bodies*. The vector of the considered physical systems must be in the *state space* $\mathcal{S}$, the set of all possible sets:

$$(K_1, \dots, K_N) \in \mathcal{S} \subseteq \mathcal{K}(\mathbb{E}^3)^N$$

The state space is parametrized with a transformation q in the *configuration space* $\mathcal{C}$:

$$q = (\phi_1, \dots, \phi_M) \in \mathcal{C}$$

where each ϕ_i is a *transformation* of a system, that is:

$$\phi_i : \mathcal{K}(\mathbb{E}^3) \to \mathcal{K}(\mathbb{E}^3)$$

Note that M corresponds to the number of systems of the robot, the other physical systems being obstacles. Therefore, given a configuration q, the space S_q is:

$$S_q = \underbrace{\phi_1(K_1^\circ), \dots, \phi(K_M^\circ)}_{\text{robot}}, \underbrace{K_{M+1}, \dots, K_N}_{\text{obstacles}}$$

where the K_i° are compacts representing each body, intuitively in their own frames. We will see below that they can be represented numerically with meshes or polynomials.

In general, $\mathcal{C}$ is a manifold, its dimension being the number of degrees of freedom of the system. Note that the numerical dimension used to represent $\mathcal{C}$ is not necessarily the same as the actual dimension of the manifold: for instance, quaternions of dimension 4 can be used to represent rotations of $SO(3)$, a manifold of dimension 3.

5.1.2 Joints

Multiple types of joints can be used to connect the different parts of a robot. Formally, a joint is a mapping from a manifold of dimension $1 \leqslant p \leqslant 6$ into the solid placement $SE(3)$. The difference between the joints is the dimension of the manifold, which can also be seens as the number of degrees of freedom of the joint.

5.1.3 Examples

Moving solid Let us consider the Euclidian manifold $\mathbb{E}^2$ as a workspace, that is 2D objects. A solid object K is described as a compact subset of $\mathbb{E}^2$. The configuration space is the set of transformations of this object, $SE(2)$, the special Euclidian group of dimension 2.

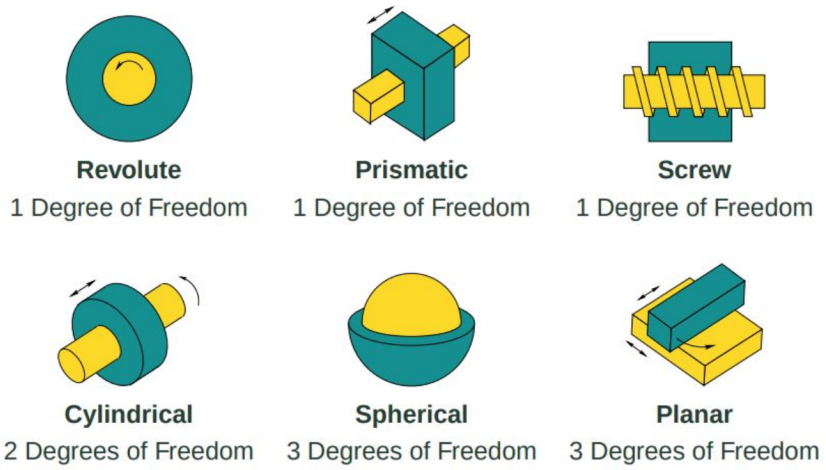


Figure 5.1: Different types of joints.

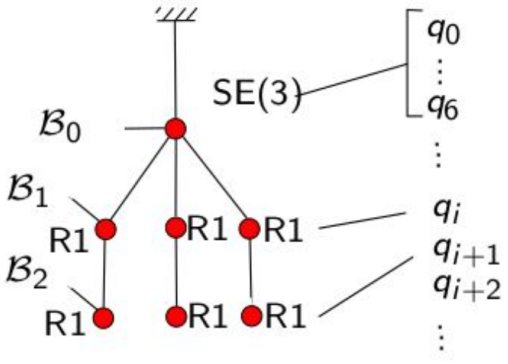
Double pendulum In the case of a double pendulum, the workspace remains $\mathbb{E}^2$. Because of the physical constraints on the structure of the system, the pendulum is parametrized by two angles θ_1 and θ_2. The configuration space is therefore a torus $T^2 = S^1 \times S^1$, but can also be seen as $[0, 2\pi) \times [0, 2\pi)$.

Rigid poly-articulated body The same general framework can be applied to more complex systems, such as humanoid robots. The workspace is $\mathbb{E}^3$, and the configuration q of a robot is represented by the concatenation of the parameters of each joint:

$$q \in \underbrace{SE(3)}_{\text{Position}} \times \underbrace{S_1 \times \cdots \times S_1}_{\text{Revolute joints}} \times \underbrace{[0,1] \times \cdots \times [0,1]}_{\text{Prismatic \& bounded joints}}$$

Recall that the forward kinematic can be used to compute the position of each joint in the global frame, given the configuration q.

A humanoid robot



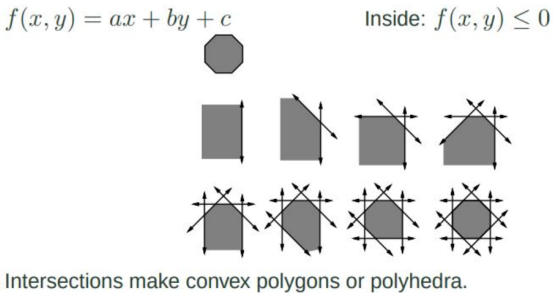
The associated configuration tree

5.2 Obstacles and collisions

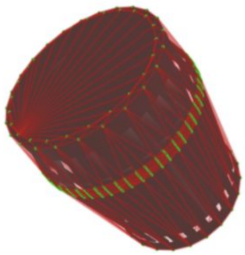
Obstacles and collision handling are crucial in motion planning, and are responsible for the complexity of the problem. We will dive deeper into this topic in another chapter, but we introduce here the fundamental notions needed for motion planning.

5.2.1 Representations and modelisation

There are multiple ways to represent obstacles, which are used in different algorithms. The *polynomial representation* consists in representing the obstacles as sets satisfying polynomial inequalities. The *mesh representation* and *primitive representation* define simple sets of points in the workspace that are respectively inside or outside the obstacles.



Polynomial representation



Mesh representation

Given a physical state S_q of the form

$$S_q = \phi_1(K_1^\circ), \ldots, \phi(K_M^\circ), K_{M+1}^\circ, \ldots, K_N^\circ$$

we can define the space of configurations $\mathcal{C}_{\text{obs}}$ that are in collision with the obstacles:

$$\mathcal{C}_{\text{obs}} = \left\{ q \in \mathcal{C} \;\middle|\; \underbrace{\exists i \neq j \leqslant M, \phi_i(K_i^\circ) \cap \phi_j(K_j^\circ) \neq \varnothing}_{\text{Auto-collision}} \text{ or } \underbrace{\exists i \leqslant M, j > M, \phi_i(K_i^\circ) \cap K_j^\circ \neq \varnothing}_{\text{Environment collision}} \right\}$$

We can then define the *free space* $\mathcal{C}_{\text{free}}$ as the set of configurations that are not in collision with the obstacles, that is:

$$\mathcal{C}_{\text{free}} = \mathcal{C} \setminus \mathcal{C}_{\text{obs}}$$

Therefore, the problem of motion planning is to find a path in the free space $\mathcal{C}_{\text{free}}$ that connects the initial configuration to the final configuration.

5.2.2 Example: translation without rotation of a rigid body

Let us consider the workspace $\mathbb{E}^2$ and a rigid body K_r that can be translated without rotating. The configuration space is $SE(2)$, and the obstacles are represented by a set of points in the workspace.

It is handy to compute the Minkowski sum of the robot K_r and the obstacles K_o:

$$K_r + K_o = \{ x + y \mid x \in K_r, y \in K_o \}$$

If there is a path outside the Minkowski sum, then it is possible to find a path to move the body without rotating it.

Figure 5.2: Use of the Minkowski sum of the robot and the obstacles.

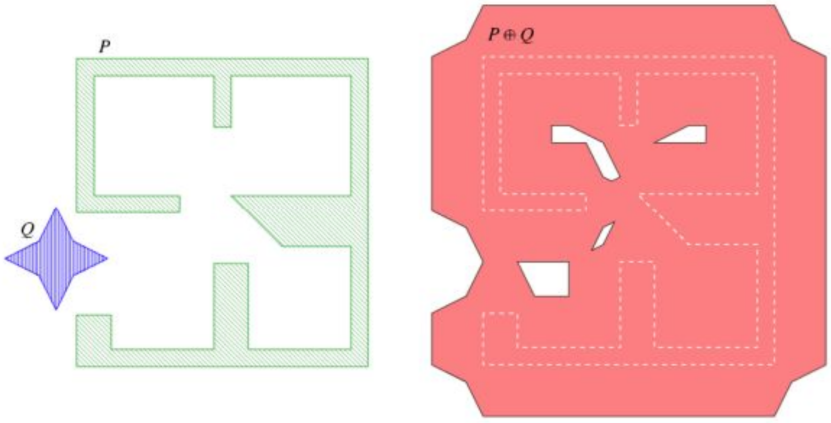


Figure 5.3: Another example, without a feasible path.

Thanks to the configuration space approach, motion planing can be reduced to a punctual trajectory problem. Given $q_i, q_f \in \mathcal{C}_{\text{free}}$, we want to find a continuous path $\gamma \in \mathcal{C}^0([0,1], \mathcal{C}_{\text{free}})$ such that $\gamma(0) = q_i$ and $\gamma(1) = q_f$.

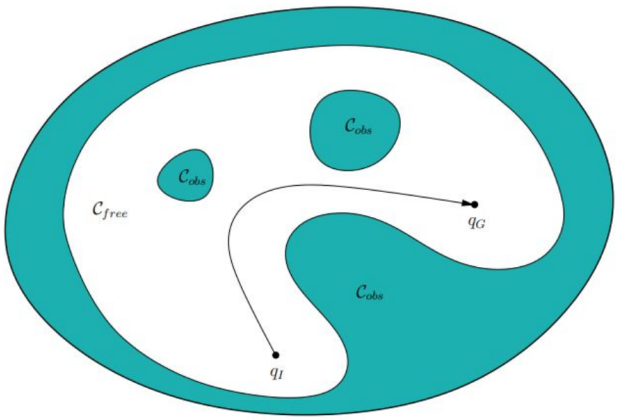


Figure 5.4: Example of a continuous path in the free configuration space.

Nevertheless, this still raises sub-questions. From a mathematical point of view, we can study the existence and optimality of such paths. From a numerical point of view, we need to choose a representation of the configuration space and obstacles, and to design algorithms to compute the path.

5.3 Motion planning algorithms

5.3.1 Algorithm paradigms

Three competing paradigms exist to solve the motion planning problem:

- *Combinatorial planning* consists in finding a global, exact planning. It is "moral" but computationally unrealistic.
- *Trajectory optimization* consists in finding a local planning. It is fast but remains a local approach.
- *Sampling-based motion planning* probabilistically explores the configuration space. It is very efficient but does not guarantee optimality. This is the most used approach in practice.

Such algorithms come with different possible level of completeness:

- A *complete* algorithm returns a solution if one exists, and reports a failure otherwise.
- A *semi-complete* algorithm returns a solution if one exists, but may run forever otherwise.
- For a *probabilistically complete* algorithm, if a solution exists, the probability that it will be found approaches 1 as the number of iterations approaches infinity.

5.3.2 Combinatorial planning

Combinatorial planning was developed in the 80s, and remains close to the mathematical problem studied. It is extremely efficient for low-dimensional problems, and are *complete* and even sometimes *resolution complete*. On the other hand, some are difficult to implement due to numerical issues, and most are intractable even with medium-sized problems.

Principle The main idea is to build finite roadmaps; a *roadmap* is a graph $\mathcal{G}$ such that:

- each vertex is a configuration $q \in \mathcal{C}_{\text{free}}$,
- each edge $e(q, q')$ is a path $\gamma \in \mathcal{C}^0([0,1], \mathcal{C}_{\text{free}})$ such that $\gamma(0) = q$ and $\gamma(1) = q'$.

Furthermore, a roadmap $\mathcal{G}$ is a *topological graph* for the space $\mathcal{C}_{\text{free}}$ if:

- $\mathcal{G}$ is *accessible*: from anywhere in $\mathcal{C}_{\text{free}}$, it is trivial to compute a path that reaches at least one point along any edge in $\mathcal{G}$
- $\mathcal{G}$ is *homotopic-preserving*: every close path in $\mathcal{C}_{\text{free}}$ can be homotopically deformed into a path which is the concatenation of edges that form a cycle in $\mathcal{G}$.

In the following, we will assume that $\mathcal{C}_{\text{free}}$ is piecewise linear, that is, it is the union of a finite number of polytopes. The robot can be either a point, a polygonal, or even a disc, which has the ability to move without rotation (that is, it can be translated).

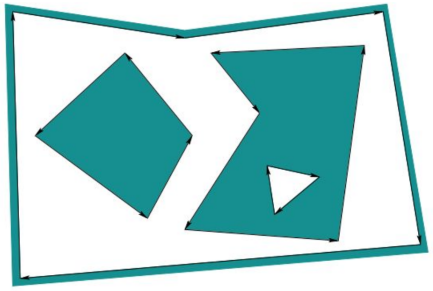


Figure 5.5: Example of piecewise linear obstacles.

Sweep A first approach to build a roadmap is to use a *sweep* algorithm. The idea is to use the plane sweep principle to efficiently determine where the rays terminate on the obstacles. We sort the vertices by abscisse coordinate, and we handle extensions from left to right, while maintaining a vertically sorted list of edges. This leads to a simple to implement, $O(n \log n)$-time algorithm, where n is the number of vertices of the obstacles.

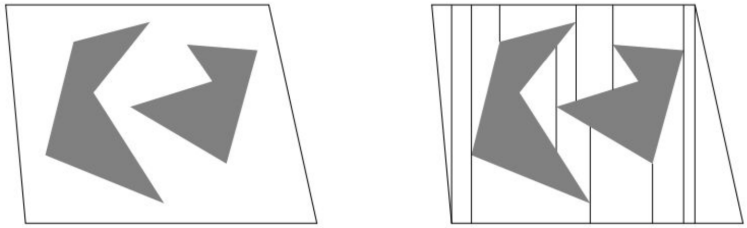


Figure 5.6: Example of sweeping.

5.3.3 Trajectory optimization (local planning)

Trajectory optimization, also known as local planning, use optimal control methods such as DDP or iLQR to find a local solution to the motion planning problem. It has a strong terminal cost, and does not guarantee to explore all the homotopy classes. To solve this issue, we need global optimization methods, that are only viable in small dimensions, or momentum methods, as introduced by El Khadir and Lassere in 2020.

5.3.4 Sampling-based motion planning (probabilistic planning)

6 Collision Detection

Collision detection is a subject at the center of physics simulators. To build a simulation of the robot in its environment, the main loop goes as follows:

1. Collision detection: finding contact points
2. Collision resolution: finding contact forces using physical principles
3. Time integration: update of the quantities of interest (position, velocity, etc.)

It is therefore crucial to have an efficient collision detection algorithm, that is to know whether two objects are in contact or not, and if so, to find the contact points.

Nevertheless, collision detection is a computational bottleneck in physics simulators. Resolving collision detection for one pair of objects takes a significant amount of time, especially for complex shapes, and the number of pairs to check grows quadratically with the number of objects. A general method to optimize such a process is to decompose one collision detection into two phases, the broad phase and the narrow phase. The broad phase uses simple geometric primitives to quickly discard pairs of objects that are far from colliding. The narrow phase then uses more complex geometric primitives to find the exact contact points.

6.1 The broad phase

6.1.1 Bounding volumes

As described previously, the broad phase uses *bounding volumes* (BVs) to prune collisions: we will only check the overlapping BVs, and leave fine-grain detection for the narrow phase. The main goal is to efficiently determine when two objects are far from overlapping, to prune such pairs of objects. Therefore, we will always use an over-approximation of the shape of the object, but never an under-approximation. We accept to mark as a likely collision two objects that are not intersecting, but we do not want to miss any.

The narrow phase will later determine which pairs are actually colliding, and which aren't, using exact representation of the objects. This two-phases approach allows for improved performance without sacrificing the precision of the detector.

6.1.2 Example of bounding volumes

The choice of a bounding volume is always a tradeoff between performance and precision. Larger BVs such as spheres or axis-aligned bounding boxes rely on extremly efficient algorithms, at the cost of a poor representation of the actual volume of the object. On the other hand, complex volumes such as the convex hull represent precisely the objects while requiring a longer time to compute collisions.

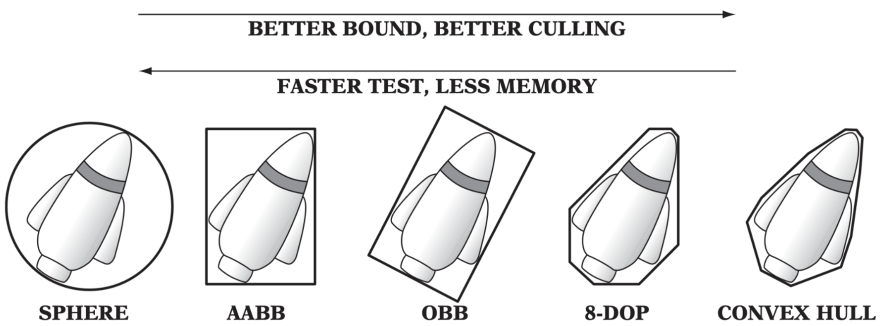
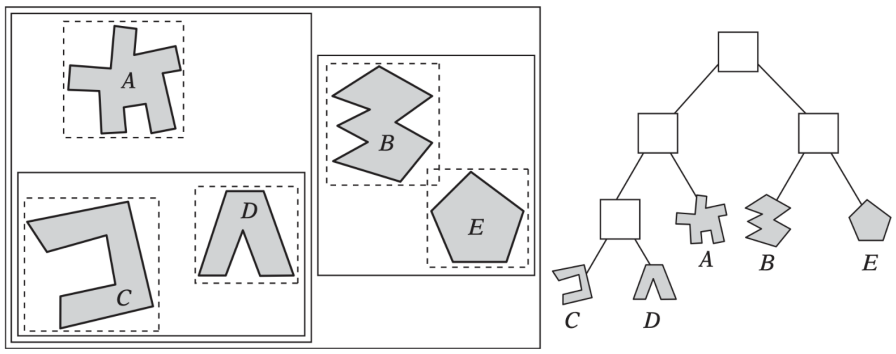


Figure 6.1: Examples of bounding volumes.

It is worth noting two important examples: Axis-Aligned Bounding Boxes (AABB) and Oriented Bounding Boxes (OBB), which can be efficiently computed, and represent accurately most objects we might have to deal with. Note that the choice of the axes is arbitrary.

6.1.3 Dynamic tree representation

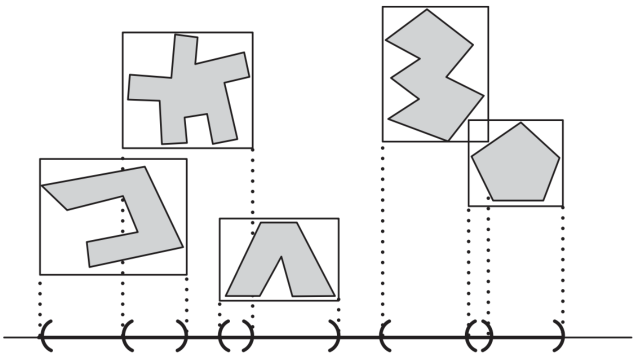
Given any representation of the bounding volumes of the objects of the scene, an efficient way to compute the potential collisions between these objects is to build a *dynamic tree*.



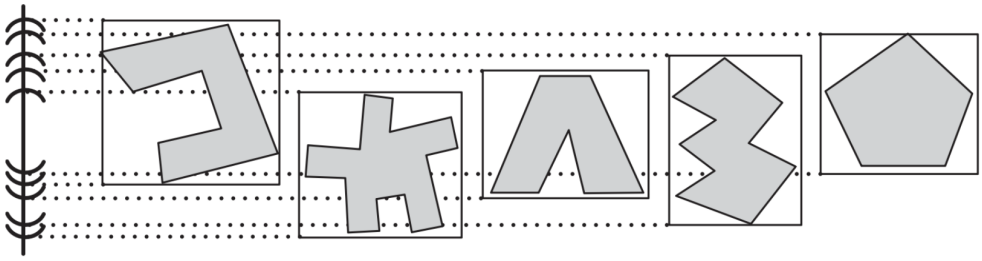
The idea is to split the scene into primitives that are easy to compute, for instance AABBs, such that any object is in exactly one subpart of the scene. Therefore, two objects are coliding only if they are children of the same node of the tree, which drastically reduces the number of collisions to check.

6.1.4 Sweep and Prune algorithm

When using exclusively AABBs, another efficient approach is to use the *Sweep and Prune* (SaP) algorithm. Its idea is to sort the bounding boxes along each axis of their axis. Objects may overlap at their starts or ends; if two objects are overlapping for every axis, one can conclude that their AABBs collide.



Such a method can become problematic if multiple objects are almost aligned on the same line.



6.2 The narrow phase