Industrial automation and robotics

Motion planning

Prof. Paolo Rocco (paolo.rocco@polimi.it)
Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria
After the description of the motion of the robot through its kinematic model, let's now move on to the problem of motion planning.

We want to define the way in which the robot evolves from an initial posture to a final one.

Motion planning is one of the essential problems of robotics. Most of a robot's market success depends on the quality of motion planning.
Instruction stack: list of instructions to be executed, specified using the proprietary programming language

Trajectory generation: converts an instruction into a trajectory to be executed

Inverse kinematics: maps the trajectory from the Cartesian space to the joint space (if needed)

Axis controllers & drives: closes the control loop ensuring tracking performance
Elements of a motion planning and control system

- **Instruction stack**: list of instructions to be executed, specified using the proprietary programming language
- **Trajectory generation**: converts an instruction into a trajectory to be executed
- **Inverse kinematics**: maps the trajectory from the Cartesian space to the joint space (if needed)
- **Axis controllers & drives**: closes the control loop ensuring tracking performance
Motion programming

- To have a manipulator make appropriate movements, it is necessary to instruct it.

- This is done with appropriate commands that induce the robot to subsequently reach the points that correspond to the execution of the desired task.

- A robot program is a sequence of motion commands through the consecutive points ("targets").
A first mode for motion programming is the so called teaching-by-showing (also known as lead-through programming).

Using the teach-pendant, the operator moves the manipulator along the desired path. Position transducers memorize the positions the robot has to reach, which will be then jointed by a software for trajectory generation, possibly using some of the intermediate points as via points. The robot will be then able to autonomously repeat the motion.

No particular programming skills are requested to the operator. On the other hand the method has limitations, since producing a program requires that the programmer has the robot at his/her disposal (and then the robot is not productive).
The generation of the robot program can also be done in a robot programming environment. The robot programmer can move the robot in a virtual environment with high fidelity rendering of the robot motion in the robotic cell.

There are tools to record positions and to make the robot move along a path formed by such positions.

At the end the robot programming environment produces the code ready to be downloaded into the robot controller.
The robot programmer can also write the robot program directly using a robotic programming language.

With a **robotic programming language** the operator can program the motion of the robot as well as complex operations where the robot, inside a work-cell, interacts with other machines and devices. With respect to a general purpose programming language, the language provides specific robot oriented functionalities.

In the following, we will discuss some elements of the **PDL2** programming language by **COMAU Robotics**.
We need first to introduce some terminology concerning concepts which are often confused:

- **Path**: it is a geometric concept and stands for a line in a certain space (the space of Cartesian positions, the space of the orientations, the joint space,..) to be followed by the object whose motion has to be planned

- **Timing law**: it is the time dependence with which we want the robot to travel along the assigned path

- **Trajectory**: it is a path over which a timing law has been assigned
Trajectories in the operational space: the path (position and orientation) of the robot end effector is specified in the common Cartesian space.

It is necessary to specify:

- the end point of the movement
- the path that the end effector must follow

- task description is natural
- constraints on the path can be accounted for
- singular points or redundant degrees of freedom generate problems
- online kinematic inversion is needed
Trajectories in joint space: the desired joint positions are directly specified.

It is necessary to specify:
- the end point of the movement
- problems related to kinematic singularities and redundant degrees of freedom are solved directly
- it is a mode of interest when we just want that the axes move from an initial to a final pose (and we are not interested in the resulting motion of the end effector)
- online kinematic inversion is not needed
Reference frames

In PDL2 for each manipulator a world reference frame is defined. The operator might redefine the base frame ($\text{BASE}$) relative to the world frame. This is useful when the robot has to be repositioned in the work area, since it avoids re-computation of all the positions of objects in the cell.

Similarly the programmer can define a frame ($\text{TOOL}$) relative to the tool frame of the manipulator, which is useful whenever the tool mounted on the end effector is changed.
The program moves pieces from a feeder to a table or to a discard bin, depending on digital input signals:

Program example

```plaintext
PROGRAM pack
VAR
    home, feeder, table, discard : POSITION
BEGIN CYCLE
    MOVE TO home
    OPEN HAND 1
    WAIT FOR $DIN[1] = ON
    -- signals feeder ready
    MOVE TO feeder
    CLOSE HAND 1
    IF $DIN[2] = OFF THEN
    -- determines if good part
        MOVE TO table
    ELSE
        MOVE TO discard
    ENDIF
    OPEN HAND 1
    -- drop part on table or in bin
END pack
```

1. Feeder
2. Robot
3. Discard Bin
4. Table
Besides data types typical of any programming language (integer, real, boolean, string, array), some types specific to robotic applications are defined in PDL2. Among them:

**VECTOR**: representation of a vector through three components

**POSITION**: three components of Cartesian position, three orientation components (Euler angles) and a configuration string (which indicates whether it is a shoulder/elbow/wrist upper/lower configuration)

**JOINTPOS**: positions of the joints, measured in degrees
The instruction MOVE

With the instruction **MOVE**, **motion commands** of the arms are given. The format of the instruction is as follows:

```
MOVE <ARM[n]> <trajectory> dest_clause <opt_clauses> <sync_clause>
```

(note that a single controller can manage several arms).

The **trajectory clause** can take one of the following values:

- **LINEAR** (linear motion in Cartesian space)
- **CIRCULAR** (circular motion in Cartesian space)
- **JOINT** (motion in joint space)

The default is a motion in joint space.
There are several destination clauses for the instruction MOVE. Main ones are:

**MOVE TO**

It moves the arm towards the specified destination, which might be a variable of type POSITION or JOINTPOS. For example:

MOVE LINEAR TO POS(x, y, z, e1, e2, e3, config)
MOVE TO home

The optional **VIA** clause can be used with the MOVE TO destination clause to specify a position through which the arm passes between the initial position and the destination. The VIA clause is used most commonly to define an arc for circular moves. For example:

MOVE TO initial
MOVE CIRCULAR TO destination VIA arc
Using the optional clause `WITH` it is possible to assign values to some predefined temporary variables. In particular we can operate over the following variables:

$\text{PROG\_SPD\_OVR}$
It is a percentage by which we can modify the default speed value used by the controller for joint space motions.

$\text{PROG\_ACC\_OVR},\, \text{PROG\_DEC\_OVR}$
They are percentages by which we can modify the default values of acceleration and deceleration used by the controller for joint space motions.

$\text{LIN\_SPD}$
It is the value of linear speed for a Cartesian motion, expressed in m/s:

**Examples:**

```
MOVE TO p1 WITH $\text{PROG\_SPD\_OVR}=50
MOVE LINEAR TO p2 WITH $\text{LIN\_SPD}=0.6
```
Once the sequence of motion instructions has been prepared, the trajectory must be generated instant by instant.

It is therefore necessary to define at each instant (with an appropriate sampling time) how the motion of the robot must evolve in order to reach the final configuration provided by the movement instruction.

Depending on the given motion command, the generation of trajectories can take place in the joint space or in the operating space.
Instruction stack: list of instructions to be executed, specified using the proprietary programming language

Trajectory generation: converts an instruction into a trajectory to be executed

Inverse kinematics: maps the trajectory from the Cartesian space to the joint space (if needed)

Axis controllers & drives: closes the control loop ensuring tracking performance
When we plan the trajectory in the joint space we want to generate a function $q(t)$ which defines how each joint variable should evolve from the initial to the final value.

It is therefore sufficient to operate component by component (i.e. consider a single joint variable $q_i(t)$): in what follows we will then consider the generation of a single scalar coordinate.

When planning in the joint space, the definition of the path as a geometric entity is not relevant, since we are not interested in a coordinated movement of the joints (except for the fact that all the joints complete their movement at the same time).
The simplest case of trajectory planning for **point-to-point motion** is when some initial and final conditions are assigned on positions, velocities and possibly on acceleration and jerk and the travel time.

**Polynomial functions** of the following kind can be considered:

\[
q(t) = a_0 + a_1 t + a_2 t^2 + \cdots + a_n t^n
\]

The higher the degree \( n \) of the polynomial, the larger the number of boundary conditions that can be satisfied and the smoother the trajectory will be.
Cubic polynomials

Suppose that the following boundary conditions are assigned:

- an initial and a final instants $t_i$ and $t_f$
- initial position and velocity $q_i$ and $\dot{q}_i$
- final position and velocity $q_f$ and $\dot{q}_f$

We then have four boundary conditions. In order to satisfy them we need a polynomial of order at least equal to three (cubic polynomial):

$$q(t) = a_0 + a_1(t - t_i) + a_2(t - t_i)^2 + a_3(t - t_i)^3$$

If we impose the boundary conditions:

- $q(t_i) = q_i$
- $\dot{q}(t_i) = \dot{q}_i$
- $q(t_f) = q_f$
- $\dot{q}(t_f) = \dot{q}_f$

we obtain:

- $a_0 = q_i$
- $a_1 = \dot{q}_i$
- $a_2 = \frac{-3(q_i - q_f) - (2\dot{q}_i + \dot{q}_f)T}{T^2}$
- $a_3 = \frac{2(q_i - q_f) + (\dot{q}_i + \dot{q}_f)T}{T^3}$

$$T = t_f - t_i$$
Cubic polynomials: example

t_i = 0, \ t_f = 1 \ s,

q_i = 10^\circ, \ q_f = 30^\circ,

\dot{q}_i = \dot{q}_f = 0^\circ/s
A quite common industrial practice to generate the trajectory consists in planning a linear position profile adjusted at the beginning and at the end of the trajectory with parabolic bends. The resulting velocity profile has the typical trapezoidal shape.

The trajectory is then composed of three parts:

1. Constant accel., linear velocity, parabolic position;
2. Zero acceleration, constant velocity, linear position;
3. Constant deceleration, linear velocity, parabolic position.

Often the duration $T_a$ of the acceleration phase (phase 1) is set equal to the duration of the deceleration phase (phase 3): this way a trajectory is obtained, which is symmetric with respect to the central time instant. Of course it has to be $T_a \leq (t_f - t_i)/2$. 

### Trapezoidal velocity profile (TVP)

![Trapezoidal velocity profile diagram](image)
\[ t_i = 0, \quad t_f = 4s, \quad T_a = 1s, \]

\[ q_i = 0^\circ, \quad q_f = 30^\circ, \quad \dot{q}_v = 10^\circ/s \]
Trajectories in the operational space

Trajectory planning in the joint space yields unpredictable motions of the end-effector. When we want the motion to evolve along a predefined path in the operational space, it is necessary to plan the trajectory directly in this space.

Trajectory planning in the operational space entails both a path planning problem and a timing law planning problem: both the path and the timing law can be expressed analytically, as it will be shown in the following.

We will first address the trajectory planning for the position, and then we will concentrate on the orientation.
Let us consider a parametric representation of a curve in space. The parameterization can be performed with respect to the **natural coordinate** (length of the arc of trajectory) \( p = p(s) \).

We can define the tangent unit vector as:

\[
t = \frac{dp(s)}{ds}
\]
As an example of path parameterization we can consider a segment in space (linear Cartesian path):

\[ p(s) = p_1 + \frac{s}{\|p_2 - p_1\|}(p_2 - p_1) \]
Linear paths can be concatenated in order to obtain more elaborated paths.

The intermediate point between two consecutive segments can be considered as a *via point*, meaning that there is no need to pass and stop there.

During the *over-fly*, i.e. the passage near a via point, the path remains always in the plane specified by the two lines intersecting in the via point. This means that the problem of planning the over-fly is planar.

Formulas can be derived to define the blending (typically a parabolic one)
A circular path can be defined assigning three points in space belonging to the same plane:
For the **position trajectories**, taking into account the parameterization of the path with respect to the natural coordinate $\mathbf{p} = \mathbf{p}(s)$, we will assign the **timing law** through the function $s(t)$.

In order to determine function $s(t)$ we can use any of the timing laws (polynomials, trapezoidal velocity profile, etc.) Also we notice that:

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

$|\dot{s}|$ is then the **norm of the velocity**. The time law $s(t)$ takes then an immediate meaning!
Orientation trajectories

Orientation can be planned by resorting to the *axis/angle representation*.

The axis is kept fixed while the angle changes with time.
Elements of a motion planning and control system

- **Instruction stack**: list of instructions to be executed, specified using the proprietary programming language
- **Trajectory generation**: converts an instruction into a trajectory to be executed
- **Inverse kinematics**: maps the trajectory from the Cartesian space to the joint space (if needed)
- **Axis controllers & drives**: closes the control loop ensuring tracking performance