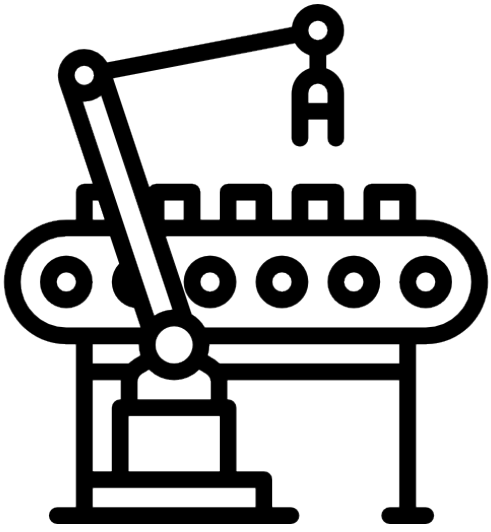
UNIVERSITÀ
DI SIENA
1240

Dipartimento di Ingegneria dell'Informazione e Scienze Matematiche

M.Sc. in Artificial Intelligence and Automation Engineering

INDUSTRIAL ROBOTICS

Prof. Gionata Salvietti



Authors: Alberto Vaglio & Alessio De Bona

Contents

Chapter 1

Review of Robotics fundamentals

From *Robotics, Modelling, Planning and Control, Sciavicco, Siciliano, Villani, Oriolo*

1.1 Kinematics

1.1.1 Pose of a rigid body

A *rigid body* is completely described in space by its ***position*** and ***orientation*** (in brief pose) for a reference frame. As shown in Fig. 1.1, let *O–xyz* be the orthonormal reference frame and $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z}$ be the unit vectors of the frame axes.
The **position** of a point *O'* on the rigid body for the coordinate frame *O–xyz* is expressed by the relation:

$$\boldsymbol{o'} = o'_x\boldsymbol{x} + o'_y\boldsymbol{y} + o'_z\boldsymbol{z}$$

where o'_x, o'_y, o'_z denote the components of the vector $\boldsymbol{o'} \in \mathbb{R}^3$ along the frame axes; the position of O can be compactly written as the (3×1) vector:

$$\boldsymbol{o'} = [o'_x \quad o'_y \quad o'_z]^T \tag{1.1}$$

Vector $\boldsymbol{o'}$ is a bound vector since its line of application and point of application are both prescribed, in addition to their direction and norm.

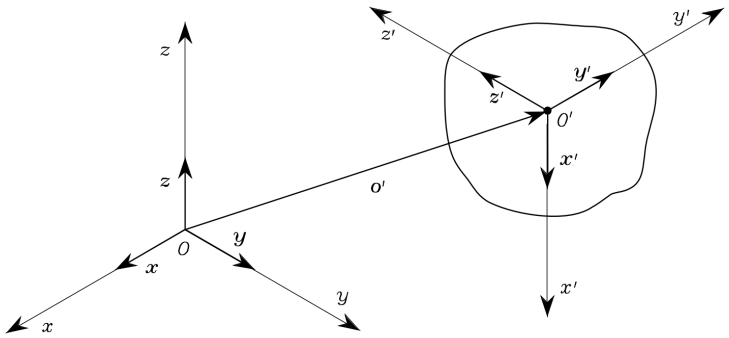


Figure 1.1: Position and orientation of a rigid body

To describe the rigid body **orientation**, it is convenient to consider an orthonormal frame attached to the body and express its unit vectors concerning the reference frame.

Let then O–xyz be such a frame with the origin in O and x, y, z be the unit vectors of the frame axes. These unit vectors are expressed concerning the reference frame O–xyz by the equations:

$$\begin{aligned} \boldsymbol{x'} &= x'_x\boldsymbol{x} + x'_y\boldsymbol{y} + x'_z\boldsymbol{z} \\ \boldsymbol{y'} &= y'_x\boldsymbol{x} + y'_y\boldsymbol{y} + y'_z\boldsymbol{z} \\ \boldsymbol{z'} &= z'_x\boldsymbol{x} + z'_y\boldsymbol{y} + z'_z\boldsymbol{z} \end{aligned} \tag{1.2}$$

The components of each unit vector are the direction cosines of the axes of frame O'–$x'y'z'$ with respect to the reference frame O–xyz.

1.1.2 Rotation Matrix

By adopting a compact notation, the three unit vectors in Fig. 1.2 describing the body orientation concerning the reference frame can be combined in the (3×3) matrix:

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{x'} & \boldsymbol{y'} & \boldsymbol{z'} \end{bmatrix} = \begin{bmatrix} x'_x & y'_x & z'_x \\ x'_y & y'_y & z'_y \\ x'_z & y'_z & z'_z \end{bmatrix} = \begin{bmatrix} \boldsymbol{x'}^T\boldsymbol{x} & \boldsymbol{y'}^T\boldsymbol{x} & \boldsymbol{z'}^T\boldsymbol{x} \\ \boldsymbol{x'}^T\boldsymbol{y} & \boldsymbol{y'}^T\boldsymbol{y} & \boldsymbol{z'}^T\boldsymbol{y} \\ \boldsymbol{x'}^T\boldsymbol{z} & \boldsymbol{y'}^T\boldsymbol{z} & \boldsymbol{z'}^T\boldsymbol{z} \end{bmatrix} \tag{1.3}$$

which is termed *rotation matrix*. It is worth noting that the column vectors of the matrix $\boldsymbol{R}$ are mutually orthogonal since they represent the unit vectors

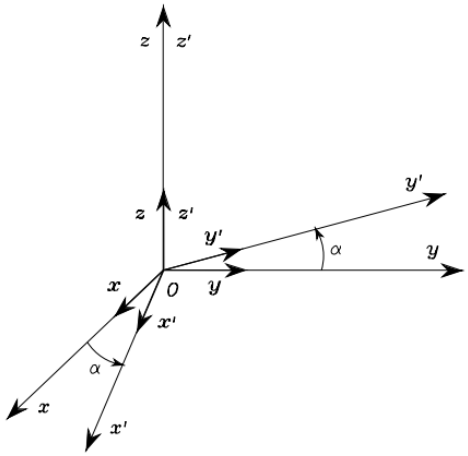


Figure 1.2: Rotation of frame O–xyz by an angle α about axis z

of an orthonormal frame, also they have unit norm. As a consequence, $\boldsymbol{R}$ is an *orthogonal matrix* meaning that:

$$\boldsymbol{R}^T\boldsymbol{R} = \boldsymbol{I} \tag{1.4}$$

Hence:

$$\boldsymbol{R}^T = \boldsymbol{R}^{-1} \tag{1.5}$$

1.1.3 Elementary Rotations

Consider the frames that can be obtained via *elementary rotations* of the reference frame about one of the coordinate axes. These rotations are positive if they are made counter-clockwise about the relative axis. Suppose that the reference frame O–xyz is rotated by an angle α about axis z (Fig. 1.2), and let O–$x'y'z'$ be the rotated frame. The unit vectors of the new frame can be described in terms of their components with respect to the reference frame. Consider the frames that can be obtained via elementary rotations of the reference frame about one of the coordinate axes. These rotations are positive if they are made counter-clockwise about the relative axis. Suppose that the reference frame O–xyz is rotated by an angle α about axis z (Fig. 1.2), and let O–$x'y'z'$ be the rotated frame. The unit vectors of the new frame can be described in terms of its components with respect to

the reference frame, i.e.:

$$\boldsymbol{x'} = \begin{bmatrix} cos(\alpha) \\ sin(\alpha) \\ 0 \end{bmatrix} \quad \boldsymbol{y'} = \begin{bmatrix} -sin(\alpha) \\ cos(\alpha) \\ 0 \end{bmatrix} \quad \boldsymbol{z'} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Hence, the rotation matrix of frame O–$x'y'z'$ with respect to frame O–xyz is

$$\boldsymbol{R}_z(\alpha) = \begin{bmatrix} cos(\alpha) & -sin(\alpha) & 0 \\ sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.6}$$

Similarly, it can be shown that the rotations by an angle β about axis y and by an angle γ about axis x are respectively given by:

$$\boldsymbol{R}_y(\beta) = \begin{bmatrix} cos(\beta) & 0 & sin(\beta) \\ 0 & 1 & 0 \\ -sin(\beta) & 0 & cos(\beta) \end{bmatrix} \tag{1.7}$$

$$\boldsymbol{R}_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\gamma) & -sin(\gamma) \\ 0 & sin(\gamma) & cos(\gamma) \end{bmatrix} \tag{1.8}$$

These matrices will be useful to describe rotations about an arbitrary axis in space. It is easy to verify that for the elementary rotation matrices in (1.6)–(1.8) the following property holds:

$$\boldsymbol{R}_k(-\theta) = \boldsymbol{R}_k^T(\theta) \quad k = x, y, z. \tag{1.9}$$

Namely, the matrix $\boldsymbol{R}$ describes the rotation about an axis in space needed to align the axes of the reference frame with the corresponding axes of the body frame.

1.1.4 Representation of a Vector

To understand a further the zgeometrical meaning of a rotation matrix, consider the case when the origin of the body frame coincides with the origin of the reference frame (Fig. 1.3); it follows that $\boldsymbol{o'} = \mathbf{0}$ (the origin), where $\mathbf{0}$

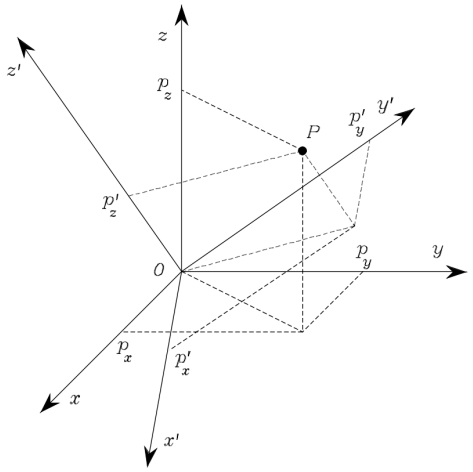


Figure 1.3: Representation of a point P in two different coordinate frames

denotes the (3×1) null vector. A point P in space can be represented either as:

$$\boldsymbol{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

with respect to frame, or as

$$\boldsymbol{p'} = \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix}$$

with respect to frame O–$x'y'z'$. Since $\boldsymbol{p}$ and $\boldsymbol{p'}$ are representations of the same point P, it is:

$$\boldsymbol{p} = p'_x\boldsymbol{x'} + p'_y\boldsymbol{y'} + p'_z\boldsymbol{z'} = \begin{bmatrix} \boldsymbol{x'} & \boldsymbol{y'} & \boldsymbol{z'} \end{bmatrix} \boldsymbol{p'}$$

Therefore:

$$\boldsymbol{p} = \boldsymbol{R}\boldsymbol{p'}$$

The rotation matrix R represents the *transformation matrix* of the vector coordinates in frame O–$x'y'z'$ into the coordinates of the same vector in frame O–xyz. In view of the orthogonality property (1.9), the inverse transformation is simply given by

$$\boldsymbol{p'} = \boldsymbol{R^T}\boldsymbol{p} \tag{1.10}$$

.

A rotation matrix attains *three equivalent geometrical meanings*:

1. It describes the mutual orientation between two coordinate frames; its column vectors are the direction cosines of the axes of the rotated frame with respect to the original frame.

2. It represents the coordinate transformation between the coordinates of a point expressed in two different frames (with common origin).

3. It is the operator that allows the rotation of a vector in the same coordinate frame.

1.1.5 Composition of Rotation Matrices

In order to derive composition rules of rotation matrices, it is useful to consider the expression of a vector in two different reference frames. Let then O–$x_0y_0z_0$, O–$x_1y_1z_1$, O–$x_2y_2z_2$ be three frames with common origin O. The vector $\boldsymbol{p}$ describing the position of a generic point in space can be expressed in each of the above frames; let $\boldsymbol{p^0}$, $\boldsymbol{p^1}$, $\boldsymbol{p^2}$ denote the expressions of $\boldsymbol{p}$ in the three frames. At first, consider the relationship between the expression $\boldsymbol{p^2}$ of the vector $\boldsymbol{p}$ in Frame 2 and the expression $\boldsymbol{p^1}$ of the same vector in Frame 1. If $\boldsymbol{R}_i^j$ denotes the rotation matrix of Frame i with respect to Frame j, it is:

$$\boldsymbol{p}^1 = \boldsymbol{R}_2^1 \boldsymbol{p}^2 \tag{1.11}$$

Similarly, it turns out that:

$$\boldsymbol{p}^0 = \boldsymbol{R}_1^0 \boldsymbol{p}^1 \tag{1.12}$$

$$\boldsymbol{p}^0 = \boldsymbol{R}_2^0 \boldsymbol{p}^2 \tag{1.13}$$

So in conclusion we have that:

$$\boldsymbol{R}_2^0 = \boldsymbol{R}_1^0 \boldsymbol{R}_2^1 \tag{1.14}$$

The relationship in (1.14) can be interpreted as the **composition of successive rotations**. Consider a frame initially aligned with the frame O–$x_0y_0z_0$. The rotation expressed by matrix $\boldsymbol{R}_2^0$ can be regarded as obtained in two steps:

- First rotate the given frame according to $\boldsymbol{R}_1^0$, so as to align it with frame O–$x_1y_1z_1$.
- Then rotate the frame, now aligned with frame O–$x_1y_1z_1$, according to $\boldsymbol{R}_2^1$, so as to align it with frame O–$x_2y_2z_2$.

Notice that the overall rotation can be expressed as a sequence of partial rotations; each rotation is defined with respect to the preceding one. The frame with respect to which the rotation occurs is termed *current frame*. Composition of successive rotations is then obtained by post multiplication of the rotation matrices following the given order of rotations, as in (1.14). With the adopted notation, in view of (1.5), it is:

$$\boldsymbol{R}_i^j = (\boldsymbol{R}_j^i)^{-1} = (\boldsymbol{R}_j^i)^T \tag{1.15}$$

Successive rotations can be also specified by constantly referring them to the initial frame; in this case, the rotations are made with respect to a *fixed frame*. Let $\boldsymbol{R}_1^0$ be the rotation matrix of frame O–$x_1y_1z_1$ with respect to the fixed frame O–$x_0y_0z_0$.

1.1.6 Euler Angles

Rotation matrices give a **redundant description of frame orientation**; in fact, they are characterized by nine elements which are not independent but related by six constraints due to the orthogonality conditions given in (1.4). This implies that *three parameters* are sufficient to describe orientation of a rigid body. in space. A representation of orientation in terms of three independent parameters constitutes a *minimal representation*. A minimal representation of orientation can be obtained by using a set of three angles $\boldsymbol{\phi} = (\varphi, \theta, \psi)$. Consider the rotation matrix expressing the elementary rotation about one of the coordinate axes as a function of a single angle. Then, a generic rotation matrix can be obtained by composing a suitable sequence of three elementary rotations while guaranteeing that two successive rotations are not made about parallel axes: therefore only 12 distinct sets of angles are allowed out of all 27 (3^3) possible combinations; each set represents a triplet of *minimal representation*. In the following, two sets of Euler

angles are analyzed; namely, the ***ZYZ angles*** and the ***Roll–Pitch–Yaw (or ZYX) angles***.

ZYZ Angles

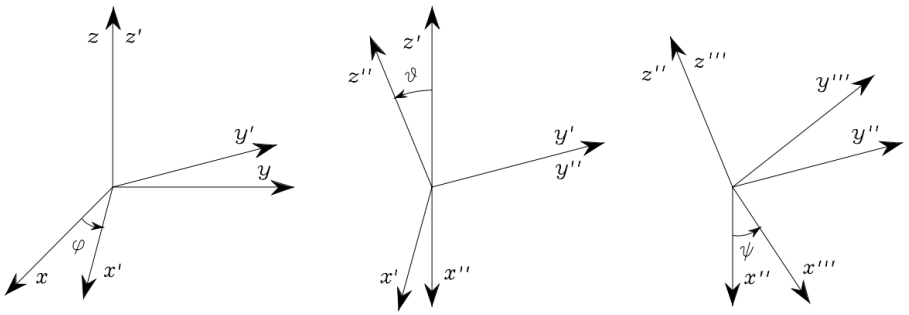


Figure 1.4: Representation of Euler angles ZYZ

The rotation described by ZYZ angles is obtained as composition of the following elementary rotations (Fig. 1.4):

- Rotate the reference frame by the angle φ about axis z; this rotation is described by the matrix $\boldsymbol{R}_z(\varphi)$ which is formally defined in (1.6).
- Rotate the current frame by the angle θ about axis y; this rotation is described by the matrix $\boldsymbol{R}_y(\theta)$ which is formally defined in (1.7).
- • Rotate the current frame by the angle ψ about axis z; this rotation is described by the matrix $\boldsymbol{R}_z(\psi)$ which is again formally defined in (1.6).

The resulting frame orientation is obtained by composition of ***rotations with respect to current frames***, and then it can be computed via post-multiplication of the matrices of elementary rotation, i.e.:

$$\boldsymbol{R}(\boldsymbol{\phi}) = \boldsymbol{R}_z(\varphi)\boldsymbol{R}_{y'}(\theta)\boldsymbol{R}_{z''}(\psi) = \begin{bmatrix} c_\varphi c_\theta c_\psi - s_\varphi s_\psi & -c_\varphi c_\theta s_\psi - s_\varphi c_\psi & c_\varphi s_\theta \\ s_\varphi c_\theta c_\psi + c_\varphi s_\psi & -s_\varphi c_\theta s_\psi + c_\varphi c_\psi & s_\varphi s_\theta \\ -s_\theta c_\psi & s_\theta s_\psi & c_\theta \end{bmatrix} \tag{1.16}$$

It is useful to solve the ***inverse problem***, that is to determine the set of Euler angles corresponding to a given rotation matrix:

$$\boldsymbol{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

Compare this expression with that of $\boldsymbol{R(\phi)}$ in (1.16). By considering the elements [1,3] and [2,3], under the assumption that $r_{13} \neq 0$ and $r_{23} \neq 0$ (otherwise the computation would be impossible), it follows that:

$$\begin{aligned} \boldsymbol{\varphi} &= Atan2(r_{23}, r_{13}) \\ \boldsymbol{\theta} &= Atan2(\sqrt{r_{13}^2 + r_{23}^2}, r_{33}) \\ \boldsymbol{\varphi} &= Atan2(r_{32}, -r_{31}) \end{aligned} \tag{1.17}$$

The choice of the positive sign for the term $\sqrt{r_{13}^2 + r_{23}^2}$ limits the range of feasible values of θ to $(0, \pi)$.
It is possible to derive another solution which produces the same effects as solution (1.17). Choosing θ in the range $(-\pi, 0)$ leads to:

$$\begin{aligned} \boldsymbol{\varphi} &= Atan2(-r_{23}, -r_{13}) \\ \boldsymbol{\theta} &= Atan2(-\sqrt{r_{13}^2 + r_{23}^2}, r_{33}) \\ \boldsymbol{\varphi} &= Atan2(-r_{32}, r_{31}) \end{aligned} \tag{1.18}$$

Solutions (1.17), (1.18) degenerate when $s_\theta = 0$; in this case, it is possible to determine only the sum or difference of φ and ψ. In fact, if $\theta = 0$, π, the successive rotations of φ and ψ are made about axes of current frames which are parallel, thus giving equivalent contributions to the rotation.

R-P-Y Angles

Another set of Euler angles originates from a representation of orientation in the (aero)nautical field. These are the ZYX angles, also called *Roll–Pitch–Yaw angles*, to denote the typical changes of attitude of an (air)craft. In this case, the angles $\boldsymbol{\phi} = [\varphi \quad \theta \quad \psi]^T$ represent ***rotations defined with respect to a fixed frame*** attached to the centre of mass of the craft (Fig. 1.5).

The rotation resulting from Roll–Pitch–Yaw angles can be obtained as follows:

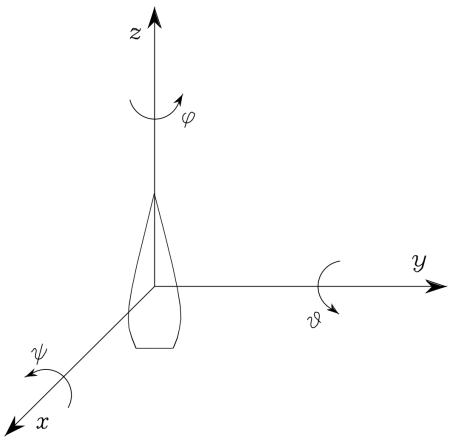


Figure 1.5: Representation of Roll–Pitch–Yaw angles

- Rotate the reference frame by the angle ψ about axis x (yaw); this rotation is described by the matrix $\boldsymbol{R}_x(\psi)$ which is formally defined in (1.8).
- Rotate the reference frame by the angle θ about axis y (pitch); this rotation is described by the matrix $\boldsymbol{R}_y(\theta)$ which is formally defined in (1.7)
- Rotate the reference frame by the angle φ about axis z (roll); this rotation is described by the matrix $\boldsymbol{R}_z(\varphi)$ which is formally defined in (1.6)

The resulting frame orientation is obtained by composition of rotations with respect to the *fixed frame*, and then it can be computed via premultiplication of the matrices of elementary rotation, i.e.:

$$\boldsymbol{R}(\boldsymbol{\phi}) = \boldsymbol{R}_z(\varphi)\boldsymbol{R}_y(\theta)\boldsymbol{R}_x(\psi) = \begin{bmatrix} c_\varphi c_\theta & c_\varphi s_\theta s_\psi - s_\varphi c_\psi & c_\varphi s_\theta c_\psi + s_\varphi s_\psi \\ s_\varphi c_\theta & s_\varphi s_\theta s_\psi + c_\varphi c_\psi & s_\varphi s_\theta c_\psi - c_\varphi s_\psi \\ -s_\theta c & c_\theta s_\psi & c_\theta c_\psi \end{bmatrix} \tag{1.19}$$

As for the Euler angles ZYZ, the *inverse solution* to a given rotation matrix:

$$\boldsymbol{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

can be obtained by comparing it with the expression of $\boldsymbol{R}(\boldsymbol{\phi})$ in (1.19). The solution for θ in the range $(-\pi/2, \pi/2)$ is:

$$\begin{aligned} \boldsymbol{\varphi} &= Atan2(r_{21}, r_{11}) \\ \boldsymbol{\theta} &= Atan2(-r_{31}, \sqrt{r_{32}^2 + r_{33}^2}) \\ \boldsymbol{\varphi} &= Atan2(r_{32}, r_{33}) \end{aligned} \tag{1.20}$$

The other equivalent solution for θ in the range $(\pi/2, 3\pi/2)$ is:

$$\begin{aligned} \boldsymbol{\varphi} &= Atan2(-r_{21}, -r_{11}) \\ \boldsymbol{\theta} &= Atan2(-r_{31}, -\sqrt{r_{32}^2 + r_{33}^2}) \\ \boldsymbol{\varphi} &= Atan2(-r_{32}, -r_{33}) \end{aligned} \tag{1.21}$$

Solutions (1.20), (1.21) degenerate when $c_\theta = 0$; in this case, it is possible to determine only the sum or difference of φ and ψ.

1.1.7 Homogeneous Transformations

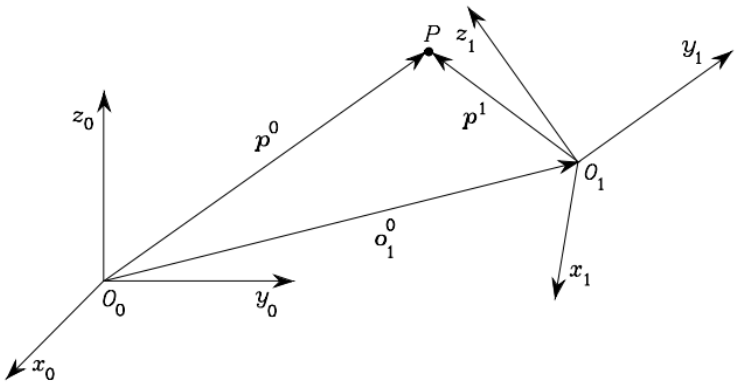


Figure 1.6: Representation of a point P in different coordinate frames

As illustrated at the beginning of the chapter, the position of a rigid body in space is expressed in terms of the **position** of a suitable point on the body with respect to a reference frame (translation), while its **orientation** is expressed in terms of the components of the unit vectors of a frame attached to the body — with origin in the above point — with respect to the same reference frame (rotation). As shown in Fig. (1.6), consider an arbitrary

point P in space. Let $\boldsymbol{p}^0$ be the vector of coordinates of P with respect to the reference frame O_0–$x_0y_0z_0$. Consider then another frame in space O_1–$x_1y_1z_1$. Let $\boldsymbol{o}_1^0$ be the vector describing the origin of Frame 1 with respect to Frame 0, and $\boldsymbol{R}_1^0$ be the rotation matrix of Frame 1 with respect to Frame 0. Let also $\boldsymbol{p}^1$ be the vector of coordinates of P with respect to Frame 1. On the basis of simple geometry, the position of point P with respect to the reference frame can be expressed as:

$$\boldsymbol{p}^0 = \boldsymbol{o}_1^0 + \boldsymbol{R}_1^0 \boldsymbol{p}^1 \tag{1.22}$$

Hence, (1.22) represents the *coordinate transformation* (*translation + rotation*) of a bound vector between two frames. The inverse transformation can be obtained by premultiplying both sides of (1.22) by $\boldsymbol{R}_1^{0T}$ (or $-\boldsymbol{R}_0^1$) ; in view of (1.4), it follows that

$$\boldsymbol{p}^1 = -\boldsymbol{R}_0^1 \boldsymbol{o}_1^0 + \boldsymbol{R}_0^1 \boldsymbol{p}^0 \tag{1.23}$$

In order to achieve a compact representation of the relationship between the coordinates of the same point in two different frames, the ***homogeneous representation*** of a generic vector $\boldsymbol{p}$ can be introduced as the vector $\tilde{\boldsymbol{p}}$ formed by adding a fourth unit component, i.e.:

$$\tilde{\boldsymbol{p}} = \begin{bmatrix} \boldsymbol{p} \\ 1 \end{bmatrix} \tag{1.24}$$

By adopting this representation for the vectors $\boldsymbol{p}^0$ and $\boldsymbol{p}^1$ in (1.22), the coordinate transformation can be written in terms of the (4×4) matrix:

$$\boldsymbol{A}_1^0 = \begin{bmatrix} \boldsymbol{R}_1^0 & \boldsymbol{o}_1^0 \\ \boldsymbol{0}^T & 1 \end{bmatrix} \tag{1.25}$$

which, according to (1.24), is the ***homogeneous transformation matrix***. As can be easily seen from (1.25), the transformation of a vector from Frame 1 to Frame 0 is expressed by a single matrix containing the rotation matrix of Frame 1 with respect to Frame 0 and the translation vector from the origin of Frame 0 to the origin of Frame 1. Therefore, the coordinate transformation (1.22) can be compactly rewritten as:

$$\tilde{\boldsymbol{p}}^0 = \boldsymbol{A}_1^0 \tilde{\boldsymbol{p}}^{\boldsymbol{1}} \tag{1.26}$$

Notice that for the homogeneous transformation matrix **the orthogonality property does not hold**; hence, in general:

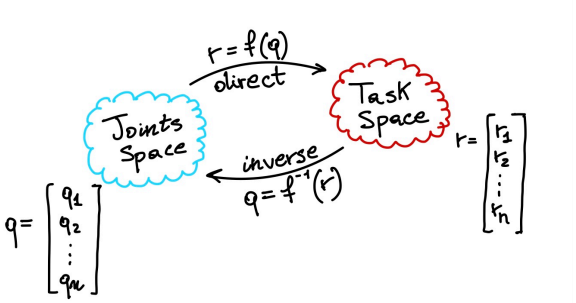
$$\boldsymbol{A}^{-1} \neq \boldsymbol{A}^{T} \tag{1.27}$$

In sum, a homogeneous transformation matrix expresses the coordinate transformation between two frames in a compact form. If the frames have the same origin, it reduces to the rotation matrix previously defined. Instead, if the frames have distinct origins, it allows the notation with superscripts and subscripts to be kept which directly characterize the current frame and the fixed frame. Analogously to what is presented for the rotation matrices, it is easy to verify that a sequence of coordinate transformations can be composed by the product:

$$\tilde{\boldsymbol{p}}^{0} = \boldsymbol{A}_{1}^{0}\boldsymbol{A}_{2}^{1}...\boldsymbol{A}_{n}^{n-1}\tilde{\boldsymbol{p}}^{n} \tag{1.28}$$

where $\boldsymbol{A}_{i}^{i-1}$ denotes the homogeneous transformation relating the description of a point in Frame i to the description of the same point in Frame $i-1$.

1.2 Direct Kinematics



A manipulator consists of a series of rigid bodies (*links*) connected by means of kinematic pairs or *joints*. Joints can be essentially of two types: *revolute* and *prismatic*. The whole structure forms a *kinematic chain*. One end of the chain is constrained to a base. An *end-effector* (gripper, tool) is connected to the other end allowing manipulation of objects in space. From a topological viewpoint, the kinematic chain is termed *open* when there is only one sequence of links connecting the two ends of the chain. Alternatively, a manipulator contains a *closed* kinematic chain when a sequence of links

forms a loop. The mechanical structure of a manipulator is characterized by a number of degrees of freedom (DOFs) which uniquely determine its *posture*. Each DOF is typically associated with a joint articulation and constitutes a *joint variable*. **The aim of *direct kinematics* is to compute the pose of the end-effector as a function of the joint variables**. with respect

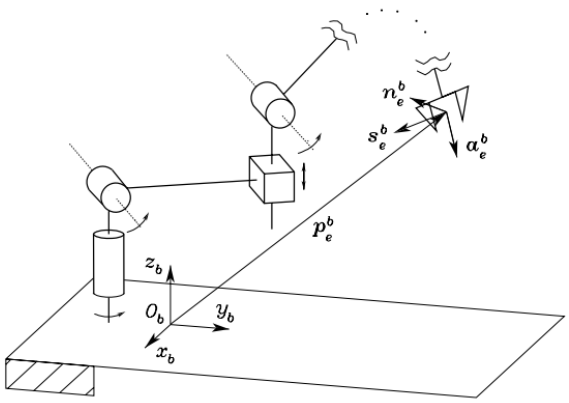


Figure 1.7: Description of the position and orientation of the end-effector frame

to a reference frame O_b–$x_b y_b z_b$, the direct kinematics function is expressed by the homogeneous transformation matrix:

$$\boldsymbol{T}_e^b(\boldsymbol{q}) = \begin{bmatrix} \boldsymbol{n}_e^b(\boldsymbol{q}) & \boldsymbol{s}_e^b(\boldsymbol{q}) & \boldsymbol{a}_e^b(\boldsymbol{q}) & \boldsymbol{p}_e^b(\boldsymbol{q}) \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.29}$$

where $\boldsymbol{q}$ is the $(n \times 1)$ vector of joint variables, $\boldsymbol{n}_e$, $\boldsymbol{s}_e$, $\boldsymbol{a}_e$ are the unit vectors of a frame attached to the end-effector, and $\boldsymbol{p}_e$ is the position vector of the origin of such a frame with respect to the origin of the base frame O_b–$x_b y_b z_b$ (Fig. 1.7). Note that $\boldsymbol{n}_e$, $\boldsymbol{s}_e$, $\boldsymbol{s}_e$ and $\boldsymbol{p}_e$ are a function of $\boldsymbol{q}$. The frame O_b–$x_b y_b z_b$ is termed *base frame*. The frame attached to the end-effector is termed *end-effector frame* and is conveniently chosen according to the particular task geometry. If the end-effector is a gripper, the origin of the end-effector frame is located at the centre of the gripper, the unit vector $\boldsymbol{a}_e$ is chosen in the *approach* direction to the object, the unit vector $\boldsymbol{s}_e$ is chosen normal to $\boldsymbol{a}_e$ in the *sliding* plane of the jaws, and the unit vector $\boldsymbol{n}_e$ is chosen normal to the other two so that the frame ($\boldsymbol{n}_e$, $\boldsymbol{s}_e$, $\boldsymbol{a}_e$) is right-handed. A first way to compute direct kinematics is offered by a geometric analysis of the structure of the given manipulator.

1.2.1 Open Chain Manipulator

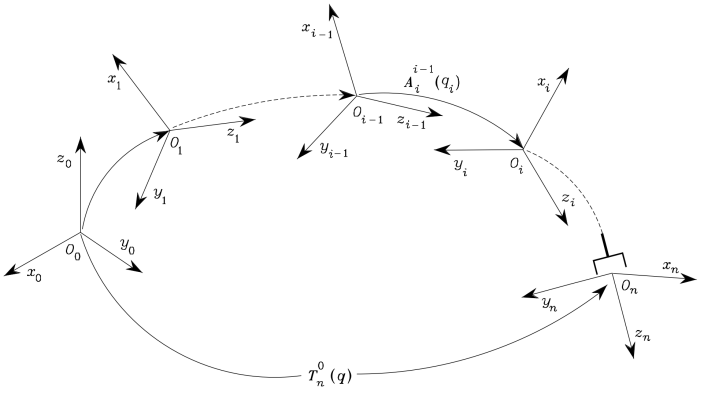


Figure 1.8: Coordinate transformations in an open kinematic chain

Consider an *open-chain* manipulator constituted by $n+1$ links connected by n joints, where Link 0 is conventionally fixed to the ground. It is assumed that each joint provides the mechanical structure with a single DOF, corresponding to the joint variable. It is reasonable to consider first the description of kinematic relationship between consecutive links and then to obtain the overall description of manipulator kinematics in a recursive fashion. To this purpose, it is worth defining a coordinate frame attached to each link, from Link 0 to Link n. Then, the coordinate transformation describing the position and orientation of Frame n with respect to Frame 0 (Fig. 1.8) is given by

$$\boldsymbol{T}_n^0(\boldsymbol{q}) = \boldsymbol{A}_1^0(q_1)\boldsymbol{A}_2^1(q_2)...\boldsymbol{A}_n^{n-1}(q_n) \tag{1.30}$$

As requested, the computation of the direct kinematics function is recursive and is obtained in a systematic manner by simple products of the homogeneous transformation matrices $\boldsymbol{A}_i^{i-1}(q_i)$ (for i = 1,...,n), each of which is a function of a single joint variable q_i, that contains information about position and orientation about the i-th link's frame. The actual coordinate transformation describing the position and orientation of the end-effector frame with respect to the base frame can be obtained as

$$\boldsymbol{T}_e^b(\boldsymbol{q}) = \boldsymbol{T}_0^b\boldsymbol{T}_n^0(\boldsymbol{q})\boldsymbol{T}_e^n \tag{1.31}$$

where $\boldsymbol{T}_0^b$ and $\boldsymbol{T}_e^n$ are two (typically) constant homogeneous transformations

describing the position and orientation of Frame 0 with respect to the base frame, and of the end-effector frame with respect to Frame n, respectively.

1.2.2 Denavit-Hartenberg Convention

In order to compute the direct kinematics equation for an open-chain manipulator according to the recursive expression in (1.30), a systematic, **general method is to be derived to define the relative position and orientation of two consecutive links**; the problem is that to determine two frames attached to the two links and compute the coordinate transformations between them. In general, the *frames can be arbitrarily chosen* (z is fixed but x and y can vary) as long as they are attached to the link they are referred to. Nevertheless, it is convenient to set some rules also for the definition of the link frames.

With reference to Fig. 1.9, let Axis i denote the axis of the joint connecting

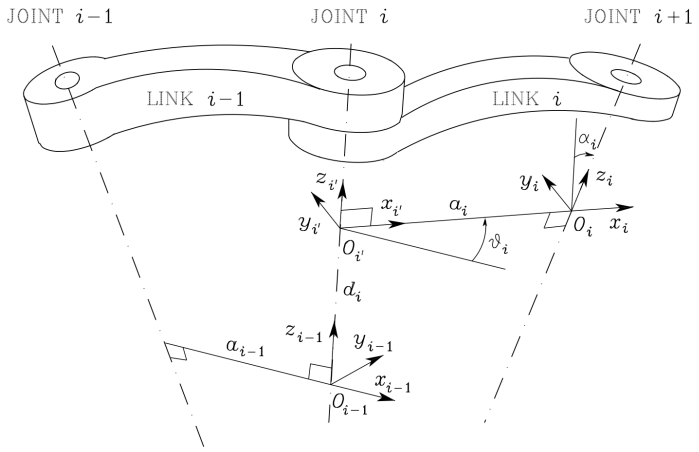


Figure 1.9: Denavit–Hartenberg kinematic parameters

Link $i-1$ to Link i; the so-called *Denavit–Hartenberg convention* (DH) is adopted to define link Frame i:

- Choose axis z_i along the axis of Joint $i+1$
- Locate the origin O_i at the intersection of axis z_i with the common normal to axes z_{i-1} and z_i ($x_{i'}$, the other third axes that is not oriented as z_i and $z_{i'}$). Also, locate $O_{i'}$ at the intersection of the common normal with axis z_{i-1}

- Choose axis x_i along the common normal to axes z_{i-1} and z_i with direction from Joint i to Joint $i+1$.
- Choose axis y_i so as to complete a right-handed frame

The Denavit–Hartenberg convention gives a nonunique definition of the link frame in the following cases:

- For Frame 0, only the direction of axis z_0 is specified; then O_0 and x_0 can be arbitrarily chosen.
- For Frame n, since there is no Joint $n+1$, z_n is not uniquely defined while x_n has to be normal to axis z_{n-1}. Typically, Joint n is revolute, and thus z_n is to be aligned with the direction of z_{n-1}.
- When two consecutive axes are parallel, the common normal between them is not uniquely defined.
- When two consecutive axes intersect, the direction of x_i is arbitrary.
- When Joint i is prismatic, the direction of z_{i-1} is arbitrary.

In all such cases, the indeterminacy can be exploited to simplify the procedure; for instance, the axes of consecutive frames can be made parallel. Once the link frames have been established, the position and orientation of Frame i with respect to Frame $i-1$ are completely specified by the following ***parameters***:

- a_i, that is the distance between O_i and $O_{i'}$
- d_i coordinate of O_i along z_{i-1}
- α_i angle between axes z_{i-1} and z_i about axis x_i to be taken positive when rotation is made counter-clockwise
- θ_i angle between axes x_{i-1} and x_i about axis z_{i-1} to be taken positive when rotation is made counter-clockwise

Two of the four parameters (a_i and α_i) are always constant and depend only on the geometry of connection between consecutive joints established by Link i. Of the remaining two parameters, only one is variable depending on the type of joint that connects Link $i-1$ to Link i. In particular:

- if Joint i is *revolute* the variable is θ_i
- if Joint i is *prismatic* the variable is d_i

At this point, it is possible to express the coordinate transformation between Frame i and Frame $i-1$ according to the following steps:

- Choose a frame aligned with Frame $i-1$.
- Translate the chosen frame by d_i along axis z_{i-1} and rotate it by θ_i about axis z_{i-1}; this sequence aligns the current frame with Frame i and is described by the homogeneous transformation matrix

$$\boldsymbol{A}_{i'}^{i-1} = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i} & 0 & 0 \\ s_{\theta_i} & c_{\theta_i} & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Translate the frame aligned with Frame i by a_i along axis x_i and rotate it by α_i about axis x_i; this sequence aligns the current frame with Frame i and is described by the homogeneous transformation matrix

$$\boldsymbol{A}_{i}^{i'} = \begin{bmatrix} 1 & 0 & 0 & \alpha_i \\ 0 & c_{\alpha_i} & -s_{\alpha_i} & 0 \\ 0 & s_{\alpha_i} & c_{\alpha_i} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- The resulting coordinate transformation is obtained by postmultiplication of the single transformations as

$$\boldsymbol{A}_{i}^{i-1}(q_i) = \boldsymbol{A}_{i'}^{i-1}\boldsymbol{A}_{i}^{i'} = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & \alpha_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & \alpha_i s\theta_i \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.32}$$

Notice that the transformation matrix from Frame i to Frame $i-1$ is a function only of the joint variable q_i, that is, θ_i for a revolute joint or d_i for a prismatic joint. To summarize, the Denavit–Hartenberg convention allows the construction of the direct kinematics function by composition of

the individual coordinate transformations expressed by (1.32) into one homogeneous transformation matrix as in (1.30). **The procedure can be applied to any open kinematic chain and can be easily rewritten in an operating form as follows**.

1. Find and number consecutively the joint axes; set the directions of axes $z_0, ..., z_{n-1}$.

2. Choose Frame 0 by locating the origin on axis z_0; axes x_0 and y_0 are chosen so as to obtain a right-handed frame. If feasible, it is worth choosing Frame 0 to coincide with the base frame.

Execute steps from **3** to **5** for $i = 1, ..., n-1$:

3. Locate the origin O_i at the intersection of z_i with the common normal to axes z_{i-1} and z_i. If axes z_{i-1} and z_i are parallel and Joint i is revolute, then locate O_i so that $d_i = 0$; if Joint i is prismatic, locate O_i at a reference position for the joint range, e.g., a mechanical limit.

4. Choose axis x_i along the common normal to axes z_{i-1} and z_i with direction from Joint i to Joint $i+1$

5. Choose axis y_i so as to obtain a right-handed frame

To complete:

6. Choose Frame n; if Joint n is revolute, then align z_n with z_{n-1}, otherwise, if Joint n is prismatic, then choose z_n arbitrarily. Axis x_n is set according to step **4**

7. For $i = 1, ..., n$, form the table of parameters a_i, d_i, α_i, θ_i.

8. On the basis of the parameters in **7**, compute the homogeneous transformation matrices $A_i^{i-1}(q_i)$ for $i = 1, ..., n$.

9. Compute the homogeneous transformation $\boldsymbol{T}_n^0(q) = \boldsymbol{A}_1^0 ... \boldsymbol{A}_n^{n-1}$ that yields the position and orientation of Frame n with respect to Frame 0.

10. Given $\boldsymbol{T}_0^b$ and $\boldsymbol{T}_e^n$, compute the direct kinematics function as $\boldsymbol{T}_e^b(q) = \boldsymbol{T}_0^b \boldsymbol{T}_n^0 \boldsymbol{T}_e^n$ that yields the position and orientation of the end-effector frame with respect to the base frame.

1.3 Joint Space and Operational (or Task) Space

The direct kinematics equation of a manipulator allows the position and orientation of the end-effector frame to be expressed as a function of the joint variables with respect to the base frame. If a task is to be specified for the end-effector, it is necessary to assign the end-effector position and orientation, eventually as a function of time (trajectory). This is quite easy for the position. On the other hand, specifying the orientation through the unit vector triplet ($\boldsymbol{n}_e$, $\boldsymbol{s}_e$, $\boldsymbol{a}_e$) is quite difficult, since their nine components must be guaranteed to satisfy the orthonormality constraints at each time instant. The problem of describing end-effector orientation admits a natural solution if one of the above minimal representations is adopted. In this case, indeed, a motion trajectory can be assigned to the set of angles chosen to represent orientation. Therefore, the position can be given by a minimal number of coordinates with regard to the geometry of the structure, and the orientation can be specified in terms of a minimal representation (Euler angles) describing the rotation of the end-effector frame with respect to the base frame. In this way, it is possible to describe the end-effector pose by means of the $(m \times 1)$ vector, with $m \leq n$,

$$\boldsymbol{x}_e = \begin{bmatrix} \boldsymbol{p}_e \\ \boldsymbol{\phi}_e \end{bmatrix} \tag{1.33}$$

where $\boldsymbol{p}_e$ describes the end-effector position and $\boldsymbol{\phi}_e$ its orientation. This representation of position and orientation allows the description of an end-effector task in terms of a number of inherently independent parameters. The vector $\boldsymbol{x}_e$ is defined in the space in which the manipulator task is specified; hence, this space is typically called ***operational space***.
On the other hand, the ***joint space*** (configuration space) denotes the space in which the $(n \times 1)$ vector of joint variables

$$\boldsymbol{q} = \begin{bmatrix} q_1 \\ \vdots \\ q_n \end{bmatrix} \tag{1.34}$$

is defined; it is $q_i = \theta_i$ for a revolute joint and $q_i = d_i$ for a prismatic joint. Accounting for the dependence of position and orientation on the joint variables, the direct kinematics equation can be written in a form other than (1.30), i.e.,

$$\boldsymbol{x}_e = \boldsymbol{k}(\boldsymbol{q}) \tag{1.35}$$

The $(m \times 1)$ vector function $\boldsymbol{k}(\cdot)$ — nonlinear in general — allows computation of the operational space variables from the knowledge of the joint space variables. It is worth noticing that the dependence of the orientation components of the function $\boldsymbol{k}(\boldsymbol{q})$ in (1.35) on the joint variables is not easy to express except for simple cases. In fact, in the most general case of a six-dimensional operational space ($m = 6$, for position and orientation), the computation of the three components of the function $\boldsymbol{\phi}_e(\boldsymbol{q})$ cannot be performed in closed form but goes through the computation of the elements of the rotation matrix, i.e., $\boldsymbol{n}_e(\boldsymbol{q})$, $\boldsymbol{s}_e(\boldsymbol{q})$, $\boldsymbol{a}_e(\boldsymbol{q})$.

1.4 Trajectory Planning

The goal of *trajectory planning* is to generate the reference inputs to the motion control system which ensures that the manipulator executes the planned trajectories. The user typically specifies a number of parameters to describe the desired trajectory. Planning consists of generating a time sequence of the values attained by an interpolating function (typically a polynomial) of the desired trajectory.

Definition 1.4.1 (**Path**)**.** Locus of the points in the joint or operational space which the manipulator has to follow in the execution of an assigned motion. A Path is a pure geometric description of the motion.

Definition 1.4.2 (**Trajectory**)**.** It is a path on which a timing law is specified (e.g. velocities or acceleration at each point).

1.4.1 Joint-Space Trajectory

A manipulator motion is typically assigned in the operational space in terms of trajectory parameters such as the **initial and final end-effector**

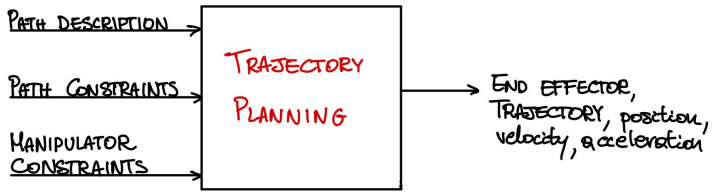


Figure 1.10

pose, **possible intermediate poses**, and **travelling time** along particular geometric paths. If it is desired to plan a trajectory in the *joint space*, the values of the joint variables have to be determined first from the end-effector position and orientation specified by the user. It is then necessary to resort to an inverse kinematics algorithm, if planning is done offline, or to directly measure the above variables if planning is done by the teaching-by-showing technique.

The planning algorithm generates a function $\boldsymbol{q}(t)$ interpolating the given vectors of joint variables at each point, with respect to the imposed constraints. In general, a joint space trajectory planning algorithm is required to have the following features:

- The generated trajectories should be not very demanding from a computational viewpoint
- the joint positions and velocities should be continuous functions of time (Continuity of accelerations may be imposed, too)
- undesirable effects should be minimized, e.g., non-smooth trajectories interpolating a sequence of points on a path.

Point-to-Point Motion

In *point-to-point motion*, **the manipulator has to move from an initial to a final joint configuration in a given time** t_f. In this case, the actual end-effector path is of no concern. The algorithm should generate a

trajectory that, with respect to the above general requirements, is also capable of optimizing some performance index when the joint is moved from one position to another. A suggestion for choosing the motion primitive may stem from the analysis of an incremental motion problem. Let I be the moment of inertia of a rigid body about its rotation axis. It is required to take the angle q from an initial value q_i to a final value q_f in a time t_f. It is evident that infinite solutions exist to this problem. Assumed that rotation is executed through a torque τ supplied by a motor, a solution can be found that minimizes the energy dissipated in the motor. This **optimization problem** can be formalized as follows. Having set $\dot{q} = \omega$, determine the solution to the differential equation:

$$I\dot{\omega} = \tau$$

subject to condition:

$$\int_0^{t_f} \omega(t)dt = q_f - q_i$$

so as to minimize the performance index (energy consumption):

$$\int_0^{t_f} \tau^2(t)dt$$

It can be shown that the resulting solution is of the type

$$w(t) = at^2 + bt + c$$

Even though the joint dynamics cannot be described in the above simple manner, the choice of a third-order polynomial function to generate a joint trajectory represents a valid solution for the problem at issue. Therefore, to determine a joint motion, the *cubic polynomial*

$$q(t) = a_3t^3 + a_2t^2 + a_1t + a_0 \tag{1.36}$$

can be chosen, resulting into a parabolic velocity profile

$$\dot{q}(t) = 3a_3t^2 + 2a_2t + a_1$$

and a linear acceleration profile

$$\ddot{q}(t) = 6a_3t + 2a_2$$

Since four coefficients are available, it is possible to impose, besides the initial and final joint position values q_i and q_f, also the initial and final joint velocity values $\dot{q}_i$ and $\dot{q}_f$ which are usually set to zero. Determination of a specific trajectory is given by the solution to the following system of equations:

$$\begin{aligned} a_0 &= q_i \\ a_1 &= \dot{q}_i \\ a_3 t_f^3 + a_2 t_f^2 + a_1 t_f + a_0 &= q_f \\ 3a_3 t_f^2 + 2a_2 t_f + a_1 &= \dot{q}_f \end{aligned}$$

that allows the computation of the coefficients of the polynomial in (1.36). Figure 1.11 illustrates the timing law obtained with the following data: $q_i = 0$, $q_f = \pi$, $t_f = 1$, and $q_i = q_f = 0$. As anticipated, velocity has a parabolic profile, while acceleration has a linear profile with initial and final discontinuity.

If it is desired to assign the initial and final values of acceleration, six constraints have to be satisfied and then a polynomial of at least fifth order is needed. The motion timing law for the generic joint is then given by

$$q(t) = a_5 t^5 + a_4 t^4 + a_3 t^3 + a_2 t^2 + a_1 t + a_0 \tag{1.37}$$

whose coefficients can be computed, as for the previous case, by imposing the conditions for $t = 0$ and $t = t_f$ on the joint variable $q(t)$ and on its first two derivatives. With the choice (1.37), one obviously gives up minimizing the above performance index.

An alternative approach with timing laws of blended polynomial type is frequently adopted in industrial practice, which *allows a direct verification of whether the resulting velocities and accelerations can be supported by the physical mechanical manipulator.*

In this case, a **trapezoidal velocity profile** is assigned, which imposes a constant acceleration in the start phase, a cruise velocity, and a constant deceleration in the arrival phase. The resulting trajectory is formed by a linear segment connected by two parabolic segments to the initial and final positions.

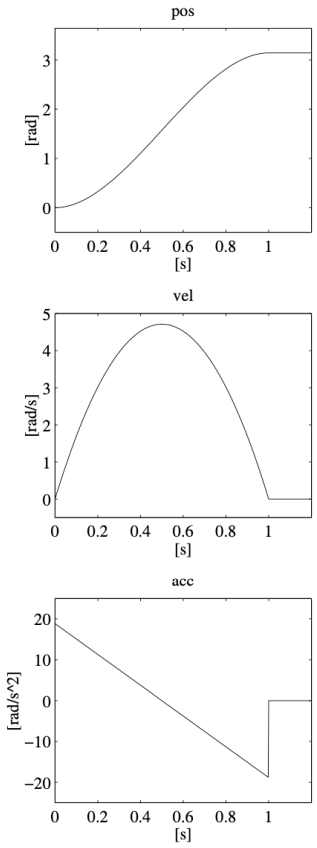


Figure 1.11: Time history of position, velocity, and acceleration with a cubic polynomial timing law

In the following, the problem is formulated by assuming that the final time of trajectory duration has been assigned. However, in industrial practice, the user is offered the option to specify the velocity percentage with respect to the maximum allowable velocity; this choice is aimed at avoiding occurrences when the specification of a much too short motion duration would involve much too large values of velocities and/or accelerations, beyond those achievable by the manipulator. As can be seen from the velocity profiles in Fig. 1.12, it is assumed that both initial and final velocities are null and the segments with constant accelerations have the same time duration; this implies an equal magnitude $\ddot{q}_c$ in the two segments. Notice also that the above choice leads to a symmetric trajectory with respect to the average point $q_m = (q_f + q_i)/2$ at $t_m = t_f/2$.

The trajectory has to satisfy some constraints to ensure the transition

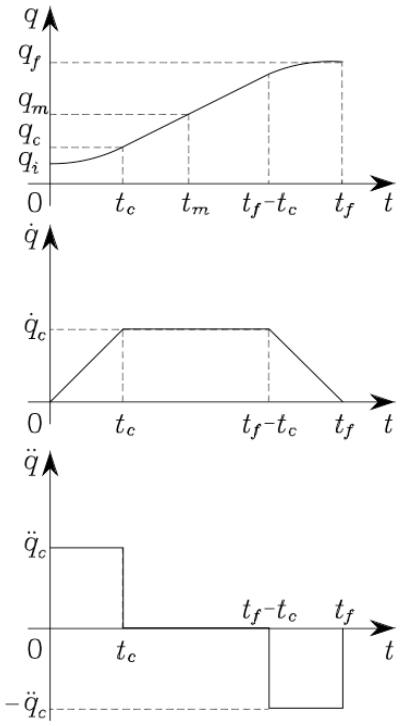


Figure 1.12: Characterization of a timing law with trapezoidal velocity profile in terms of position, velocity and acceleration

from q_i to q_f in a time t_f. The velocity at the end of the parabolic segment must be equal to the (constant) velocity of the linear segment, i.e. we must impose:

$$\ddot{q}_c t_c = \frac{q_m - q_c}{t_m - t_c} \tag{1.38}$$

where q_c is the value attained by the joint variable at the end of the parabolic segment at time t_c with constant acceleration $\ddot{q}_c$ (recall that $\dot{q}(0) = 0$). It is then:

$$q_c = q_i + \frac{1}{2}\ddot{q}_c t_c^2 \tag{1.39}$$

Combining (1.38), (1.39) gives

$$\ddot{q}_c t_c^2 - \ddot{q}_c t_f t_c + q_f - q_i = 0 \tag{1.40}$$

Usually, $\ddot{q}_c$ is specified with the constraint that $sgn(\ddot{q}_c) = sgn(q_f - q_i)$; hence for given t_f, q_i and q_f, the solution for t_c is computed from (1.40) as ($t_c \leq$

$t_f/2$)

$$t_c = \frac{t_f}{2} - \frac{1}{2}\sqrt{\frac{t_f^2\ddot{q}_c - 4(q_f - q_i)}{\ddot{q}_c}} \tag{1.41}$$

Therefore acceleration is subject to the constraint (the discriminant must be ≥ 0):

$$|\ddot{q}_c| \geq \frac{4|q_f - q_i|}{t_f^2} \tag{1.42}$$

When the acceleration $\ddot{q}_c$ is chosen so as to satisfy (1.42) with the equality sign, the resulting trajectory does not feature the constant velocity segment any more and has only the acceleration and deceleration segments (triangular profile).

Given q_i, q_f and t_f, and thus also an average transition velocity, the constraint in (1.42) allows the imposition of a value of acceleration consistent with the trajectory. Then, t_c is computed from (1.41), and the following sequence of polynomials is generated:

$$q(t) = \begin{cases} q_i + \frac{1}{2}\ddot{q}_c t^2 & 0 \leq t \leq t_c \\ q_i + \ddot{q}_c t_c (t - \frac{t_c}{2}) & t_c < t \leq t_f - t_c \\ q_f - \frac{1}{2}\ddot{q}_c (t_f - t)^2 & t_f - t_c < t \leq t_f \end{cases} \tag{1.43}$$

Figure 1.13 illustrates a representation of the motion timing law obtained by imposing the data: $q = 0$, $q = \pi$, $t = 1$,and $|\ddot{q}| = 6\pi$. Specifying acceleration in the parabolic segment is not the only way to determine trajectories with trapezoidal velocity profile. Besides q_i, q_f and t_f, one can specify also the cruise velocity $\dot{q}_c$ which is subject to the constraint:

$$\frac{|q_f - q_i|}{t_f} < |\dot{q}_c| \leq \frac{2|q_f - q_i|}{t_f} \tag{1.44}$$

By recognizing that $\dot{q}_c = \ddot{q}_c t_c$, (1.40) allows the computation of t_c as:

$$t_c = \frac{q_i - q_f + \dot{q}_c t_f}{\dot{q}_c} \tag{1.45}$$

and thus the resulting acceleration is:

$$\ddot{q}_c = \frac{\dot{q}_c^2}{q_i - q_f + \dot{q}_c t_f} \tag{1.46}$$

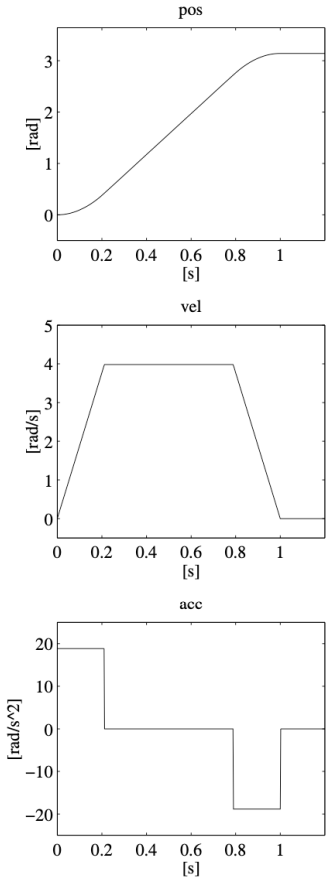


Figure 1.13: Time history of position, velocity and acceleration with a trapezoidal velocity profile timing law

The computed values of t_c and $\ddot{q}_c$ as in (1.45), (1.46) allow the generation of the sequence of polynomials expressed by (1.43). The adoption of a trapezoidal velocity profile results in a worse performance index compared to the cubic polynomial. The decrease is, however, limited; the term $\int_0^{t_f} \tau^2 d\tau$ increases by 12.5% with respect to the optimal case.

Motion Through a Sequence of Points

In several applications, the path is described in terms of a number of points greater than two. For instance, even for the simple point-to-point motion of a pick-and-place task, it may be worth assigning two intermediate points between the initial point and the final point; suitable positions can be set for lifting off and setting down the object, so that reduced velocities

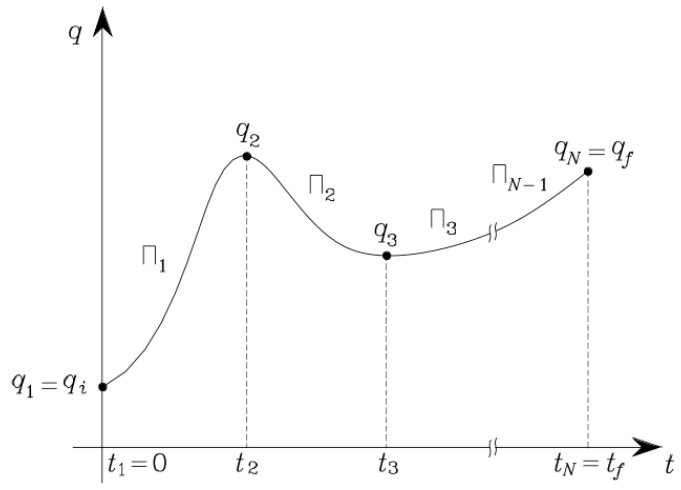


Figure 1.14: Characterization of a trajectory on a given path obtained through interpolating polynomials

are obtained with respect to direct transfer of the object. For more complex applications, it may be convenient to assign a *sequence of points* so as to guarantee better monitoring on the executed trajectories; the points are to be specified more densely in those segments of the path where obstacles have to be avoided or a high path curvature is expected.

It should not be forgotten that **the corresponding joint variables have to be computed from the operational (task) space poses**. Therefore, the problem is to generate a trajectory when N points, termed *path points*, are specified and have to be reached by the manipulator at certain instants of time. For each joint variable there are N constraints, and then one might want to use an $(N-1)$ order polynomial. This choice, however, has the following disadvantages:

- It is not possible to assign the initial and final velocities.
- As the order of a polynomial increases, its oscillatory behaviour increases, and this may lead to trajectories which are not natural for the manipulator.
- Numerical accuracy for computation of polynomial coefficients decreases as order increases.

- The resulting system of constraint equations is heavy to solve.
- Polynomial coefficients depend on all the assigned points; thus, if it is desired to change a point, all of them have to be recomputed.

These drawbacks can be overcome if a suitable number of low-order interpolating polynomials, continuous at the path points, are considered instead of a single high-order polynomial. According to the previous section, the interpolating polynomial of lowest order is the *cubic polynomial*, since it allows the imposition of continuity of velocities at the path points.
With reference to the single joint variable, a function $q(t)$ is sought, formed by a sequence of $N-1$ cubic polynomials $\Pi_k(t)$, for $k = 1, ..., N-1$, continuous with continuous first derivatives. The function $q(t)$ reaches the values q_k for $t = t_k$ $(k = 1, ..., N)$, and $q_1 = q_i$, $t_1 = 0$, $q_N = q_f$, $t_N = t_f$; the q_k 's represent the path points describing the desired trajectory at $t = t_k$ (Fig. 1.14). The following situations can be considered:

- Arbitrary values of $\dot{q}(t)$ are imposed at the path points.
- The values of $\dot{q}(t)$ at the path points are assigned according to a certain criterion.
- The acceleration $\ddot{q}(t)$ has to be continuous at the path points.

To simplify the problem, it is also possible to find interpolating polynomials of order less than three which determine trajectories passing nearby the path points at the given instants of time. The following situations can happen:

- **Interpolating polynomials with imposed velocities at path points**: This solution requires the user to be able to specify the desired velocity at each path point. The system of equations allowing computation of the coefficients of the $N-1$ cubic polynomials interpolating the N path points is obtained by imposing the following conditions on the generic polynomial $\Pi_k(t)$ interpolating q_k and q_{k+1}, for $k = 1, ..., N-1$:

$$\Pi_k(t_k) = q_k$$
$$\Pi_k(t_{k+1}) = q_{k+1}$$
$$\dot{\Pi}_k(t_k) = \dot{q}_k$$
$$\dot{\Pi}_k(t_{k+1}) = \dot{q}_{k+1}$$

The result is $N-1$ systems of four equations in the four unknown coefficients of the generic polynomial; these can be solved one independently of the other. The initial and final velocities of the trajectory are typically set to zero ($\dot{q}_1 = \dot{q}_N = 0$) and continuity of velocity at the path points is ensured by setting:

$$\dot{\Pi}_k(t_{k+1}) = \dot{\Pi}_{k+1}(t_{k+1})$$

- **Interpolating polynomials with computed velocities at path points** In this case, the joint velocity at a path point has to be computed according to a certain criterion. By interpolating the path points with linear segments, the relative velocities can be computed according to the following rules:

$$\dot{q}_1 = 0$$

$$\dot{q}_k = \begin{cases} 0 & sgn(v_k) \neq sgn(v_{k+1}) \\ \frac{1}{2}(v_k + v_{k+1}) & sgn(v_k) = sgn(v_{k+1}) \end{cases}$$

$$\dot{q}_N = 0$$

where

$$v_k = \frac{q_k - q_{k-1}}{t_k - t_{k-1}}$$

and it gives the slope of the segment in the time interval $[t_{k-1}, t_k]$.
With the above settings, the determination of the interpolating polynomials is reduced to the previous case.

- **Interpolating polynomials with continuous accelerations at path points (Splines)**
Both the above two solutions do not ensure continuity of accelerations at the path points. Given a sequence of N path points, the acceleration is also continuous at each t_k if four constraints are imposed, namely, two position constraints for each of the adjacent cubics and two constraints guaranteeing continuity of velocity and acceleration.

The following equations have then to be satisfied:

$$\begin{aligned}
\Pi_{k-1}(t_k) &= q_k \\
\Pi_{k-1}(t_k) &= \Pi_k(t_k) \\
\dot{\Pi}_{k-1}(t_k) &= \dot{\Pi}_k(t_k) \\
\ddot{\Pi}_{k-1}(t_k) &= \ddot{\Pi}_k(t_k)
\end{aligned}$$

Having a control of the acceleration while the robot reaches a certain position it's really important, for instance in collaborative robotics where there's person close to the robot.
Once the spline coefficients are computed, evaluating the trajectory at different points is computationally efficient, making real-time execution feasible. This is important for robotics applications where trajectory planning needs to be performed quickly to respond to changing environments.

1.4.2 Operational Space Trajectory

A joint space trajectory planning algorithm generates a time sequence of values for the joint $\boldsymbol{q}(t)$ so that the manipulator is taken from the initial to the final configuration, eventually by moving through a sequence of intermediate configurations. The resulting end-effector motion is not easily predictable, in view of the nonlinear effects introduced by direct kinematics. Whenever it is desired that the **end-effector motion follows a geometrically specified path in the *operational space*, it is necessary to plan trajectory execution directly in the same space**. Planning can be done either by interpolating a sequence of prescribed path points or by generating the analytical primitive and the relative trajectory in a punctual way.
In both cases, the time sequence of the values by the operational space variables is utilized in real time to obtain the corresponding sequence of values of the joint space variables, via an inverse kinematics algorithm. In this regard, the computational complexity induced by trajectory generation in the operational space and related kinematic inversion sets an upper limit on the maximum sampling rate to generate to the above sequences (because

the computational burden is heavy). Since these sequences constitute the reference inputs to the motion control system, a linear *microinterpolation* is typically carried out. In this way, the frequency at which reference inputs are updated is increased so as to enhance dynamic performance of the system.

Whenever the path is not to be followed exactly, its characterization can be performed through the *assignment of N points specifying the values of the variables $\boldsymbol{x}_e$* chosen to describe the end-effector pose in the operational space at given time instants t_k, for $k = 1, ..., N$. Similar to what was presented in the above sections, the trajectory is generated by determining a smooth interpolating vector function between the various path points. Such function can be computed by applying to each component of $\boldsymbol{x}_e$ any of the interpolation techniques. Therefore, for given path (or via) points $\boldsymbol{x}_e(t_k)$, the corresponding components $x_{ei}(t_k)$, for $i = 1, ..., r$ (where r is the dimension of the operational space of interest) can be interpolated with a sequence of cubic polynomials, a sequence of linear polynomials with parabolic blends, and so on.

On the other hand, if the end-effector motion has to follow a prescribed trajectory o motion, this must be expressed analytically. It is then necessary to refer to *motion primitives* defining the geometric features of the path and time primitives defining the timing law on the path itself.

Path Primitives

For the definition of *path primitives* it is convenient to refer to the parametric description of paths in space. Then let $\boldsymbol{p}$ be a (3×1) vector and $\boldsymbol{f}(\sigma)$ a continuos vector function defined in the interval $[\sigma_i, \sigma_f]$. Consider the equation

$$\boldsymbol{p} = \boldsymbol{f}(\sigma) \tag{1.47}$$

with reference to its geometric description, the sequence of values of $\boldsymbol{p}$ with σ varying in $[\sigma_i, \sigma_f]$ is termed path in space. The equation in (1.47) defines the parametric representation of the path Γ and the scalar σ is called parameter. As σ increases, the point $\boldsymbol{p}$ moves on the path in a given direction.

This direction is said to be the direction induced on Γ by the parametric representation (1.47). A path is closed when $\boldsymbol{p}(\sigma_i) = \boldsymbol{p}(\sigma_f)$; otherwise it is open. Let $\boldsymbol{p}_i$ be a point on the open path Γ on which a direction has been fixed. The arc length s of the generic point $\boldsymbol{p}$ is the length of the arc of Γ with extremes $\boldsymbol{p}$ and $\boldsymbol{p}_i$ if $\boldsymbol{p}$ follows $\boldsymbol{p}_i$, the opposite of this length if $\boldsymbol{p}$ precedes $\boldsymbol{p}_i$. The point $\boldsymbol{p}_i$ is said to be the origin of the arc length ($s = 0$). From the above presentation, it follows that to each value of s a well-determined path, point corresponds, and then the arc length can be used as a parameter in a different parametric representation of the path Γ:

$$\boldsymbol{p} = \boldsymbol{f}(s) \tag{1.48}$$

the range of variation of the parameter s will be the sequence of arc lengths associated with the points of Γ. Consider a path Γ represented by (1.48). Let $\boldsymbol{p}$ be a point corresponding to the arc length s. Except for special cases, $\boldsymbol{p}$ allows the definition of **three unit vectors characterizing the path**. The orientation of such vectors depends exclusively on the path geometry, while their direction depends also on the direction induced by (1.48) on the path.

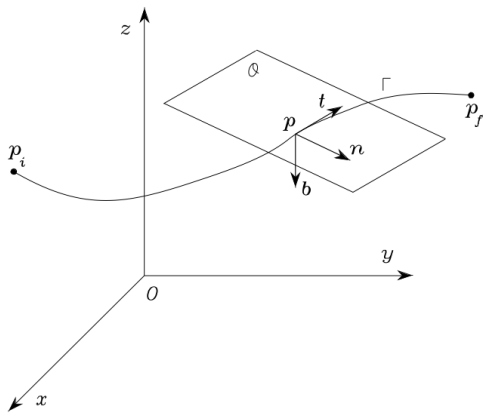


Figure 1.15: Parametric representation of a path in space

- $\boldsymbol{t}$ is the *tangent unit vector*. It is oriented along the direction induced on the path by s.
- $\boldsymbol{n}$ is the second unit vector and it is the *normal unit vector*. It is oriented along the line intersecting $\boldsymbol{p}$ at a right angle with $\boldsymbol{t}$ and lies in the so-called *osculating plane* $\mathcal{O}$ (Fig. 1.15).

- $\boldsymbol{b}$ is the third unit vector and it is the *binormal unit vector*. This vector is so that the frame ($\boldsymbol{t}$, $\boldsymbol{n}$, $\boldsymbol{b}$) is right-handed (Fig. 1.15). Notice that is not always possible to define uniquely such a frame.

The *osculating plane* $\mathcal{O}$ is the limit position of the plane containing the unit vector $\boldsymbol{t}$ and a point $\boldsymbol{p}' \in \Gamma$ when $\boldsymbol{p}'$ tends to $\boldsymbol{p}$ along the path. The direction of $\boldsymbol{n}$ is so that the path Γ, in the neighborhood of $\boldsymbol{p}$ with respect to the plane containing $\boldsymbol{t}$ and normal to $\boldsymbol{n}$, lies on the same side of $\boldsymbol{n}$. It can be shown that the above three unit vectors are related by simple relations to the path representation Γ as a function of the arc length. In particular, it is:

$$\begin{aligned} \boldsymbol{t} &= \frac{d\boldsymbol{p}}{ds} \\ \boldsymbol{n} &= \frac{1}{||\frac{d^2\boldsymbol{p}}{ds^2}||}\frac{d^2\boldsymbol{p}}{ds^2} \\ \boldsymbol{b} &= \boldsymbol{t} \times \boldsymbol{n} \end{aligned} \tag{1.49}$$

Rectilinear path

Consider the linear segment connecting point $\boldsymbol{p}_i$ to point $\boldsymbol{p}_f$. The parametric representation of this path is:

$$\boldsymbol{p}(s) = \boldsymbol{p}_i + \frac{s}{||\boldsymbol{p}_f - \boldsymbol{p}_i||}(\boldsymbol{p}_f - \boldsymbol{p}_i) \tag{1.50}$$

Notice that $\boldsymbol{p}(0) = \boldsymbol{p}_i$ and $\boldsymbol{p}(||\boldsymbol{p}_f - \boldsymbol{p}_i||) = \boldsymbol{p}_f$. Hence, the direction induced on Γ by the parametric representation (1.50) is that going from $\boldsymbol{p}_i$ to $\boldsymbol{p}_f$. Differentiating (1.50) with respect to s gives:

$$\frac{d\boldsymbol{p}}{ds} = \frac{1}{||\boldsymbol{p}_f - \boldsymbol{p}_i||}(\boldsymbol{p}_f - \boldsymbol{p}_i) = \boldsymbol{t} \tag{1.51}$$

$$\frac{d^2\boldsymbol{p}}{ds^2} = \boldsymbol{0} \tag{1.52}$$

In this case, it is not possible to define the frame ($\boldsymbol{t}$, $\boldsymbol{n}$, $\boldsymbol{b}$) uniquely.

Circular Path

Consider a circle Γ in space. Before deriving its parametric representation, it is necessary to introduce its significant parameters. Suppose that the circle is specified by assigning (Fig. 1.16):

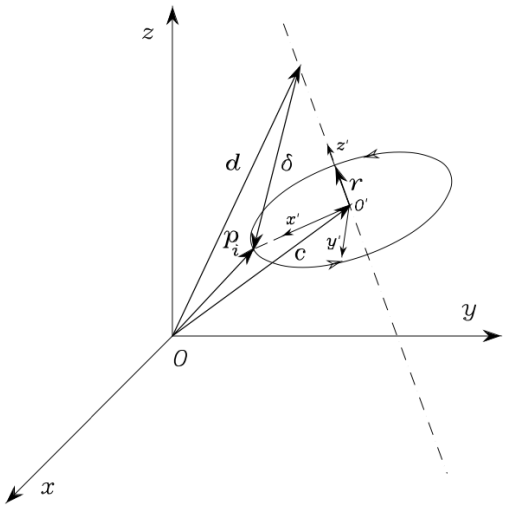


Figure 1.16: Parametric representation of a circle in space

- $\boldsymbol{r}$, the unit vector of the circle axis
- $\boldsymbol{d}$, the position vector of a point along the circle axis
- $\boldsymbol{p}_i$, the position of a point on the circle

With these parameters, the position vector $\boldsymbol{c}$ of the center of the circle can be found. Let $\boldsymbol{\delta} = \boldsymbol{p}_i - \boldsymbol{d}$; for $\boldsymbol{p}_i$ not to be on the axis, i.e., for the circle not to degenerate into a point, it must be:

$$|\boldsymbol{\delta}^T \boldsymbol{r}| < ||\boldsymbol{\delta}||$$

in this case, it is:

$$\boldsymbol{c} = \boldsymbol{d} + (\boldsymbol{\delta}^T \boldsymbol{r})\boldsymbol{r} \tag{1.53}$$

It is now desired to find a parametric representation of the circle as a function of the arc length. Notice that this representation is very simple for a suitable choice of reference frame. To see this, consider the frame O'–$x'y'z'$, where O' coincides with the center of the circle, axis x' is oriented along the direction of the vector $\boldsymbol{p_i} - \boldsymbol{c}$, axis z' is oriented along $\boldsymbol{r}$ and axis y' is chosen so as to complete a right-handed frame. When expressed in this reference frame, the parametric representation of the circle is

$$\boldsymbol{p}'(s) = \begin{bmatrix} \rho cos(s/\rho)) \\ \rho sin(s/\rho)) \\ 0 \end{bmatrix} \tag{1.54}$$

where $\rho = |p_i - c|$ is the radius of the circle and the point $\boldsymbol{p}_i$ has been assumed as the origin of the arc length. For a different reference frame, the path representation becomes:

$$\boldsymbol{p}(s) = \boldsymbol{c} + \boldsymbol{R}\boldsymbol{p}'(s) \tag{1.55}$$

where $\boldsymbol{c}$ is expressed in the frame $O - xyz$ and $\boldsymbol{R}$ is the rotation matrix of frame $O' - x'y'z'$. Differentiating (1.55) with respect to s gives:

$$\frac{d\boldsymbol{p}}{ds} = \boldsymbol{R}\begin{bmatrix} -sin(s/\rho) \\ cos(s/\rho) \\ 0 \end{bmatrix} \tag{1.56}$$

$$\frac{d^2\boldsymbol{p}}{ds^2} = \boldsymbol{R}\begin{bmatrix} -cos(s/\rho)/\rho \\ -sin(s/\rho)/\rho \\ 0 \end{bmatrix} \tag{1.57}$$

Position

Let $\boldsymbol{x}_e$ be the vector of operational space variables expressing the pose of the manipulator's end-effector as in (1.33). Generating a trajectory in the operational space means determining a function $\boldsymbol{x}_e(t)$ taking the end-effector frame from the initial to the final pose in a time t_f along a given path with a specific motion timing law. First, consider the end-effector position. Orientation will follow. Let $\boldsymbol{p}_e = \boldsymbol{f}(s)$ be the (3×1) vector of the parametric representation of the path Γ as a function of the arc length s; the origin of the end-effector frame moves from $\boldsymbol{p}_i$ to $\boldsymbol{p}_f$ in a time t_f. For simplicity, suppose that the origin of the arc length is at $\boldsymbol{p}_i$ and the direction induced on Γ is that going from $\boldsymbol{p}_i$ to $\boldsymbol{p}_f$. The arc length then goes from the value $s = 0$ at $t = 0$ to the value $s = s_f$ (path length) at $t = t_f$. The timing law along the path is described by the function $s(t)$. In order to find an analytic expression for $s(t)$, any of the above techniques for joint trajectory generation can be employed. In particular, either a cubic polynomial or a sequence of linear segments with parabolic blends can be chosen for $s(t)$. It is worth making some remarks on the time evolution of $\boldsymbol{p}_e$ on Γ, for a given timing law $s(t)$. The velocity of point $\boldsymbol{p}_e$ is given by the time derivative of $\boldsymbol{p}_e$

$$\dot{\boldsymbol{p}}_e = \dot{s}\frac{d\boldsymbol{p}_e}{ds} = \dot{s}\boldsymbol{t}$$

the positive or negative sign depending on the direction of $\dot{\boldsymbol{p}}$ along $\boldsymbol{t}$, where $\boldsymbol{t}$ is the tangent vector to the path at point $\boldsymbol{p}$ in (1.49). Then, $\dot{s}$ represents the magnitude of the velocity vector relative to point $\boldsymbol{p}$, taken with the magnitude of $\dot{\boldsymbol{p}}$ starts from zero at $t = 0$, then it varies with a parabolic or trapezoidal profile as per either of the above choices for $s(t)$ and finally it returns to zero at $t = t_f$. As a first example, consider the segment connecting point p_i with point p_f. The parametric representation of this path is given by (1.50). Velocity and acceleration of p_e can be easily computed by recalling the rule of differentiation of compound functions, i.e.:

$$\dot{\boldsymbol{p}}_e = \frac{\dot{s}}{||\boldsymbol{p}_f - \boldsymbol{p}_i||}(\boldsymbol{p}_f - \boldsymbol{p}_i) = \dot{s}\boldsymbol{t} \tag{1.58}$$

$$\ddot{\boldsymbol{p}}_e = \frac{\ddot{s}}{||\boldsymbol{p}_f - \boldsymbol{p}_i||}(\boldsymbol{p}_f - \boldsymbol{p}_i) = \ddot{s}\boldsymbol{t} \tag{1.59}$$

As a further example, consider a circle Γ in space. From the parametric representation derived above, in view of (1.56), (1.57), velocity and acceleration of point $\boldsymbol{p}_e$ on the circle are:

$$\dot{\boldsymbol{p}}_e = \boldsymbol{R}\begin{bmatrix} -\dot{s}sin(s/\rho) \\ \dot{s}cos(s/\rho) \\ 0 \end{bmatrix} \tag{1.60}$$

$$\ddot{\boldsymbol{p}}_e = \boldsymbol{R}\begin{bmatrix} -\dot{s}^2cos(s/\rho)/\rho - \ddot{s}sin(s/\rho) \\ -\dot{s}^2sin(s/\rho)/\rho + \ddot{s}cos(s/\rho) \\ 0 \end{bmatrix} \tag{1.61}$$

Notice that the velocity vector is aligned with $\boldsymbol{t}$, and the acceleration vector is given by two contributions: the first is aligned with $\boldsymbol{n}$ and represents the centripetal acceleration, while the second is aligned with $\boldsymbol{t}$ and represents the tangential acceleration.

Finally, consider the path consisting of a sequence of $N + 1$ points, $\boldsymbol{p}_0, \boldsymbol{p}_1, ..., \boldsymbol{p}_N$, connected by N segments. A feasible parametric representation of the overall path is the following:

$$\boldsymbol{p}_e = \boldsymbol{p}_0 + \sum_{j=1}^{N} \frac{s_j}{||\boldsymbol{p}_j - \boldsymbol{p}_{j-1}||}(\boldsymbol{p}_j - \boldsymbol{p}_{j-1}) \tag{1.62}$$

with $j = 1, ..., N$. In (1.62) s_j is the arc length associated with the j-th segment of the path, connecting point $\boldsymbol{p}_{j-1}$ to point $\boldsymbol{p}_j$, defined as

$$s_j(t) = \begin{cases} 0 & 0 \leq t \leq t_{j-1} \\ s'_j(t) & t_{j-1} < t < t_j \\ ||\boldsymbol{p}_j - \boldsymbol{p}_{j-1}|| & t_j \leq t \leq t_f \end{cases} \tag{1.63}$$

where $t_0 = 0$ and $t_N = t_f$ are respectively the initial and final time instants of the trajectory, t_j is the time instant corresponding to point $\boldsymbol{p}_j$ and $s'_j(t)$ can be an analytical function of cubic polynomial type, linear type with parabolic blends, and so forth, which varies continuously from the value $s'_j = 0$ at $t = t_{j-1}$ to the value $s'_j = ||\boldsymbol{p}_j - \boldsymbol{p}_{j-1}||$ at $t = t_j$. The velocity and acceleration of p_e can be easily found by differentiating (1.62):

$$\dot{\boldsymbol{p}}_e = \sum_{j=1}^{N} \frac{\dot{s}_j}{||\boldsymbol{p}_j - \boldsymbol{p}_{j-1}||}(\boldsymbol{p}_j - \boldsymbol{p}_{j-1}) = \sum_{j=1}^{N} \dot{s}_j \boldsymbol{t}_j \tag{1.64}$$

$$\ddot{\boldsymbol{p}}_e = \sum_{j=1}^{N} \frac{\ddot{s}_j}{||\boldsymbol{p}_j - \boldsymbol{p}_{j-1}||}(\boldsymbol{p}_j - \boldsymbol{p}_{j-1}) = \sum_{j=1}^{N} \ddot{s}_j \boldsymbol{t}_j \tag{1.65}$$

where $\boldsymbol{t}_j$ is the tangent unit vector of the j-th segment. Because of the discontinuity of the first derivative at the path points between two non-aligned segments, the manipulator will have to stop and then go along the direction of the following segment. Assumed a relaxation of the constraint to pass through the path points, it is possible to avoid a manipulator stop by connecting the segments near the above points, which will then be named *operational space via points* so as to guarantee, at least, continuity of the first derivative. As already illustrated for planning of interpolating linear polynomials with parabolic blends passing by the via points in the joint space, the use of trapezoidal velocity profiles for the arc lengths allows the development of a rather simple planning algorithm. In detail, it will be sufficient to properly anticipate the generation of the single segments, before the preceding segment has been completed. This leads to modifying (1.66) as follows:

$$s_j(t) = \begin{cases} 0 & 0 \leq t \leq t_{j-1} - \Delta t_j \\ s'_j(t + \Delta t_j) & t_{j-1} - \Delta t_j < t < t_j - \Delta t_j \\ ||\boldsymbol{p}_j - \boldsymbol{p}_{j-1}|| & t_j - \Delta t_j \leq t \leq t_f - \Delta t_N \end{cases} \tag{1.66}$$

where Δt_j is the time advance at which the j-th segment is generated, which can be recursively evaluated as:

$$\Delta t_j = \Delta t_{j-1} + \delta t_j$$

with $j = 1, ..., N$ and $\Delta t_0 = 0$. Notice that this time advance is given by the sum of two contributions: the former, Δt_{j-1}, accounts for the sum of the time advances at which the preceding segments have been generated, while the latter, δt_j, is the time advance at which the generation of the current segment starts.

Orientation

Consider now end-effector orientation. Typically, this is specified in terms of the rotation matrix of the (time-varying) end-effector frame with respect to the base frame. As is well known, the three columns of the rotation matrix represent the three unit vectors of the end-effector frame with respect to the base frame. To generate a trajectory, however, a linear interpolation on the unit vectors $\boldsymbol{n}_e, \boldsymbol{s}_e, \boldsymbol{a}_e$ describing the initial and final orientation does not guarantee orthonormality of the above vectors at each instant of time.

Euler angles

In view of the above difficulty, for trajectory generation purposes, orientation is often described in terms of the Euler angles triplet $\phi_e = (\varphi, \theta, \psi)$ for which a timing law can be specified. Usually, ϕ_e moves along the segment connecting its initial value ϕ_i to its final value ϕ_f. Also in this case, it is convenient to choose a cubic polynomial or a linear segment with parabolic blends timing law. In this way, in fact, the angular velocity ω_e of the timevarying frame, which is related to $\dot{\phi}_e$ by the linear relationship:

$$\boldsymbol{\omega}_e = \boldsymbol{T}(\phi_e)\dot{\phi}_e \tag{1.67}$$

it will have continuous magnitude. Therefore, for given ϕ_i and ϕ_f and timing law, the position, velocity and acceleration profiles are

$$\begin{aligned}
\phi_e &= \phi_i + \frac{s}{||\phi_f - \phi_i||}(\phi_f - \phi_i) \\
\dot{\phi}_e &= \frac{\dot{s}}{||\phi_f - \phi_i||}(\phi_f - \phi_i) \\
\ddot{\phi}_e &= \frac{\ddot{s}}{||\phi_f - \phi_i||}(\phi_f - \phi_i)
\end{aligned} \tag{1.68}$$

where the timing law for $s(t)$ has to be specified. The three unit vectors of the end-effector frame can be computed — with reference to Euler angles ZYZ — as in (1.16), the end-effector frame angular velocity as in (1.67), and the angular acceleration by differentiating (1.67) itself.

Chapter 2

Motion Planning

From MODERN ROBOTICS MECHANICS, PLANNING, AND CONTROL, Kevin M. Lynch and Frank C. Park

Motion planning is the problem of finding a robot motion from a start state to a goal state that avoids obstacles in the environment and satisfies other constraints, such as joint limits or torque limits. Motion planning is one of the most active subfields of robotics, and it is the subject of entire books.

2.1 Overview of Motion Planning

A key concept in motion planning is configuration space, or **$\mathcal{C}$-space** for short. Every point in the $\mathcal{C}$-space $\mathcal{C}$ corresponds to a unique configuration q of the robot, and every configuration of the robot can be represented as a point in $\mathcal{C}$-space. For example, the configuration of a robot arm with n joints can be represented as a list of n joint positions, $q = [\theta_1, ..., \theta_n]^T$. The **free $\mathcal{C}$-space** $\mathcal{C}_{free}$ consists of the configurations where the robot neither penetrates an obstacle nor violates a joint limit. In this chapter, unless otherwise stated, we assume that q is an n-vector and that $\mathcal{C} \subset \mathbf{R}^n$. With some generalization, the concepts of this chapter apply to non-Euclidean $\mathcal{C}$-spaces such as $\mathcal{C} = SE(3)$. The control inputs available to drive the robot are written as an m-vector $u \in \mathcal{U} \subset \mathbb{R}^m$, where $m = n$ for a typical robot arm. If the robot has second-order dynamics, such as that for a robot arm, and the

control inputs are forces (equivalently, accelerations), the state of the robot is defined by its configuration and velocity, $x = (q, v) \in \mathcal{X}$. For $q \in \mathbb{R}^n$, typically we write $v = \dot{q}$. If we can treat the control inputs as velocities, the state x is simply the configuration q. The notation $q(x)$ indicates the configuration q corresponding to the state x, and $\mathcal{X}_{free} = \{x : q(x) \in C_{free}\}$. The equations of motion of the robot are written

$$\dot{x} = f(x, u) \tag{2.1}$$

or, in integral form

$$x(T) = x(0) + \int_0^T f(x(t), u(t))dt \tag{2.2}$$

2.2 Types of Motion Planning Problems

Definition 2.2.1 (**Motion Planning Problem**). *Given an initial state $x(0) = x_{start}$ and a desired final state x_{goal}, find a time T and a set of controls $u : [0, T] \to \mathcal{U}$ such that the motion* (2.2) *satisfies $x(T) = x_{goal}$ and $q(x(t)) \in C_{free}$ for all $t \in [0, T]$*

It is assumed that a feedback controller is available to ensure that the planned motion $x(t)$, $t \in [0, T]$, is followed closely. It is also assumed that an accurate geometric model of the robot and environment is available to evaluate $\mathcal{C}_{free}$ during motion planning. There are many variations of the basic problem:

Path planning versus motion planning

The path planning problem is a subproblem of the general motion planning problem. Path planning is the purely geometric problem of finding a collision-free path $q(s)$, $s \in [0, 1]$, from a start configuration $q(0) = q_{start}$ to a goal configuration $q(1) = q_{goal}$, without concern for the dynamics, the duration of motion, or constraints on the motion or on the control inputs. It is assumed that the path returned by the path planner can be time-scaled to create a feasible trajectory. This problem is sometimes called the *piano mover's problem*, emphasizing the focus on the geometry of cluttered spaces.

Control inputs: $m = n$ versus $m < n$

If there are fewer control inputs m than degrees of freedom n, then the robot is incapable of following many paths, even if they are collision-free. For example, a car has $n = 3$ (the position and orientation of the chassis in the plane) but $m = 2$ (forward–backward motion and steering); it cannot slide directly sideways into a parking space.

Online versus offline

A motion planning problem requiring an immediate result, perhaps because obstacles appear, disappear, or move unpredictably, calls for a fast, online, planner. If the environment is static then a slower offline planner may suffice.

Optimal versus satisficing

In addition to reaching the goal state, we might want the motion plan to minimize (or approximately minimize) a cost J, e.g.:

$$J = \int_0^T L(x(t), u(t))dt \tag{2.3}$$

For example, minimizing with $L = 1$ yields a time-optimal motion while minimizing with $L = u^T(t)u(t)$ yields a "minimum-effort" motion

Exact versus approximate

We may be satisfied with a final state $x(T)$ that is sufficiently close to x_{goal}, e.g., $||x(T) - x_{goal}|| < \epsilon$

With or without obstacles

The motion planning problem can be challenging even in the absence of obstacles, particularly if $m < n$ or optimality is desired.

2.3 Properties of Motion Planners

Planners must conform to the properties of the motion planning problem as outlined above. In addition, planners can be distinguished by the following

properties:

Multiple-query versus single-query planning

If the robot is being asked to solve a number of motion planning problems in an unchanging environment, it may be worth spending the time building a data structure that accurately represents $\mathcal{C}_{free}$. This data structure can then be searched to solve multiple planning queries efficiently. Single-query planners solve each new problem from scratch.

"Anytime" planning

An anytime planner is one who continues to look for better solutions after the first solution is found. The planner can be stopped at any time, for example when a specified time limit has passed, and the best solution returned.

Completeness

A motion planner is said to be **complete** if it is guaranteed to find a solution in finite time if one exists, and to report failure if there is no feasible motion plan. A weaker concept is **resolution completeness**. A planner is resolution complete if it is guaranteed to find a solution if one exists at the resolution of a discretized representation of the problem, such as the resolution of a grid representation of $\mathcal{C}_{free}$. Finally, a planner is **probabilistically complete** if the probability of finding a solution, if one exists, tends to 1 as the planning time goes to infinity.

Computational complexity

The computational complexity refers to characterizations of the amount of time the planner takes to run or the amount of memory it requires. These are measured in terms of the description of the planning problem, such as the dimension of the $\mathcal{C}$-space or the number of vertices in the representation of the robot and obstacles. For example, the time for a planner to run may be exponential in n, the dimension of the $\mathcal{C}$-space. The computational complexity may be expressed in terms of the average case or the worst case.

Some planning algorithms lend themselves easily to computational complexity analysis, while others do not.

2.4 Motion Planning Methods

There is no single planner applicable to all motion planning problems. Below is a broad overview of some of the many motion planners available.

Grid methods

These methods discretize $\mathcal{C}_{free}$ into a grid and search the grid for a motion from q_{start} to a grid point in the goal region. Modifications of the approach may discretize the state space or control space or they may use multi-scale grids to refine the representation of $\mathcal{C}_{free}$ near obstacles. These methods are relatively easy to implement and can return optimal solutions but, for a fixed resolution, the memory required to store the grid, and the time to search it, grow exponentially with the number of dimensions of the space. This limits the approach to low-dimensional problems.

Sampling methods

A generic sampling method relies on a random or deterministic function to choose a sample from the $\mathcal{C}$-space or state space; a function to evaluate whether the sample is in $\mathcal{X}_{free}$; a function to determine the "closest" previous free-space sample; and a local planner to try to connect to, or move toward, the new sample from the previous sample. This process builds up a graph or tree representing the feasible motions of the robot. Sampling methods are easy to implement, tend to be probabilistically complete, and can even solve high-degree-of-freedom motion planning problems. The solutions tend to be satisfying, not optimal, and it can be difficult to characterize the computational complexity.

Virtual potential fields

Virtual potential fields create forces on the robot that pull it toward the goal and push it away from obstacles. The approach is relatively easy to im-

plement, even for high-degree-of-freedom systems, and fast to evaluate, often allowing online implementation. The drawback is local minima in the potential function: the robot may get stuck in configurations where the attractive and repulsive forces cancel but the robot is not at the goal state.

Nonlinear optimization

The motion planning problem can be converted to a nonlinear optimization problem by representing the path or controls by a finite number of design parameters, such as the coefficients of a polynomial or a Fourier series. The problem is to solve for the design parameters that minimize a cost function while satisfying constraints on the controls, obstacles, and goals. While these methods can produce near-optimal solutions, they require an initial guess at the solution. Because the objective function and feasible solution space are generally not convex, the optimization process can get stuck far away from a feasible solution, let alone an optimal solution.

Smoothing

Often the motions found by a planner are jerky. A smoothing algorithm can be run on the result of the motion planner to improve the smoothness.

A major trend in recent years has been toward **sampling methods**, which are easy to implement and can handle high-dimensional problems.

2.5 Foundations

Before discussing motion planning algorithms, we establish concepts used in many of them: configuration space obstacles, collision detection, graphs, and graph search.

2.5.1 Configuration Space Obstacles

Determining whether a robot at a configuration q is in collision with a known environment generally requires a complex operation involving a CAD model of the environment and robot. There are a number of free and

commercial software packages that can perform this operation, and we will not delve into them here. For our purposes, it is enough to know that the workspace obstacles partition the configuration space $\mathcal{C}$ into two sets, the free space $\mathcal{C}_{free}$ and the obstacle space C_{obs}, where $\mathcal{C} = \mathcal{C}_{free} \cup \mathcal{C}_{obs}$. Joint limits are treated as obstacles in the configuration space. With the concepts of $\mathcal{C}_{free}$ and $\mathcal{C}_{obs}$, the path planning problem reduces to the problem of finding a path for a point robot among the obstacles $\mathcal{C}_{obs}$. If the obstacles break $\mathcal{C}_{free}$ into separate connected components, and q_{start} and q_{goal} do not lie in the same connected component, then there is no collision-free path. The explicit mathematical representation of a $\mathcal{C}_{obs}$ can be exceedingly complex, and for that reason, $\mathcal{C}_{obs}$ are rarely represented exactly. Despite this, the concept of $\mathcal{C}_{obs}$ is very important for understanding motion planning algorithms. The ideas are best illustrated by examples:

A 2R Planar Arm

Figure 2.1 shows a 2R planar robot arm, with configuration $q = [\theta_1 \quad \theta_2]^T$, among obstacles A, B, and C in the workspace. The $\mathcal{C}$-space of the robot is represented by a portion of the plane with $0 \leq \theta_1 < 2\pi, \quad 0 \leq \theta_2 < 2\pi$.

The topology of the $\mathcal{C}$-space, for a 2R planar arm, is a torus (or doughnut, in the sense that is periodic) since the edge of the square at $\theta_1 = 2\pi$ is connected to the edge $\theta_1 = 0$; similarly, $\theta_2 = 2\pi$ is connected to $\theta_2 = 0$. The

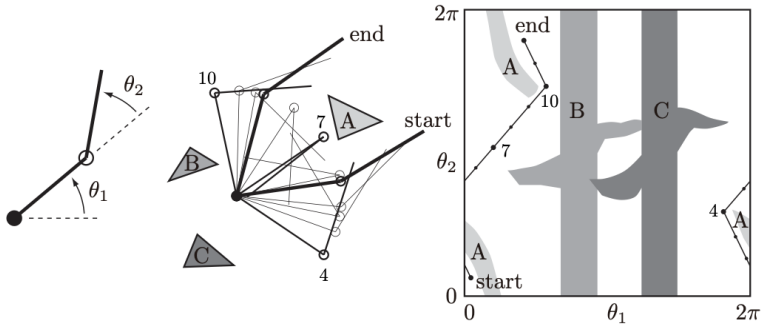


Figure 2.1: (Left) The joint angles of a 2R robot arm. (Middle) The arm navigating among obstacles A, B, and C. (Right) The same motion in $\mathcal{C}$-space. Three intermediate points, 4, 7, and 10, along the path are labeled

square region of $\mathbb{R}^2$ is obtained by slicing the surface of the doughnut twice,

at $\theta_1 = 0$ and $\theta_2 = 0$, and laying it flat on the plane. The $\mathcal{C}$-space on the right in Figure 2.1 shows the workspace obstacles A, B, and C represented as $\mathcal{C}$-obstacles. Any configuration lying inside a $\mathcal{C}$-obstacle corresponds to penetration of the obstacle by the robot arm in the workspace. A free path for the robot arm from one configuration to another is shown in both the workspace and $\mathcal{C}$-space. The path and obstacles illustrate the topology of the $\mathcal{C}$-space. Note that the obstacles break $\mathcal{C}_{free}$ into three connected components.

A Circular Planar Mobile Robot

Figure 2.2 shows a top view of a circular mobile robot whose configuration is given by the location of its center, $(x, y) \in \mathbb{R}^2$. The robot translates (moves without rotating) in a plane with a single obstacle. The corresponding $\mathcal{C}$-obstacle is obtained by "growing" (enlarging) the workspace obstacle by the radius of the mobile robot. Any point outside this $\mathcal{C}$-obstacle represents a free configuration of the robot. Figure 2.3 shows the workspace and $\mathcal{C}$-space for two obstacles, indicating that in this case the mobile robot cannot pass between the two obstacles.

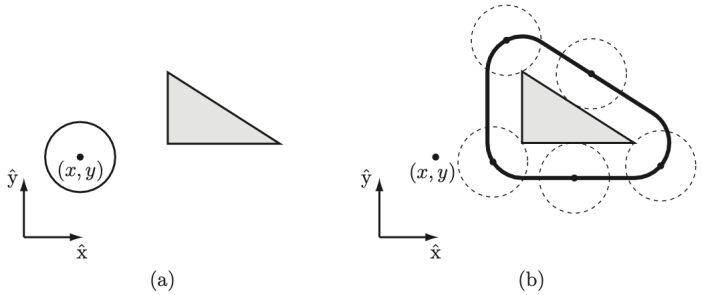


Figure 2.2: (a) A circular mobile robot (open circle) and a workspace obstacle (gray triangle). The configuration of the robot is represented by (x, y), the center of the robot. (b) In the $\mathcal{C}$-space, the obstacle is "grown" by the radius of the robot and the robot is treated as a point. Any (x, y) configuration outside the bold line is collision-free.

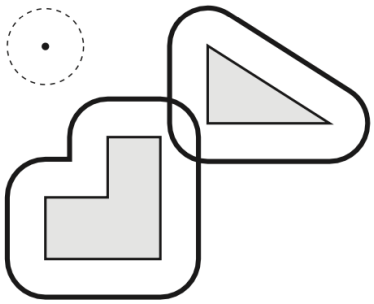


Figure 2.3: The "grown" $\mathcal{C}$-space obstacles corresponding to two workspace obstacles and a circular mobile robot. The overlapping boundaries mean that the robot cannot move between the two obstacles.

A Polygonal Planar Mobile Robot That Translates

Figure 2.4 shows the $\mathcal{C}$-obstacle for a polygonal mobile robot translating in the presence of a polygonal obstacle. The $\mathcal{C}$-obstacle is obtained by sliding the robot along the boundary of the obstacle and tracing the position of the robot's reference point.

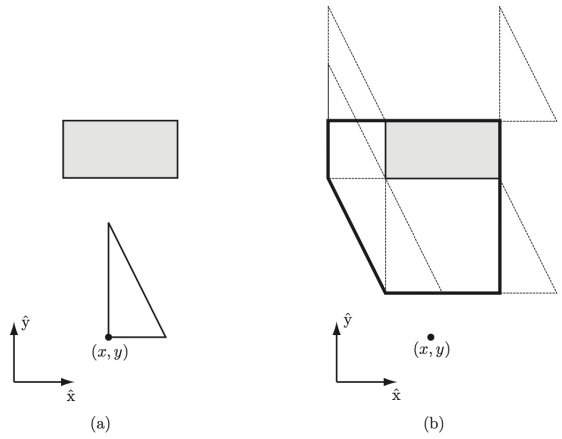


Figure 2.4: (a) The configuration of a triangular mobile robot, which can translate but not rotate, is represented by the (x, y) location of a reference point. Also shown is a workspace obstacle in gray. (b) The corresponding $\mathcal{C}$-space obstacle (bold outline) is obtained by sliding the robot around the boundary of the obstacle and tracing the position of the reference point.

A Polygonal Planar Mobile Robot That Translates and Rotates

Figure 2.5 illustrates the $\mathcal{C}$-obstacle for the workspace obstacle and triangular mobile robot of Figure 10.5 if the robot is now allowed to rotate. The $\mathcal{C}$-space is now three dimensional, given by $(x, y, \theta) \in \mathbb{R}^2 \times S^1$. The three-dimensional $\mathcal{C}$-obstacle is the union of two-dimensional $\mathcal{C}$-obstacle slices at angles $\theta \in [0, 2\pi)$. Even for this relatively low-dimensional $\mathcal{C}$-space, an exact representation of the $\mathcal{C}$-obstacle is quite complex. For this reason, $\mathcal{C}$-obstacles are rarely described exactly.

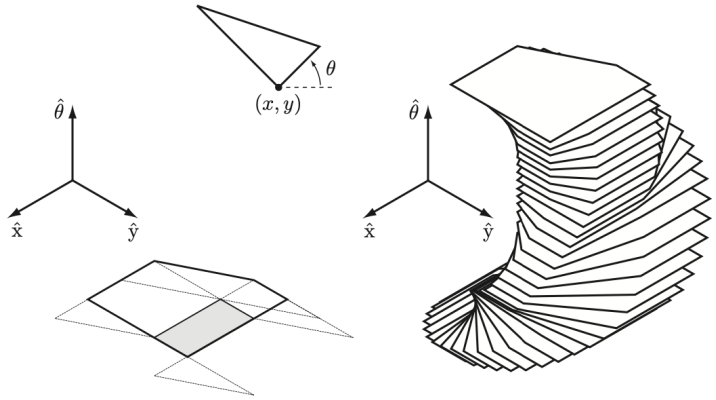


Figure 2.5: (Top) A triangular mobile robot that can both rotate and translate, represented by the configuration (x, y, θ). (Left) The $\mathcal{C}$-space obstacle from Figure 2.4(b) when the robot is restricted to $\theta = 0$. (Right) The full three-dimensional $\mathcal{C}$-space obstacle shown in slices at 10° increments.

2.5.2 Distance to Obstacles and Collision Detection

Given a $\mathcal{C}$-obstacle $\mathcal{B}$ and a configuration q, let $d(q, \mathcal{B})$ be the distance between the robot and the obstacle, where

$$d(q, \mathcal{B}) > 0 \quad \text{(no contact with the obstacle)}$$
$$d(q, \mathcal{B}) = 0 \quad \text{(contact)}$$
$$d(q, \mathcal{B}) < 0 \quad \text{(penetration)}$$

The distance could be defined as the Euclidean distance between the two closest points of the robot and the obstacle, respectively. A distance-measurement

algorithm is one that determines $d(q, \mathcal{B})$. A collision–detection routine determines whether $d(q, \mathcal{B}_i) \leq 0$ for any $\mathcal{C}$-obstacle $\mathcal{B}_i$. A collision-detection routine returns a binary result and may or may not utilize a distance-measurement algorithm at its core. One popular distance-measurement algorithm is the **Gilbert–Johnson–Keerthi (GJK) algorithm**, which efficiently computes the distance between two convex bodies, possibly represented by triangular meshes. Any robot or obstacle can be treated as the union of multiple convex bodies. Extensions of this algorithm are used in many distance-measurement algorithms and collision-detection routines for robotics, graphics, and game-physics engines.

A simpler approach is to **approximate the robot and obstacles as unions of overlapping spheres**. Approximations must always be conservative – the approximation must cover all points of the object – so that if a collision-detection routine indicates a free configuration q, then we are guaranteed that the actual geometry is collision-free. As the number of spheres in the representation of the robot and obstacles increases, the closer the approximations come to the actual geometry. An example is shown in Figure 2.6. Given a robot at q represented by k spheres of radius R_i centered at

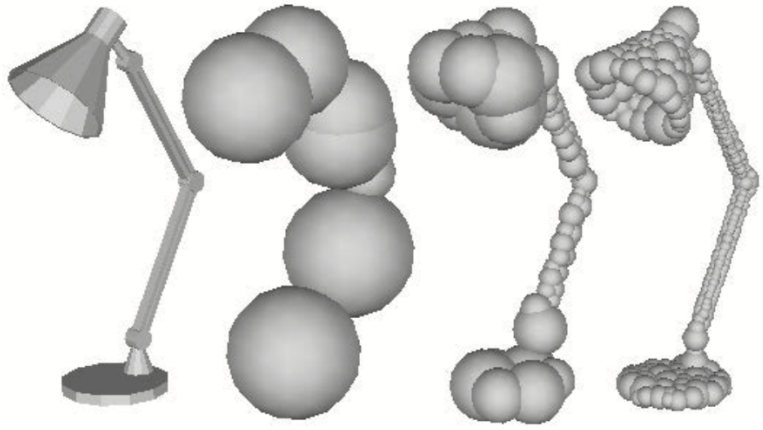


Figure 2.6: A lamp represented by spheres. The approximation improves as the number of spheres used to represent the lamp increases

$r_i(q)$, $i = 1, ..., k$, and an obstacle $\mathcal{B}$ represented by l spheres of radius B_j centered at $b_j, j = 1, ..., l$, the distance between the robot and the obstacle

can be calculated as

$$d(q, \mathcal{B}) = \min_{i,j}\{||r_i(q) - b_j|| - (R_i + B_j)\}$$

Apart from determining whether a particular configuration of the robot is in collision, another useful operation is determining whether the robot collides during a particular motion segment. While exact solutions have been developed for particular object geometries and motion types, the general approach is to sample the path at finely spaced points and to "grow" the robot to ensure that if two consecutive configurations are collision-free for the grown robot then the volume swept out by the actual robot between the two configurations is also collision-free.

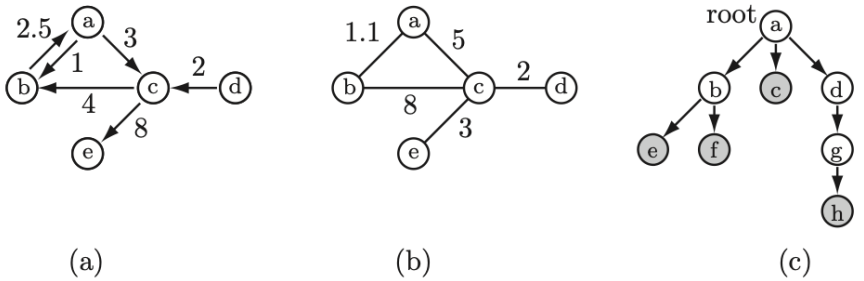
2.5.3 Graphs and Trees

Many motion planners explicitly or implicitly represent the $\mathcal{C}$-space or state space as a graph. A graph consists of a collection of nodes N and a collection of edges $\mathcal{E}$, where each edge e connects two nodes.

In motion planning, a node typically represents a configuration or state while an edge between nodes n_1 and n_2 indicates the ability to move from n_1 to n_2 without penetrating an obstacle or violating other constraints. A graph can be either **directed** or **undirected**. In an undirected graph, each edge is bidirectional: if the robot can travel from n_1 to n_2 then it can also travel from n_2 to n_1. In a directed graph, or **digraph** for short, each edge allows travel in only one direction. The same two nodes can have two edges between them, allowing travel in opposite directions. Graphs can also be **weighted** or **unweighted**. In a weighted graph, each edge has a positive cost associated with traversing it. In an unweighted graph each edge has the same cost (e.g., 1). Thus the most general type of graph we consider is a weighted digraph.

A **tree** is a digraph in which (1) there are no cycles and (2) each node has at most one **parent** node (i.e., at most one edge leading to the node). A tree has one **root** node with no parents and a number of leaf **nodes** with no **child**. A digraph, undirected graph, and tree are illustrated in Figure 2.7. Given N nodes, any graph can be represented by a matrix $A \in \mathbb{R}^{N \times N}$, where

element a_{ij} of the matrix represents the cost of the edge from node i to node j; a zero or negative value indicates no edge between the nodes. Graphs and trees can be represented more compactly as a list of nodes, each with links to its neighbors.



(a) (b) (c)

Figure 2.7: (a) A weighted digraph. (b) A weighted undirected graph. (c) A tree. The leaves are shaded gray.

2.5.4 Graph Search

Once the free space is represented as a graph, a motion plan can be found by searching the graph for a path from the start to the goal. One of the most powerful and popular graph search algorithms is ***A**** (pronounced "A star") search.

*A** Search

The A^* search algorithm is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied. At each iteration of its main loop, the algorithm needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes:

$$f(n) = g(n) + h(n)$$

where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal. A^* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended. The heuristic function is problem-specific. If the heuristic function is admissible – meaning that it never overestimates the actual cost to get to the goal – the algorithm is guaranteed to return a least-cost path from start to goal.

Given a graph described by a set of nodes $\mathcal{N} = 1, ..., N$, where node 1 is the start node, and a set of edges $\mathcal{E}$, the A^* algorithm makes use of the following data structures:

- a sorted list `OPEN` of the nodes from which exploration is still to be done, and a list `CLOSED` of nodes for which exploration has already taken place;
- a matrix cost `[node1,node2]` encoding the set of edges, where a positive value corresponds to the cost of moving from `node1` to `node2` (a negative value indicates that no edge exists);
- an array `past_cost[node]` of the minimum cost found so far to reach node node from the start node; and
- a search tree defined by an array `parent[node]`, which contains for each `node` a link to the node preceding it in the shortest path found so far from the start node to that node.

To initialize the search, the matrix cost is constructed to encode the edges, the list `OPEN` is initialized to the start node 1, the cost to reach the start node (`past_cost[1]`) is initialized to 0, and `past_cost[node]` for node $\in 2, ..., N$ is initialized to infinity (or a large number), indicating that currently we have no idea of the cost of reaching those nodes. At each step of the algorithm, the first node in `OPEN` is removed from `OPEN` and called current. The node current is also added to `CLOSED`. The first node in `OPEN` is one that minimizes the total estimated cost of the best path to the goal that passes through that node. The estimated cost is calculated as:

```
est_total_cost[node] = past_cost[node] +
        heuristic_cost_to_go(node)
```

where `heuristic_cost_to_go(node)` ≥ 0 is an optimistic (underestimating) estimate of the actual cost-to-go to the goal from `node`. For many path planning problems, an appropriate choice for the heuristic is the straight-line distance to the goal, ignoring any obstacles. Because `OPEN` is a list sorted according to the estimated total cost, inserting a new node at the correct location in `OPEN` entails a small computational price. If the node `current` is in the goal set then the search is finished and the path is reconstructed from the `parent` links. If not, for each neighbor `nbr` of current in the graph which is not also in `CLOSED`, the `tentative_past_cost` for `nbr` is calculated as `past_cost[current] + cost[current,nbr]`. If:

```
tentative_past_cost < past_cost[nbr]
```

then `nbr` can be reached with less cost than previously known, so `past_cost[nbr]` is set to `tentative_past_cost` and `parent[nbr]` is set to `current`. The node `nbr` is then added (or moved) in `OPEN` according to its estimated total cost. The algorithm then returns to the beginning of the main loop, removing the first node from `OPEN` and calling it current. If `OPEN` is empty then there is no solution. The A^* algorithm is guaranteed to return a minimum-cost path, as nodes are only checked for inclusion in the goal set when they have the minimum total estimated cost of all nodes. If the node `current` is in the goal set then `heuristic_cost_to_go(current)` is zero and, since all edge costs are positive, we know that any path found in the future must have a cost greater than or equal to `past_cost[current]`. Therefore the path to current must be a shortest path. (There may be other paths of the same cost.) If the heuristic "cost-to-go" is calculated exactly, considering obstacles, then A^* will explore from the minimum number of nodes necessary to solve the problem. Of course, calculating the cost-to-go exactly is equivalent to solving the path planning problem, so this is impractical. Instead, the heuristic cost-to- go should be calculated quickly and should be as close as possible to the actual cost-to-go to ensure that the algorithm runs efficiently. Using an optimistic cost-to-go ensures an optimal solution. The A^* algorithm is an example of the general class of best-first searches, which always explore

from the node currently deemed "best" by some measure. The A^* search algorithm is described in pseudocode below:

Algorithm 10.1 A^* search.

```
1: OPEN ← {1}
2: past_cost[1] ← 0, past_cost[node] ← infinity for node ∈ {2, ..., N}
3: while OPEN is not empty do
4:   current ← first node in OPEN, remove from OPEN
5:   add current to CLOSED
6:   if current is in the goal set then
7:     return SUCCESS and the path to current
8:   end if
9:   for each nbr of current not in CLOSED do
10:    tentative_past_cost ← past_cost[current]+cost[current,nbr]
11:    if tentative_past_cost < past_cost[nbr] then
12:      past_cost[nbr] ← tentative_past_cost
13:      parent[nbr] ← current
14:      put (or move) nbr in sorted list OPEN according to
               est_total_cost[nbr] ← past_cost[nbr] +
                       heuristic_cost_to_go(nbr)
15:    end if
16:  end for
17: end while
18: return FAILURE
```

Other Search Methods

- **Dijkstra's algorithm** If the heuristic cost-to-go is always estimated as zero then A^* always explores from the OPEN node that has been reached with minimum past cost. This variant is called Dijkstra's algorithm, which preceded A^* historically. Dijkstra's algorithm is also guaranteed to find a minimum-cost path but on many problems it runs more slowly than A^* owing to the lack of a heuristic look-ahead function to help guide the search.

- **Breadth-first search** If each edge in $\mathcal{E}$ has the same cost, Dijkstra's algorithm reduces to breadth-first search. All nodes one edge away from the start node are considered first, then all nodes two edges away, etc. The first solution found is therefore a minimum-cost path.

- **Suboptimal A^* search.** If the heuristic cost-to-go is overestimated

by multiplying the optimistic heuristic by a constant factor $\eta > 1$, the A^* search will be biased to explore from nodes closer to the goal rather than nodes with a low past cost. This may cause a solution to be found more quickly but, unlike the case of an optimistic cost-to-go heuristic, the solution will not be guaranteed to be optimal. One possibility is to run A^* with an inflated cost-to-go to find an initial solution, then rerun the search with progressively smaller values of η until the time allotted for the search has expired or a solution is found with $\eta = 1$.

2.6 Virtual Potential Fields

Virtual potential field methods are **inspired by potential energy fields in nature**, such as gravitational and magnetic fields. From physics we know that a potential field $\mathcal{P}(q)$ defined over $\mathcal{C}$ induces a force $F = -d\mathcal{P}/dq$ that drives an object from high to low potential. For example, if h is the height above the Earth's surface in a uniform gravitational potential field ($g = 9.81$ m/s^2) then the potential energy of a mass m is $\mathcal{P}(h) = mg$ hand the force acting on it is

$$F = -\frac{d\mathcal{P}}{dh} = -mg$$

The force will cause the mass to fall to the Earth's surface. In robot motion control, the goal configuration q_{goal} is assigned a low virtual potential and obstacles are assigned a high virtual potential. *Applying a force to the robot proportional to the negative gradient of the virtual potential naturally pushes the robot toward the goal and away from the obstacles.*

A virtual potential field is very different from the planners we have seen so far. Typically the gradient of the field can be calculated quickly, so the motion can be calculated in real time (reactive control) instead of planned in advance. With appropriate sensors, the method can even handle obstacles that move or appear unexpectedly. The drawback of the basic method is that the robot can get stuck in local minima of the potential field, away from the goal, even when a feasible motion to the goal exists. In certain cases it is possible to design the potential to guarantee that the only local minimum is at the goal, eliminating this problem.

2.6.1 A Point in $\mathcal{C}$-space

Let's begin by assuming a point robot in its $\mathcal{C}$-space. A goal configuration q_{goal} is typically encoded by a **quadratic potential energy "bowl" with zero energy at the goal**,

$$\mathcal{P}_{goal}(q) = \frac{1}{2}(q - q_{goal})^T K (q - q_{goal})$$

where K is a symmetric positive-definite weighting matrix (for example, the identity matrix). The force induced by this potential is

$$F_{goal}(q) = -\frac{d\mathcal{P}_{goal}}{dq} = K(q_{goal} - q)$$

an attractive force proportional to the distance from the goal.
The **repulsive potential induced by a $\mathcal{C}$-obstacle $\mathcal{B}$** can be calculated from the distance $d(q, \mathcal{B})$ to the obstacle:

$$\mathcal{P}_{\mathcal{B}(q)} = \frac{k}{2d^2(q, \mathcal{B})} \tag{2.4}$$

where $k > 0$ is a scaling factor. The potential is only properly defined for points outside the obstacle, $d(q, \mathcal{B}) > 0$. The force induced by the obstacle potential is

$$F_{\mathcal{B}}(q) = -\frac{d\mathcal{P}_{\mathcal{B}}}{dq} = \frac{k}{d^3(q, \mathcal{B})}\frac{d}{dq}d$$

The total potential is obtained by summing the attractive goal potential and the repulsive obstacle potentials,

$$\mathcal{P}(q) = \mathcal{P}_{goal}(q) + \sum_i \mathcal{P}_{\mathcal{B}_i}(q)$$

yielding a total force

$$F(q) = F_{goal}(q) + \sum_i F_{\mathcal{B}_i}(q)$$

Note that the sum of the attractive and repulsive potentials may not give a minimum (zero force) exactly at q_{goal}. Also, it is common to put a bound on the maximum potential and force, as the simple obstacle potential (2.4) would otherwise yield unbounded potentials and forces near the boundaries of obstacles. Figure 2.8 shows a potential field for a point in $\mathbb{R}^2$ with three

circular obstacles. The contour plot of the potential field clearly shows the global minimum near the center of the space (near the goal marked with a +), a local minimum near the two obstacles on the left, as well as saddles (critical points that are a maximum in one direction and a minimum in the other direction) near the obstacles. Saddles are generally not a problem, as a small perturbation allows continued progress toward the goal.
Local minima away from the goal are a problem, however, as they attract nearby states.

To actually control the robot using the calculated $F(q)$, we have several options, two of which are:

- **Apply the calculated force plus damping**

$$u = F(q) + B\dot{q} \tag{2.5}$$

  If B is positive definite then it dissipates energy for all $\dot{q} \neq 0$, reducing oscillation and guaranteeing that the robot will come to rest. If $B = 0$, the robot continues to move while maintaining constant total energy, which is the sum of the initial kinetic energy $\frac{1}{2}\dot{q}^T(0)M(q(0))\dot{q}(0)$ and the initial virtual potential energy $\mathcal{P}(q(0))$ The motion of the robot under the control law 2.5 can be visualized as a ball rolling in gravity on the potential surface of Figure 2.8, where the dissipative force is rolling friction.

- **Treat the calculated force as a commanded velocity**:

$$\dot{q} = F(q) \tag{2.6}$$

  This automatically eliminates oscillations.

Using the simple obstacle potential (2.4), even distant obstacles have a nonzero effect on the motion of the robot. To speed up the evaluation of the repulsive terms, distant obstacles could be ignored. We can define a range of influence of the obstacles $d_{range} > 0$ so that the potential is zero for all $d(q, \mathcal{B}) \geq d_{range}$:

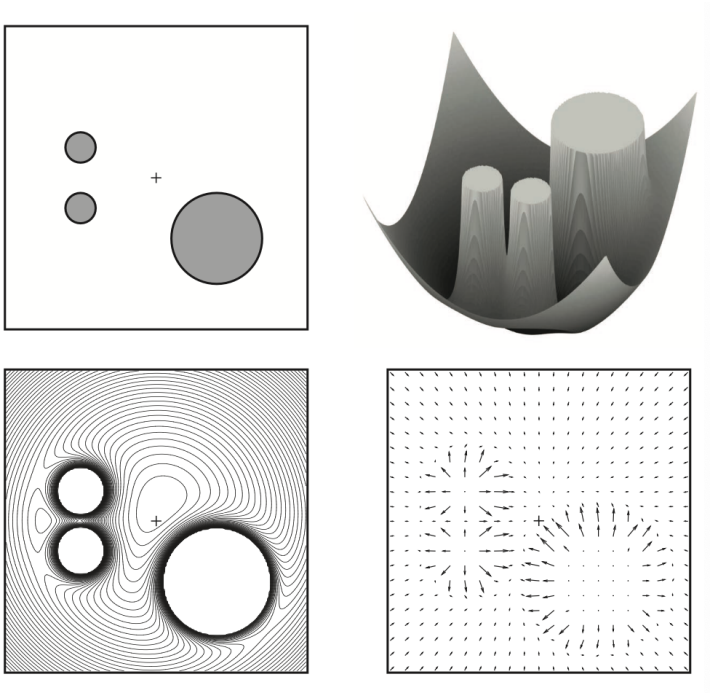


Figure 2.8: (**Top left**) Three obstacles and a goal point, marked with a +, in $\mathbb{R}^2$.
(**Top right**) The potential function summing the bowl-shaped potential pulling the robot to the goal with the repulsive potentials of the three obstacles. The potential function saturates at a specified maximum value.
(**Bottom left**) A contour plot of the potential function, showing the global minimum, a local minimum, and four saddles: between each obstacle and the boundary of the workspace, and between the two small obstacles.
(**Bottom right**) Forces induced by the potential function.

$$U_B(q) = \begin{cases} \frac{k}{2}(\frac{d_{range} - d_(q,\mathcal{B})}{d_{range} d(q,\mathcal{B})})^2 & \text{if } d(q,\mathcal{B}) < d_{range} \\ 0 & \text{otherwise} \end{cases}$$

Another issue is that $d(q,\mathcal{B})$ and its gradient are generally difficult to calculate.

Chapter 3

Navigation for Mobile Robots

From Robotics, Vision, and Control - Fundamental Algorithms in Matlab, Peter Corke

Robot navigation is the problem of guiding a robot towards a goal. Many robotic tasks can be achieved without any map at all, using an approach referred to as *reactive navigation*. For example, navigating by heading towards a light, following a white line on the ground, moving through a maze by following a wall, or vacuuming a room by following a random path. The robot is reacting directly to its environment: the intensity of the light, the relative position of the white line, or contact with a wall.

Human-style *map-based navigation* is used by more sophisticated robots and is also known as motion planning. This approach supports more complex tasks but is itself more complex. It imposes several requirements, not the least of which is having a map of the environment. It also requires that the robot's position is always known.

3.1 Reacting Navigation

Surprisingly complex tasks can be performed by a robot even if it has no map and no real idea about where it is. As already mentioned robotic vacuum cleaners use only random motion and information from contact sensors to perform a complex task.

3.1.1 Braitenberg Vehicles

A very simple class of goal-achieving robots are known as Braitenberg vehicles and are characterized by direct connection between sensors and motors. They have no explicit internal representation of the environment in which they operate nor do they make explicit plans. Consider the problem of a **robot moving in two dimensions that is seeking the local maxima of a scalar field** – the field could be light intensity or the concentration of some chemical. In Matlab, you can launch the Simulink model using:

```
>> sl_braitenberg
```

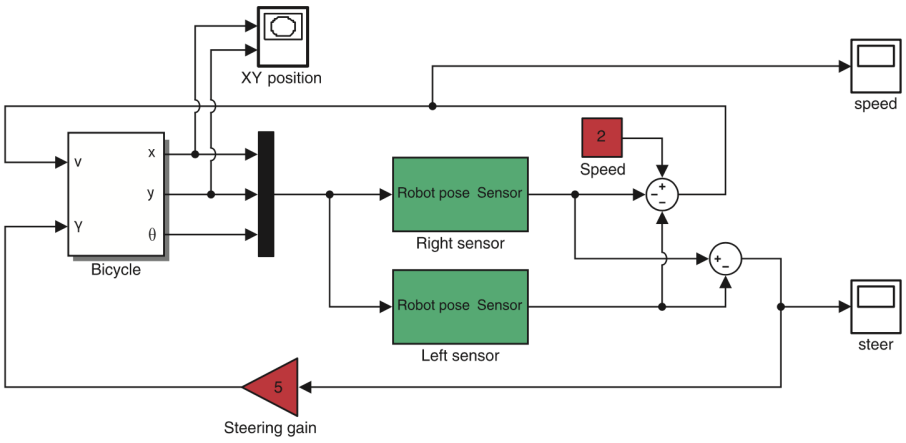


Figure 3.1: The Simulink model `sl_braitenberg` drives the vehicle toward the maxima of a provided scalar function. The vehicle plus controller is an example of a Braitenberg vehicle

To ascend the gradient we need to estimate the gradient direction at the current location and this requires at least two measurements of the field. In this example, we use two sensors, bilateral sensing, with one on each side of the robot's body. The sensors are modeled by the green sensor blocks shown in Fig. 3.1 and are parameterized by the position of the sensor with respect to the robot's body, and the sensing function. In this example, the sensors are at ± 2 units in the vehicle's lateral or y-direction. The field to be sensed is a simple inverse square field defined by

$$sensor = \frac{200}{(x - x_c^2) + (y - y_c)^2 + 200}$$

where $x_c = 60$ and $y_c = 90$. The function returns the sensor value $s(x, y) \in [0, 1]$ which is a function of the sensor's position in the plane. This particular function has a peak value at the point $(60, 90)$. The vehicle speed is:

$$v = 2 - s_R - s_L$$

where s_R and s_L are the right and left sensor readings respectively. At the goal, where $s_R = s_L = 1$ the velocity becomes zero. Steering angle is based on the difference between the sensor readings:

$$\gamma = k(s_R - s_L)$$

so when the field is equal in the left- and right-hand sensors the robot moves straight ahead.
We start the simulation from the Simulink menu or the command line:

```
>> sim('sl_braitenberg');
```

and the path of the robot is shown in Fig. 3.2. We see that the robot turns

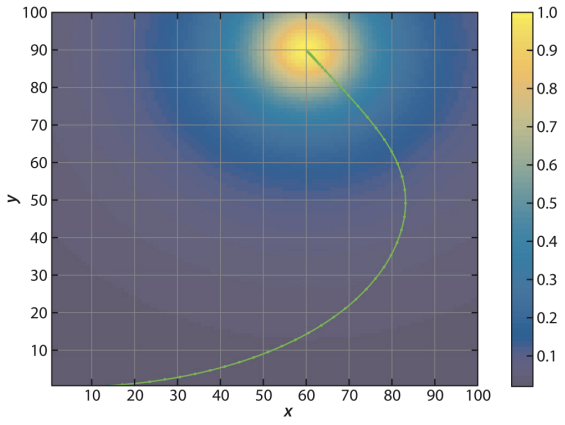


Figure 3.2: Path of the Braitenberg vehicle moving toward the maximum of a 2D scalar field whose magnitude is shown color-coded

toward the goal and slows down as it approaches, asymptotically achieving the goal position. This particular sensor-action control law results in a specific robotic behavior. We could add additional logic to the robot to detect that it had arrived near the goal and then switch to a stopping behavior. An obstacle would block this robot since its only behavior is to steer toward the goal, but an additional behavior could be added to handle this case and drive

around an obstacle. We could add another behavior to search randomly for the source if none was visible. Multiple behaviors and the ability to switch between them lead to an approach known as *behavior-based robotics*. Complex, some might say *intelligent looking*, behaviors can be manifested by such systems. However, as more behaviors are added the complexity of the system grows rapidly and interactions between behaviors become more complex to express and debug. Ultimately the penalty of not using a map becomes too great.

3.1.2 Simple Automata

Another class of reactive robots is known as *bugs* – simple automata that perform goal-seeking in the presence of nondriveable areas or obstacles. There are a large number of *bug* algorithms and they share the ability to sense when they are in proximity to an obstacle. In this respect, they are similar to the Braitenberg class vehicle, but the bug includes a state machine and other logic in between the sensor and the motors. **The automata have memory which our earlier Braitenberg vehicle lacked**.
In this section, we will investigate a specific *bug* algorithm known as *bug2*. We start by loading an obstacle field to challenge the robot:

```
>> load house
```

which defines a matrix variable `house` in the workspace. The elements are zero or one representing free space or obstacle respectively and this is shown in Fig. 3.3. This command also loads a list of named places within the house, as elements of a structure. At this point we state some assumptions. Firstly, the robot operates in a grid world and occupies one grid cell. Secondly, the robot is capable of omnidirectional motion and can move to any of its eight neighboring grid cells. Thirdly, it is able to determine its position on the plane which is a nontrivial problem. Finally, the robot can only sense its goal and whether adjacent cells are occupied. We create an instance of the `bug2` class:

```
>> bug = Bug2(house);
```

and pass in the occupancy grid. The *bug2* algorithm does not use the map

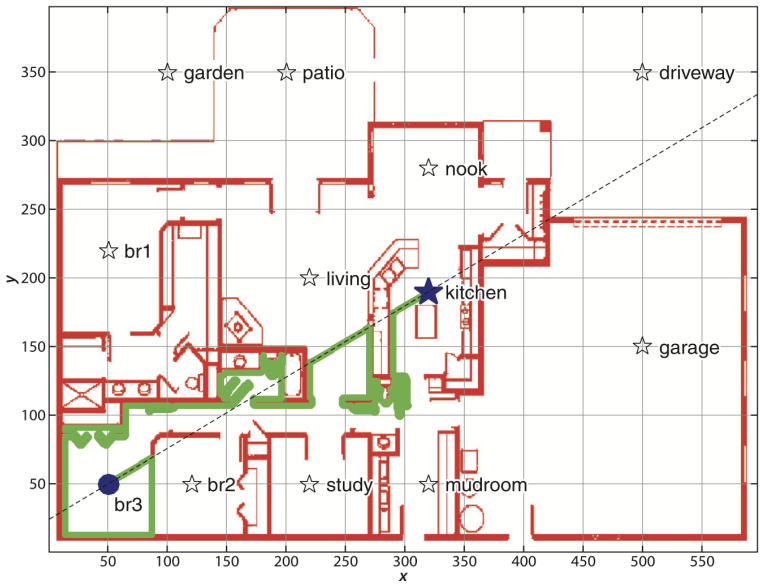


Figure 3.3: Obstacles are indicated by *red pixels*. Named places are indicated by *hollow black stars*. Approximate scale is 4.5 cm per cell. The start location is a solid blue circle and the goal is a *solid blue star*. The path taken by the bug2 algorithm is marked by a *green line*. The *black dashed line* is the m-line, the direct path from the start to the goal

to plan a path – the map is used by the simulator to provide sensory inputs to the robot. We can display the robot's environment by

```
>> bug.plot();
```

The simulation is run using the `query` method

```
>> bug.query(place.br3, place.kitchen, 'animate');
```

whose arguments are the start and goal positions of the robot within the house. The method displays an animation of the robot moving toward the goal and the path is shown as a series of green dots in Fig. 3.3. The strategy of the bug2 algorithm is quite simple. It is given a straight line – the m-line – towards its goal. If it encounters an obstacle it turns right and continues until it encounters a point on the m-line that is closer to the goal than when it departed from the m-line. If an output argument is specified:

```
>> p = bug.query(place.br3, place.kitchen)
```

it returns the path as a matrix `p`, which has one row per point. Invoking the function with an empty matrix:

```
>> p = bug.query([], place.kitchen);
```

will prompt for the corresponding point to be selected by clicking on the plot. In this example the *bug2* algorithm has reached the goal but it has taken a very suboptimal route, traversing the inside of a wardrobe, behind doors and visiting two bathrooms. It would perhaps have been quicker in this case to turn left, rather than right, at the first obstacle but that strategy might give a worse outcome somewhere else. Many variants of the bug algorithm have been developed, but while they improve the performance for one type of environment they can degrade performance in others. **Fundamentally the robot is limited by not using a map**. It cannot see the big picture and therefore takes paths that are locally, rather than globally, optimal.

3.2 Map-Based Planning

The key to achieving the best path between points A and B, as we know from everyday life, is to use a map. Typically best means the shortest distance but it may also include some penalty term or cost related to traversability which is how easy the terrain is to drive over.

There are many ways to represent a map and the position of the vehicle within the map. **Graphs** can be used to represent places and paths between them. Graphs can be efficiently searched to find a path that minimizes some measure or cost, most commonly the distance traveled.

A simpler and very computer-friendly representation is the **occupancy grid** which is widely used in robotics. An occupancy grid treats the world as a grid of cells and each cell is marked as occupied or unoccupied. **We use zero to indicate an unoccupied cell or free space where the robot can drive. A value of one indicates an occupied or non-driveable cell**. The size of the cell depends on the application. The memory required to hold the occupancy grid increases with the spatial area represented and inversely with the cell size. However, for modern computers, this representation is very feasible. To create uniformity the planners are all implemented as classes

derived from the `Navigation` superclass. The *bug2* class we used previously was also an instance of this class so the remaining examples follow a familiar pattern.

Once again we state some assumptions. Firstly, the robot operates in a grid world and occupies one grid cell. Secondly, the robot does not have any nonholonomic constraints and can move to any neighboring grid cell. Thirdly, it is able to determine its position on the plane. Fourthly, the robot is able to use the map to compute the path it will take

3.2.1 Distance Transform

Consider a matrix of zeros with just a single nonzero element representing the goal. The distance transform of this matrix is another matrix, of the same size, but the value of each element is its distance from the original nonzero pixel. For robot path planning we use the default Euclidean distance. The distance between two points (x_1, y_1) and (x_2, y_2) where $\Delta_x = x_2 - x_1$ and $\Delta_y = y_2 - y_1$ can be **Euclidean**:

$$d_E = \sqrt{\Delta_x^2 + \Delta_y^2}$$

or City Block (also known as **Manhattan**) distance:

$$d_M = |\Delta_x| + |\Delta_y|$$

Making a Map. An occupancy grid is a matrix that corresponds to a region of 2-dimensional space. Elements containing zeros are free space where the robot can move, and those with ones are obstacles where the robot cannot move. We can use many approaches to create a map. For example we could create a matrix filled with zeros (representing all free space):

```
>> map = zeros(100, 100);
```

and use MATLAB operations such as:

```
>> map(40:50,20:80) = 1;
```

or the MATLAB builtin matrix editor to create obstacles but this is quite cumbersome. Instead we can use the Toolbox map editor `makemap` to create more complex maps using an interactive editor

```
>> map = makemap(100)
```

that allows you to add rectangles, circles and polygons to an occupancy grid. In this example the grid is 100×100. Note that the occupancy grid is a matrix whose coordinates are conventionally expressed as (row, column) and the row is the vertical dimension of a matrix. We use the Cartesian convention of a horizontal x-coordinate first, followed by the y-coordinate therefore the matrix is always indexed as y, x in the code.

To use the distance transform for robot navigation we create a `DXform` object, which is derived from the `Navigation` class:

```
>> dx = DXform(house);
```

and then create a plan to reach a specific goal:

```
>> dx.plan(place.kitchen)
```

which can be visualized:

```
>> dx.plot()
```

as shown in Fig. 3.6. We see the obstacle regions in red overlaid on the distance map whose grey level at any point indicates the distance from that point to the goal, in grid cells, taking into account travel *around* obstacles. The hard work has been done and to find the shortest path from any point to the goal we simply consult or query the plan. For example a path from the bedroom to the goal is:

```
>> dx.query(place.br3, 'animate');
```

which displays an animation of the robot moving toward the goal. The path is indicated by a series of green dots as shown in Fig. 3.6.

The plan is the distance map. **Wherever the robot starts, it moves to the neighboring cell that has the smallest distance to the goal. The process is repeated until the robot reaches a cell with a distance value of zero which is the goal.** If the `path` method is called with an output argument the path:

```
>> p = dx.query(place.br3);
```

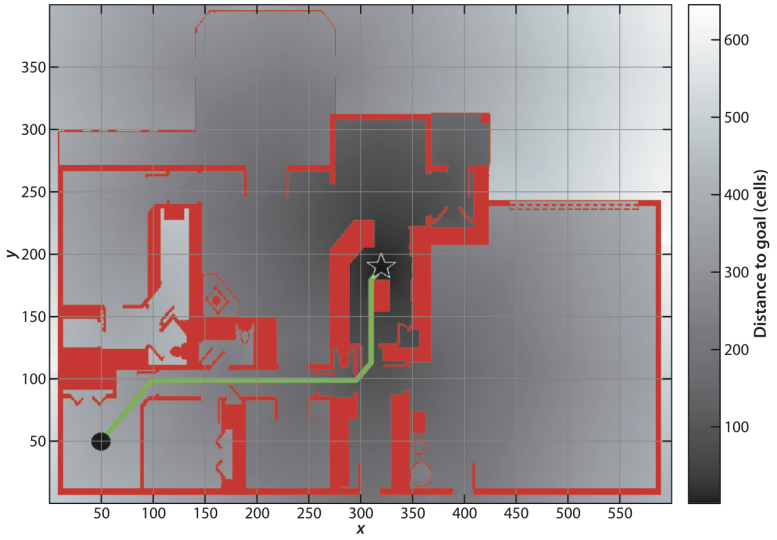


Figure 3.4: The distance transform path. Obstacles are indicated by *red cells*. The background grey intensity represents the cell's distance from the goal in units of cell size as indicated by the *scale* on the right-hand side

is returned as a matrix, one row per point.

This navigation algorithm has exploited its global view of the world and has, through exhaustive computation, found the shortest possible path. In contrast, *bug2* without the global view has just bumped its way through the world.

The penalty for achieving the optimal path is computational cost. This particular implementation of the distance transform is iterative. Each iteration has a cost of $O(N^2)$ and the number of iterations is at least $O(N)$, where N is the dimension of the map. We can visualize the iterations of the distance transform by:

```
>> dx.plan(place.kitchen, 'animate');
```

which shows the distance values propagating as a wavefront outward from the goal. The wavefront moves outward, and spills through doorways into adjacent rooms and outside the house. Although the plan is expensive to create, once it has been created it can be used to plan a path from any initial point to that goal. We have converted a fairly complex planning problem into one that can now be handled by a Braitenberg-class robot that makes local decisions based on the distance to the goal. Effectively the robot is

rolling *downhill* on the distance function which we can plot as a 3D surface:

```
>> dx.plot3d(p)
```

shown in Fig. 3.5 with the robot's path and room locations overlaid. For large occupancy grids, this approach to planning will become impractical. The roadmap methods that we discuss later in this chapter provide an effective means to find paths in large maps at greatly reduced **computational cost**. The scale associated with this occupancy grid is $4.5cm$ per cell and we have

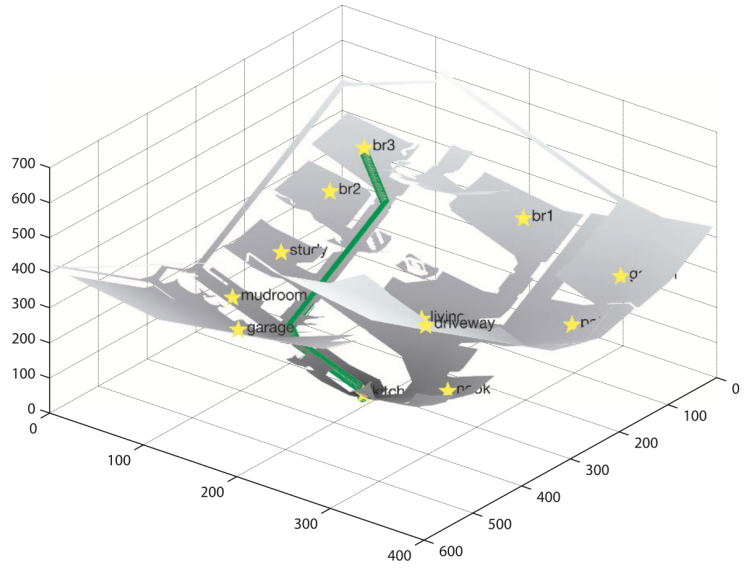


Figure 3.5: The distance transform as a 3D function, where height is distance from the goal. Navigation is simply a downhill run. Note the discontinuity in the distance transform where the split wavefronts met

assumed the robot occupies a single grid cell – this is a very small robot. The planner could therefore find paths that a larger real robot would be unable to fit through. A common solution to this problem is to *inflate* the occupancy grid – making the obstacles bigger is equivalent to leaving the obstacles unchanged and making the robot bigger. For example, if we inflate the obstacles by 5 cells:

```
>> dx = DXform(house, 'inflate', 5);
```

3.2.2 D* Algorithm

A popular algorithm for robot path planning is D* which it's an extension of the A* algorithm and which **finds the best path through a graph, which it first computes, that corresponds to the input occupancy grid**. D* has a number of features that are useful for real-world applications.

Firstly, it generalizes the occupancy grid to a cost map which represents the cost $c \in \mathbb{R}$, $c > 0$ of traversing each cell in the horizontal or vertical direction. The cost of traversing the cell diagonally is $c\sqrt{2}$. For cells corresponding to obstacles $c = \infty$ (Inf in MATLAB). Secondly, **D* supports incremental replanning**. This is important if, while we are moving, we discover that the world is different from our map. If we discover that a route has a higher-than-expected cost or is completely blocked we can incrementally replan to find a better path. The incremental replanning has a lower computational cost than completely replanning as would be required using the distance transform method just discussed.

D* finds the path that minimizes the total cost of travel. If we are interested in the shortest time to reach the goal then cost is the time to drive across the cell and it is inversely related to traversability. If we are interested in minimizing damage to the vehicle or maximizing passenger comfort then cost might be related to the roughness of the terrain within the cell. The costs assigned to cells will also depend on the characteristics of the vehicle: a large 4-wheel drive vehicle may have a finite cost to cross a rough area whereas for a small car that cost might be infinite. To implement the D* planner using the Toolbox we use a similar pattern and first create a D* navigation object:

```
>> ds = Dstar(house);
```

The D* planner converts the passed occupancy grid `map` into a cost map which we can retrieve:

```
>> c = ds.costmap();
```

where the elements of c will be 1 or ∞ representing free and occupied cells respectively. A plan for moving to the goal is generated by:

```
>> ds.plan(place.kitchen);
```

which creates a very dense directed graph. *Every cell is a graph vertex and has a cost, a distance to the goal, and a link to the neighboring cell that is closest to the goal.* Each cell also has a state $t \in \{\text{NEW, OPEN, CLOSED}\}$. Initially every cell is in the NEW state, the cost of the goal cell is zero and its state is OPEN. We can consider the set of all cells in the OPEN state as a wavefront propagating outward from the goal. The cost of reaching cells that are neighbors of an OPEN cell is computed and these cells in turn are set to OPEN and the original cell is removed from the open list and becomes CLOSED. In MATLAB this initial planning phase is quite slow and takes over a minute and you can get the iterations of the planning loop:

```
>> ds.niter
```

The path from an arbitrary starting point to the goal

```
>> ds.query(place.br3);
```

is shown in Fig. 3.6. The robot has again taken a short and efficient path around the obstacles that is almost identical to that generated by the distance transform. *The real power of D* comes from being able to efficiently change*

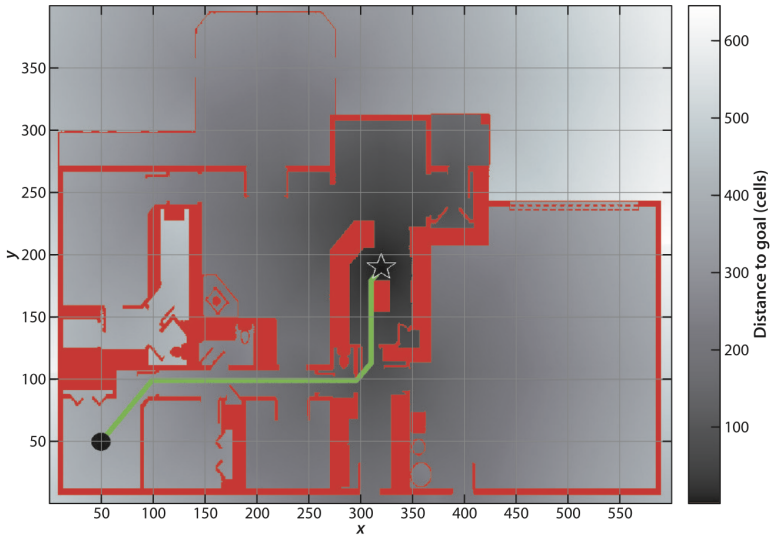


Figure 3.6: The D* planner path. Obstacles are indicated by *red cells* and all driveable cells have a cost of 1. The background grey intensity represents the cell's distance from the goal in units of cell size as indicated by the *scale* on the right-hand side

the cost map during the mission. This is actually quite a common requirement in robotics since real sensors have a finite range and a robot discovers more of world as it proceeds. We inform D* about changes using the `modify_cost` method, for example to raise the cost of entering the kitchen via the bottom doorway:

```
>> ds.modify_cost( [300,325; 115,125], 5 );
```

we have raised the cost to 5 for a small rectangular region across the doorway. The other driveable cells have a default cost of 1. The plan is updated by invoking the planning algorithm again:

```
>> ds.plan();
```

and this time the number of iterations is only the 70% of that required to create the original plan. The new path for the robot is shown in Fig. 3.7. The cost change is relatively small but we notice that the increased cost of driving within this region is indicated by a subtle brightening of those cells – in a cost sense these cells are now further from the goal. Compared to Fig. 3.6 the robot has taken a different route to the kitchen and avoided the bottom door. D* allows updates to the map to be made at any time while the robot is moving. After re-planning the robot simply moves to the adjacent cell with the lowest cost which ensures continuity of motion even if the plan has changed.

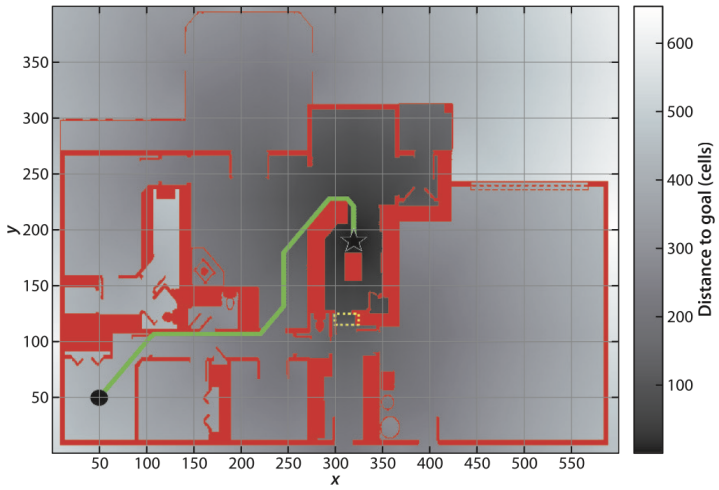


Figure 3.7: Path from D* planner with modified map.

3.2.3 Introduction to roadmap Methods

In robotic path planning the analysis of the map is referred to as the planning phase. The query phase uses the result of the planning phase to find a path from A to B. The two previous planning algorithms, distance transform and D*, require a significant amount of computation for the planning phase, but the query phase is very cheap. However, the plan depends on the goal. If the goal changes the expensive planning phase must be re-executed. Even though D* allows the path to be recomputed as the costmap changes it does not support a changing goal. **The disparity in planning and query costs has led to the development of roadmap methods where the query can include both the start and goal positions**. The planning phase provides analysis that supports changing starting points and changing goals. *A good analogy is making a journey by train.* We first find a local path to the nearest train station, travel through the train network, get off at the station closest to our goal, and then take a local path to the goal. The train network is invariant and planning a path through the train network is straightforward. Planning paths to and from the entry and exit stations respectively is also straightforward since they are, ideally, short paths. The robot navigation problem then becomes one of building a network of obstacle-free paths through the environment which serves the function of the train network. In the literature, such a network is referred to as a roadmap. The roadmap needs only to be computed once and can then be used like the train network to get us from any start location to any goal location. We will illustrate the principles by creating a roadmap from the occupancy grid's free space using some image processing techniques. The essential steps in creating the roadmap are shown in Fig. 3.8. The first step is to find the free space in the map which is simply the complement of the occupied space:

```
>> free = 1 - house
```

and is a matrix with nonzero elements where the robot is free to move. The boundary is also an obstacle so we mark the outermost cells as being not free

```
>> free(1,:) = 0; free(end,:) = 0;
>> free(:,1) = 0; free(:,end) = 0;
```

and this map is shown in Fig. 3.8a where free space is depicted as white. The topological skeleton of the free space is computed by a morphological image processing algorithm known as thinning applied to the free space of Fig. 3.8a

```
>> skeleton = ithin(free);
```

and the result is shown in Fig. 3.8b. We see that the obstacles have grown and the free space, the white cells, has become a thin network of connected white cells that are equidistant from the boundaries of the original obstacles. Figure 3.8 shows the free space network overlaid on the original map. We have created a network of paths that span the space and which can be used for obstacle-free travel around the map.

These paths are the edges of a ***generalized Voronoi diagram***. We could obtain a similar result by computing the distance transform of the obstacles, Fig. 3.8a, and this is shown in Fig. 3.8d. The value of each pixel is the distance to the nearest obstacle and the ridge lines correspond to the skeleton of Fig. 3.8b. Thinning or skeletonization, like the distance transform, is a computationally expensive iterative algorithm but it illustrates well the principles of finding paths through free space. In the next section, we will examine a cheaper alternative.

The Voronoi tessellation (Fig.3.9) of a set of planar points, known as sites, is a set of Voronoi cells as shown to the left. Each cell corresponds to a site and consists of all points that are closer to its site than to any other site. The edges of the cells are the points that are equidistant to the two nearest sites.

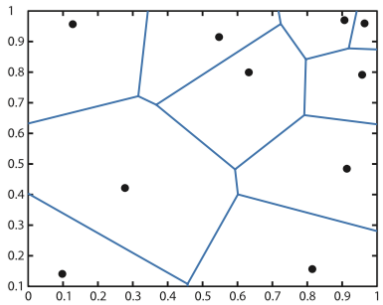


Figure 3.9: Voronoi tessellation

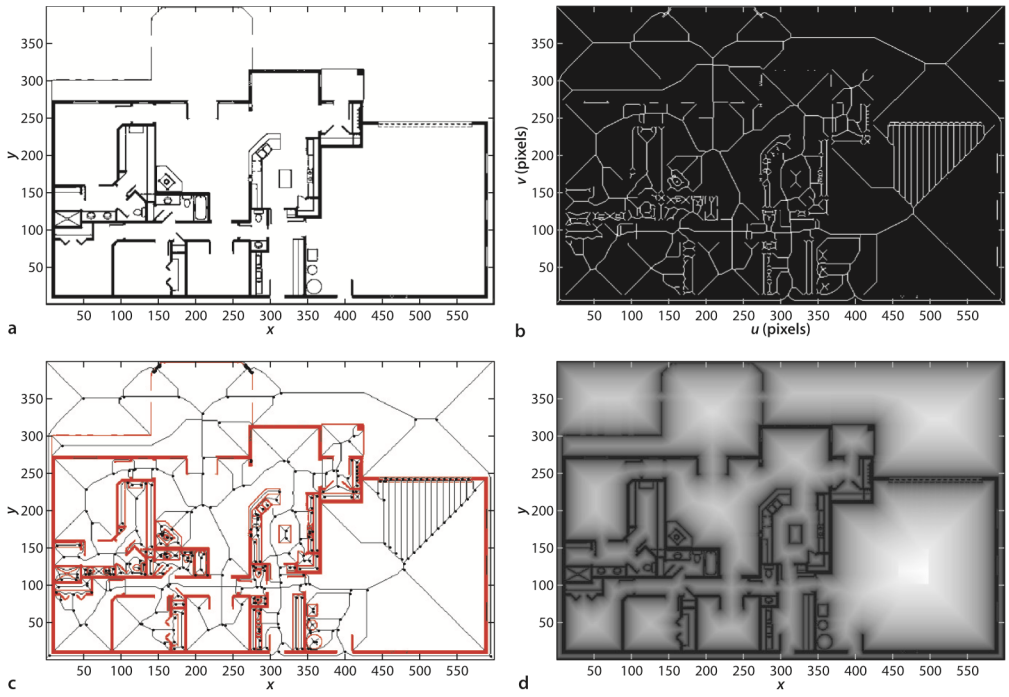


Figure 3.8: Steps in the creation of a Voronoi roadmap. **a** Free space is indicated by *white cells*; **b** the skeleton of the free space is a network of adjacent cells no more than one cell thick; **c** the skeleton with the obstacles overlaid in red and road-map junction points indicated by black dots; **d** the distance transform of the obstacles, pixel values correspond to distance to the nearest obstacle

3.2.4 Probabilistic Roadmap Method (PRM)

The high computational cost of the distance transform and skeletonization methods makes them infeasible for large maps and has led to the development of probabilistic methods. *These methods sparsely sample the world map* and the most well-known of these methods is the **probabilistic roadmap or PRM method**. To use the Toolbox PRM planner for our problem we first create a `PRM` object:

```
>> prm = PRM(house)
```

and then create the plan

```
>> prm.plan('npoints', 150)
```

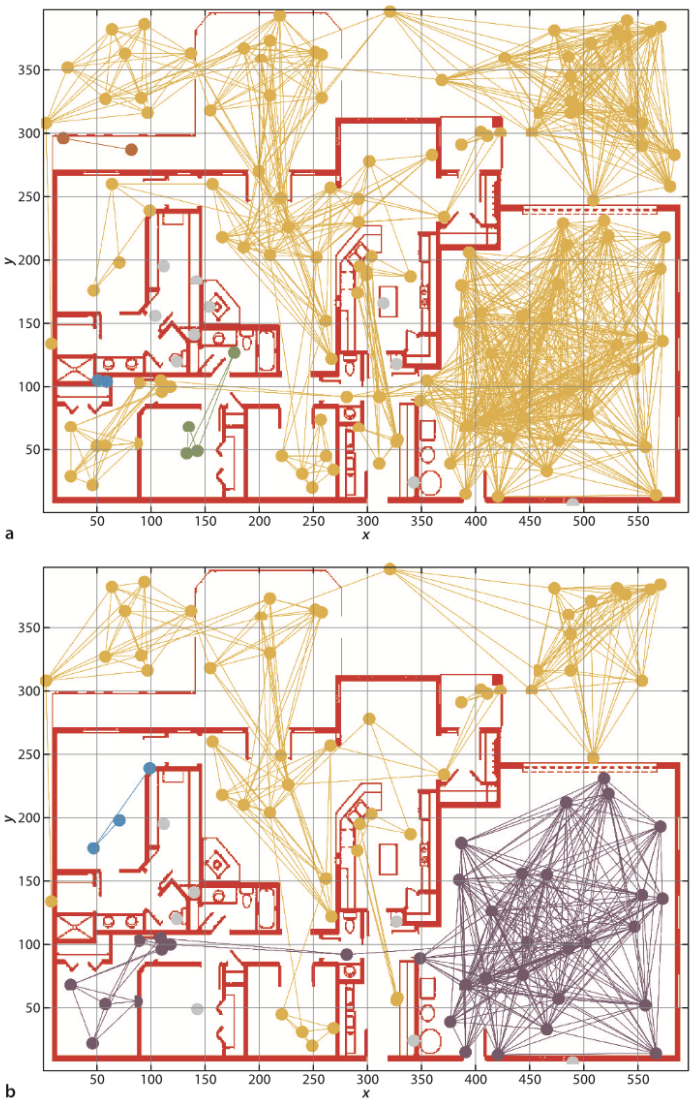


Figure 3.10: Probabilistic roadmap (PRM) planner and the random graphs produced in the planning phase. **a** Well connected network with 150 nodes; **b** poorly connected network with 100 nodes

with 150 roadmap nodes. Note that we do not pass the goal as an argument since the plan is independent of the goal.

Creating the path is a two-phase process: **planning**, and **query**. The planning phase finds N random points, 150 in this case, that lie in free space. Each point is connected to its nearest neighbors by a straight line path that does not cross any obstacles, so as to create a network, or graph, with a minimal number of disjoint components and no cycles. *The advantage of PRM is that relatively few points need to be tested to ascertain that the points and the paths between them are obstacle-free.* The resulting network is stored within the `PRM` object. which indicates the number of edges and connected components in the graph. The graph can be visualized

```
>> prm.plot()
```

as shown in Fig. 3.11a. The dots represent the randomly selected points and the lines are obstacle-free paths between the points. Only paths less than 178.8 cells long are selected which is the distance threshold parameter of the `PRM` class. Each edge of the graph has an associated cost which is the distance between its two nodes. The color of the node indicates which component it belongs to and each component is assigned a unique color. In this case, there are 14 components but the bulk of nodes belong to a single large component.

The query phase finds a path from the start point to the goal. This is simply a matter of moving to the closest node in the roadmap (the start node), following a minimum cost A^* route through the roadmap, getting off at the node closest to the goal, and then traveling to the goal. For our standard problem, this is

```
>> prm.query(place.br3, place.kitchen)
>> prm.plot()
```

The path that has been found is quite efficient although there are two areas where the path doubles back on itself. Note that we provide the start and the goal position to the query phase. An advantage of this planner is that once the roadmap is created by the planning phase we can change the goal and starting points very cheaply, only the query phase needs to be repeated. The path taken is a list of the node coordinates that the robot passes through – via points. There are some important tradeoffs in achieving

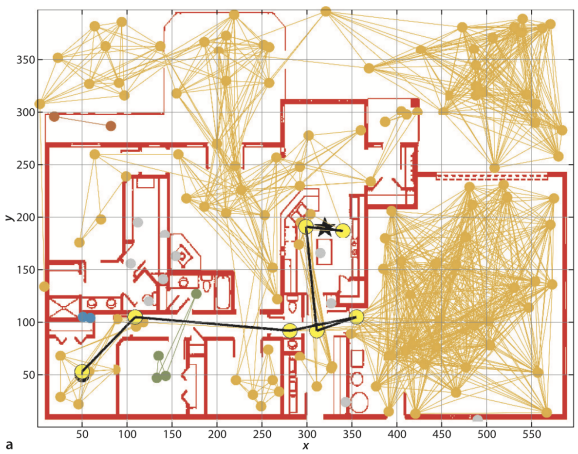


Figure 3.11: Probablistic roadmap (PRM) planner showing the path taken by the robot via nodes of the roadmap which are highlighted in yellow

this computational efficiency. Firstly, the underlying random sampling of the free space means that a different roadmap is created every time the planner is run, resulting in different paths and path lengths. Secondly, the planner can fail by creating a network consisting of disjoint components. The roadmap with only 100 nodes has several large disconnected components and the nodes in the kitchen and bedrooms belong to different components. If the start and goal nodes are not connected by the roadmap, that is, they are close to different components the query method will report an error. The only solution is to rerun the planner and/or increase the number of nodes. Thirdly, long narrow gaps between obstacles such as corridors are unlikely to be exploited since the probability of randomly choosing points that lie along such spaces is very low.

3.2.5 Lattice Planner

The planners discussed so far have generated paths independent of the motion that the vehicle can achieve, and we know that wheeled vehicles have significant motion constraints. One common approach is to use the output of the planners we have discussed and move a point along the paths at constant velocity and then follow that point. An alternative is to design a path from the outset that we know the vehicle can follow. The next two planners that we introduce take into account the **motion model of the**

vehicle, and relax the assumption we have so far made that the robot is capable of omnidirectional motion. We consider that the robot is moving between discrete points in its 3-dimensional configuration space. The robot is initially at the origin and can drive forward to the three points shown in black in Fig. 3.12a. Each path is an arc that requires a constant steering wheel setting and the arc radius is chosen so that at the end of each arc the robot's heading direction is some multiple of $\pi/2$ radians. At the end of each branch, we can add the same set of three motions suitably rotated and translated, and this is shown in Fig. 3.12b. The graph now contains 13 nodes and represents 9 paths each 2 segments long. We can create this lattice by using the `Lattice` planner class

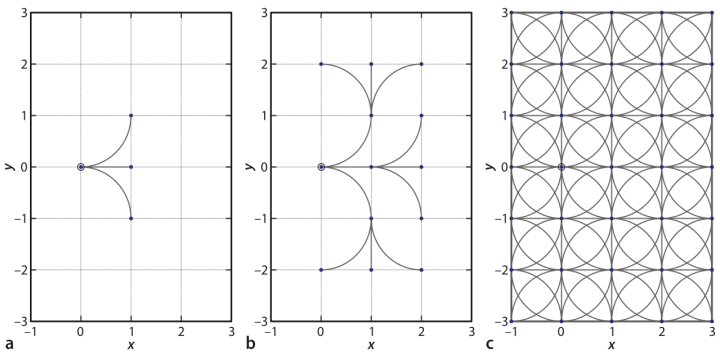


Figure 3.12: Lattice plan after 1, 2 and 8 iterations

```
>> lp = Lattice();
>> lp.plan('iterations', 2);
>> lp.plot()
```

which will generate a plot like Fig. 3.12b. Each node represents a configuration (x, y, θ), not just a position, and if we rotate the plot we can see in Fig. 3.13 that the paths lie in the 3-dimensional configuration space. While the paths appear smooth and continuous the curvature is in fact discontinuous – at some nodes, the steering wheel angle would have to change instantaneously from hard left to hard right for example. By increasing the number of iterations

```
>> lp.plan('iterations', 8)
```

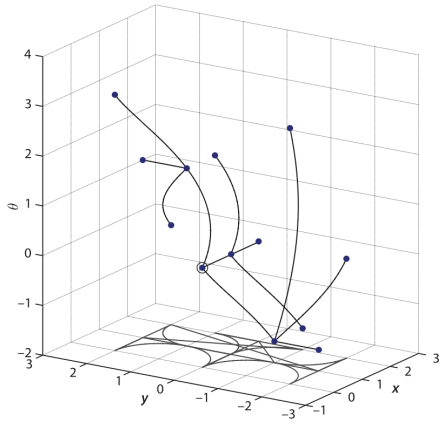


Figure 3.13: Lattice plan after 1, 2 and 8 iterations

We can fill in more possible paths as shown in Fig. 3.12c and the paths now extend well beyond the area shown. Now that we have created the lattice we can compute a path between any two nodes using the query method

```
>> lp.query([1 2 pi/2], [2 -2 0]);
```

where the start and goal are specified as configurations (x, y, θ) and the lowest cost path found by an A* search is reported: We can overlay this on the vertices and is shown in Fig. 3.14a. This is a path that takes into account the fact that the vehicle has an orientation and preferred directions of motion, as do most wheeled robot platforms. We can also access the configuration-space coordinates of the nodes, where each row represents the configuration-space coordinates (x, y, θ) of a node in the lattice along the path from start to goal configuration. Implicit in our search for the lowest cost path is the cost of traversing each edge of the graph which by default gives equal cost to the three steering options: straight ahead, turn left, and turn right. We can increase the cost associated with turning:

```
>> lp.plan('cost', [1 10 10])
```

and now we have the path shown in Fig. 3.14b which has only 3 turns compared to 5 previously. However, the path is longer – having 8 rather than 6 segments.

Consider a more realistic scenario with obstacles in the environment. Specifically, we want to find a path to move the robot 2 m in the lateral direction with its final heading angle the same as its initial heading angle:

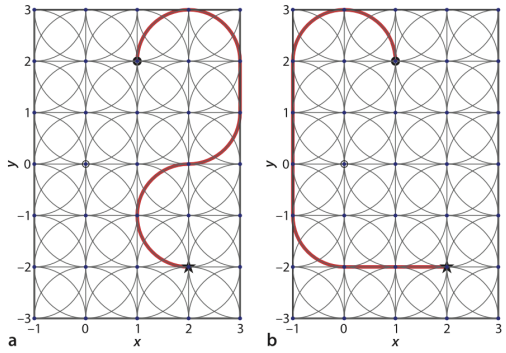


Figure 3.14: Paths over the lattice graph. **a** With uniform cost; **b** with increased penalty for turns

```
>> load road
>> lp = Lattice(road, 'grid', 5, 'root', [50 50 0]);
>> lp.plan();
```

where we have loaded an obstacle grid that represents a simple parallel-parking scenario and planned a lattice with a grid spacing of 5 units and the root node at a central obstacle-free configuration. In this case the planner continues to iterate until it can add no more nodes to the free space. We query for a path from the road to the parking spot:

```
>> lp.query([30 45 0], [50 20 0])
```

and the result is shown in Fig. 3.15. Paths generated by the lattice planner are inherently driveable by the robot but there are clearly problems driving along a diagonal with this simple lattice. The planner would generate a continual sequence of hard left and right turns which would cause undue wear and tear on a real vehicle and give a very uncomfortable ride. More sophisticated version of lattice planners are able to deal with this by using motion primitives with hundreds of arcs, such as shown in Fig. 3.16, instead of the three shown in these examples.

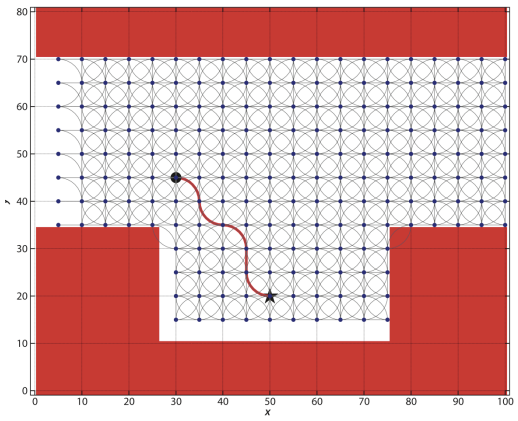


Figure 3.15: A simple parallel parking scenario based on the lattice planner with an occupancy grid (cells are 10 cm square)

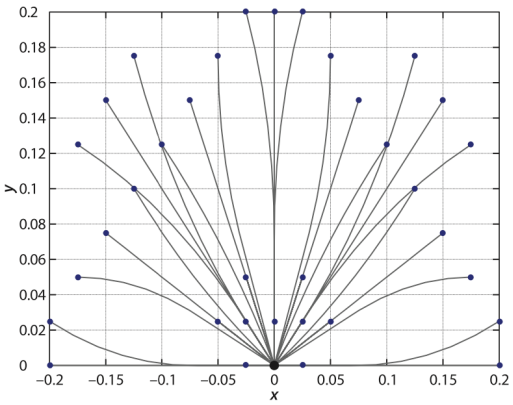


Figure 3.16: A more sophisticated lattice generated by the package **sbpl** with 43 paths based on the kinematic model of a unicycle

3.2.6 Rapidly-Exploring Random Tree (RRT)

The final planner that we introduce is also able to take into account the motion model of the vehicle. Unlike the lattice planner which plans over a regular grid, the RRT uses probabilistic methods like the PRM planner. The underlying insight is similar to that for the lattice planner and Fig. 3.17 shows a family of paths that the bycicle would follow in configuration space. The paths are computed over a fixed time interval for discrete values of velocity, forward or backward, and various steering angles. This demonstrates clearly the subset of all possible configurations that a nonholonomic vehicle can reach from a given initial configuration.

The main steps in creating an RRT are as follows, with the notation shown in the figure to the right. A graph of robot configurations is maintained and each node is a configuration $\boldsymbol{q} \in \mathbb{R}^2 \times \mathbb{S}^1$ which is represented by a 3-vector $\boldsymbol{q} \sim (x, y, \theta)$. The first, or root, node in the graph is the goal configuration of the robot. A random configuration $\boldsymbol{q}_{rand}$ is chosen, and the node with the closest configuration $\boldsymbol{q}_{near}$ is found – this configuration is near in terms of a cost function that includes distance and orientation. A control is computed that moves the robot from $\boldsymbol{q}_{near}$ toward $\boldsymbol{q}_{rand}$ over a fixed path simulation time. The configuration that it reaches is $\boldsymbol{q}_{new}$ and this is added to the graph. For any desired starting configuration we can find the closest configuration in the graph, and working backward toward the starting configuration we can determine the sequence of steering angles and velocities needed to move from the start to the goal configuration. This has some similarities to the roadmap methods discussed previously, but the limiting factor is the combinatoric explosion in the number of possible poses. We first of all create a model to describe the vehicle's kinematics

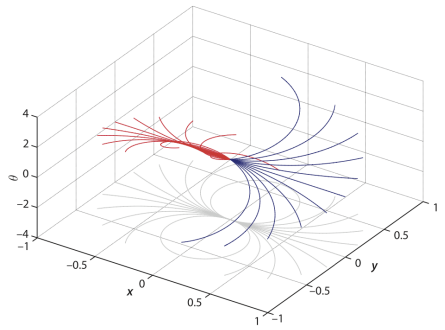


Figure 3.17: A set of possible paths that the bicycle model robot could follow from an initial configuration

```
>> car = Bicycle('steermax', 0.5);
```

and here we have specified a car-like vehicle with a maximum steering angle of 0.5 rad. Following our familiar programming pattern we create an RRT object

```
>> rrt = RRT(car, 'npoints', 1000)
```

for an obstacle free environment which by default extends from –5 to +5 in the x- and y-directions and create a plan:

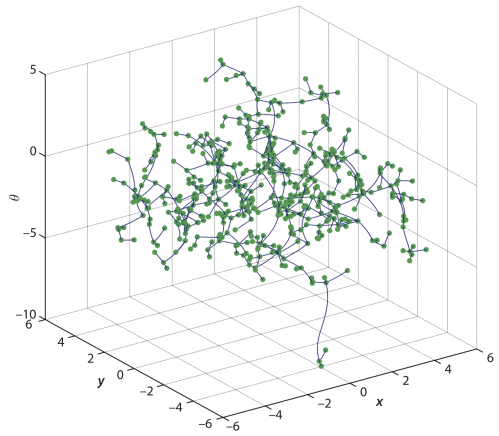


Figure 3.18: An RRT computed for the bicycle model with a velocity of ±1 m s^{-1}, steering angle limits of ±0.5 rad, integration period of 1 s, and initial configuration of (0, 0, 0). Each node is indicated by a green circle in the 3-dimension- al space of vehicle poses (x, y, θ)

```
>> rrt.plan();
>> rrt.plot();
```

The random tree is shown in Fig. 3.18 and we see that the paths have a good coverage of the configuration space, not just in the x- and y-directions but also in orientation, which is why the algorithm is known as rapidly exploring.

An important part of the RRT algorithm is **computing the control input** that moves the robot from an existing configuration in the graph to $\boldsymbol{q}_{rand}$. Driving a nonholonomic vehicle to a specified configuration can be difficult. Rather than the complex nonlinear controller we will use something simpler that fits with the randomized sampling strategy used in this class of planner. The controller randomly chooses whether to drive forwards or backwards and randomly chooses a steering angle within the limits. It then simulates motion of the vehicle model for a fixed period of time, and computes the closest distance to $\boldsymbol{q}_{rand}$. This is repeated multiple times and the control input with the best performance is chosen. The configuration on its path that was closest to $\boldsymbol{q}_{rand}$ is chosen as $\boldsymbol{q}_{near}$ and becomes a new node in the graph.

Handling obstacles with the RRT is quite straightforward. The configuration $\boldsymbol{q}_{rand}$ is discarded if it lies within an obstacle, and the point $\boldsymbol{q}_{near}$ will not

be added to the graph if the path from $\boldsymbol{q}_{near}$ toward $\boldsymbol{q}_{rand}$ and intersects an obstacle. The result is a set of paths, a roadmap, that is collision free and driveable by this nonholonomic vehicle. We will repeat the parallel parking example from the last section

```
>> rrt = RRT(car, road, 'npoints', 1000,
     'root', [50 22 0], 'simtime', 4);
>> rrt.plan();
```

where we have specified the vehicle kinematic model, an occupancy grid, the number of sample points, the location of the first node, and that each random motion is simulated for 4 seconds. We can query the RRT plan for a path between two configurations

```
>> p = rrt.query([40 45 0], [50 22 0]);
```

and the result is a continuous path which will take the vehicle from the street to the parking slot. We can overlay the path on the occupancy grid and RRT and the result is shown in Fig. 5.20 with some vehicle configurations overlaid. We can also animate the motion along the path

```
>> plot\_vehicle(p, 'box', 'size', [20 30], 'fill',
     'r', 'alpha\', 0.1)
```

where we have specified the vehicle be displayed as a red translucent shape of width 20 and length 30 units. This example illustrates some important points about the RRT. Firstly, as for the PRM planner, there may be some distance (and orientation) between the start and goal con- figuration and the nearest node. Minimizing this requires tuning RRT parameters such as the number of nodes and path simulation time. Secondly, the path is feasible but not quite optimal. In this case the vehicle has changed direction twice before driving into the parking slot. This is due to the random choice of nodes – rerunning the planner and/or increasing the number of nodes may help. Finally, we can see that the vehicle body collides with the obstacle, and this is very apparent if you view the animation. This is actually not surprising since the collision check we did when adding a node only tested if the node's position lay in an obstacle – it should properly check if a finite-sized vehicle with that configuration intersects an obstacle. Alternatively we could inflate the obstacles by the radius of the smallest disk that contains the robot.

3.3 Wrapping Up

Robot navigation is the problem of guiding a robot towards a goal and we have covered a spectrum of approaches. The simplest was the purely reactive Braitenberg-type vehicle. Then we added limited memory to create state machine based automata such as *bug2* which can deal with obstacles, however the paths that it finds are far from optimal. A number of different map-based planning algorithms were then introduced. The distance transform is a computationally intense approach that finds an optimal path to the goal. D* also finds an optimal path, but supports a more nuanced travel cost – individual cells have a continuous traversability measure rather than being considered as only free space or obstacle. D* also supports computationally cheap incremental replanning for small changes in the map. PRM reduces the computational burden significantly by probabilistic sampling but at the expense of somewhat less optimal paths. In particular it may not discover narrow routes between areas of free space. The lattice planner takes into account the motion constraints of a real vehicle to create paths which are feasible to drive, and can readily account for the orientation of the vehicle as well as its position. RRT is another random sampling method that also generates kinematically feasible paths. All the map-based approaches require a map and knowledge of the robot's location.

Chapter 4

Localization for Mobile Robots

In our discussion of map-based navigation we assumed that the robot had a means of knowing its position. In this chapter we discuss some of the common techniques used to estimate the location of a robot in the world – a process known as localization.

4.1 Dead Reckoning

Dead reckoning is the estimation of a robot's pose based on its estimated speed, direction and time of travel with respect to a previous estimate. An odometer is a sensor that measures distance traveled and sometimes also change in heading direction. For wheeled vehicles this can be determined by measuring the angular rotation of the wheels. The direction of travel can be measured using an electronic compass, or the change in heading can be measured using a gyroscope or differential odometry. These sensors are imperfect due to systematic errors such an incorrect wheel radius or gyroscope bias, and random errors such as slip between wheels and the ground. Robots without wheels, such as aerial and underwater robots, can use visual odometry

4.1.1 Modeling the Vehicle

The first step in estimating the robot's pose is to write a function, $f(\cdot)$, that describes how the vehicle's configuration (x, y, θ) changes from one time

step to the next. A vehicle model such as

$$\begin{aligned} \dot{x} &= v\, cos(\theta) \\ \dot{y} &= v\, sin(\theta) \\ \dot{\theta} &= \frac{v}{L} tan(\gamma) \end{aligned} \tag{4.1}$$

where L is the length of the vehicle and γ is the steering angle, or

$$\dot{\theta} = \frac{v_R - v_L}{W} \tag{4.2}$$

describes the evolution of the robot's configuration as a function of its control inputs, however for real robots we rarely have access to these control inputs. Most robotic platforms have proprietary motion control systems that accept motion commands from the user (speed and direction) and report odometry information. Instead of using Eq. 4.1 or 4.2 directly we will write a **discrete-time model for the *evolution of configuration* based on odometry** where $\delta\langle k\rangle = (\delta_d,\ \delta_\theta)^T$ is the distance traveled and change in heading direction over the preceding interval, and k is the time step. The initial pose is represented in $\boldsymbol{SE(2)}$ as

$$\xi\langle k\rangle \sim \begin{bmatrix} cos(\theta\langle k\rangle) & -sin(\theta\langle k\rangle) & x\langle k\rangle \\ sin(\theta\langle k\rangle) & cos(\theta\langle k\rangle) & y\langle k\rangle \\ 0 & 0 & 1 \end{bmatrix}$$

We make a simplifying assumption that motion over the time interval is *small* so the order of applying the displacements is not significant.
Let's consider to move forward the vehicle in the x-direction by δ_d, and then rotate it by δ_θ giving the new configuration

$$\xi\langle k\rangle \sim \begin{bmatrix} cos(\theta\langle k\rangle) & -sin(\theta\langle k\rangle) & x\langle k\rangle \\ sin(\theta\langle k\rangle) & cos(\theta\langle k\rangle) & y\langle k\rangle \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \delta_d \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos(\delta_\theta) & -sin(\delta_\theta) & 0 \\ sin(\delta_\theta) & cos(\delta_\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\sim \begin{bmatrix} cos(\theta\langle k\rangle + \delta_\theta) & -sin(\theta\langle k\rangle + \delta_\theta) & x\langle k\rangle + \delta_d cos(\theta\langle k\rangle) \\ sin(\theta\langle k\rangle + \delta_\theta) & cos(\theta\langle k\rangle + \delta_\theta) & y\langle k\rangle + \delta_d sin(\theta\langle k\rangle) \\ 0 & 0 & 1 \end{bmatrix}$$

which we can represent concisely as a 3-vector $\boldsymbol{x} = [x, y, \theta]^T$

$$\boldsymbol{x}\langle k+1\rangle = \begin{bmatrix} x\langle k\rangle + \delta_d cos(\theta\langle k\rangle) \\ y\langle k\rangle + \delta_d sin(\theta\langle k\rangle) \\ \theta\langle k\rangle + \delta_\theta \end{bmatrix} \tag{4.3}$$

which gives the new configuration in terms of the previous configuration and the odometry.

In practice odometry is not perfect and we model the error by imagining a random number generator that corrupts the output of a perfect odometer. The measured output of the real odometer is the perfect, but unknown, odometry (δ_d, δ_θ) plus the output of the random number generator (v_d, v_θ). Such random errors are often referred to as noise, or more specifically as sensor noise. The random numbers are not known and cannot be measured, but we assume that we know the distribution from which they are drawn. The *robot's configuration at the next time step, including the odometry error*, is:

$$\boldsymbol{x}\langle k\rangle = f(\boldsymbol{x}\langle k\rangle, \delta\langle k\rangle, v\langle k\rangle) = \begin{bmatrix} x\langle k\rangle + (\delta_d + v_d) cos(\theta\langle k\rangle) \\ y\langle k\rangle + (\delta_d + v_d) sin(\theta\langle k\rangle) \\ \theta\langle k\rangle + \delta_\theta + v_\theta \end{bmatrix} \tag{4.4}$$

where k is the time step, $\delta\langle k\rangle = (\delta_d, \delta_\theta)^T \in \mathbb{R}^{2\times 1}$ is the odometry measurement and $\boldsymbol{v}\langle k\rangle = (v_d, v_\theta)^T \in \mathbb{R}^{2\times 1}$ is the random measurement noise over the preceding interval. In the abscence of any information to the contrary we model the odometry noise as $\boldsymbol{v} = (v_d, v_\theta)^T \sim N(0, V)$, a zero-mean multivariate Gaussian process with variance matrix:

$$V = \begin{bmatrix} \sigma_d^2 & 0 \\ 0 & \sigma_d^2 \end{bmatrix}$$

This constant matrix, the covariance matrix, is diagonal which means that the errors in distance and heading are *independent*. Choosing a value for V is not always easy but we can conduct experiments or make some reasonable engineering assumptions. In the examples which follow, we choose $\sigma_d = 2cm$ and $\sigma_\theta = 0.5°$ per sample interval which leads to a covariance matrix of

```
>> V = diag([0.02, 0.5*pi/180].^2);
```

All objects of the Toolbox `Vehicle` superclass provide a method `f()` that implements the appropriate odometry update equation. For the case of a vehicle with bicycle kinematics that has the motion model of Eq. 4.1 and the odometric update Eq. 4.4, we create a `Bicycle` object

```
>> veh = Bicycle('covar', V);
```

The object provides a method to simulate motion over one time step:

```
>> odo = veh.step(1, 0.3)
```

where we have specified a speed of 1 m s^{-1} and a steering angle of 0.3 rad.

If you find a problem with the Toolbox code, you need to go to `Home` and then open "`Set Path`" and to find at the bottom of the list the folder that ends as "`.../spatial-math`". Select it and move it to the top. This operation should solve the problems

---

The function updates the robot's true configuration and returns a noise corrupted odometer reading. With a sample interval of 0.1 s the robot reports that is moving approximately 0.1 m each interval and changing its heading by approximately 0.03 rad. The robot's true (but 'unknown') configuration can be seen by

```
>> veh.x'
```

Given the reported odometry we can estimate the configuration of the robot after one time step using Eq. 4.4 which is implemented by

```
>> veh.f([0 0 0], odo)
```

where the discrepancy with the exact value is due to the use of a noisy odometry measurement. For the scenarios that we want to investigate we require the simulated robot to drive for a long time period within a defined spatial region. The `RandomPath` class is a *driver* that steers the robot to randomly selected waypoints within a specified region. We create an instance of the driver object and connect it to the robot

```
>> veh.add_driver(RandomPath(10))
```

where the argument to the `RandomPath` constructor specifies a working region that spans ±10 m in the x- and y-directions. We can display an animation of the robot with its driver by

```
>> veh.run()
```

which repeatedly calls the `step` method and maintains a history of the true state of the vehicle over the course of the simulation within the `Bicycle` object. The `RandomPath` and `Bicycle` classes have many parameters and methods which are described in the online documentation.

4.1.2 Estimating Pose

The problem we face is how to estimate our new pose given the previous pose and noisy odometer. We want the best estimate of where we are and how certain we are about that. The mathematical tool that we will use is the **Kalman filter** that provides the optimal estimate of the system state, vehicle configuration in this case, assuming that the noise is zero-mean and Gaussian.

The filter is a recursive algorithm that updates, at each time step, the optimal estimate of the unknown true configuration and the uncertainty associated with that estimate based on the previous estimate and noisy measurement data. The Kalman filter is formulated for linear systems but our model of the vehicle's motion Eq. 4.4 is nonlinear – the tool of choice is the **Extended Kalman filter (EKF)**. For this problem the state vector is the vehicle's configuration

$$\boldsymbol{x} = \begin{bmatrix} x_v \\ y_v \\ \theta_v \end{bmatrix}$$

and the predictions equations

$$\hat{\boldsymbol{x}}^+\langle k+1\rangle = \boldsymbol{f}(\hat{\boldsymbol{x}}\langle k\rangle), \hat{\boldsymbol{u}}\langle k\rangle) \tag{4.5}$$

$$\hat{\boldsymbol{P}}^+\langle k+1\rangle = \boldsymbol{F}_x\hat{\boldsymbol{P}}\langle k\rangle\hat{\boldsymbol{F}}_x^T + \boldsymbol{F}_v\hat{\boldsymbol{V}}\boldsymbol{F}_v^T \tag{4.6}$$

describe how the state and covariance evolve with time. The term $\hat{\boldsymbol{x}}^+\langle k+1\rangle$ indicates an estimate of $\boldsymbol{x}$ at time $k+1$ based on information up to, and including, time k. $\hat{\boldsymbol{u}}\langle k\rangle$ is the input to the process, which in this case is

the measured odometry, so $\hat{\boldsymbol{u}}\langle k\rangle = \delta\langle k\rangle$. $\hat{\boldsymbol{P}} \in \mathbb{R}^{3\times3}$ is a covariance matrix representing uncertainty in the estimated vehicle configuration. $\hat{\boldsymbol{V}}$ is our estimate of the covariance of the odometry noise which in reality we do not know. F_x and F_v are Jacobian matrices. They are obtained by differentiating Eq. 4.4 and evaluating the result at $v = 0$ giving

$$\boldsymbol{F}_x = \frac{\partial \boldsymbol{f}}{\partial x}\Big|_{v=0} = \begin{bmatrix} 1 & 0 & -\delta_d sin(\theta_v) \\ 0 & 1 & \delta_d cos(\theta_v) \\ 0 & 0 & 1 \end{bmatrix} \tag{4.7}$$

$$\boldsymbol{F}_v = \frac{\partial \boldsymbol{f}}{\partial v}\Big|_{v=0} = \begin{bmatrix} cos(\theta_v) & 0 \\ sin(\theta_v) & 0 \\ 0 & 1 \end{bmatrix} \tag{4.8}$$

which are functions of the current state and odometry. All objects of the `Vehicle` superclass provide methods `Fx` and `Fv` to compute these Jacobians, for example

```
>> veh.Fx([0,0,0], [0.5, 0.1])
```

where the first argument is the state at which the Jacobian is computed and the second is the odometry. To simulate the vehicle and the EKF using the Toolbox we define the initial covariance to be quite small since, we assume, we have a good idea of where we are to begin with

```
>> P0 = diag([0.005, 0.005, 0.001].^2);
```

and we pass this to the constructor for an `EKF` object

```
>> ekf = EKF(veh, V, P0);
```

Running the filter for 1000 time steps

```
>> ekf.run(1000);
```

drives the robot as before, along a random path. At each time step the filter updates the state estimate using various methods provided by the `Vehicle` superclass. We can plot the true path taken by the vehicle, stored within the `Vehicle` superclass object, by

```
>> veh.plot_xy()
```

and the filter's estimate of the path stored within the EKF object.

```
>> hold on
>> ekf.plot_xy('r')
```

These are shown in Fig. 4.1 and we see some divergence between the true and estimated robot path. The covariance at the 700th time step is

```
>> P700 = ekf.history(700).P
```

The matrix is symmetric and the diagonal elements are the estimated variance associated with the states, that is σ_x^2, σ_y^2 and σ_θ^2 respectively. The standard deviation σ_x of the PDF associated with the vehicle's x-coordinate is

```
>> sqrt(P700(1,1))
```

There is a 95% chance that the robot's x-coordinate is within the $\pm 2\sigma$ bound or ± 2.75 m in this case. We can compute uncertainty for y and θ similarly. The off-diagonal terms are correlation coefficients and indicate that the uncertainties between the corresponding variables are related. For example the value $P_{1,3} = P_{3,1} = -0.5575$ indicates that the uncertainties in x and θ are related – error in heading angle causes error in x-position and vice versa. Conversely new information about θ can be used to correct θ as well as x. The uncertainty in position is described by the top-left 2×2 covariance submatrix of $\hat{\boldsymbol{P}}$. This can be interpreted as an ellipse defining a confidence bound on position. We can overlay such ellipses on the plot by

```
>> ekf.plot_ellipse('g')
```

as shown in Fig. 4.1. These correspond to the default 95% confidence bound and are plotted by default every 20 time steps. The vehicle started at the origin and as it progresses we see that the ellipses become larger as the estimated uncertainty increases. The ellipses only show x- and y-position but uncertainty in θ also grows. The total uncertainty, position and heading, is given by $\sqrt{det(\hat{\boldsymbol{P}})}$ and is plotted as a function of time and we observe that it never decreases. This is because the second term in Eq. 4.6 is positive definite which means that $\boldsymbol{P}$, the position uncertainty, can never decrease.

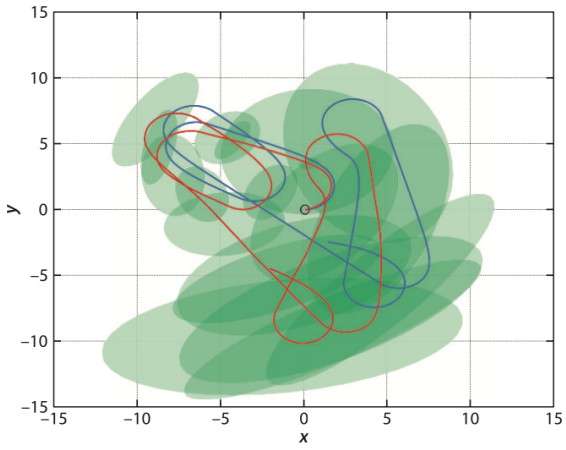


Figure 4.1: Deadreckoning using the EKF. The true path of the robot, *blue*, and the path estimated from odometry in *red*. 95% confidence ellipses are indicated in *green*. The robot starts at the origin

4.2 Localizing with a map

We have seen how uncertainty in position grows without bound using dead-reckoning alone. The solution is to bring in **additional information from observations of known features in the world**. In the examples that follow we will use a map that contains N fixed but randomly located landmarks whose positions are known. The Toolbox supports a `LandmarkMap` object

```
>> map = LandmarkMap(20, 10)
```

that in this case contains $N = 20$ landmarks uniformly randomly spread over a region spanning ± 10 m in the x- and y-directions and this can be displayed by

```
>> map.plot()
```

The robot is equipped with a sensor that provides observations of the landmarks *with respect to the robot* as described by

$$\boldsymbol{z} = h(\boldsymbol{x}, \boldsymbol{p}_i) \tag{4.9}$$

where $\boldsymbol{x} = (x_v, y_v, \theta_v)^T$ is the vehicle state, and $\boldsymbol{p}_i = (x_i, y_i)^T$ is the known location of the i^{th} landmark in the world frame. To make this tangible we will consider a common type of sensor that measures the range and bearing angle

to a landmark in the environment, for instance a radar or a scanning-laser rangefinder. The sensor is mounted on-board the robot so the observation of the i^{th} landmark is

$$\boldsymbol{z} = h(\boldsymbol{x}, \boldsymbol{p}_i) = \begin{bmatrix} \sqrt{(y_i - y_v)^2 + (x_i - x_v)^2} \\ tan^{-1}(y_i - y_v)/(x_i - x_v) - \theta_v \end{bmatrix} + \begin{bmatrix} w_r \\ w_\beta \end{bmatrix} \tag{4.10}$$

where $\boldsymbol{z} = (r, \beta)^T$ and r is the range, β the bearing angle, and $\boldsymbol{w} = (w_r, w_\beta)^T$ is a zero-mean Gaussian random variable that models errors in the sensor

$$\begin{bmatrix} w_r \\ w_\beta \end{bmatrix} \sim N(0, W), \quad W = \begin{bmatrix} \sigma_r^2 & 0 \\ 0 & \sigma_\beta^2 \end{bmatrix}$$

The constant diagonal covariance matrix indicates that range and bearing errors are independent. For this example we set the sensor uncertainty to be $\theta_r = 0.1$m and $\sigma_\beta = 1^\circ$ giving a sensor covariance matrix

```
>> W = diag([0.1, 1*pi/180].\string^2);
```

We model this type of sensor with a `RangeBearingSensor` object

```
>> sensor = RangeBearingSensor(veh, map, 'covar', W)
```

which is connected to the vehicle and the map, and the sensor covariance matrix $\boldsymbol{W}$ is specified along with the maximum range and the bearing angle limits. The reading method provides the range and bearing to a randomly selected visible landmark along with its identity, for example

```
>> [z,i] = sensor.reading()
```

The identity is an integer $i \in [1, 20]$ since the map was created with 20 landmarks. We have avoided the data association problem by assuming that we know the identity of the sensed landmark.

Using Eq. 4.10 the robot can estimate the range and bearing angle of the landmark based on its own estimated position and the known position of the landmark from the map. Any difference between the observation $z^\#$ and the estimated observation indicates an error in the robot's pose estimate $\hat{\boldsymbol{x}}$ – it isn't where it thought it was. However this *difference*

$$\nu = z^\#\langle k+1 \rangle - \boldsymbol{h}(\hat{\boldsymbol{x}}^+\langle k+1 \rangle, \boldsymbol{p}_i) \tag{4.11}$$

has real value and is key to the operation of the Kalman filter. It is called the **innovation** since it represents *new* information. The Kalman filter uses the innovation to correct the state estimate and update the uncertainty estimate in an optimal way. The predicted state computed earlier using Eq. 4.5 and Eq. 4.6 is updated by

$$\hat{\boldsymbol{x}}\langle k+1\rangle = \hat{\boldsymbol{x}}^{+}\langle k+1\rangle + \boldsymbol{K}\boldsymbol{\nu} \tag{4.12}$$

$$\hat{\boldsymbol{P}}\langle k+1\rangle = \hat{\boldsymbol{P}}^{+}\langle k+1\rangle - \boldsymbol{K}\boldsymbol{H}_x\hat{\boldsymbol{P}}^{+}\langle k+1\rangle \tag{4.13}$$

which are the Kalman filter update equations. These take the predicted values for the next time step denoted with the $^+$ and compute the optimal estimate by applying landmark measurements from time step $k+1$. The innovation is added to the estimated state after multiplying by the Kalman gain matrix $\boldsymbol{K}$ which is defined as

$$\boldsymbol{K} = \boldsymbol{P}^{+}\langle k+1\rangle \boldsymbol{H}_x^T \boldsymbol{S}^{-1} \tag{4.14}$$

$$\boldsymbol{S} = \boldsymbol{H}_x \boldsymbol{P}^{+}\langle k+1\rangle \boldsymbol{H}_x^T + \boldsymbol{H}_{\boldsymbol{w}}\hat{\boldsymbol{W}}\boldsymbol{H}_{\boldsymbol{w}}^{\boldsymbol{T}} \tag{4.15}$$

where $\hat{\boldsymbol{W}}$ is the estimated covariance of the sensor noise and $\boldsymbol{H}_x$ and $\boldsymbol{H}_w$ are Jacobians obtained by differentiating Eq. 4.10 yielding

$$\boldsymbol{H}_x = \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}\Big|_{w=0} = \begin{bmatrix} -\frac{x_i - x_v}{r} & -\frac{y_i - y_v}{r} & 0 \\ \frac{y_i - y_v}{r^2} & -\frac{x_i - x_v}{r^2} & -1 \end{bmatrix} \tag{4.16}$$

which is a function of landmark position, vehicle pose and landmark range; and

$$\boldsymbol{H}_w = \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{w}}\Big|_{w=0} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.17}$$

The `RangeBearingSensor` object above includes methods `h` to implement Eq. 4.10 and $\boldsymbol{H}_x$ and $\boldsymbol{H}_2$ to compute these Jacobians respectively. The Kalman gain matrix $\boldsymbol{K}$ in Eq. 4.12 distributes the innovation from the landmark observation, a 2-vector, to update every element of the state vector – the position and orientation of the vehicle. Note that the second term in Eq. 4.13 is subtracted from the estimated covariance and this provides a means for covariance to decrease which was not possible for the dead-reckoning case of Eq. 4.6. The EKF comprises two phases: prediction and update, and these are summarized in Fig. 4.3. We now have all the piece to build an

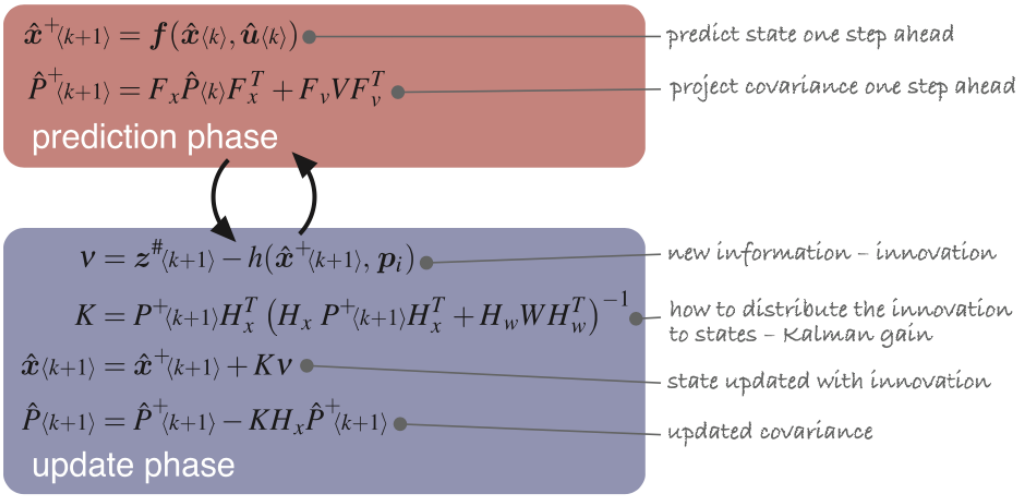


Figure 4.2: Summary of extended Kalman filter algorithm showing the prediction and update phases

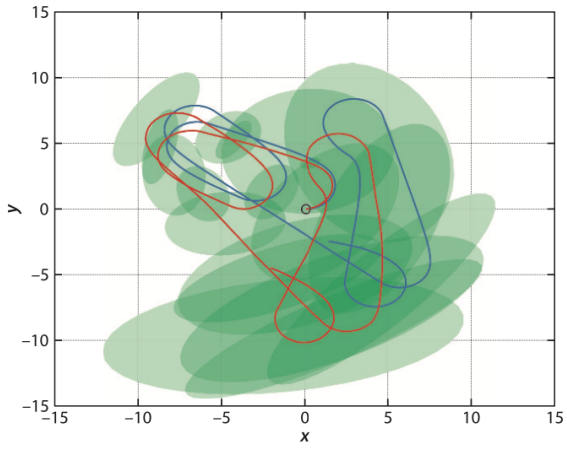


Figure 4.3: Summary of extended Kalman filter algorithm showing the prediction and update phases

estimator that uses odometry and observations of map features. The Toolbox implementation is

```
>> map = LandmarkMap(20);
>> veh = Bicycle('covar', V);
>> veh.add_driver(RandomPath(map.dim));
>> sensor = RangeBearingSensor(veh, map, 'covar', W, 'angle',
   [-pi/2 pi/2], 'range', 4, 'animate');
>> ekf = EKF(veh, V, P0, sensor, W, map);
```

The `LandmarkMap` constructor has a default map dimension of ±10 m which is accessed by its `dim` property. Running the simulation for 1000 time steps

```
>> ekf.run(1000);
```

shows an animation of the robot moving and observations being made to the landmarks. We plot the saved results

```
>> map.plot()
>> veh.plot_xy();
>> ekf.plot_xy('r');
>> ekf.plot_ellipse('k')
```

which are shown in Fig. 4.4a. The error ellipses are now much smaller and many can hardly be seen. Figure 4.4b shows a zoomed view of the robot's actual and estimated path – the robot is moving from top to bottom. We can see the error ellipses growing as the robot moves and then shrinking, just after a jag in the estimated path. This corresponds to the observation of a landmark. New information, beyond odometry, has been used to correct the state in the Kalman filter update phase.

The overall uncertainty is no longer growing monotonically. When the robot sees a landmark it is able to dramatically reduce its estimated covariance. The error associated with each component of the pose and the pink background is the estimated 95% confidence bound (derived from the covariance matrix) and we see that the error is mostly within this envelope. Below this is plotted the landmark observations and we see that the confidence bounds are tight (indicating low uncertainty) while landmarks are being observed but that they start to grow once observations stop. However, as soon as an observation is made the uncertainty rapidly decreases.

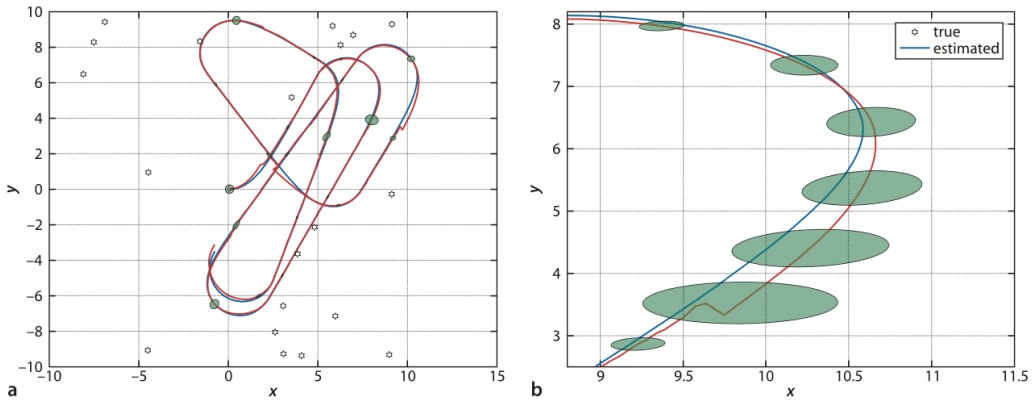


Figure 4.4: Summary of extended Kalman filter algorithm showing the prediction and update phases

This EKF framework allows data from many and varied sensors to update the state which is why the estimation problem is also referred to as **sensor fusion**. For example, heading angle from a compass, yaw rate from a gyroscope, target bearing angle from a camera, position from GPS could all be used to update the state. For each sensor we need only to provide the observation function $\boldsymbol{h}(\cdot)$, the Jacobians $\boldsymbol{H}_x$ and $\boldsymbol{H}_w$ and some estimate of the sensor covariance $\boldsymbol{W}$. The function $\boldsymbol{h}(\cdot)$ can be nonlinear and even noninvertible – the EKF will do the rest.

4.3 Creating a Map

So far we have taken the existence of the map for granted, an understandable mindset given that maps today are common and available for free via the internet. Nevertheless, somebody, or something, has to create the maps we will use. Our next example considers the problem of **a robot moving in an environment with landmarks and creating a map of their locations**. As before we have a range and bearing sensor mounted on the robot which measures, imperfectly, the position of landmarks with respect to the robot. There are a total of N landmarks in the environment and as for the previous example, we assume that the sensor can determine the identity of each observed landmark. However for this case **we assume that the robot knows its own location perfectly** – it has ideal localization. This is unre-

alistic, but this scenario is an important stepping stone to the next section. Since the vehicle pose is known perfectly we do not need to estimate it, but we do need to estimate the coordinates of the landmarks. For this problem, the **state vector** comprises the estimated coordinates of the M landmarks that have been observed so far

$$\hat{\boldsymbol{x}} = (x_1, y_1, x_2, y_2, ..., x_M, y_M)^T \in \mathbb{R}^{2M \times 1}$$

The corresponding **estimated covariance** $\hat{\boldsymbol{P}}$ will be a $2M \times 2M$ matrix. The state vector has a variable length since we do not know in advance how many landmarks exist in the environment. Initially $M = 0$ and is incremented every time a previously unseen landmark is observed. The prediction equation is straightforward in this case since the landmarks are assumed to be stationary

$$\hat{\boldsymbol{x}}^+\langle k+1 \rangle = \hat{\boldsymbol{x}}\langle k \rangle \tag{4.18}$$

$$\hat{\boldsymbol{P}}^+\langle k+1 \rangle = \hat{\boldsymbol{P}}\langle k \rangle \tag{4.19}$$

We introduce the function $g(\cdot)$ which is the inverse of $h(\cdot)$ and gives the coordinates of the observed landmark based on the known vehicle pose and the sensor observation

$$g(\boldsymbol{x}, \boldsymbol{z}) = \begin{bmatrix} x_v + rcos(\theta_v + \beta) \\ y_v + rsin(\theta_v + \beta) \end{bmatrix} \tag{4.20}$$

Since $\hat{\boldsymbol{x}}$ has a variable length we need to extend the state vector and the covariance matrix whenever we encounter a landmark we have not previously seen. The state vector is extended by the function $y(\cdot)$

$$\boldsymbol{x}\langle k \rangle' = y(\boldsymbol{x}\langle k \rangle, \boldsymbol{z}\langle k \rangle, x_v\langle k \rangle) \tag{4.21}$$

which appends the sensor-based estimate of the new landmark's coordinates to those already in the map. The order of feature coordinates within, therefore depends on the order in which they are observed. The covariance matrix also needs to be extended when a new landmark is observed and this is achieved by

$$\hat{\boldsymbol{P}}\langle k \rangle' = \boldsymbol{Y}_z \begin{bmatrix} \hat{\boldsymbol{P}}\langle k \rangle & 0 \\ 0 & \hat{\boldsymbol{W}} \end{bmatrix} \boldsymbol{Y}_z^T$$

where $\boldsymbol{Y}_z$ is the **Insertion Jacobian**

$$\boldsymbol{Y}_z = \frac{\partial y}{\partial \boldsymbol{z}} = \begin{bmatrix} \boldsymbol{I}_{n\times n} & & \boldsymbol{0}_{n\times 2} \\ \boldsymbol{G}_x & \boldsymbol{0}_{2\times n-3} & \boldsymbol{G}_z \end{bmatrix} \tag{4.22}$$

that relates the rate of change of the extended state vector to the new observation. n is the dimension of $\hat{\boldsymbol{P}}$ prior to it being extended and

$$\boldsymbol{G}_x = \frac{\partial g}{\partial \boldsymbol{x}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4.23}$$

$$\boldsymbol{G}_z = \frac{\partial g}{\partial \boldsymbol{z}} = \begin{bmatrix} cos(\theta_v + \beta) & -rsin(\theta_v + \beta) \\ sin(\theta_v + \beta) & rcos(\theta_v + \beta) \end{bmatrix} \tag{4.24}$$

$\boldsymbol{G}_x$ is zero since $g(\cdot)$ is independent of the map in x. An additional Jacobian for $h(\cdot)$ is

$$\boldsymbol{H}_{\boldsymbol{p}_i} = \frac{\partial h}{\partial \boldsymbol{p}_i} = \begin{bmatrix} -\frac{x_i - x_v}{r} & -\frac{y_i - y_v}{r} \\ \frac{y_i - y_v}{r^2} & -\frac{x_i - x_v}{r^2} \end{bmatrix} \tag{4.25}$$

which describes how the landmark observation changes with respect to landmark position for a particular robot pose, and is implemented by the method `Hp`. For the mapping case the Jacobian $\boldsymbol{H}_x$ used in Eq. 6.11 describes how the landmark observation changes with respect to the full state vector. However, the observation depends only on the position of that landmark so this Jacobian is mostly zeros

$$\boldsymbol{H}_x = \frac{\partial h}{\partial \boldsymbol{x}}\Big|_{w=0} = (0...\boldsymbol{H}_{\boldsymbol{p}_i}...0) \in \mathbb{R}^{2\times 2M} \tag{4.26}$$

where $\boldsymbol{H}_{\boldsymbol{p}_i}$ is at the location in the vector corresponding to the state $\boldsymbol{p}_i$. This structure represents the fact that observing a particular landmark provides information to estimate the position of that landmark, but no others. The Toolbox implementation is

```
>> map = LandmarkMap(20);
>> veh = Bicycle();  % error free vehicle
>> veh.add_driver( RandomPath(map.dim) );
>> W = diag([0.1, 1*pi/180].^2);
>> sensor = RangeBearingSensor(veh, map, 'covar', W);
>> ekf = EKF(veh, [], [], sensor, W, []);
```

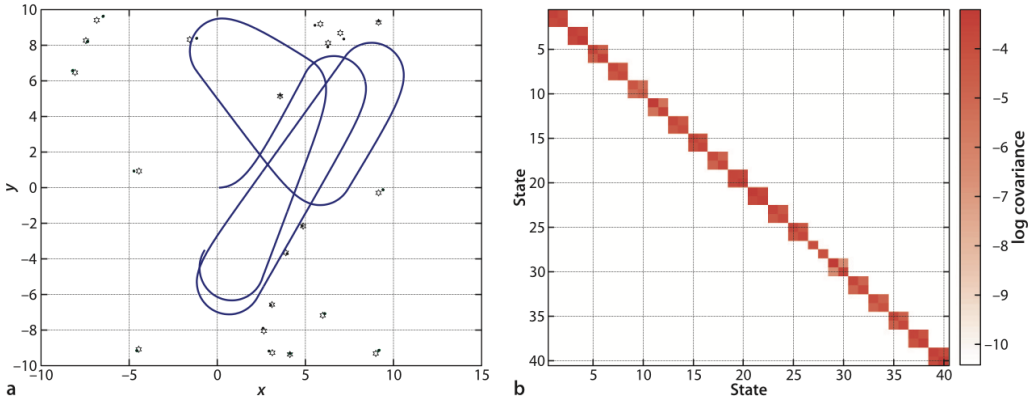


Figure 4.5: EKF mapping results. **a** The estimated landmarks are indicated by *black dots* with 95% confidence ellipses (*green*), the true location (*black star-marker*) and the robot's path (*blue*). The landmark estimates have not fully converged on their true values and the estimated covariance ellipses can only be seen by zooming; **b** the nonzero elements of the final covariance matrix

The empty matrices passed to `EKF` indicate respectively that there is no estimated odometry covariance for the vehicle (the estimate is perfect), no initial vehicle state covariance and the map is unknown. We run the simulation for 1000 time steps

```
>> ekf.run(1000);
```

and see an animation of the robot moving and the covariance ellipses associated with the map features evolving over time. The estimated landmark positions

```
>> map.plot();
>> ekf.plot_map('g');
>> veh.plot_xy('b');
```

are shown in Fig. 4.5a as 95% confidence ellipses along with the true landmark positions and the path taken by the robot. The covariance matrix has a block diagonal structure which is shown graphically in Fig. 4.5b. The off-diagonal elements are zero, which implies that the landmark estimates are uncorrelated or independent. This is to be expected since observing one landmark provides no new information about any other landmark. Internally

the `EKF` object maintains a table to relate the landmark's identity, returned by the `RangeBearingSensor`, to the position of that landmark's coordinates in the state vector.

4.4 Localization and Mapping

Finally, we tackle the problem of determining our position and creating a map at the same time. This is an old problem in marine navigation and cartography – incrementally extending maps while also using the map for navigation. It can be done without GPS from a moving ship with poor odometry and infrequent celestial position "fixes". In robotics, this problem is known as **simultaneous localization and mapping (SLAM)** or concurrent mapping and localization (CML). This is often considered to be a "chicken and egg" problem – we need a map to localize and we need to localize to make the map. However, based on what we have learned in the previous sections this problem is now quite straightforward to solve.

The state vector comprises the vehicle configuration and the coordinates of the M landmarks that have been observed so far

$$\hat{\boldsymbol{x}} = (x_v, y_v, \theta_v, x_1, y_1, x_2, y_2, ..., x_M, y_M)^T \in \mathbb{R}^{2M+3\times 1}$$

The estimated covariance is a $(2M + 3) \times (2M + 3)$ matrix and has the structure

$$\hat{\boldsymbol{P}} = \begin{bmatrix} \hat{\boldsymbol{P}}_{vv} & \hat{\boldsymbol{P}}_{vm} \\ \hat{\boldsymbol{P}}_{vm}^T & \hat{\boldsymbol{P}}_{mm} \end{bmatrix}$$

where $\hat{\boldsymbol{P}}_{vv}$ is the covariance of the vehicle pose, $\hat{\boldsymbol{P}}_{mm}$ the covariance of the map landmark positions, and $\hat{\boldsymbol{P}}_{vm}$ is the correlation between vehicle and landmark states. The predicted vehicle state and covariance are given by Eq. 4.5 and Eq. 4.6 and the sensor-based update is given by Eq. 4.12 to 4.17. When a new feature is observed the state vector is updated using the insertion Jacobian $\boldsymbol{Y}_z$ given by Eq. 4.22 but in this case $\boldsymbol{G}_x$ is nonzero

$$\boldsymbol{G}_x = \frac{\partial g}{\partial \boldsymbol{x}} = \begin{bmatrix} 1 & 0 & -rsin(\theta_v + \beta) \\ 0 & 1 & rcos(\theta_v + \beta) \end{bmatrix} \tag{4.27}$$

since the estimate of the new landmark depends on the state vector which now contains the vehicle's pose. For the SLAM case the Jacobian $\boldsymbol{H}_x$ used

in Eq. 4.13 describes how the landmark observation changes with respect to the state vector. The observation will depend on the position of the vehicle and on the position of the observed landmark and is

$$\boldsymbol{H}_x = \frac{\partial h}{\partial \boldsymbol{x}}\Big|_{w=0} = (\boldsymbol{H}_{x_v} ... 0 ... \boldsymbol{H}_{p_i} ... 0) \in \mathbb{R}^{2\times(2M+3)} \tag{4.28}$$

where $\boldsymbol{H}_{p_i}$ is at the location corresponding to the landmark $\boldsymbol{p}_i$. This is similar to Eq. 4.26 but with an extra nonzero block $\boldsymbol{H}_{x_v}$ given by Eq. 4.16. The Kalman gain matrix $\boldsymbol{K}$ distributes innovation from the landmark observation, a 2-vector, to update every element of the state vector – the pose of the vehicle and the position of every landmark in the map. The Toolbox implementation is by now quite familiar

```
>> P0 = diag([.01, .01, 0.005].^2);
>> map = LandmarkMap(20);
>> veh = Bicycle('covar', V);
>> veh.add_driver( RandomPath(map.dim) );
>> sensor = RangeBearingSensor(veh, map, 'covar', W);
>> ekf = EKF(veh, V, P0, sensor, W, []);
```

and the empty matrix passed to `EKF` indicates that the map is unknown. `P0` is the initial 3×3 covariance for the vehicle state. We run the simulation for 1000 time steps

```
>> ekf.run(1000);
```

and as usual an animation is shown of the vehicle moving. We also see the covariance ellipses associated with the map features evolving over time. We can plot the results

```
>> map.plot();
>> ekf.plot_map('g');
>> ekf.plot_xy('r');
>> veh.plot_xy('b');
```

which are shown in Fig. 4.6. Figure 4.7a shows that uncertainty is decreasing over time.

The final covariance matrix is shown graphically in Fig. 4.7b and we see a complex structure. Unlike the mapping case in Fig. 4.5 $\hat{\boldsymbol{P}}_{mm}$ is not block

diagonal, and the finite off-diagonal terms represent the correlation between the landmarks in the map. The landmark uncertainties never increase, the position prediction model is that they do not move, but they also never drop below the initial uncertainty of the vehicle which was set in $\boldsymbol{P}_0$. The block $\hat{\boldsymbol{P}}_{vm}$ is the correlation between errors in the vehicle pose and the landmark locations. A landmark's location estimate is a function of the vehicle's location and mistakes in the vehicle location appear as errors in the landmark location – and vice versa.

The Kalman filter uses the correlations to connect the observation of any landmark to an improvement in the estimate of every other landmark in the map as well as the vehicle pose. Conceptually it is as if all the states were connected by springs and the movement of any one affects all the others.

The Extended Kalman filter, introduced here, has a number of drawbacks... Firstly the size of the matrices involved increases with the number of landmarks and can lead to memory and computational bottlenecks as well as numerical problems. The underlying assumption of the Kalman filter is that all errors are Gaussian and this is far from true for sensors like laser rangefinders. We also need reasonable estimates of covariance of the noise sources which in practice is challenging.

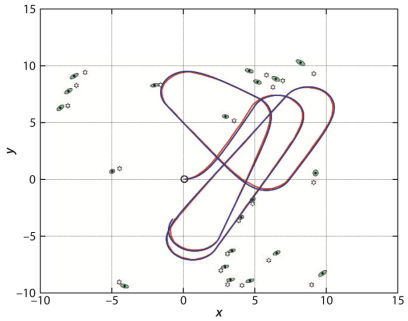


Figure 4.6: EKF mapping results. **a** SLAM showing the true (*blue*) and estimated (*red*) robot path superimposed on the true map (*black star-marker*). The estimated map features are indicated by black dots with 95% confidence ellipses (*green*)

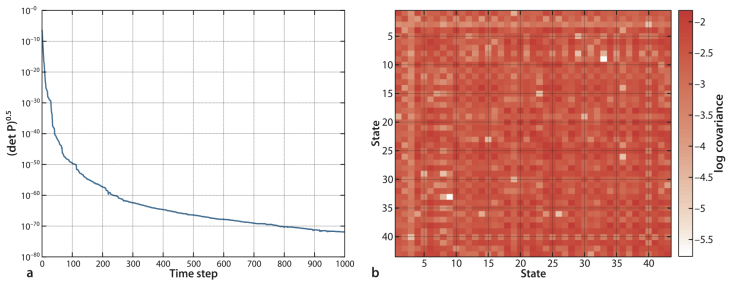


Figure 4.7: SLAM **a** Covariance versus time; **b** the final covariance matrix

SUMMARY for SLAM using KF

SLAM, which stands for Simultaneous Localization and Mapping, is a fundamental problem in robotics where a robot navigates through an environment, creates a map of that environment, and at the same time estimates its own position within that map. The Kalman Filter is a mathematical technique commonly used in robotics and other fields for estimation and control purposes, including SLAM. It helps the robot fuse sensor measurements and motion information to maintain an accurate estimate of its position and the map it's building.

Here's an explanation of SLAM using a Kalman Filter:

Mapping: The robot starts by moving through the environment and collecting sensor measurements, such as from cameras, lidar, or other range sensors. These measurements can be features in the environment, like landmarks or distinctive points. The robot then uses these measurements to create a map of the environment. In SLAM, this map is typically represented as a set of landmarks or features with their positions.

Localization: Simultaneously, the robot needs to estimate its own position within the map it's building. This is done by using its motion information, which comes from its own sensors like wheel encoders, gyroscopes, etc. However, over time, errors in the motion estimation accumulate, leading to uncertainty in the robot's position.

Kalman Filter: The Kalman Filter is a recursive mathematical algorithm that helps the robot estimate the state of a dynamic system based on noisy measurements. In the context of SLAM, the robot's state consists of its current position and orientation, as well as the positions of the landmarks

it's observing. The Kalman Filter consists of two main steps: prediction and correction.

Prediction: Using the robot's motion model, the Kalman Filter predicts its new state (position and orientation) based on its previous state and the control inputs (e.g., velocity or wheel rotation) from its sensors. The prediction also includes an estimation of the uncertainty associated with this prediction.

Correction: Once new sensor measurements (landmark positions, for instance) are obtained, the Kalman Filter corrects its predicted state using these measurements. This involves updating the estimated state and its uncertainty based on the sensor measurements' accuracy and the predicted state's uncertainty. The Kalman Filter effectively combines the predictions from the motion model with the measurements from sensors to provide a more accurate and optimal estimate of the robot's state.

Iterative Process: SLAM using a Kalman Filter is an iterative process. As the robot moves through the environment, it continually collects sensor measurements, updates its map with new landmark positions, estimates its own position, and refines its map. The Kalman Filter helps in maintaining a consistent and accurate estimate despite the inherent noise and uncertainties in the measurements and motion.

It's worth noting that while the basic concept of using a Kalman Filter for SLAM is outlined here, practical implementations can be more complex. For instance, Extended Kalman Filters (EKF) or more advanced techniques like Particle Filters are often used to handle nonlinearities and uncertainties in a more sophisticated manner. Additionally, modern SLAM systems often use advanced sensor fusion techniques, loop closure detection, and optimization algorithms to enhance accuracy and robustness.

4.5 Application: Scanning Laser Rangefinder

As we have seen, robot localization is informed by measurements of range and bearing to landmarks. Sensors that measure range can be based on many principles such as laser rangefinding (Fig. 4.8a, 4.8b), ultrasonic ranging (Fig. 4.8c), computer vision or radar. A laser rangefinder emits short pulses

of infra-red laser light and measures how long it takes for the reflected pulse to return. Operating range can be up to 50 m with an accuracy of the order of centimeters. A *scanning* laser rangefinder, as shown in Fig. 4.8a, contains a rotating laser rangefinder and typically emits a pulse every quarter, half or one degree over an angular range of 180 or 270 degrees and returns a planar cross-section of the world in polar coordinate form $(r_i, \theta_i), i \in 1, ..., N$. Some scanning laser rangefinders also measure the return signal strength, remission,

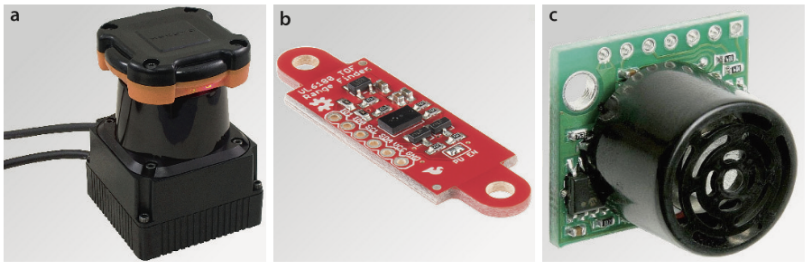


Figure 4.8: Robot rangefinders:
a: A scanning laser rangefinder with a maximum range of 30 m, an angular range of 270 deg in 0.25 deg intervals at 40 scans per second
b: a low-cost time-of-flight rangefinder with maximum range of 20 cm at 10 measurements per second
c: a low-cost ultrasonic rangefinder with maximum range of 6.5 m at 20 measurements per second

which is a function of the infra-red reflectivity of the surface. The rangefinder is typically configured to scan in a plane parallel to, and slightly above, the ground. Laser rangefinders have advantages and disadvantages compared to cameras and computer vision. On the positive side laser scanners provide metric data, that is, the actual range to points in the world in units of meters, and they can work in the dark. However laser rangefinders work less well than cameras outdoors since the returning laser pulse is overwhelmed by infra-red light from the sun. Other disadvantages include providing only a linear cross section of the world, rather than an area as a camera does; inability to discern fine texture or color; having moving parts; as well as being bulky, power hungry and expensive compared to cameras.

4.5.1 Laser Odometry

A common application of scanning laser rangefinders is laser odometry, estimating the change in robot pose using laser scan data rather than wheel encoder data. We will illustrate this with laser scan data from a real robot

```
>> pg = PoseGraph('killian.g2o', 'laser');
```

and each scan is associated with a vertex of this already optimized pose graph. The range and bearing data for the scan at node 2580 is

```
>> [r, theta] = pg.scan(2580);
```

represented by two vectors each of 180 elements. We can plot these in polar form

```
>> polar(theta, r)
```

or convert them to Cartesian coordinates and plot them

```
>> [x,y] = pol2cart(theta, r);
>> plot(x, y, '.')
```

The method `scanxy` is a simpler way to perform these operations. We load scans from two closely spaced nodes

```
>> p2580 = pg.scanxy(2580);
>> p2581 = pg.scanxy(2581);
```

which creates two matrices whose columns are Cartesian point coordinates and these are overlaid in Fig. 4.9a. To determine the change in pose of the robot between the two scans we need to align these two sets of points and this can be achieved with iterated closest-point-matching or ICP. This is implemented by the Toolbox function `icp` and we pass in the second and first set of points, each organized as a $2 \times N$ matrix

```
>> T = icp( p2581, p2580, 'verbose' , 'T0',
        transl2(0.5, 0), 'distthresh', 3)
```

and the algorithm converges after a few iterations with an estimate of $T \sim^{2580} \xi_{2581} \in \mathbf{SE}(2)$. This transform maps points from the second scan so that they are as close as possible to the points in the first scan. Figure 4.9b shows the first set of points trans- formed and overlaid on the second set and we see

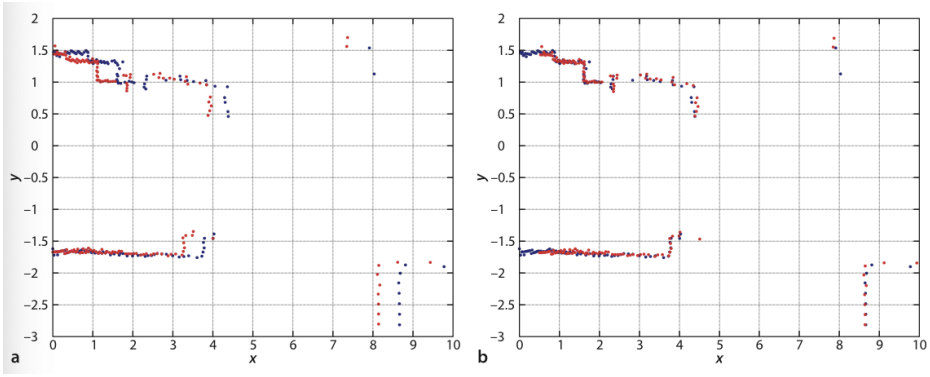


Figure 4.9: Laser scan matching. **a** Laser scans from location 2580 (*blue*) and 2581 (*red*); **b** location 2580 points (*blue*) and transformed points from location 2581 (*red*)

good alignment. The translational part of this transform is an estimate of the robot's motion between scans – around 0.50 m in the x-direction. The nodes of the graph also hold time stamp information and these two scans were captured

```
>> pg.time(2581)-pg.time(2580)
```

seconds apart which indicates that the robot is moving quite slowly – a bit under 0.3 m s^{-1}. At each iteration ICP assigns each point in the second set to the closest point in the first set and then computes a transform that minimizes the sum of distances between all corresponding points. Some points may not actually be corresponding but as long as enough are, the algorithm will converge. The '`verbose`' option causes data about each iteration to be displayed and `d` is the total distance between corresponding points which is decreasing but does not reach zero. This is due to many factors. The beams from the laser at the two different poses will not strike the walls at the same location so ICP's assumption about point correspondence is not actually valid. In practice there are additional challenges. Some laser pulses will not return to the sensor if they fall on a surface with low reflectivity or on an oblique polished surface that specularly reflects the pulse away from the sensor – in these cases the sensor typically reports its maximum value. People moving through the environment change the shape of the world and temporarily cause a shorter range to be reported. In very large spaces all the

walls may be beyond the maximum range of the sensor. Outdoors the beams can be reflected from rain drops, absorbed by fog or smoke and the return pulse can be overwhelmed by ambient sunlight. Finally the laser rangefinder, like all sensors, has measurement noise.

Chapter 5

Learning for Adaptive and Reactive Robot Control

Before the introduction of Machine Learning to Robotic Factories everything was predetermined, there was no room for change. In traditional robot factories, robots' motions are pre-programmed, they are always the same and they are optimized for maximum efficiency. After the arrival of Industry 4.0 things have begun to change: robots can work outside cells and work collaboratively with humans: This also has led to an increase in productivity and a saving of space. That induced a need for **real-time planning** of robots, but often the environment is only partially predictable, so there is:

- need to learn from data
- needs guarantees on the stability of the learned controller

Also, the environment is often only partially observable, so there is also a need to react in milliseconds to avoid collisions.

The most popular example in this field is the **robot arm catching flying objects** (*RACFO* for short): this means that the system has to be able to compute the trajectory for the robot hand to get in the point and close it on time. The object is flying with very complex dynamics, it's not a ballistic motion, it completely depends on how the object is thrown at the beginning.

So there's a need to combine:

- **Control**: traditional control in robotics
- **Machine Learning**: because there is high uncertainty, highly complex and nonlinear systems. The only way to have a good estimate of the internal dynamics of the trajectories is to do it with ***Nonlinear Regression***

Everything must happen in a few milliseconds, which means that we cannot wait until we have a good estimate: In the RACFO example, we start initial estimate of how the object is flying, then we plan an initial path, we keep tracking the object with cameras and sensors and then we iterate. Sensors are inaccurate (frame drops, obstructions, light reflections, outliers, etc.) and therefore our tracking with time will also become inaccurate. We have to perform **predictions**, we need to have estimates when we cannot "see". But when we "re-open our eyes" we are still on our way to the target, so we need a way to quickly re-estimate where to go and to move there.
Our focus is to make a *re-planning* in milliseconds: not only do we re-plan immediately without sampling but we also have an infinite number of paths that are given to us in a closed analytical form and we also are guaranteed that we'll always reach the target. The only way to accomplish this is to have solutions that are easy to compute, essentially **matrix products or summations of matrices**.

Real-time planners face some challenges:

- Environment is **dynamic**
- Environment is only **partially observable**
- The model of the environment cannot always be explicitly described by known equations
- One must **learn appropriate dynamics** for the robot to move in the environment
- The learned controller must **offer guarantees to ensure the safety of users and bystanders, and to ensure successful task completion**.

5.1 Traditional Planning Approaches in Robotics

5.1.1 Path Planning in 2D

Path planning, also known as motion planning, was for a long time thought of the problem of moving a vehicle (wheel-based) in a 2D environment. The complexity of the planning relates, primarily, to three factors:

- The vehicle is holonomic or non-holonomic
- The environment is fully or partially known: you augment it using GPS, and map, but the uncertainties are always present
- The environment is deterministic or stochastic

5.1.2 Global Path Planning

- Compute all paths - complete search
- Determine the set of paths that are optimal

PROS: it guarantees:

- optimality
- completeness of the search
- convergence to the goal
- feasibility of the paths

CONS:

- Requires complete enumeration
- Depends on global knowledge of the world
- Does not apply to continuous world representations

5.1.3 Local Path Planning

- Compute a subset of paths in a neighborhood
- Determine the optimal paths among this set

PROS: Requires only local knowledge of the world. It guarantees:

- fast and reactive control
- adapted to real-time control
- adapted to local robot perception

CONS:

- May not find a feasible path to the goal
- Paths may all be suboptimal
- Depends on a heuristics to determine the paths

Chapter 6

Gathering Data for Learning

This chapter presents techniques and interfaces that can be used to gather data that can then be used to train the robot's controller.

6.1 Approaches to generate data

One popular approach to generate data to train robots is to have an expert provide examples of the task we want the robot to learn. This is known as ***learning from demonstration (LfD)*** or *programming by demonstration (PbD)*. LfD/PbD, sometimes referred to as imitation learning and apprenticeship learning, is a paradigm for enabling robots to learn new tasks from observing experts (usually humans) performing the task. The main principle of robot learning from demonstration is that end users can teach robots new tasks without programming. Hence, for a long time, LfD/PbD assumed that the demonstrations were provided by a human on site. This was limiting, as it required having an expert available to produce the data, and hence it restricted learning to tasks doable by humans. More recent approaches use simulations or optimal control to generate solutions that would be exhaustive or impossible to generate with a human.

An alternative to LfD/PbD is to let the robot learn on its own, through trial and error. This is broadly known as ***reinforcement learning (RL)***. An expert is still needed, but only to provide a reward function, not a demonstration of the entire task. One drawback of RL is that it takes a long time

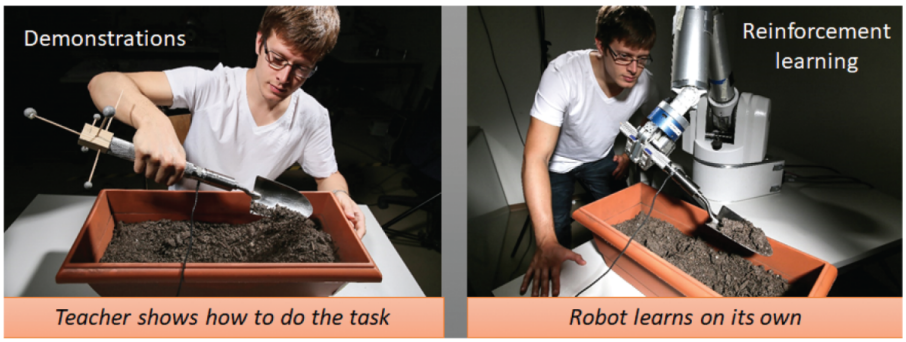


Figure 6.1: Data for training robots can either be provided by human demonstrations or gathered by the robot on its own through trial and error (known as reinforcement learning, or RL). The two modes can be combined. Demonstrations can be used to bootstrap the search, providing a good example of how to move the tool. Trial and error can then be conducted in a restricted search space around the demonstrated trajectory to gather more information on the force and impedance to apply when interacting with gravel.

before it converges to an optimal solution. To tackle this issue, two main trends are followed. The first combines learning from demonstration with RL. The demonstrations are used to bootstrap the search, providing a feasible solution or binding the search space (see Fig. 6.1). The second trend uses realistic simulators to perform the search offline and use the real world only to refine the initial search. Another generic criticism of RL is that it requires one to carefully craft the reward. Inverse reinforcement learning contours this problem by allowing the robot to automatically infer the reward and the optimal controller. To do so, however, the robot must have access to examples of good and bad solutions to the problem. These are usually provided by a human, which again raises the problem of limiting the number of required demonstrations so it would be bearable to the human.

6.1.1 Which Method Should Be Used, and When?

The methods presented here have different requirements. Some need the user to have prior knowledge about the robot, task, and environment, while others need extensive time to generate data. To help, we provide a summary of what each method needs and provides

Method to generate the data	Online mode	Need model of robot or world	Trainer	Number of training examples
Learning from human demonstrations	YES	NO	Anyone	<20
Optimal control	NO	YES	Skilled programmer	>100
RL (live)	NO	YES (model-based RL) NO (model-free RL)	Anyone (reward)	>100
RL (simulation)	YES	YES	Skilled programmer	>1,000

Figure 6.2

Human demonstrations:

Data can be gathered from a human showing the task to the robot. This has the advantage that the robot immediately obtains examples of feasible solutions. The teacher does not need to be an expert in robot control and can be the end user of the platform. This approach allows the end user to customize the robot's behavior to their preferred way of moving. This is particularly useful for robots that are meant to work in coordination with the user, such as co-bots, and for control of prostheses and exoskeletons. The disadvantage is that the number of examples is limited to only about twenty trajectories for training to be bearable to the end user, which may provide too few statistics. Finally, the dynamics of motion performed by the human sometimes may not be doable for the robot's hardware.

Optimal control:

If we can write a model of the task and of the dynamics of the robot, one can use optimal control to generate a large range of feasible trajectories that satisfy all the task requirements and the robot's constraints. Optimal control depends on solvers for non-convex optimization or integer and constraint programming. Solvers may take minutes or hours to find solutions, and they are not certain to converge. There is, hence, an advantage to converting all the solutions found offline through optimal control into closed-form expressions, with guarantees to retrieve always a feasible solution in real time. This em-

bedding is done using machine learning techniques. Using optimal control to generate data for learning is advantageous, as it can provide a large number of trajectories (by randomizing the initial conditions) and the trajectories are by construction feasible for the robot's dynamics.

Reinforcement learning:

Data can be gathered by the robot through trial and error. At each trial, the robot receives a reward that informs it on whether this is a good or poor solution. The advantage of using this technique over the previous two is that it provides a set of both feasible and unfeasible trajectories. Unfeasible trajectories can then be used during learning to delimit the range of doable motion. The disadvantage is that this is fairly slow, and it may take many trials before a feasible solution is found. It may not be doable to run this on board the robot, and it may require an accurate simulator.

6.2 Interfaces for Teaching Robots

When one chooses to train a robot through human demonstrations, the choice of interface is crucial, as the interace plays a key role in the way that the information is gathered and transmitted. We primarily consider interfaces that provide information about the kinematic of the motion (speed, position) and haptic information (force, torque, touch location). In this context, the user must hence choose an interface following the three possibilities given here.

6.2.1 Motion-Tracking Systems

To record the kinematic of human motion, one may use any of the existing motion-tracking systems, whether these are based on vision, exoskeleton, or other types of wearable motion sensors. Figu. 6.3 shows an example of using motion tracking to monitor a human catching a ball. The motion-capture system consist of a set of infrared cameras that track at high speed the markers worn by the subject and attached to the ball at a high frame rate of 250 frames per second within millimeters precision. Given the speed

at which the motion happens, it is necessary to sample this rapidly and to have this level of precision. This can be done only with motion-capture systems or time-of-flight cameras. When one considers teaching a motion that requires a much slower pace, using standard cameras that sample at 50 Hz is an appropriate alternative. There are several systems, commercial but also open source, that can reconstruct the human motion from a camera.

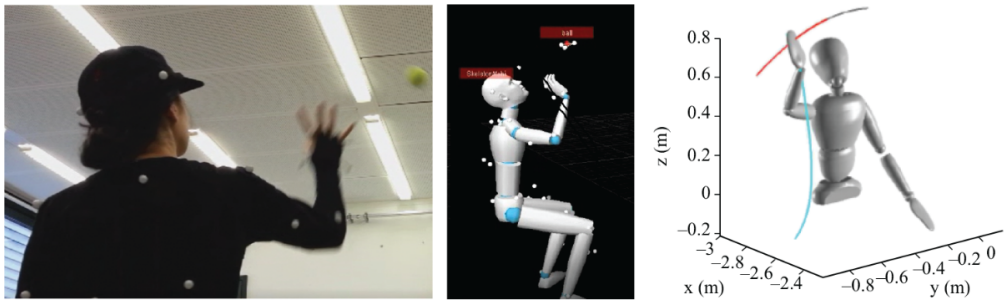


Figure 6.3: Demonstration of catching a ball recorded with a motion-capture system. The subject wears a suit covered with markers to track all upper-body joints. The reconstructed trajectories are highly accurate (c).

6.2.2 Correspondence Problem

These external means of tracking human motion return precise measurements of the angular displacement of the limbs and joints. They have been used in various instances for LfD of full-body motion. These methods allow the human to move freely and to perform the task naturally. However, they require solutions to the correspondence problem (i.e., the problem of how to transfer motion from human to robot when both differ in the kinematic and dynamics of their bodies), or, in other words, if the configuration spaces are of different dimensions and sizes. This is typically done when mapping the motion of the joints that are tracked visually to a model of the human body that matches closely with that of the robot. Such mapping is always challenging to do. For instance, a robotic arm with 7 degrees of freedom (DOF) does not have a joint alignment that resembles the human arm's joint alignment. In other words, the two kinematic chains differ significantly. They also differ in terms of the range of motion. A simple scaling is not sufficient, and one

must often perform double inverse kinematics, transferring displacement of the human's end point into the corresponding robot's joint angles. Given the current human's joint measurement q^h, one computes the human's end point position x^h, using Jacobian $J(q)^h$, and similarly for the robot's end point's position x^r, Jacobian J^r, and joint angles q^r, by solving the following:

$$\begin{aligned} \min_{\theta^r} \|x^h - x^r\| \\ x^h = J^h \theta^h \\ x^r = J^r \theta^r \end{aligned} \tag{6.1}$$

The problem is particularly difficult when the two systems differ dramatically in the number of degrees of freedom. For instance, the human hand is said to have between 22 and 28 DOF, but most robotic hands have much fewer degrees of freedom. A traditional prosthetic hand would have only 5 DOF (1 for each finger), while more sophisticated hands would have 9 (Humanoid iCub hand),1 13 (DLR hand), 16 (Gifu hand [67] and Allegro hand2), and 24 (Shadow hand).

6.2.3 Kinesthetic Teaching

The problem becomes even more difficult when one tries to control a full humanoid robot, or when the robot's body bears little resemblance to the human's body (e.g., an hexapod differs significantly from the human body). One way to overcome the correspondence problem is to put the human in the robot's shoes, such as by having the human guide the robot through kinesthetic teaching, as illustrated in figure 2.4. Kinesthetic teaching is a technique in which the teacher embodies the robot using the mechanical backdrivibility of its joint to move it passively. Kinesthetic teaching also enables the transmission of information about forces, which is not readily available from motion-tracking systems. In addition, it provides a natural teaching interface to correct a skill reproduced by the robot. One main drawback of kinesthetic teaching is that the human must often use more degrees of freedom to move the robot than the number of degrees of freedom moved on the robot. This is visible in the example on the right side of figure 6.4. To move the fingers of one hand of the robot, the teacher must

use both hands. This limits the type of tasks that can be taught through kinesthetic teaching. Typically, tasks that would require moving the two hands simultaneously could not be taught in this way. One could either proceed incrementally, teaching first the task for the right hand and then, while the robot replays the motion with its right hand, teach the motion of the left hand. However, this may prove to be cumbersome. The use of external trackers, as reviewed previously, is more amenable to teaching coordinated motion among several limbs.

Figure 6.4: **Left figure**: Demonstration of a task requiring force control through teleoperation via a haptic interface. **Middle figure**: Example of kinesthetic teaching, in which the human puts herself in the robot's shoes to teach the robot how to peel vegetables. Forces are measured through tactile sensors at the robot's end points. **Right figure**: Example of kinesthetic teaching to teach how to manipulate an object. Forces are measured through tactile sensors at the robot's fingertips.

6.2.4 Teleoperation

Teleoperation, via joystick and remote control, can also be used to transmit the kinematics of motion. Such immersive teleoperation scenarios, where a human operator uses the robot's own sensors and effectors to perform the task, are powerful techniques to train a robot from its own viewpoint. For instance, in, acrobatic trajectories of an helicopter are learned by recording the motion of the helicopter when teleoperated by an expert pilot. A robot dog is taught to play soccer via a human guiding it via a joystick. Teleoperation, however, is often limited in the field of view rendered, which prevents the teacher from observing all the sensorial information required to perform

the task. Teleoperation is advantageous compared to external motion tracking systems, as it entirely solves the correspondence problem because the system records directly the perception and action from the robot's configuration space. Teleoperation using a simple joystick allows one to guide only a subset of degrees of freedom. To control for all degrees of freedom, very complex, exoskeleton-type devices must be used, which is cumbersome.

6.2.5 Interface to Transfer Forces

All of the methods described thus far provide only kinematic information. This is sufficient when we want to teach tasks that can be completely described using position and velocity, such as gestures. However, when the task depends on modulating forces at contact and this is done without large movements (e.g., when inserting a peg into a hole), then one needs to use interfaces that can measure and transmit these forces. In place of using a simple joystick, one can employ haptic devices that can render and transmit forces (see figure 6.4 left). It is advantageous compared to kinesthetic training, as it allows to train robots from a distance and hence is particularly well suited for teaching navigation and locomotion patterns. The teacher no longer needs to share the same space with the robot. This method was used, for instance, to teach balancing techniques to a humanoid robot. As the teacher moves, a haptic interface attached to the torso of the demonstrator measures the interaction forces. The kinematics of motion of the demonstrator are directly transmitted to the robot through teleoperation and are combined with haptic information to train a model of motion conditioned on perceived forces. The disadvantage of teleoperation techniques is that the teacher often needs training in order to learn to use the remote control device. Also, haptic devices can sometimes poorly render the contacts perceived at the robot's end-effector, or do so with a delay. To adapt to this, one may provide the teacher with visualization interfaces to simulate the interaction forces. To measure forces, one can also use the robot's onboard tactile or force/torque sensors. This can be used in conjunction with either kinesthetic teaching (see figure 6.4), or through the use of nonhaptic devices. While with kinesthetic teaching, the teacher can perceive the forces and modulate their

gestures in response, when using a nonhaptic joystick, this perception is lost and may negatively affect the teaching.

6.2.6 Combining Interfaces

Each teaching interface has pros and cons. For that reason, nowadays people use several of these interfaces in conjunction. Figure 6.5 shows one example, where a robot is trained to scoop melons. The task is challenging, as the melon is made of soft material and the robot must learn to vary the force and stiffness as it scoops, in response to the resistance that it perceives from the melon. When the melon is hard, a stiff controller may be appropriate, whereas a more compliant controller may be needed when the melon is softer. Melons are not homogeneous, and modifying compliance on the go may be necessary. To teach the robot, the user shown in the figure exploits a combination of interfaces. She uses kinesthetic teaching to embody the robot, which allows her to better perceive and transfer the forces. Forces and torques are measured through two force/torque sensors mounted on the tool and the robot's end-effector, and also through tactile sensors covering the glove worn by the user. These sensors measure the evolution of the forces applied as a response to the change in resistance of the melon. This evolution of force measurement provides key information to allow one to adapt the impedance to changes in the fruit's resistance. In addition, marker-based vision and a data glove track kinematic information on arm and hand motions, so as to detect changes in posture and the grasp of the user's hand as she adapts to the task.

6.3 Desire for the Data

A few general considerations are in order:

Gathering enough data

Data are inherently noisy. Because all machine learning techniques used in this book are based on statistics, the data must include enough statistics for the algorithm to tell noise from the signal. A good rule in statistics is to

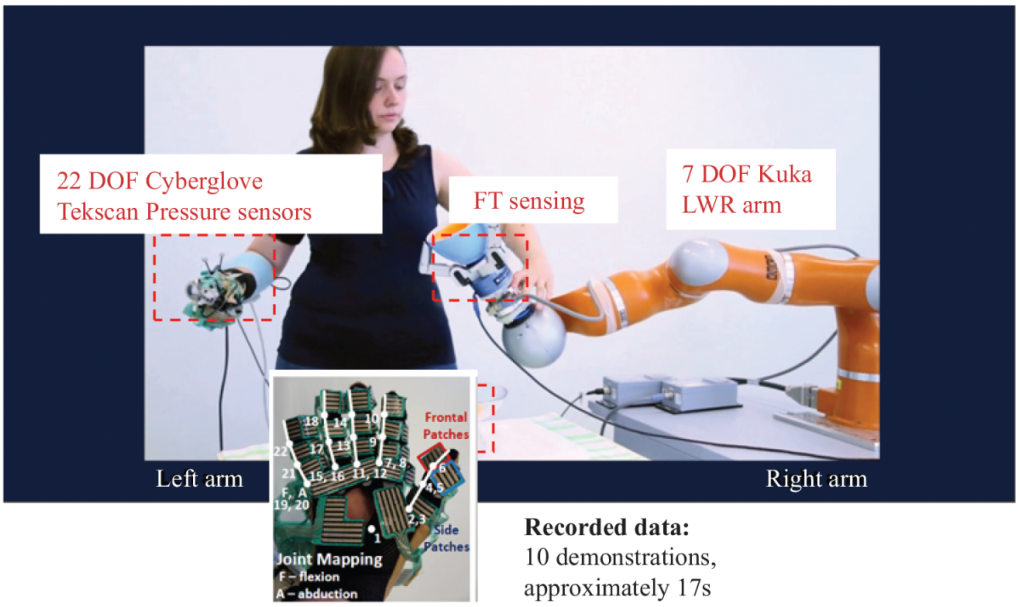


Figure 6.5: Use of multiple interfaces (i.e., vision, haptic) for training a robot to scoop melons.

have at least ten samples for each dimension. If you are teaching a task that requires control of position and orientation ($N = 6$), you would want to have at least 106 data points. If you sample trajectories at 50 Hz, the data will include enough statistics after as few as ten demonstrations, each one lasting less than a few seconds. However, while this may be statistically sufficient, it is not certain that this will be enough to teach the robot well. We need to make sure that the data are relevant to the task. Consider, for instance, that you want the robot to learn how to play golf (. To teach the robot how to put a ball into a hole, you must present examples illustrative of how to do this. While showing ten examples may be sufficient to generate the required statistics, showing quasi-identical ones will provide little insight into the task. It is useful to generate multiple examples, starting from different configurations and approaching the target with different orientations. It may also be useful to change the experimental setup and change the position and height of the hole.

Data must be well chosen

For the focus of this book, data must be composed of samples of dynamics of movement that can be realized by the robot. Speed, acceleration profiles, torques, and impedance parameters must be within the robot's capacities. Also, the data must be representative of what we want the robot to learn. They must entail useful examples of the task at hand. While one often provides only examples of what the robot must do, it is often quite useful to provide examples of what not to do. This is akin to providing examples of feasible and unfeasible dynamics—dynamics that would lead the task to succeed or fail.

For instance, when teaching a robot to play golf, you may want to show both examples when the ball goes into the hole and examples when the ball misses or bounces off the edge of the hole. This additional information will help the algorithm learn that tiny changes in orientation and speed may lead the task to shift from success to failure. When teaching is performed by a human, one aims to reduce the number of demonstrations to its bare minimum to lighten the burden on the teacher. A rule of thumb is generally not to exceed twenty demonstrations for each task. Thus, it is important to choose well what you demonstrate to keep from wasting demonstrations. If you work in simulation and use optimization to generate example trajectories, you are limited only by your patience and the speed of your solver.

Yet we should still be sure to collect a wide enough sample of solutions and to generate solutions that are both feasible and infeasible for the robot to learn well. Be aware that the number of feasible and infeasible solutions should be similar (or of the same order of magnitude) to avoid unbalanced datasets.

The concept of an unbalanced data set refers to a situation when one has many more samples of one type than of another (e.g., more samples of noncancer cells than of cancer cells). The imbalance can be detrimental to learning, as the group with the largest number of samples will have more influence and be better represented in the model than the group with fewer examples.

Generalization

One major goal of machine learning is to ensure that the model generalizes outside the examples used for training. This is crucial, as we are quite aware of the fact that the training examples provide only a subset of all that exists.

The usual practice in machine learning is to decompose the data set into a training set, a validation set, and a testing set. Training and validation are used in conjunction to determine the best hyperparameters through grid search and cross-validation. The testing set is used to assess how good the algorithm is at generalizing to unseen data. The training/test set ratio is a good metric of generalization. The smaller the number of training data required to ensure good performance of the testing set, the stronger our confidence in the model's generalization.

However, for a robot to generalize, it is important that training and testing sets be chosen in a way that leads to this generalization. Generalization can take many forms. In the golf task presented previously, limited generalization would be needed for the robot to put the ball correctly only if the initial ball locations differ by a few centimeters from those seen during training. But proper generalization would be demonstrated if the system can put the ball for a variety of configurations of both ball and hole.

Impressive generalization capabilities would be demonstrated if the robot were capable of sinking the ball on more complex terrain than seen during training. This is rarer, however, and transferring knowledge to a new context often requires the system to be trained again.

This is known as transfer learning. The testing set to measure the generalization capabilities of robots hence must not be the result of random sampling across the data set, but rather be chosen with care to demonstrate the expected generalization capabilities.

To conclude, the data for training robots must be chosen with care. They must be representative of what you would like the robot to achieve. They must include examples of what is a good solution to the task and what is not. Data must not be split randomly for training and testing. Rather, the data used at testing must be chosen to demonstrate the extent to which the robot generalizes the knowledge involved in the training examples.

6.4 Gathering Data for Optimal Control

In the two examples presented previously, we assumed that the speed and trajectories demonstrated by the human could be reproduced by the robot. In other words, we assumed that the demonstrations satisfy the kinematics and dynamics constraints of the robot.

Kinematics constraints are satisfied when using kinesthetic teaching, as it consists of moving the robot's joint. However, the dynamics displayed in the human demonstration may be too fast, requiring larger accelerations than are possible on the robot. Conversely, some human motions are sometimes too slow and when converted into velocities for the robot, they cannot be generated if the acceleration is too small and does not overcome the robot's friction.

An alternative to using human demonstrations is to search for possible solutions through RL, as described earlier in this chapter. Reinforcement learning is closely related to optimal control. In order to generate trajectories that are feasible for the robot, one must have a model of the robot. Not having such a model may be risky when running the search in the real world, as it may lead to hardware damage. In the golf example, generating trajectories that would be feasible and lead the robot to move as fast as possible could be expressed through the following iterative optimization:

Algorithm 2.1 Generate Kinematically Feasible Data Trajectories

Initialization:
$x(0)$ and x^* *Randomly define the initial robot's position and final target position.*
$q(0) = F^{-1}(x(0))$ *Inverse kinematics for the initial joint position.*
$t = 0$. *Initialize time.*
Main loop:
While $\epsilon \leq \|F(q(t)) - x^*\|$:
$\dot{q}(t+dt) = \arg\max_{\dot{q}} \|\dot{q}\|$. *Maximizing the velocity of the joints.*
subject to:
$\frac{\dot{q}}{\|\dot{q}\|} = J^{+}(q(t)) \frac{x^* - F(q(t))}{\|x^* - F(q(t))\|}$. Moving on the straight line toward the target at the task space.
$\dot{q}_{min}[i] \leq \dot{q}[i] \leq \dot{q}_{max}[i] \ \ \forall i \in \{1, \ldots, D\}$. Kinematic feasibility at the velocity level.
$q_{min} \leq q(t)[i] + \dot{q}[i]dt \leq q_{max}[i] \ \ \forall i \in \{1, \ldots, D\}$. Kinematic feasibility at the position level.
$q(t+dt) = q(t) + \dot{q}(t+dt)\, dt$
$t = t + dt$

$F(.)$ and $J(.)$ are the forward kinematics and the Jacobin matrices, respectively. $+$ is Moore–Penrose inverse.

Algorithm 2.1 makes sure that the trajectory is feasible using both bounds on maximal acceleration and velocity for each joint and model of the forward

and inverse kinematics.

This optimization can be run offline. If one samples different initial and final end-effector and joint locations $(x(0), q(0), x^*)$, this enables one to populate a database of feasible trajectories. All the configurations that lead to infeasible solutions from the solver may be used as examples of a failed set of parameters. Running this in simulation is advantageous, in that it will offer many examples (much more than are doable from human demonstration) and hence have a fine-grained model.

Yet this is just a simulation, remaining a proxy of reality. In the golf task, for instance, one can certainly run some of the trials in simulation using a good model of the robot and the environment. This would allow one to generate a set of initial values for feasible regions of parameters. However, real implementation will be needed to fine-tune the parameters and add more data to overcome the non-linearities of the terrain.

Chapter 7

Learning a control law

This chapter presents methods to learn a control law for robot motion generation using time-invariant dynamical systems (DSs). We assume that the motion of the robotic system is fully defined by its state $x \in \mathbb{R}^N$ and characterized by a system of ordinary differential equations (ODEs). Let $f(x)$ be a first-order, autonomous DS describing a nominal motion plan for a robot, such that:

$$f : \mathbb{R}^N \to \mathbb{R}^N, \quad \dot{x} = f(x) \tag{7.1}$$

where $f : \mathbb{R}^N \to \mathbb{R}^N$ is a continuous differentiable, vector-valued function representing. Learning consists of estimating the function f, which is a map from the N-dimensional input state $x \in \mathbb{R}^N$ to its time-derivative $\dot{x} \in \mathbb{R}^N$. Training data consist of sample trajectories, which we assume to represent path integrals of the underlying DS. These trajectories cover a limited portion of the state space. The goal of the learning algorithm is to ensure that the learned DS reproduces the training data well, while generalizing over regions not covered by the data. We will, in particular, want to ensure that the system does not diverge from the training data points. To tackle this problem, we will embed constraints explicitly into the learning algorithm so that the learned dynamics offers guarantees from control theory. One desirable property is stability at an attractor, x^*. Hence, f must be such that:

$$\dot{x}^* = f(x^*) = 0, \quad \lim_{t \to \infty} x = x^* \tag{7.2}$$

Here, $f(\cdot) : \mathbb{R}^N \to \mathbb{R}^N$ is a continuous differentiable, vector-valued function representing a DS that converges on a single stable equilibrium point x^*, which is also called the *target* or *attractor*. To learn the function, we choose an expression for the function to be learned, which is parameterized through a set of parameters Θ such that $f(x; \Theta)$. Learning consists of updating the parameters Θ until the function fits as accurately as possible the reference trajectories, which can be measured through a loss function $L(X, f, \Theta)$. The problem at stake can then be rephrased as follows:

Given a dataset of M reference trajectories

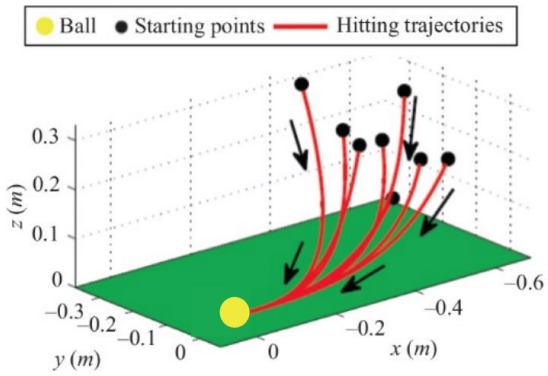
$$\{\boldsymbol{X}, \dot{\boldsymbol{X}}\} = \{X^m, \dot{X}^m\}_{m=1}^{M} = \{\{x^{t,m}, \dot{x}^{t,m}\}_{t=1}^{T_m}\}_{m=1}^{M}$$

where T_m is the length of each m-th trajectory, estimate the parameters Θ of the function $f(x; \Theta)$, to represent at best the reference trajectories according to a loss $L(X, f, \Theta)$. Furthermore, function f must be capable of generating motion that retains some of the characteristics of the reference trajectories in the state space not covered by the training data. This can be assessed by measuring the loss incurred on a testing set composed of reference trajectories not used at training. Finally, the function f must ensure that the target x^* is reached from any point in space. This can be summarized as two objectives:

to be continued...

Learning from Demonstration: Using dynamical systems

Going back to the previous chapter, looking at the golf ball example we can see that as the teacher moves the robot's joints passively, the robot records the state $x \in \mathbb{R}^N$ and velocity $\dot{x} \in \mathbb{R}^N$ of its end-effector at each time step. The set of demonstrations, then, consist in a set of M trajectories. We embed the dynamics of movement in a dynamical system (DS) of the form $\dot{x} = g(x)$. To do so, we use the stable estimator of dynamical systems (SEDS) method, which guarantees that the flow asymptotically reaches and stabilizes at the ball's location such that $\lim_{t\to\infty} x = x^*$ with x^* being the ball location.



This approach generates a flow of motion for the endeffector that ends up correctly for hitting the ball. However, this model is not sufficient for our task because with such a system, the robot would stop once it reaches the ball. To hit the ball so that it goes into the hole, the robot needs to reach the ball with a very specific velocity. To achieve this, we modulate the initial stable DS flow so that the velocity vector at the attractor is oriented and matches the amplitude.

To achieve this, we use our demonstration data points twice: first, we learn a stable flow at the ball that embeds the demonstrated orientation when approaching it. As this flow vanishes at the ball, we store only the unitary vector field $||g(x)|| = 1\ \forall x$. Second, we learn a (not necessarily stable)

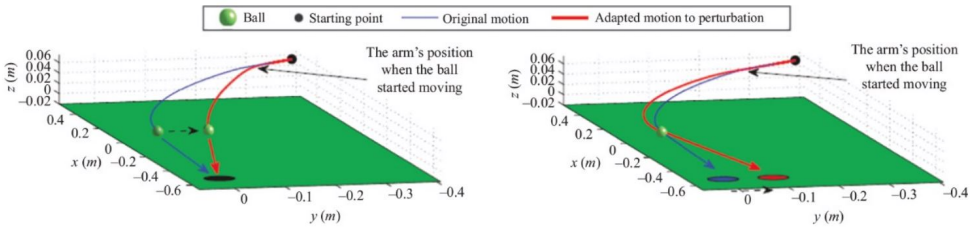
function $M(x)$ to represent the amplitude of the demonstrated velocity flow: $M(x) = ||\dot{x}||$. $M(x)$ can be learned using any machine learning technique for Regression.

The final dynamics is generated as a combination of the learned functions:

$$\dot{x} = f(x) = M(x)g(x)$$

Then $f(x)$ converges to the ball as it follows the flow indicated by $g(x)$ with the correct amplitude specified by $M(x^*)$.

Expressing the dynamics from the relative position of the ball to the sink allows one to nicely generalize the orientation toward the ball without further demonstrations. The learned DS can adapt at run time to perturbations, such as pushing the robot away from its trajectory (left) and moving the hole (right) by generating a new trajectory that reaches the target correctly.



Intro Dynamic Systems

Model

We assume that the motion of the robotic system is fully defined by its state $x \in \mathbb{R}^N$ and characterized by a system of ordinary differential equations ($ODEs$). Let $f(x)$ be a first-order, autonomous DS describing a nominal motion plan for a robot, such that:

$$f : \mathbb{R}^N \to \mathbb{R}^N, \quad \dot{x} = f(x),$$

where $f(\cdot)$ is a continuous differentiable, vector-valued function representing. Learning consists of estimating the function f, which is a map from the N-dimensional input state $x \in \mathbb{R}^N$ to its derivative $\dot{x} \in \mathbb{R}^N$.

Training data consists of sample trajectories, which we assume to represent path integrals of the underlying DS. These trajectories cover a limited portion of the state space. The goal of the learning algorithm is to ensure that the learned DS reproduces the training data well, while generalizing over regions not covered by the data. We will, in particular, want to ensure that the system does not diverge from the training data points.

To tackle this problem, we will embed constraints explicitly into the learning algorithm so that the learned dynamics offers guarantees from control theory. One desirable property is stability at an attractor, x^*. Hence, f must be such that:

$$\dot{x}^* = f(x^*) = 0, \quad \lim_{t \to \infty} x = x^*$$

Here, $f(\cdot)$ is a continuous differentiable, vector-valued function representing a DS that **converges on a single stable equilibrium point x^*, which is also called attractor** (or target).

The stability of an equilibrium point x^* can be classified as

- **Unstable** if x moves away from x^* with time.
- **Asymptotically stable** if $x \to x^*$ from all $x(0) \in \mathcal{D}$ as t tends to infinity, $\mathcal{D}$ is a subset of the state space.
- **Globally asymptotically stable** if $x \to x^*$ as t tend to infinity, from all $x(0)$ in the state space.
- **Exponentially stable** if the rate of convergence to x^* is exponentially fast within $\mathcal{D}$.
- **Globally exponentially stable** if the rate of convergence to x^* is exponentially fast from everywhere in the state space.

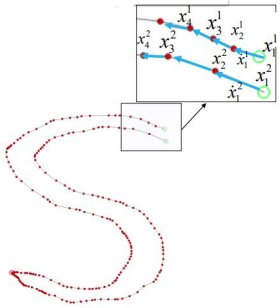
We make these notions mathematically precise after a study of stability in linear DS.

We can model the robot as a point mass moving according to a time invariant autonomous dynamical system (DS).

Training Data

Given a set of M reference trajectories

$$\left\{X,\ \dot{X}\right\} = \left\{\left\{x_t^i,\ \dot{x}_t^i\right\}_{t=1}^{T_m}\right\}_{m=1}^{M},$$



where T_m is the length of each m-th trajectory, estimate the parameters Θ of the function $f(x;\Theta)$, to represent at best the reference trajectories according to a loss $L(X, f, \Theta)$ (Any regression technique could be used).

This can be summarized as two objectives:

1. Reproduce the reference dynamics.
2. Converge to the attractor.

From a machine learning perspective, estimating $\dot{x} = f(x)$ from data can be framed as a regression problem, where the inputs are the state variables x and the outputs are their first-order derivatives $\dot{x}$.

We will see two methods from machine learning for estimating f:

- **Gaussian Mixture Regression**

  With GMR, a first-order DS, $\dot{x} = f(x)$, is estimated by learning about a joint density of position and velocity measurements through a K-component Gaussian mixture model (GMM) as follows:

$$p(x, \dot{x}|\Theta_{GMR}) = \sum_{k=1}^{K} \pi_k\ p(x, \dot{x}|\mu^k, \Sigma^k) = \sum_{k=1}^{K} \pi_k\ \mathcal{N}(\ \cdot\ |\ \mu, \Sigma)$$

We query for the velocity by conditioning the output for each velocity. This can be written as a compact form:

$$\dot{x} = f(x; \Theta_{GMR}) = E\left\{p(\dot{x}|x)\right\} = \sum_{k=1}^{K} \gamma_k(x)\tilde{\mu}^k(x).$$

- **Support vector regression**

  If we use SVR we must fit each of the velocity dimensions independently. With SVR, a first-order DS $\dot{x} = f(x)$, is estimated by learning N regressive functions, one for each n-th output dimension (i.e., $f(x) = [f_1(x), \ldots, f_n(x), \ldots, f_N(x)]^T$) as follows:

$$\dot{x}_n = f_n(x; \Theta^n_{SVR}) = \sum_{i=1}^{L} (\alpha_i - \alpha_i^*)_n k_n(x, x^i) + b_n$$

  with L being the number of all input/output pairs (position/velocity measurements) in the collection of reference trajectories. Here $\alpha_i,\ \alpha_i^*$, are the weights that indicate which i-th input data point is a support vector that is, when $(\alpha_i - \alpha_i^*)_n > 0$, x_i is a support vector for the n-th regressor. Here, $b_n \in \mathbb{R}$ is the bias term. The term $k_n(x, x^i)$ represents the kernel function used in each n-th regressive function.

Summing up:

- Learning a control law composed of a dynamical system can be formulated as a regression problem.
  - We regress on the velocity given the state.
- Using out of the box machine learning techniques for regression is insufficient:
  - It may lead to imprecise estimate of the attractor.
  - Many trajectories may drift away from the attractor.
  - Spurious fixed point attractors with spurious local dynamics may arise

Lyapunov's Theorem for Global Asymptotic Stability

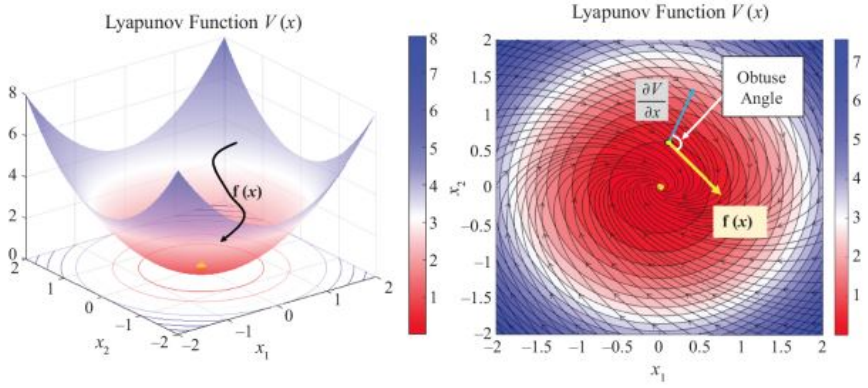
Theorem A function $\dot{x} = f(x)$ is a DS that is **Global Asymptotically Stable (GAS)** at the attractor x^*, if there exists a continuous and continuously differentiable Lyapunov candidate function $V(x) : \mathbb{R}^N \to \mathbb{R}$ that is radially unbounded; i.e., $V(x) \to \infty$ as $||x|| \to \infty$, C^1 and satisfies the following conditions:

$$(i)\ V(x^*) = 0, \qquad (ii)\ V(x^*) > 0\ \forall x \in \mathbb{R}^N \setminus \{x = x^*\}$$

$$(iii)\ \dot{V}(x^*) = 0, \qquad (iiii)\ \dot{V}(x^*) > 0\ \forall x \in \mathbb{R}^N \setminus \{x = x^*\}$$

DS to be GAS there should be a corresponding energy-like function $V(x)$, that is non-increasing along all the trajectories of $f(x)$.

$$\dot{V}(x^*) = \frac{\partial V(x)}{\partial x} f(x) < 0$$



In control theory, the most common Lyapunov function used for linear and/or linearized DS is the **Quadratic Lyapunov Function**, (QLF) that is:

$$V(x) = \frac{1}{2}(x - x^*)^T(x - x^*)$$

we showcase the derivation of sufficient stability conditions for a linear time-invariant (LTI) DS of the following form:

$$\dot{x} = f(x) = Ax + b$$

where b can be seen as an offset or translation of the origin of the DS, and A is the linear system matrix that defines the dynamics of the LTI system.

How to ensure $\dot{V}(x)$ is always negative? By enforcing the eigenvalues to be negative! Hence, the linear DS is GAS at the attractor x^* if:

$$\begin{cases} b = -Ax^* \\ A^T + A \prec 0 \end{cases}$$

where $\prec$ refers to negative definiteness of a matrix, respectively. A matrix A is deemed negative definite if its symmetric part has all negative eigenvalues.

What if $f(x)$ is non-linear?

- Not easy to assess whether the system is stable.
- Traditionally, the following has been done: local linearization, numerical estimation of stability, analytical solution in special cases.

Stable Estimator of Dynamical Systems (SEDS)

How to model this non-linear dynamical system?

Start with sampled trajectories from a nonlinear DS. Model the data with a probabilistic model:

$$p(x, \dot{x}; \Theta_{GMR}) = \sum_{k=1}^{K} \pi_k \; p(\dot{x}, x; \mu^k, \Sigma^k)$$

$$\text{with} \quad p(\dot{x}, x; \mu^k, \Sigma^k) = \mathcal{N}(\mu^k, \Sigma^k), \quad 0 < \pi_k \leq 1$$

where $\Theta_{GMR} = \{\pi_k, \mu^k, \Sigma^k\}$ priors probability, means and covariance matrices of the K Gauss functions.

Generate an estimate of the DS using GMR:

$$\dot{x} = f(x; \Theta_{GMR}) = E\{p(\dot{x}|x)\} = \sum_{k=1}^{K} \gamma_k(x)(A^k x + b^k)$$

where $A^k x + b^k$ generates K linear DS, but we have a nonlinearity coming from $\gamma_k(x) = \frac{\pi_k \cdot p(x; \mu_x^k, \Sigma_x^k)}{\sum_{k=1}^{K} \alpha_k \cdot p(x; \mu_x^k, \Sigma_x^k)}$

Each linear dynamics $A^k x + b^k$ corresponds to a line that passes through the centers of the Gaussian μ^k with slope A^k. First eigenvector of each A^k matrix gives direction of velocity of DS locally. Model is parametrized only by A^k and b^k vectors. To guarantee that the learned DS will be GAS at a target x^*, we propose the following sufficient conditions.

Theorem The nonlinear DS defined by $\dot{x} = f(x; \Theta_{GMR})$ parametrized by $\Theta_{GMR} = \{\pi_k, \mu^k, \Sigma^k\}_{k=1}^K$ is GAS at the target $x^* \in \mathbb{R}^N$ if:

$$\begin{cases} b^k = -A^k x^* & \textbf{Stability at attractor} \\ A^k + (A^k)^T \prec 0 & \textbf{Energy decreases} \end{cases} \quad \forall k = 1 \dots K$$

We now introduce a procedure for computing the unknown parameters Θ_{GMR}, which we use to describe the SEDS. Θ_{GMR} is found solving an optimization problem under the GAS constraints defined previous equation of the *Theorem*.

We need to ensure that the flow is aligned as closely as possible to demonstrated trajectories. We, thus, consider two candidates for the objective function of the SEDS algorithm:

- **Maximum Likelihood** $\rightarrow$ fits at best the entire density.
- **Mean Square Error** $\rightarrow$ fits at best the state space trajectories and velocities.

When trained with mean-square error (MSE) as objective function, the Gauss function no longer need to fit the distribution of the data.

Hyperparameter and pre-selections for SEDS

Prior to training SEDS, the user must make a number of choices that will influence the quality of the learned model. The choices are:

- Type of objective function

  $\rightarrow$ This will affect the placement of the Gauss functions.

- Number of Gauss functions

  → This can be automated by using the Bayesian Information Criterion(BIC) – BIC finds a balance between improved quality of the fit and increase in number of parameters.

SEDS Summary

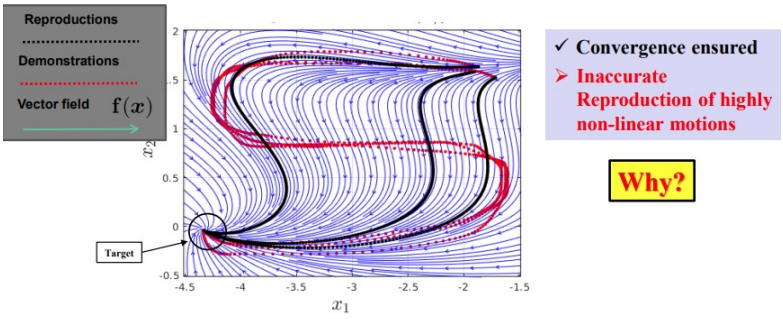
Automatically estimate globally asymptotically stable dynamical systems from sampled trajectories

- Extension of Gaussian Mixture Model
  - Uses same objective function (maximum likelihood)
  - Add new set of constraints to enforce stability
- Stability is guaranteed through Lyapunov stability constraints
  - Assumes a quadratic Lyapunov function
- High accuracy for a large number of nonlinear dynamics
- Limitations:
  - Non convex optimization
  - Poor accuracy for highly nonlinear dynamics (high curvature)

We started this section by formulating learning of the control law using constrained GMR. We presented one approach, SEDS, to learn the parameters of the GMR using a constrained nonconvex optimization. This formulation guarantees that the resulting control law is GAS at a single attractor. Moreover, we showed that robot motion generated with such a model provides online adaptation to changes in the target's location. However, the approach suffers from a number of limitations, which led to an alternative approach.

Linear Parameter Varying Dynamical Systems (LPVDS)

SEDS on Highly Non-Linear Trajectories



In case V is too conservative, highly non-linear trajectories violate stability condition

$$\dot{V}(x^*) = \frac{\partial V(x)}{\partial x} f(x) < 0$$

Observe that the mixture of linear DS, defined

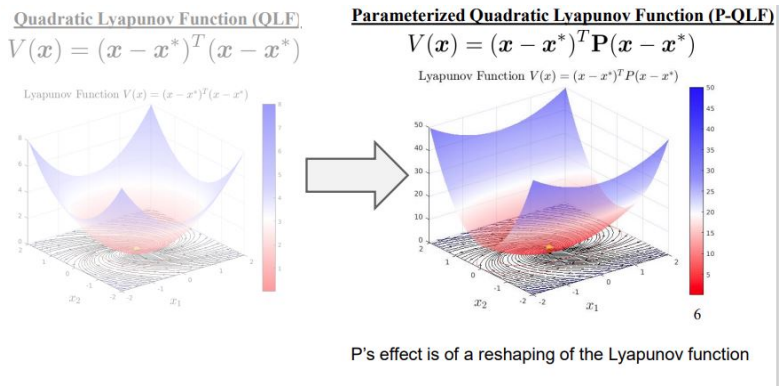
$$\dot{x} = \sum_{k=1}^{K} \gamma_k(x)(A^k x + b^k)$$

is a linear parameter varying (LPV) system, with each $(A^k x + b^k)$ as an LTI system, and $\gamma_k(x)$ a state-dependent parameter vector $\gamma = [\gamma_1, \ldots, \gamma_K]$.
In LTI systems, Lyapunov functions of the form $V(x) = (x - x^*)^T P(x - x^*)$ are commonly used to ensure stability. The matrix P becomes a parameter of the QLF. Replacing the QLF of SEDS with a P-QLF function yields the sufficient conditions described next to ensure GAS at x^*.

Theorem The nonlinear DS $\dot{x}$ is GAS at an attractor x^* if $\exists P = P^T$ and $P \succ 0$, with $V(x) = (x - x^*)^T P(x - x^*)$, such that:

$$\begin{cases} (A^k)^T P + P A^k = Q^k, \quad Q^k = (Q^k)^T \prec 0 \\ b^k = -A^k x^* \end{cases} \qquad \forall k = 1 \ldots K$$

Goal: learn the parameters of a non-linear DS with P-QLF.



Has we can see, P's effect is of a reshaping of the Lyapunov function.

LPVDS Summary

LPV-DS was offered as an alternative to SEDS to enable learning of more complex, and nonlinear DS from demonstrations.

SEDS	LPV - DS
Fix by hand number of Gaussians	**Learns automatically number of Gaussians**
Conservative stability constraints → Cannot learn highly non-linear trajectories	**Less conservative stability constraints → Can embed large non-linearities**

Adapting and Modulating an Existing Control Law

There are, however, many occasions when it would be useful to be able to train the system again, such as to enable a robot to take a different approach path toward a target. Often, the changes apply only to a small region of the state space. Hence, it would be useful to be able to retrain the controller by modifying the original flow only locally. This chapter shows how one can learn to modulate an initial (nominal) dynamical system (DS) to generate new dynamics. We consider modulations that act locally to preserve the

generic properties of the nominal DS (e.g., asymptotic or global stability). Let us assume that we have at our disposal a nominal DS $\dot{x} = f(x)$ which is asymptotically stable at a fixed-point attractor x^*, s.t. $\dot{x}^* = f(x^*) = 0$.

How can we modulate the DS while preserving the stability properties?

Modulate the original DS in such a way that the new DS:

- Remains a first order DS
- Preserves asymptotic stability at its attractor x^*
- Has still a single attractor

We can modulate this nominal DS to generate new dynamics, $g(x)$, by multiplying $f(x)$ by a continuous matrix function $M(x) \in \mathbb{R}^{N \times N}$. The modulated dynamics is then given by:

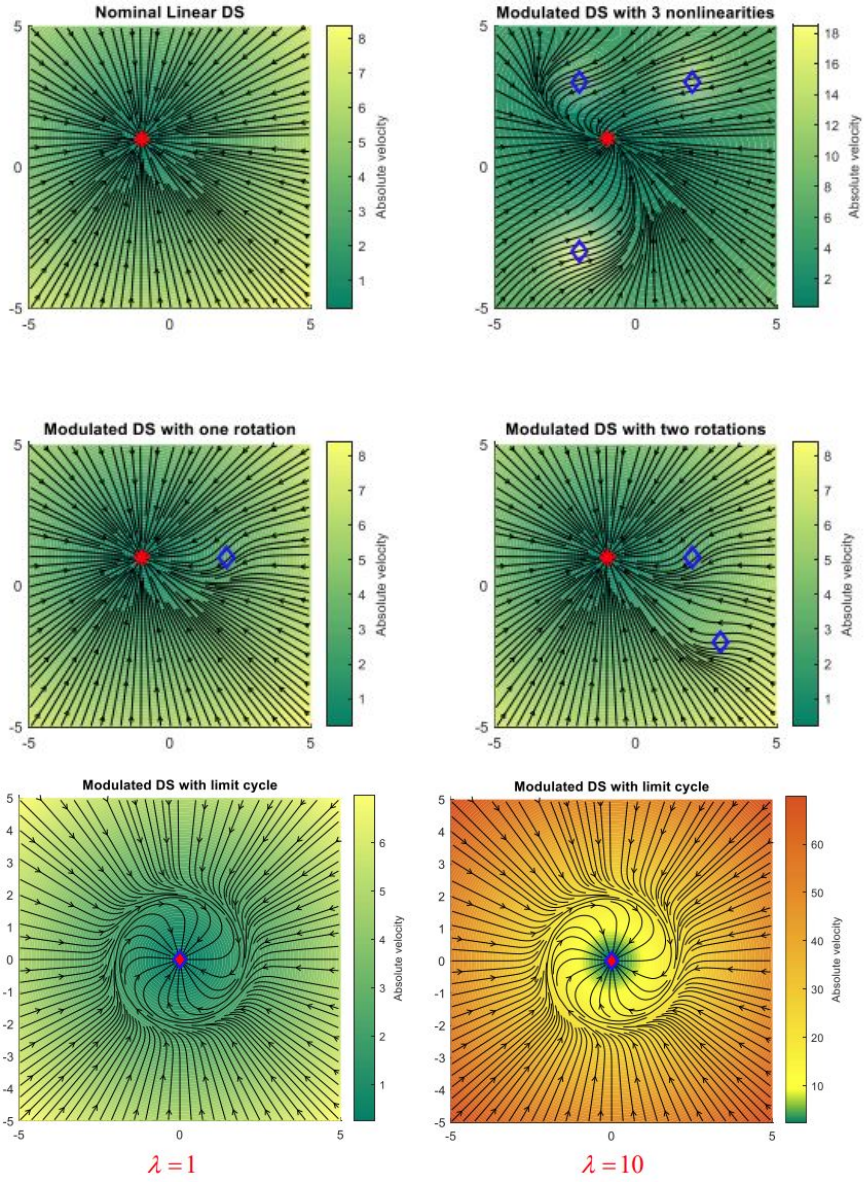
$$\dot{x} = g(x) = M(x)f(x)$$

The modulation should be local.

We want that:

- Preserve stability at attractor $M(x^*) = I$
- Uniqueness of the attractor $M(x)f(x) \neq 0 \quad \forall x \neq x^*$, M must be full rank
- Locality of the modulation $\exists$ a compact region $B(x, x^*)$ around x^* where $M(x) = I$

The simplest activation one can think of is a rotation, active only locally. The effect of the modulation vanishes exponentially from that point. Modulation can be used to generate new properties for the DS. Also multiplicative term in front of the modulation will increase/decrease the speed of the DS but will not modify the type of dynamics $\dot{x} = \lambda M(x)f(x)$.



Parameterization of the modulation

$$\dot{x} = M(x)f(x), \quad M(x) = (1 + \kappa(x))R(x)$$

where $\kappa(x)$ modulates speed and $R(x)$ is the rotation. They are active only locally, zero elsewhere. The modulation function can rotate and speed up the dynamics locally.

Bibliography

[1] David R. Woolley (12 February 2013). *PLATO: The Emergence of Online Community*,

[2] Crow, W. B. & Din, H. (2009)

[3] *Unbound By Place or Time: Museums and Online Learning*, Washington, DC: American Association of Museums, pp. 9-10

Graziadei, W. D., et al., 1997 *Building Asynchronous and Synchronous Teaching-Learning Environments: Exploring a Course/-Classroom Management System Solution*, `http://horizon.unc.edu/projects/monograph/CD/Technological_Tools/Graziadei.html`