CHAPTER 12

Robot programming languages and systems

12.1 INTRODUCTION

In this chapter, we begin to consider the interface between the human user and an industrial robot. It is by means of this interface that a user takes advantage of all the underlying mechanics and control algorithms we have studied in previous chapters.

The sophistication of the user interface is becoming extremely important as manipulators and other programmable automation are applied to more and more demanding industrial applications. It turns out that the nature of the user interface is a very important concern. In fact, most of the challenge of the design and use of industrial robots focuses on this aspect of the problem.

Robot manipulators differentiate themselves from fixed automation by being "flexible," which means programmable. Not only are the movements of manipulators programmable, but, through the use of sensors and communications with other factory automation, manipulators can *adapt* to variations as the task proceeds.

In considering the programming of manipulators, it is important to remember that they are typically only a minor part of an automated process. The term **workcell** is used to describe a local collection of equipment, which may include one or more manipulators, conveyor systems, parts feeders, and fixtures. At the next higher level, workcells might be interconnected in factorywide networks so that a central control computer can control the overall factory flow. Hence, the programming of manipulators is often considered within the broader problem of programming a variety of interconnected machines in an automated factory workcell.

Unlike that in the previous 11 chapters, the material in this chapter (and the next chapter) is of a nature that constantly changes. It is therefore difficult to present this material in a detailed way. Rather, we attempt to point out the underlying fundamental concepts, and we leave it to the reader to seek out the latest examples, as industrial technology continues to move forward.

12.2 THE THREE LEVELS OF ROBOT PROGRAMMING

There have been many styles of user interface developed for programming robots. Before the rapid proliferation of microcomputers in industry, robot controllers resembled the simple sequencers often used to control fixed automation. Modern approaches focus on computer programming, and issues in programming robots include all the issues faced in general computer programming—and more.

Teach by showing

Early robots were all programmed by a method that we will call **teach by showing**, which involved moving the robot to a desired goal point and recording its position in a memory that the sequencer would read during playback. During the teach phase, the user would guide the robot either by hand or through interaction with a **teach pendant**. Teach pendants are handheld button boxes that allow control of each manipulator joint or of each Cartesian degree of freedom. Some such controllers allow testing and branching, so that simple programs involving logic can be entered. Some teach pendants have alphanumeric displays and are approaching hand-held terminals in complexity. Figure 12.1 shows an operator using a teach pendant to program a large industrial robot.

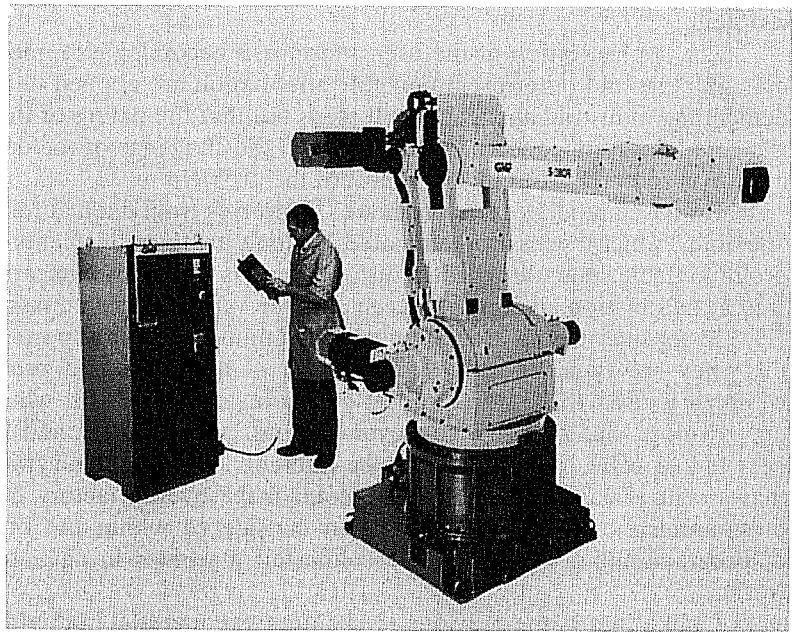

FIGURE 12.1: The GMF S380 is often used in automobile-body spot-welding applications. Here an operator uses a teach-pendant interface to program the manipulator. Photo courtesy of GMFanuc Corp.

Explicit robot programming languages

Ever since the arrival of inexpensive and powerful computers, the trend has been increasingly toward programming robots via programs written in computer programming languages. Usually, these computer programming languages have special features that apply to the problems of programming manipulators and so are called **robot programming languages** (RPLs). Most of the systems that come equipped with a robot programming language have nonetheless retained a teach-pendant-style interface also.

Robot programming languages have likewise taken on many forms. We will split them into three categories:

1. **Specialized manipulation languages**. These robot programming languages have been built by developing a completely new language that, although addressing robot-specific areas, might well be considered a general computer programming language. An example is the VAL language developed to control the industrial robots by Unimation, Inc [1]. VAL was developed especially as a manipulator control language; as a general computer language, it was quite weak. For example, it did not support floating-point numbers or character strings, and subroutines could not pass arguments. A more recent version, V-II, provided these features [2]. The current incarnation of this language, V+, includes many new features [13]. Another example of a specialized manipulation language is AL, developed at Stanford University [3]. Although the AL language is now a relic of the past, it nonetheless provides good examples of some features still not found in most modern languages (force control, parallelism). Also, because it was built in an academic environment, there are references available to describe it [3]. For these reasons, we continue to make reference to it.
2. **Robot library for an existing computer language**. These robot programming languages have been developed by starting with a popular computer language (e.g., Pascal) and adding a library of robot-specific subroutines. The user then writes a Pascal program making use of frequent calls to the predefined subroutine package for robot-specific needs. An examples is AR-BASIC from American Cimflex [4], essentially a subroutine library for a standard BASIC implementation. JARS, developed by NASA's Jet Propulsion Laboratory, is an example of such a robot programming language based on Pascal [5].
3. **Robot library for a new general-purpose language**. These robot programming languages have been developed by first creating a new general-purpose language as a programming base and then supplying a library of predefined robot-specific subroutines. Examples of such robot programming languages are RAPID developed by ABB Robotics [6], AML developed by IBM [7], and KAREL developed by GMF Robotics [8].

Studies of actual application programs for robotic workcells have shown that a large percentage of the language statements are not robot-specific [7]. Instead, a great deal of robot programming has to do with initialization, logic testing and branching, communication, and so on. For this reason, a trend might develop to

move away from developing special languages for robot programming and move toward developing extensions to general languages, as in categories 2 and 3 above.

Task-level programming languages

The third level of robot programming methodology is embodied in **task-level programming languages**. These languages allow the user to command desired subgoals of the task directly, rather than specify the details of every action the robot is to take. In such a system, the user is able to include instructions in the application program at a significantly higher level than in an explicit robot programming language. A task-level robot programming system must have the ability to perform many planning tasks automatically. For example, if an instruction to "grasp the bolt" is issued, the system must plan a path of the manipulator that avoids collision with any surrounding obstacles, must automatically choose a good grasp location on the bolt, and must grasp it. In contrast, in an explicit robot programming language, all these choices must be made by the programmer.

The border between explicit robot programming languages and task-level programming languages is quite distinct. Incremental advances are being made to explicit robot programming languages to help to ease programming, but these enhancements cannot be counted as components of a task-level programming system. True task-level programming of manipulators does not yet exist, but it has been an active topic of research [9, 10] and continues as a research topic today.

12.3 A SAMPLE APPLICATION

Figure 12.2 shows an automated workcell that completes a small subassembly in a hypothetical manufacturing process. The workcell consists of a conveyor under computer control that delivers a workpiece; a camera connected to a vision system, used to locate the workpiece on the conveyor; an industrial robot (a PUMA 560 is pictured) equipped with a force-sensing wrist; a small feeder located on the work surface that supplies another part to the manipulator; a computer-controlled press that can be loaded and unloaded by the robot; and a pallet upon which the robot places finished assemblies.

The entire process is controlled by the manipulator's controller in a sequence, as follows:

1. The conveyor is signaled to start; it is stopped when the vision system reports that a bracket has been detected on the conveyor.
2. The vision system judges the bracket's position and orientation on the conveyor and inspects the bracket for defects, such as the wrong number of holes.
3. Using the output of the vision system, the manipulator grasps the bracket with a specified force. The distance between the fingertips is checked to ensure that the bracket has been properly grasped. If it has not, the robot moves out of the way and the vision task is repeated.
4. The bracket is placed in the fixture on the work surface. At this point, the conveyor can be signaled to start again for the next bracket—that is, steps 1 and 2 can begin in parallel with the following steps.

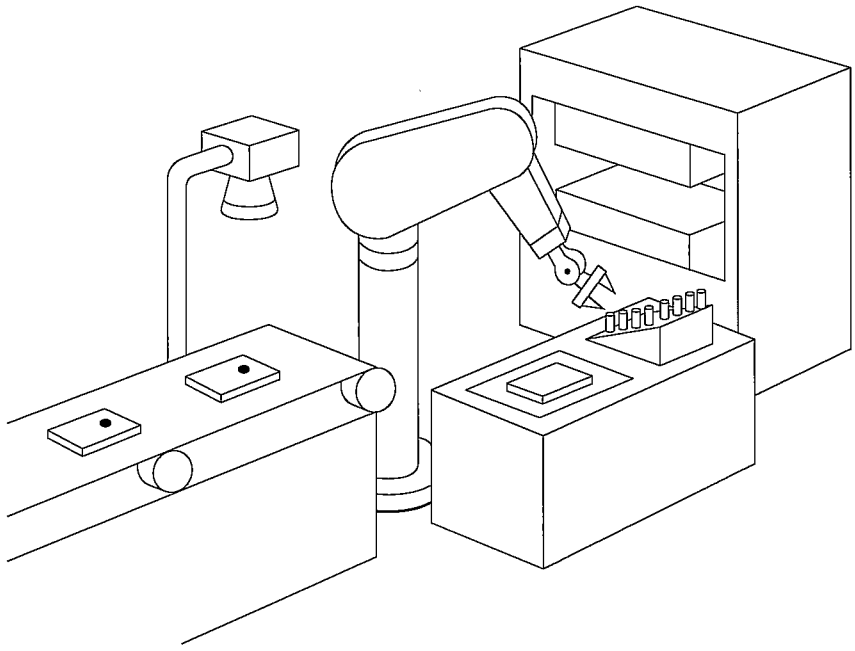

FIGURE 12.2: An automated workcell containing an industrial robot.

5. A pin is picked from the feeder and inserted partway into a tapered hole in the bracket. Force control is used to perform this insertion and to perform simple checks on its completion. (If the pin feeder is empty, an operator is notified and the manipulator waits until commanded to resume by the operator.)
6. The bracket–pin assembly is grasped by the robot and placed in the press.
7. The press is commanded to actuate, and it presses the pin the rest of the way into the bracket. The press signals that it has completed, and the bracket is placed back into the fixture for a final inspection.
8. By force sensing, the assembly is checked for proper insertion of the pin. The manipulator senses the reaction force when it presses sideways on the pin and can do several checks to discover how far the pin protrudes from the bracket.
9. If the assembly is judged to be good, the robot places the finished part into the next available pallet location. If the pallet is full, the operator is signaled. If the assembly is bad, it is dropped into the trash bin.
10. Once Step 2 (started earlier in parallel) is complete, go to Step 3.

This is an example of a task that is possible for today's industrial robots. It should be clear that the definition of such a process through "teach by showing" techniques is probably not feasible. For example, in dealing with pallets, it is laborious to have to teach all the pallet compartment locations; it is much preferable to teach only the corner location and then compute the others from knowledge of the dimensions of the pallet. Further, specifying interprocess signaling and setting up parallelism by using a typical teach pendant or a menu-style interface is usually

not possible at all. This kind of application necessitates a robot programming-language approach to process description. (See Exercise 12.5.) On the other hand, this application is too complex for any existing task-level languages to deal with directly. It is typical of the great many applications that must be addressed with an explicit robot programming approach. We will keep this sample application in mind as we discuss features of robot programming languages.

12.4 REQUIREMENTS OF A ROBOT PROGRAMMING LANGUAGE

World modeling

Manipulation programs must, by definition, involve moving objects in three-dimensional space, so it is clear that any robot programming language needs a means of describing such actions. The most common element of robot programming languages is the existence of special **geometric types**. For example, *types* are introduced to represent joint-angle sets, Cartesian positions, orientations, and frames. Predefined operators that can manipulate these types often are available. The "standard frames" introduced in Chapter 3 might serve as a possible model of the world: All motions are described as tool frame relative to station frame, with goal frames being constructed from arbitrary expressions involving geometric types.

Given a robot programming environment that supports geometric types, the robot and other machines, parts, and fixtures can be modeled by defining named variables associated with each object of interest. Figure 12.3 shows part of our example workcell with frames attached in task-relevant locations. Each of these frames would be represented with a variable of type "frame" in the robot program.

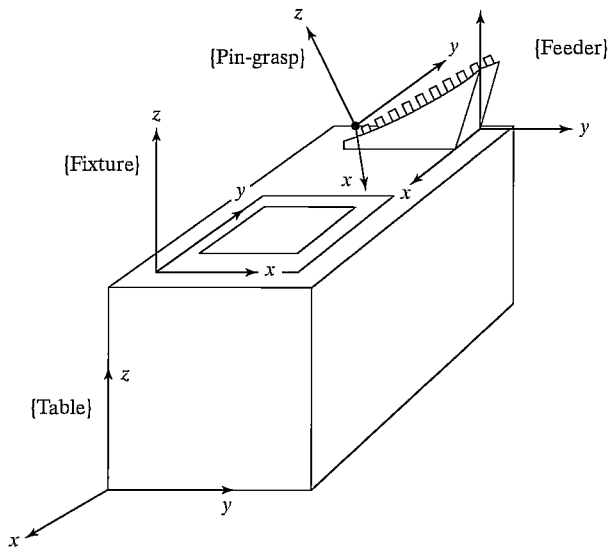

FIGURE 12.3: Often, a workcell is modeled simply, as a set of frames attached to relevant objects.

In many robot programming languages, this ability to define named variables of various geometric types and refer to them in the program forms the basis of the world model. Note that the physical shapes of the objects are not part of such a world model, and neither are surfaces, volumes, masses, or other properties. The extent to which objects in the world are modeled is one of the basic design decisions made when designing a robot programming system. Most present-day systems support only the style just described.

Some world-modeling systems allow the notion of **affixments** between named objects [3]—that is, the system can be notified that two or more named objects have become "affixed"; from then on, if one object is explicitly moved with a language statement, any objects affixed to it are moved with it. Thus, in our application, once the pin has been inserted into the hole in the bracket, the system would be notified (via a language statement) that these two objects have become affixed. Subsequent motions of the bracket (that is, changes to the value of the frame variable "bracket") would cause the value stored for variable "pin" to be updated along with it.

Ideally, a world-modeling system would include much more information about the objects with which the manipulator has to deal and about the manipulator itself. For example, consider a system in which objects are described by CAD-style models that represent the spatial shape of an object by giving definitions of its edges, surfaces, or volume. With such data available to the system, it begins to become possible to implement many of the features of a task-level programming system. These possibilities are discussed further in Chapter 13.

Motion specification

A very basic function of a robot programming language is to allow the description of desired motions of the robot. Through the use of motion statements in the language, the user interfaces to path planners and generators of the style described in Chapter 7. Motion statements allow the user to specify via points, the goal point, and whether to use joint-interpolated motion or Cartesian straight-line motion. Additionally, the user might have control over the speed or duration of a motion.

To illustrate various syntaxes for motion primitives, we will consider the following example manipulator motions: (1) move to position "goal1," then (2) move in a straight line to position "goal2," then (3) move without stopping through "via1" and come to rest at "goal3." Assuming all of these path points had already been taught or described textually, this program segment would be written as follows:

In VAL II,

```
move goal1
moves goal2
move via1
move goal3
```

In AL (here controlling the manipulator "garm"),

```
move garm to goal1;
move garm to goal2 linearly;
move garm to goal3 via via1;
```

Most languages have similar syntax for simple motion statements like these. Differences in the basic motion primitives from one robot programming language to another become more apparent if we consider features such as the following:

1. the ability to do math on such structured types as frames, vectors, and rotation matrices;
2. the ability to describe geometric entities like frames in several different convenient representations—along with the ability to convert between representations;
3. the ability to give constraints on the duration or velocity of a particular move—for example, many systems allow the user to set the speed to a fraction of maximum, but fewer allow the user to specify a desired duration or a desired maximum joint velocity directly;
4. the ability to specify goals relative to various frames, including frames defined by the user and frames in motion (on a conveyor, for example).

Flow of execution

As in more conventional computer programming languages, a robot programming system allows the user to specify the flow of execution—that is, concepts such as testing and branching, looping, calls to subroutines, and even interrupts are generally found in robot programming languages.

More so than in many computer applications, parallel processing is generally important in automated workcell applications. First of all, very often two or more robots are used in a single workcell and work simultaneously to reduce the cycle time of the process. Even in single-robot applications, such as the one shown in Fig. 12.2, other workcell equipment must be controlled by the robot controller in a parallel fashion. Hence, *signal* and *wait* primitives are often found in robot programming languages, and occasionally more sophisticated parallel-execution constructs are provided [3].

Another frequent occurrence is the need to monitor various processes with some kind of sensor. Then, either by interrupt or through polling, the robot system must be able to respond to certain events detected by the sensors. The ability to specify such **event monitors** easily is afforded by some robot programming languages [2, 3].

Programming environment

As with any computer languages, a good programming environment fosters programmer productivity. Manipulator programming is difficult and tends to be very interactive, with a lot of trial and error. If the user were forced to continually repeat the "edit-compile-run" cycle of compiled languages, productivity would be low. Therefore, most robot programming languages are now *interpreted*, so that individual language statements can be run one at a time during program development and debugging. Many of the language statements cause motion of a physical device, so the tiny amount of time required to interpret the language statements is insignificant. Typical programming support, such as text editors, debuggers, and a file system, are also required.

Sensor integration

An extremely important part of robot programming has to do with interaction with sensors. The system should have, at a minimum, the capability to query touch and force sensors and to use the response in if-then-else constructs. The ability to specify event monitors to watch for transitions on such sensors in a *background* mode is also very useful.

Integration with a vision system allows the vision system to send the manipulator system the coordinates of an object of interest. For example, in our sample application, a vision system locates the brackets on the conveyor belt and returns to the manipulator controller their position and orientation relative to the camera. The camera's frame is known relative to the station frame, so a desired goal frame for the manipulator can be computed from this information.

Some sensors could be part of other equipment in the workcell—for example, some robot controllers can use input from a sensor attached to a conveyor belt so that the manipulator can track the belt's motion and acquire objects from the belt as it moves [2].

The interface to force-control capabilities, as discussed in Chapter 9, comes through special language statements that allow the user to specify force strategies [3]. Such force-control strategies are by necessity an integrated part of the manipulator control system—the robot programming language simply serves as an interface to those capabilities. Programming robots that make use of active force control might require other special features, such as the ability to display force data collected during a constrained motion [3].

In systems that support active force control, the description of the desired force application could become part of the motion specification. The AL language describes active force control in the motion primitives by specifying six components of stiffness (three translational and three rotational) and a bias force. In this way, the manipulator's apparent stiffness is programmable. To apply a force, usually the stiffness is set to zero in that direction and a bias force is specified—for example,

```
move garm to goal
with stiffness=(80, 80, 0, 100, 100, 100)
with force=20*ounces along zhat;
```

12.5 PROBLEMS PECULIAR TO ROBOT PROGRAMMING LANGUAGES

Advances in recent years have helped, but programming robots is still difficult. Robot programming shares all the problems of conventional computer programming, plus some additional difficulties caused by effects of the physical world [12].

Internal world model versus external reality

A central feature of a robot programming system is the world model that is maintained internally in the computer. Even when this model is quite simple, there are ample difficulties in assuring that it matches the physical reality that it attempts to model. Discrepancies between internal model and external reality result in poor or failed grasping of objects, collisions, and a host of more subtle problems.

This correspondence between internal model and the external world must be established for the program's initial state and must be maintained throughout its execution. During initial programming or debugging, it is generally up to the user to suffer the burden of ensuring that the state represented in the program corresponds to the physical state of the workcell. Unlike more conventional programming, where only internal variables need to be saved and restored to reestablish a former situation, in robot programming, physical objects must usually be repositioned.

Besides the uncertainty inherent in each object's position, the manipulator itself is limited to a certain degree of accuracy. Very often, steps in an assembly will require the manipulator to make motions requiring greater precision than it is capable of. A common example of this is inserting a pin into a hole where the clearance is an order of magnitude less than the positional accuracy of the manipulator. To further complicate matters, the manipulator's accuracy usually varies over its workspace.

In dealing with those objects whose locations are not known exactly, it is essential to somehow refine the positional information. This can sometimes be done with sensors (e.g., vision, touch) or by using appropriate force strategies for constrained motions.

During debugging of manipulator programs, it is very useful to be able to modify the program and then back up and try a procedure again. Backing up entails restoring the manipulator and objects being manipulated to a former state. However, in working with physical objects, it is not always easy, or even possible, to undo an action. Some examples are the operations of painting, riveting, drilling, or welding, which cause a physical modification of the objects being manipulated. It might therefore be necessary for the user to get a new copy of the object to replace the old, modified one. Further, it is likely that some of the operations just prior to the one being retried will also need to be repeated to establish the proper state required before the desired operation can be successfully retried.

Context sensitivity

Bottom-up programming is a standard approach to writing a large computer program in which one develops small, low-level pieces of a program and then puts them together into larger pieces, eventually attaining a completed program. For this method to work, it is essential that the small pieces be relatively insensitive to the language statements that precede them and that there be no assumptions concerning the context in which these program pieces execute. For manipulator programming, this is often not the case; code that worked reliably when tested in isolation frequently fails when placed in the context of the larger program. These problems generally arise from dependencies on manipulator configuration and speed of motions.

Manipulator programs can be highly sensitive to initial conditions—for example, the initial manipulator position. In motion trajectories, the starting position will influence the trajectory that will be used for the motion. The initial manipulator position might also influence the velocity with which the arm will be moving during some critical part of the motion. For example, these statements are true for manipulators that follow the cubic-spline joint-space paths studied in Chapter 7. These effects can sometimes be dealt with by proper programming care, but often such

problems do not arise until after the initial language statements have been debugged in isolation and are then joined with statements preceding them.

Because of insufficient manipulator accuracy, a program segment written to perform an operation at one location is likely to need to be tuned (i.e., positions retaught and the like) to make it work at a different location. Changes in location within the workcell result in changes in the manipulator's configuration in reaching goal locations. Such attempts at relocating manipulator motions within the workcell test the accuracy of the manipulator kinematics and servo system, and problems frequently arise. Such relocation could cause a change in the manipulator's kinematic configuration—for example, from left shoulder to right shoulder, or from elbow up to elbow down. Moreover, these changes in configuration could cause large arm motions during what had previously been a short, simple motion.

The nature of the spatial shape of trajectories is likely to change as paths are located in different portions of the manipulator's workspace. This is particularly true of joint-space trajectory methods, but use of Cartesian-path schemes can also lead to problems when singularities are nearby.

When testing a manipulator motion for the first time, it often is wise to have the manipulator move slowly. This allows the user a chance to stop the motion if it appears to be about to cause a collision. It also allows the user to inspect the motion closely. After the motion has undergone some initial debugging at a slower speed it is then desirable to increase motion speeds. Doing so might cause some aspects of the motion to change. Limitations in most manipulator control systems cause greater servo errors, which are to be expected if the quicker trajectory is followed. Also, in force-control situations involving contact with the environment, speed changes can completely change the force strategies required for success.

The manipulator's configuration also affects the delicacy and accuracy of the forces that can be applied with it. This is a function of how well conditioned the Jacobian of the manipulator is at a certain configuration, something generally difficult to consider when developing robot programs.

Error recovery

Another direct consequence of working with the physical world is that objects might not be exactly where they should be and, hence, motions that deal with them could fail. Part of manipulator programming involves attempting to take this into account and making assembly operations as robust as possible, but, even so, errors are likely, and an important aspect of manipulator programming is how to recover from these errors.

Almost any motion statement in the user's program can fail, sometimes for a variety of reasons. Some of the more common causes are objects shifting or dropping out of the hand, an object missing from where it should be, jamming during an insertion, and not being able to locate a hole.

The first problem that arises for error recovery is identifying that an error has indeed occurred. Because robots generally have quite limited sensing and reasoning capabilities, *error detection* is often difficult. In order to detect an error, a robot program must contain some type of explicit test. This test might involve checking the manipulator's position to see that it lies in the proper range; for example, when doing an insertion, lack of change in position might indicate jamming, or too much

change might indicate that the hole was missed entirely or the object has slipped out of the hand. If the manipulator system has some type of visual capabilities, then it might take a picture and check for the presence or absence of an object and, if the object is present, report its location. Other checks might involve force, such as weighing the load being carried to check that the object is still there and has not been dropped, or checking that a contact force remains within certain bounds during a motion.

Every motion statement in the program might fail, so these explicit checks can be quite cumbersome and can take up more space than the rest of the program. Attempting to deal with all possible errors is extremely difficult; usually, just the few statements that seem most likely to fail are checked. The process of predicting which portions of a robot application program are likely to fail is one that requires a certain amount of interaction and partial testing with the robot during the program-development stage.

Once an error has been detected, an attempt can be made to recover from it. This can be done totally by the manipulator under program control, or it might involve manual intervention by the user, or some combination of the two. In any event, the recovery attempt could in turn result in new errors. It is easy to see how code to recover from errors can become the major part of the manipulator program.

The use of parallelism in manipulator programs can further complicate recovery from errors. When several processes are running concurrently and one causes an error to occur, it could affect other processes. In many cases, it will be possible to back up the offending process, while allowing the others to continue. At other times, it will be necessary to reset several or all of the running processes.

BIBLIOGRAPHY

[1] B. Shimano, "VAL: A Versatile Robot Programming and Control System," Proceedings of COMPSAC 1979, Chicago, November 1979.

[2] B. Shimano, C. Geschke, and C. Spalding, "VAL II: A Robot Programming Language and Control System," SME Robots VIII Conference, Detroit, June 1984.

[3] S. Mujtaba and R. Goldman, "AL Users' Manual," 3rd edition, Stanford Department of Computer Science, Report No. STAN-CS-81-889, December 1981.

[4] A. Gilbert et al., *AR-BASIC: An Advanced and User Friendly Programming System for Robots*, American Robot Corporation, June 1984.

[5] J. Craig, "JARS—JPL Autonomous Robot System: Documentation and Users Guide," JPL Interoffice memo, September 1980.

[6] ABB Robotics, "The RAPID Language," in the *SC4Plus Controller Manual*, ABB Robotics, 2002.

[7] R. Taylor, P. Summers, and J. Meyer, "AML: A Manufacturing Language," *International Journal of Robotics Research*, Vol. 1, No. 3, Fall 1982.

[8] FANUC Robotics, Inc., "KAREL Language Reference," FANUC Robotics North America, Inc, 2002.

[9] R. Taylor, "A Synthesis of Manipulator Control Programs from Task-Level Specifications," Stanford University AI Memo 282, July 1976.

[10] J.C. LaTombe, "Motion Planning with Uncertainty: On the Preimage Backchaining Approach," in *The Robotics Review*, O. Khatib, J. Craig, and T. Lozano-Perez, Editors, MIT Press, Cambridge, MA, 1989.

[11] W. Gruver and B. Soroka, "Programming, High Level Languages," in *The International Encyclopedia of Robotics*, R. Dorf and S. Nof, Editors, Wiley Interscience, New York, 1988.

[12] R. Goldman, *Design of an Interactive Manipulator Programming Environment*, UMI Research Press, Ann Arbor, MI, 1985.

[13] Adept Technology, *V+ Language Reference*, Adept Technology, Livermore, CA, 2002.

EXERCISES

12.1 [15] Write a robot program (in a language of your choice) to pick a block up from location *A* and place it in location *B*.

12.2 [20] Describe tying your shoelace in simple English commands that might form the basis of a robot program.

12.3 [32] Design the syntax of a new robot programming language. Include ways to give duration or speeds to motion trajectories, make I/O statements to peripherals, give commands to control the gripper, and produce force-sensing (i.e., guarded move) commands. You can skip force control and parallelism (to be covered in Exercise 12.4).

12.4 [28] Extend the specification of the new robot programming language that you started in Exercise 12.3 by adding force-control syntax and syntax for parallelism.

12.5 [38] Write a program in a commercially available robot programming language to perform the application outlined in Section 12.3. Make any reasonable assumptions concerning I/O connections and other details.

12.6 [28] Using any robot language, write a general routine for unloading an arbitrarily sized pallet. The routine should keep track of indexing through the pallet and signal a human operator when the pallet is empty. Assume the parts are unloaded onto a conveyor belt.

12.7 [35] Using any robot language, write a general routine for unloading an arbitrarily sized source pallet and loading an arbitrarily sized destination pallet. The routine should keep track of indexing through the pallets and signal a human operator when the source pallet is empty and when the destination pallet is full.

12.8 [35] Using any capable robot programming language, write a program that employs force control to fill a cigarette box with 20 cigarettes. Assume that the manipulator has an accuracy of about 0.25 inch, so force control should be used for many operations. The cigarettes are presented on a conveyor belt, and a vision system returns their coordinates.

12.9 [35] Using any capable robot programming language, write a program to assemble the hand-held portion of a standard telephone. The six components (handle, microphone, speaker, two caps, and cord) arrive in a *kit*, that is, a special pallet holding one of each kind of part. Assume there is a fixture into which the handle can be placed that holds it. Make any other reasonable assumptions needed.

12.10 [33] Write a robot program that uses two manipulators. One, called GARM, has a special end-effector designed to hold a wine bottle. The other arm, BARM, will hold a wineglass and is equipped with a force-sensing wrist that can be used to signal GARM to stop pouring when it senses the glass is full.

PROGRAMMING EXERCISE (PART 12)

Create a user interface to the other programs you have developed by writing a few subroutines in Pascal. Once these routines are defined, a "user" could write a Pascal program that contains calls to these routines to perform a 2-D robot application in simulation.

Define primitives that allow the user to set station and tool frames—namely,

```
setstation(Sre1B:vec3);
settool(Tre1W:vec3);
```

where "Sre1B" gives the station frame relative to the base frame of the robot and "Tre1W" defines the tool frame relative to the wrist frame of the manipulator. Define the motion primitives

```
moveto(goal:vec3);
moveby(increment:vec3);
```

where "goal" is a specification of the goal frame relative to the station frame and "increment" is a specification of a goal frame relative to the current tool frame. Allow multisegment paths to be described when the user first calls the "pathmode" function, then specifies motions to via points, and finally says "runpath"—for example,

```
pathmode; (* enter path mode *)
moveto(goal1);
moveto(goal2);
runpath; (* execute the path without stopping at goal1 *)
```

Write a simple "application" program, and have your system print the location of the arm every n seconds.

CHAPTER 13

Off-line programming systems

13.1 INTRODUCTION

We define an **off-line programming** (OLP) system as a robot programming language that has been sufficiently extended, generally by means of computer graphics, that the development of robot programs can take place without access to the robot itself.[1] Off-line programming systems are important both as aids in programming present-day industrial automation and as platforms for robotics research. Numerous issues must be considered in the design of such systems. In this chapter, first a discussion of these issues is presented [1] and then a closer look at one such system [2].

Over the past 20 years, the growth of the industrial robot market has not been as rapid as once was predicted. One primary reason for this is that robots are still too difficult to use. A great deal of time and expertise is required to install a robot in a particular application and bring the system to production readiness. For various reasons, this problem is more severe in some applications than in others; hence, we see certain application areas (e.g., spot welding and spray painting) being automated with robots much sooner than other application domains (e.g., assembly). It seems that lack of sufficiently trained robot-system implementors is limiting growth in some, if not all, areas of application. At some manufacturing companies, management encourages the use of robots to an extent greater than that realizable by applications engineers. Also, a large percentage of the robots delivered are being used in ways that do not take full advantage of their capabilities. These symptoms indicate that current industrial robots are not easy enough to use to allow successful installation and programming in a timely manner.

There are many factors that make robot programming a difficult task. First, it is intrinsically related to general computer programming and so shares in many of the problems encountered in that field; but the programming of robots, or of any programmable machine, has particular problems that make the development of production-ready software even more difficult. As we saw in the last chapter,

[1] Chapter 13 is an edited version of two papers: one reprinted with permission from *International Symposium of Robotics Research*, R. Bolles and B. Roth (editors), 1988 (ref [1]); the other from *Robotics: The Algorithmic Perspective*, P. Agarwal et al. (editors), 1998 (ref [2]).

most of these special problems arise from the fact that a robot manipulator interacts with its physical environment [3]. Even simple programming systems maintain a "world model" of this physical environment in the form of locations of objects and have "knowledge" about presence and absence of various objects encoded in the program strategies. During the development of a robot program (and especially later during production use), it is necessary to keep the internal model maintained by the programming system in correspondence with the actual state of the robot's environment. Interactive debugging of programs with a manipulator requires frequent manual resetting of the state of the robot's environment—parts, tools, and so forth must be moved back to their initial locations. Such state resetting becomes especially difficult (and sometimes costly) when the robot performs a irreversible operation on one or more parts (e.g., drilling or routing). The most spectacular effect of the presence of the physical environment is when a program bug manifests itself in some unintended irreversible operation on parts, on tools, or even on the manipulator itself.

Although difficulties exist in maintaining an accurate internal model of the manipulator's environment, there seems no question that great benefits result from doing so. Whole areas of sensor research, perhaps most notably computer vision, focus on developing techniques by which world models can be verified, corrected, or discovered. Clearly, in order to apply any computational algorithm to the robot command-generation problem, the algorithm needs access to a model of the robot and its surroundings.

In the development of programming systems for robots, advances in the power of programming techniques seem directly tied to the sophistication of the internal model referenced by the programming language. Early joint-space "teach by showing" robot systems employed a limited world model, and there were very limited ways in which the system could aid the programmer in accomplishing a task. Slightly more sophisticated robot controllers included kinematic models, so that the system could at least aid the user in moving the joints so as to accomplish Cartesian motions. Robot programming languages (RPLs) evolved to support many different data types and operations, which the programmer may use as needed to model attributes of the environment and compute actions for the robot. Some RPLs support such world-modeling primitives as affixments, data types for forces and moments, and other features [4].

The robot programming languages of today might be called "explicit programming languages," in that every action that the system takes must be programmed by the application engineer. At the other end of the spectrum are the so-called task-level-programming (TLP) systems, in which the programmer may state such high-level goals as "insert the bolt" or perhaps even "build the toaster oven." These systems use techniques from artificial-intelligence research to generate motion and strategy plans automatically. However, task-level languages this sophisticated do not yet exist; various pieces of such systems are currently under development by researchers [5]. Task-level-programming systems will require a very complete model of the robot and its environment to perform automated planning operations.

Although this chapter focuses to some extent on the particular problem of robot programming, the notion of an OLP system extends to any programmable device on the factory floor. An argument commonly raised in favor is that an OLP

system will not tie up production equipment when it needs to be reprogrammed; hence, automated factories can stay in production mode a greater percentage of the time. They also serve as a natural vehicle to tie computer-aided design (CAD) data bases used in the design phase of a product's development to the actual manufacturing of the product. In some applications, this direct use of CAD design data can dramatically reduce the programming time required for the manufacturing machinery.

Off-line programming of robots offers other potential benefits, ones just beginning to be appreciated by industrial robot users. We have discussed some of the problems associated with robot programming, and most have to do with the fact that an external, physical workcell is being manipulated by the robot program. This makes backing up to try different strategies tedious. Programming of robots in simulation offers a way of keeping the bulk of the programming work strictly internal to a computer—until the application is nearly complete. Under this approach, many of the problems peculiar to robot programming tend to diminish.

Off-line programming systems should serve as the natural growth path from explicit programming systems to task-level-programming systems. The simplest OLP system is merely a graphical extension to a robot programming language, but from there it can be extended into a task-level-programming system. This gradual extension is accomplished by providing automated solutions to various subtasks (as these solutions become available) and letting the programmer use them to explore options in the simulated environment. Until we discover how to build task-level systems, the user must remain in the loop to evaluate automatically planned subtasks and guide the development of the application program. If we take this view, an OLP system serves as an important basis for research and development of task-level-planning systems, and, indeed, in support of their work, many researchers have developed various components of an OLP system (e.g., 3-D models and graphic display, language postprocessors). Hence, OLP systems should be a useful tool in research as well as an aid in current industrial practice.

13.2 CENTRAL ISSUES IN OLP SYSTEMS

This section raises many of the issues that must be considered in the design of an OLP system. The collection of topics discussed will help to set the scope of the definition of an OLP system.

User interface

A major motivation for developing an OLP system is to create an environment that makes programming manipulators easier, so the user interface is of crucial importance. However, another major motivation is to remove reliance on use of the physical equipment during programming. Upon initial consideration, these two goals seem to conflict—robots are hard enough to program when you can see them, so how can it be easier without the presence of physical device? This question touches upon the essence of the OLP design problem.

Manufacturers of industrial robots have learned that the RPLs they provide with their robots cannot be utilized successfully by a large percentage of manufacturing personnel. For this and other historical reasons, many industrial robots are

provided with a two-level interface [6], one for programmers and one for nonprogrammers. Nonprogrammers utilize a teach pendant and interact directly with the robot to develop robot programs. Programmers write code in the RPL and interact with the robot in order to teach robot work points and to debug program flow. In general, these two approaches to program development trade off ease of use against flexibility.

When viewed as an extension of a RPL, an OLP system by nature contains an RPL as a subset of its user interface. This RPL should provide features that have already been discovered to be valuable in robot programming systems. For example, for use as an RPL, **interactive languages** are much more productive than compiled languages, which force the user to go through the "edit–compile–run" cycle for each program modification.

The language portion of the user interface inherits much from "traditional" RPLs; it is the lower-level (i.e., easier-to-use) interface that must be carefully considered in an OLP system. A central component of this interface is a computer-graphic view of the robot being programmed and of its environment. Using a pointing device such as a **mouse**, the user can indicate various locations or objects on the graphics screen. The design of the user interface addresses exactly how the user interacts with the screen to specify a robot program. The same pointing device can indicate items in a "menu" in order to specify modes or invoke various functions.

A central primitive is that for teaching a robot a work point or "frame" that has six degrees of freedom by means of interaction with the graphics screen. The availability of 3-D models of fixtures and workpieces in the OLP system often makes this task quite easy. The interface provides the user with the means to indicate locations on surfaces, allowing the orientation of the frame to take on a local surface normal, and then provides methods for offsetting, reorienting, and so on. Depending on the specifics of the application, such tasks are quite easily specified via the graphics window into the simulated world.

A well-designed user interface should enable nonprogrammers to accomplish many applications from start to finish. In addition, frames and motion sequences "taught" by nonprogrammers should be able to be translated by the OLP system into textual RPL statements. These simple programs can be maintained and embellished in RPL form by more experienced programmers. For programmers, the RPL availability allows arbitrary code development for more complex applications.

3-D modeling

A central element in OLP systems is the use of graphic depictions of the simulated robot and its workcell. This requires the robot and all fixtures, parts, and tools in the workcell to be modeled as three-dimensional objects. To speed up program development, it is desirable to use any CAD models of parts or tooling that are directly available from the CAD system on which the original design was done. As CAD systems become more and more prevalent in industry, it becomes more and more likely that this kind of geometric data will be readily available. Because of the strong desire for this kind of CAD integration from design to production, it makes sense for an OLP system either to contain a CAD modeling subsystem or to be, itself, a part of a CAD design system. If an OLP system is to be a stand-alone system, it must have appropriate interfaces to transfer models to and from external CAD

systems; however, even a stand-alone OLP system should have at least a simple local CAD facility for quickly creating models of noncritical workcell items or for adding robot-specific data to imported CAD models.

OLP systems generally require multiple representations of spatial shapes. For many operations, an exact analytic description of the surface or volume is generally present; yet, in order to benefit from display technology, another representation is often needed. Current technology is well suited to systems in which the underlying display primitive is a planar polygon; hence, although an object shape might be well represented by a smooth surface, practical display (especially for animation) requires a faceted representation. User-interface graphical actions, such as pointing to a spot on a surface, should internally act so as to specify a point on the true surface, even if, graphically, the user sees a depiction of the faceted model.

An important use of the three-dimensional geometry of the object models is in **automatic collision detection**—that is, when any collisions occur between objects in the simulated environment, the OLP system should automatically warn the user and indicate exactly where the collision takes place. Applications such as assembly may involve many desired "collisions," so it is necessary to be able to inform the system that collisions between certain objects are acceptable. It is also valuable to be able to generate a collision warning when objects pass within a specified tolerance of a true collision. Currently, the exact collision-detection problem for general 3-D solids is difficult, but collision detection for faceted models is quite practical.

Kinematic emulation

A central component in maintaining the validity of the simulated world is the faithful emulation of the geometrical aspects of each simulated manipulator. With regard to inverse kinematics, the OLP system can interface to the robot controller in two distinct ways. First, the OLP system could replace the inverse kinematics of the robot controller and always communicate robot positions in mechanism joint space. The second choice is to communicate Cartesian locations to the robot controller and let the controller use the inverse kinematics supplied by the manufacturer to solve for robot configurations. The second choice is almost always preferable, especially as manufacturers begin to build *arm signature* style calibration into their robots. These calibration techniques customize the inverse kinematics for each individual robot. In this case, it becomes desirable to communicate information at the Cartesian level to robot controllers.

These considerations generally mean that the forward and inverse kinematic functions used by the simulator must reflect the nominal functions used in the robot controller supplied by the manufacturer of the robot. There are several details of the inverse-kinematic function specified by the manufacturer that must be emulated by the simulator software. Any inverse-kinematic algorithm must make arbitrary choices in order to resolve singularities. For example, when joint 5 of a PUMA 560 robot is at its zero location, axes 4 and 6 line up, and a singular condition exists. The inverse-kinematic function in the robot controller can solve for the sum of joint angles 4 and 6, but then must use an arbitrary rule to choose individual values for joints 4 and 6. The OLP system must emulate whatever algorithm is used. Choosing the nearest solution when many alternate solutions exist provides another example. The simulator must use the same algorithm as the controller in

order to avoid potentially catastrophic errors in simulating the actual manipulator. A helpful feature occasionally found in robot controllers is the ability to command a Cartesian goal and specify which of the possible solutions the manipulator should use. The existence of this feature eliminates the requirement that the simulator emulate the solution-choice algorithm; the OLP system can simply force its choice on the controller.

Path-planning emulation

In addition to kinematic emulation for static positioning of the manipulator, an OLP system should accurately emulate the path taken by the manipulator in moving through space. Again, the central problem is that the OLP system needs to simulate the algorithms in the employed robot controller, and such path-planning and -execution algorithms vary considerably from one robot manufacturer to another. Simulation of the spatial shape of the path taken is important for detection of collisions between the robot and its environment. Simulation of the temporal aspects of the trajectory are important for predicting the cycle times of applications. When a robot is operating in a moving environment (e.g., near another robot), accurate simulation of the temporal attributes of motion is necessary to predict collisions accurately and, in some cases, to predict communication or synchronization problems, such as deadlock.

Dynamic emulation

Simulated motion of manipulators can neglect dynamic attributes if the OLP system does a good job of emulating the trajectory-planning algorithm of the controller and if the actual robot follows desired trajectories with negligible errors. However, at high speed or under heavy loading conditions, trajectory-tracking errors can become important. Simulation of these tracking errors necessitates both modeling the dynamics of the manipulator and of the objects that it moves and emulating the control algorithm used in the manipulator controller. Currently, practical problems exist in obtaining sufficient information from the robot vendors to make this kind of dynamic simulation of practical value, but, in some cases, dynamic simulation can be pursued fruitfully.

Multiprocess simulation

Some industrial applications involve two or more robots cooperating in the same environment. Even single-robot workcells often contain a conveyor belt, a transfer line, a vision system, or some other active device with which the robot must interact. For this reason, it is important that an OLP system be able to simulate multiple moving devices and other activities that involve **parallelism**. As a basis for this capability, the underlying language in which the system is implemented should be a multiprocessing language. Such an environment makes it possible to write independent robot-control programs for each of two or more robots in a single cell and then simulate the action of the cell with the programs running concurrently. Adding signal and wait primitives to the language enables the robots to interact with each other just as they might in the application being simulated.

Simulation of sensors

Studies have shown that a large component of robot programs consists not of motion statements, but rather of initialization, error-checking, I/O, and other kinds of statements [7]. Hence, the ability of the OLP system to provide an environment that allows simulation of complete applications, including interaction with sensors, various I/O, and communication with other devices, becomes important. An OLP system that supports simulation of sensors and multiprocessing not only can check robot motions for feasibility, but also can verify the communication and synchronization portion of the robot program.

Language translation to target system

An annoyance for current users of industrial robots (and of other programmable automation) is that almost every supplier of such systems has invented a unique language for programming its product. If an OLP system aspires to be universal in the equipment it can handle, it must deal with the problem of translating to and from several different languages. One choice for dealing with this problem is to choose a single language to be used by the OLP system and then postprocess the language in order to convert it into the format required by the target machine. An ability to upload programs that already exist on the target machines and bring them into the OLP system is also desirable.

Two potential benefits of OLP systems relate directly to the language-translation topic. Most proponents of OLP systems note that having a single, universal interface, one that enables users to program a variety of robots, solves the problem of learning and dealing with several automation languages. A second benefit stems from economic considerations in future scenarios in which hundreds or perhaps thousands of robots fill factories. The cost associated with a powerful programming environment (such as a language and graphical interface) might prohibit placing it at the site of each robot installation. Rather, it seems to make economic sense to place a very simple, "dumb," and cheap controller with each robot and have it downloaded from a powerful, "intelligent" OLP system that is located in an office environment. Hence, the general problem of translating an application program from a powerful universal language to a simple language designed to execute in a cheap processor becomes an important issue in OLP systems.

Workcell calibration

An inevitable reality of a computer model of any real-world situation is that of inaccuracy in the model. In order to make programs developed on an OLP system usable, methods for **workcell calibration** must be an integral part of the system. The magnitude of this problem varies greatly with the application; this variability makes off-line programming of some tasks much more feasible that of others. If the majority of the robot work points for an application must be retaught with the actual robot to solve inaccuracy problems, OLP systems lose their effectiveness.

Many applications involve the frequent performance of actions relative to a rigid object. Consider, for example, the task of drilling several hundred holes in a bulkhead. The actual location of the bulkhead relative to the robot can be taught by using the actual robot to take three measurements. From those data, the locations

of all the holes can be updated automatically if they are available in part coordinates from a CAD system. In this situation, only these three points need be taught with the robot, rather than hundreds. Most tasks involve this sort of "many operations relative to a rigid object" paradigm—for example, PC-board component insertion, routing, spot welding, arc welding, palletizing, painting, and deburring.

13.3 THE 'PILOT' SIMULATOR

In this section, we consider one such off-line simulator system: the 'Pilot' system developed by Adept Technology [8]. The Pilot system is actually a suite of three closely related simulation systems; here, we look at the portion of Pilot (known as "Pilot/Cell") that is used to simulate an individual workcell in a factory. In particular, this system is unusual in that it attempts to model several aspects of the physical world, as a means of unburdening the programmer of the simulator. In this section, we will discuss the "geometric algorithms" that are used to empower the simulator to emulate certain aspects of physical reality.

The need for ease of use drives the need for the simulation system to behave like the actual physical world. The more the simulator acts like the real world, the simpler the user-interface paradigm becomes for the user, because the physical world is the one we are all familiar with. At the same time, trade-offs of ease against computational speed and other factors have driven a design in which a particular "slice" of reality is simulated while many details are not.

Pilot is well-suited as a host for a variety of geometric algorithms. The need to model various portions of the real world, together with the need to unburden the user by automating frequent geometric computations, drives the need for such algorithms. Pilot provides the environment in which some advanced algorithms can be brought to bear on real problems occurring in industry.

One decision made very early on in the design of the Pilot simulation system was that the *programming paradigm* should be as close as possible to the way the actual robot system would be programmed. Certain higher level planning and optimization tools are provided, but it was deemed important to have the basic programming interaction be similar to actual hardware systems. This decision has led the product's development down a path along which we find a genuine need for various geometric algorithms. The algorithms needed range widely from extremely simple to quite complex.

If a simulator is to be programmed as the physical system would be, then the actions and reactions of the physical world must be modeled "automatically" by the simulator. The goal is to free the user of the system from having to write any "simulation-specific code." As a simple example, if the robot gripper is commanded to open, a grasped part should fall in response to gravity and possibly should even bounce and settle into a certain stable state. Forcing the user of the system to specify these real-world actions would make the simulator fall short of its goal: being programmed just as the actual system is. Ultimate ease of use can be achieved only when the simulated world "knows how" to behave like the real world without burdening the user.

Most, if not all, commercial systems for simulating robots or other mechanisms do not attempt to deal directly with this problem. Rather, they typically "allow" the user (actually, *force* the user) to embed simulation-specific commands within the

program written to control the simulated device. A simple example would be the following code sequence:

```
MOVE TO pick_part
CLOSE gripper
affix(gripper,part[i]);
MOVE TO place_part
OPEN gripper
unaffix(gripper,part[i]);
```

Here, the user has been forced to insert "affix" and "unaffix" commands, which (respectively) cause the part to move with the gripper when grasped and to stop moving with it when released. If the simulator allows the robot to be programmed in its native language, generally that language is not rich enough to support these required "simulation-specific" commands. Hence, there is a need for a second set of commands, possibly even with a different syntax, for dealing with interactions with the real world. Such a scheme is inherently *not* programmed "just as the physical system is" and must inherently cause an increased programming burden for the user.

From the preceding example, we see the first geometric algorithm that one finds a need for: From the geometry of the gripper and the relative placements of parts, figure out which part (if any) will be grasped when the gripper closes and possibly how the part will self-align within the gripper. In the case of Pilot, we solve the first part of this problem with a simple algorithm. In limited cases, the "alignment action" of the part in the gripper is computed, but, in general, such alignments need to be pretaught by the system's user. Hence, Pilot has not reached the ultimate goal yet, but has taken some steps in that direction.

Physical Modeling and Interactive Systems

In a simulation system, one always trades off complexity of the model in terms of computation time against accuracy of the simulation. In the case of Pilot and its intended goals, it is particularly important to keep the system fully interactive. This has led to designing Pilot so that it can use various approximate models—for example, the use of quasi-static approximations where a full dynamic model might be more accurate. Although there appears to be a possibility that "full dynamic" models might soon be applicable [9], given the current state of computer hardware, of dynamic algorithms, and of the complexity of the CAD models that industrial users wish to employ, we feel these trade-offs still need to be made.

Geometric Algorithms for Part Tumbling

In some feeding systems employed in industrial practice, parts tumble from some form of infeed conveyor onto a presentation surface; then computer vision is used to locate parts to be acquired by the robot. Designing such automation systems with the aid of a simulator means that the simulator must be able to predict how parts fall, bounce, and take on a stable orientation, or *stable state*.

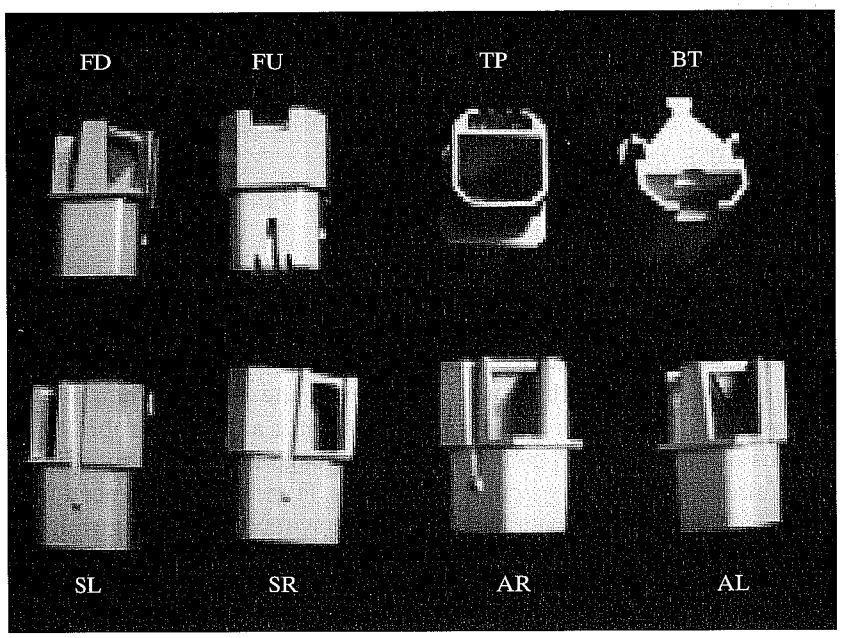

FIGURE 13.1: The eight stable states of the part.

Stable-state probabilities

As reported in [10], an algorithm has been implemented that takes as input any geometric shape (represented by a CAD model) and, for that shape, can compute the N possible ways that it can rest stably on a horizontal surface. These are called the *stable states* of the part. Further, the algorithm uses a perturbed quasi-static approach to estimate the probability associated with each of the N stable states. We have performed physical experiments with sample parts in order to assess the resulting accuracy of stable-state prediction.

Figure 13.1 shows the eight stable states of a particular test part. Using an Adept robot and vision system, we dropped this part more than 26,000 times and recorded the resulting stable state, in order to compare our stable-state prediction algorithm to reality. Table 13.1 shows the results for the test part. These results are characteristic of our current algorithm—stable-state likelihood prediction error typically ranges from 5% to 10%.

Adjusting probabilities as a function of drop height

Clearly, if a part is dropped from a gripper from a very small height (e.g., 1 mm) above a surface, the probabilities of the various stable states differ from those which occur when the part is dropped from higher than some *critical height*. In Pilot, we use probabilities from the *stable-state estimator* algorithm when parts are dropped from heights equal to or greater than the largest dimension of the part. For drop

TABLE 13.1: Predicted versus Actual Stable-State Probabilities for the Test Part

Stable State	Actual #	% Actual	% Predicted
FU	1871	7.03%	8.91%
FD	10,600	39.80%	44.29%
TP	648	2.43%	7.42%
BT	33	0.12%	8.19%
SR	6467	24.28%	15.90%
SL	6583	24.72%	15.29%
AR/AL	428	1.61%	0.00%
Total	26,630	100%	100%

heights below that value, probabilities are adjusted to take into account the initial orientation of the part and the height of the drop. The adjustment is such that, as an infinitesimal drop height is approached, the part remains in its initial orientation (assuming it is a stable orientation). This is an important addition to the overall probability algorithm, because it is typical for parts to be released a small distance above a support surface.

Simulation of bounce

Parts in Pilot are tagged with their coefficient of restitution; so are all surfaces on which parts may be placed. The product of these two factors is used in a formula for predicting how far the part will bounce when dropped. These details are important, because they affect how parts scatter or clump in the simulation of some feeding systems. When bouncing, parts are scattered radially according to a uniform distribution. The distance of bounce (away from the initial contact point) is a certain distribution function out to a maximum distance, which is computed as a function of drop height (energy input) and the coefficients of restitution that apply.

Parts in Pilot can bounce recursively from surface to surface in certain arrangements. It is also possible to mark certain surfaces such that parts are not able to bounce *off* them, but can only bounce *within* them. Entities known as *bins* in Pilot have this property –parts can fall into them, but never bounce out.

Simulation of stacking and tangling

As a simplification, parts in Pilot always rest on planar support surfaces. If parts are tangled or stacked on one another, this is displayed as parts that are intersecting each other (that is, the boolean intersection of their volumes would be non-empty). This saves the enormous amount of computation that would be needed to compute the various ways a part might be stacked or tangled with another part's geometry.

Parts in Pilot are tagged with a *tangle factor*. For example, something like a marble would have a tangle factor of 0.0 because, when tumbled onto a support surface, marbles tend never to stack or tangle, but rather tend to spread out on the

surface. On the other hand, parts like coiled springs might have a tangle factor near 1.0; they quite readily become entangled with one another. When a part falls and bounces, a *findspace* algorithm runs, in which the part tries to bounce into an open space on the surface. However, exactly "how hard it tries" to find an open space is a function of its tangle factor. By adjustment of this coefficient, Pilot can simulate parts that tumble and become entangled more or less. Currently, there is no algorithm for automatically computing the tangle factor from the part geometry—this is an interesting open problem. Through the user interface, the Pilot user can set the tangle factor to what seems appropriate.

Geometric Algorithms for Part Grasping

Much of the difficulty in programming and using actual robots has to do with the details of teaching grasp locations on parts and with the detailed design of grippers. This is an area in which additional planning algorithms in a simulator system could have a large impact. In this section, we discuss the algorithms currently in place in Pilot. The current approaches are quite simple, so this is an area of ongoing work.

Computing which part to grasp

When a tool closes, or a suction end-effector actuates, Pilot applies a simple algorithm to compute which part (if any) should become grasped by the robot. First, the system figures out which support surface is immediately beneath the gripper. Then, for all parts on that surface, it searches for each whose bounding box (for the current stable state) contains the TCP (tool center point) of the gripper. If more than one part satisfies this criterion, then it chooses the nearest among those which do.

Computation of default grasp location

Pilot automatically assigns a grasp location for each stable orientation predicted by the stable-state estimator previously described. The current algorithm is simplistic, so a graphical user interface is also provided so that the user can edit and redefine these grasp points. The current grasp algorithm is a function of the part's bounding box and the geometry of the gripper, which is assumed to be either a parallel-jaw gripper or a suction cup. Along with computing a default grasp location for each stable state, a default approach and depart height are also automatically computed.

Computation of alignment of the part during grasp

In some important cases in industrial practice, the system designer counts on the fact that, when the robot end-effector actuates, the captured part will align itself in some way with surfaces of the end-effector. This effect can be important in removing small misalignments in the presentation of parts to the robot.

A very real effect which needs to be simulated is that, with suction cup grippers, it can be the case that, when suction is applied, the part is "lifted" up against the suction cup in a way which significantly alters its orientation relative to the end-effector. Pilot simulates this effect by piercing the part geometry with a vertical line aligned with the center line of the suction cup. Whichever facet of the polygonal part model is pierced is used in computing the orientation at grasp—the normal of

this facet becomes anti-aligned with the normal of the bottom of the suction cup. In altering the part orientation, rotation about this piercing line is minimized (the part does not spin about the axis of the suction cup when picked). Without simulation of this effect, the simulator would be unable to depict realistically some pick-and-place strategies employing suction grippers.

We have also implemented a planner that allows parts to rotate about the Z axis when a parallel jaw gripper closes on them. This case is automatic only for a simple case—in other situations, the user must teach the resulting alignment (i.e., we are still waiting for a more nearly complete algorithm).

Geometric Algorithms for Part Pushing

One style of part pushing occurs between the jaws of a gripper, as mentioned in the previous section. In current industrial practice, parts sometimes get pushed by simple mechanisms. For example, after a part is presented by a bowl feeder, it might be pushed by a linear actuator right into an assembly that has been brought into the cell by a tray-conveyor system.

Pilot has support for simulating the pushing of parts: an entity called a *push-bar*, which can be attached to a pneumatic cylinder or a leadscrew actuator in the simulator. When the actuator moves the push-bar along a linear path, the leading surface of the push-bar will move parts. In the future, it is planned, push-bars will also be able to be added as guides along conveyors or placed anywhere that requires that parts motion be affected by their presence. The current pushing is still very simple, but it suffices for many real-world tasks.

Geometric Algorithms for Tray Conveyors

Pilot supports the simulation of tray-conveyor systems in which trays move along tracks composed of straight-line and circular-section components. Placed along the tracks at key locations can be *gates*, which pop up temporarily to block a tray when so commanded. Additionally, *sensors* that detect a passing tray can be placed in the track at user-specified locations. Such conveyor systems are typical in many automation schemes.

Connecting tray conveyors and sources and sinks

Tray conveyors can be connected together to allow various styles of branching. Where two conveyors "flow together," a simple collision-avoidance scheme is provided to cause trays from the *spur* conveyor to be subordinate to trays on the *main* conveyor. Trays on the spur conveyor will wait whenever a collision would occur. At "flow apart" connections, a device called a *director* is added to the main conveyor, which can be used to control which direction a tray will take at the intersection. Digital I/O lines connected to the simulated robot controller are used to read sensors, activate gates, and activate directors.

At the ends of a tray conveyor are a *source* and a *sink*. Sources are set up by the user to generate trays at certain statistical intervals. The trays generated could either be empty or be preloaded with parts or fixtures. At the end of a tray conveyor, trays (and their contents) disappear into sinks. Each time a tray enters a sink, its arrival time and contents are recorded. These so-called *sink records* can then be

replayed through a source elsewhere in the system. Hence, a line of cells can be studied in the simulator one cell at a time, by setting the source of cell $N + 1$ to the sink record from cell N.

Pushing of trays

Pushing is also implemented for trays: A push-bar can be used to push a tray off a tray conveyor system and into a particular work cell. Likewise, trays can be pushed onto a tray conveyor. The updating of various data structures when trays come off a conveyor or onto one is an automatic part of the pushing code.

Geometric Algorithms for Sensors

Simulation of various sensor systems is required, so that the user will not be burdened with the writing of code to emulate their behavior in the cell.

Proximity sensors

Pilot supports the simulation of proximity sensors and other sensors. In the case of proximity sensors, the user tags the device with its minimum and maximum range and with a threshold. If an object is within range and closer than the threshold, then the sensor will detect it. To perform this computation in the simulated world, a line segment is temporarily added to the world, one that stretches from minimum to maximum sensor range. Using a collision algorithm, the system computes the locations at which this line segment intersects other CAD geometry. The intersection point nearest the sensor corresponds to the real-world item that would have stopped the beam. A comparison of the distance to this point and the threshold gives the output of the sensor. At present, we do not make use of the angle of the encountered surface or of its reflectance properties, although those features might be added in the future.

2-D vision systems

Pilot simulates the performance of the Adept 2-D vision system. The way the simulated vision system works is closely related to the way the real vision system works, even to how it is programmed in the AIM language [11] used by Adept robots. The following elements of this vision system are simulated:

- The shape and extent of the field of view.
- The stand-off distance and a simple model of focus.
- The time required to perform vision processing (approximate).
- The spatial ordering of results in the queue in the case of many parts being found in one image.
- The ability to distinguish parts according to which stable state they are in.
- The inability to recognize parts that are touching or overlapping.
- Within the context of AIM, the ability to update robot goals based on vision results.

The use of a vision system is well integrated with the AIM robot programming system, so implementation of the AIM language in the simulator implies implementation of vision system emulation. AIM supports several constructs that make the use of vision easy for robot guidance. Picking parts that are identified visually from both indexing and tracking conveyors is easily accomplished.

A data structure keeps track of which support surface the vision system is looking at. For all parts supported on that surface, we compute which are within the vision system's field of view. We prune out any parts that are too near or too far from the camera (e.g., out of focus). We prune out any parts that are touching neighboring parts. From the remaining parts, we choose those which are in the sought-after stable state and put them in a list. Finally, this list is sorted to emulate the ordering the Adept vision system uses when multiple parts are found in one scene.

Inspector sensors

A special class of sensor is provided, called an *inspector*. The inspector is used to give a binary output for each part placed in front of it. Parts in Pilot can be tagged with a *defect rate*, and inspectors can ferret out the defective parts. Inspectors play the role of several real-world sensor systems.

Conclusion

As is mentioned throughout this section, although some simple geometric algorithms are currently in place in the simulator, there is a need for more and better algorithms. In particular, we would like to investigate the possibility of adding a quasi-static simulation capability for predicting the motion of objects in situations in which friction effects dominate any inertial effects. This could be used to simulate parts being pushed or tipped by various actions of end-effectors or other pushing mechanisms.

13.4 AUTOMATING SUBTASKS IN OLP SYSTEMS

In this section, we briefly mention some advanced features that could be integrated into the "baseline" OLP-system concept already presented. Most of these features accomplish automated planning of some small portion of an industrial application.

Automatic robot placement

One of the most basic tasks that can be accomplished by means of an OLP system is the determination of the workcell layout so that the manipulator(s) can reach all of the required workpoints. Determining correct robot or workpiece placement by trial and error is more quickly completed in a simulated world than in the physical cell. An advanced feature that automates the search for feasible robot or workpiece location(s) goes one step further in reducing burden on the user.

Automatic placement can be computed by direct search or (sometimes) by heuristic-guided search techniques. Most robots are mounted flat on the floor (or ceiling) and have the first rotary joint perpendicular to the floor, so no more is generally necessary than to search by tessellation of the three-dimensional space of robot-base placement. The search might optimize some criterion or might halt upon location of the first feasible robot or part placement. Feasibility can be

defined as collision-free ability to reach all workpoints (or perhaps be given an even stronger definition). A reasonable criterion to maximize might be some form of a *measure of manipulability*, as was discussed in Chapter 8. An implementation using a similar measure of manipulability has been discussed in [12]. The result of such an automatic placement is a cell in which the robot can reach all of its workpoints in *well-conditioned* configurations.

Collision avoidance and path optimization

Research on the planning of collision-free paths [13,14] and the planning of time-optimal paths [15,16] generates natural candidates for inclusion in an OLP system. Some related problems that have a smaller scope and a smaller search space are also of interest. For example, consider the problem of using a six-degree-of-freedom robot for an arc-welding task whose geometry specifies only five degrees of freedom. Automatic planning of the redundant degree of freedom can be used to avoid collisions and singularities of the robot [17].

Automatic planning of coordinated motion

In many arc-welding situations, details of the process require that a certain relationship between the workpiece and the gravity vector be maintained during the weld. This results in a two- or three-degree-of-freedom-orienting system on which the part is mounted, operating simultaneously with the robot and in a coordinated fashion. In such a system, there could be nine or more degrees of freedom to coordinate. Such systems are generally programmed today by using teaching-pendant techniques. A planning system that could automatically synthesize the coordinated motions for such a system might be quite valuable [17,18].

Force-control simulation

In a simulated world in which objects are represented by their surfaces, it is possible to investigate the simulation of manipulator force-control strategies. This task involves the difficult problem of modeling some surface properties and expanding the dynamic simulator to deal with the constraints imposed by various contacting situations. In such an environment, it might be possible to assess various force-controlled assembly operations for feasibility [19].

Automatic scheduling

Along with the geometric problems found in robot programming, there are often difficult scheduling and communication problems. This is particularly the case if we expand the simulation beyond a single workcell to a group of workcells. Some discrete-time simulation systems offer abstract simulation of such systems [20], but few offer planning algorithms. Planning schedules for interacting processes is a difficult problem and an area of research [21,22]. An OLP system would serve as an ideal test bed for such research and would be immediately enhanced by any useful algorithms in this area.

Automatic assessment of errors and tolerances

An OLP system might be given some of the capabilities discussed in recent work in modeling positioning-error sources and the effect of data from imperfect sensors [23,24]. The world model could be made to include various error bounds and tolerancing information, and the system could assess the likelihood of success of various positioning or assembly tasks. The system might suggest the use and placement of sensors so as to correct potential problems.

Off-line programming systems are useful in present-day industrial applications and can serve as a basis for continuing robotics research and development. A large motivation in developing OLP systems is to fill the gap between the explicitly programmed systems available today and the task-level systems of tomorrow.

BIBLIOGRAPHY

[1] J. Craig, "Issues in the Design of Off-Line Programming Systems," *International Symposium of Robotics Research*, R. Bolles and B. Roth, Eds., MIT Press, Cambridge, MA, 1988.

[2] J. Craig, "Geometric Algorithms in AdeptRAPID," *Robotics: The Algorithmic Perspective: 1998 WAFR*, P. Agarwal, L. Kavraki, and M. Mason, Eds., AK Peters, Natick, MA, 1998.

[3] R. Goldman, *Design of an Interactive Manipulator Programming Environment*, UMI Research Press, Ann Arbor, MI, 1985.

[4] S. Mujtaba and R. Goldman, "AL User's Manual," 3rd edition, Stanford Department of Computer Science, Report No. STAN-CS-81-889, December 1981.

[5] T. Lozano-Perez, "Spatial Planning: A Configuration Space Approach," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-11, 1983.

[6] B. Shimano, C. Geschke, and C. Spalding, "VAL - II: A Robot Programming Language and Control System," SME Robots VIII Conference, Detroit, June 1984.

[7] R. Taylor, P. Summers, and J. Meyer, "AML: A Manufacturing Language," *International Journal of Robotics Research*, Vol. 1, No. 3, Fall 1982.

[8] Adept Technology Inc., "The Pilot User's Manual," Available from Adept Technology Inc., Livermore, CA, 2001.

[9] B. Mirtich and J. Canny, "Impulse Based Dynamic Simulation of Rigid Bodies," Symposium on Interactive 3D Graphics, ACM Press, New York, 1995.

[10] B. Mirtich, Y. Zhuang, K. Goldberg, et al., "Estimating Pose Statistics for Robotic Part Feeders," Proceedings of the IEEE Robotics and Automation Conference, Minneapolis, April, 1996.

[11] Adept Technology Inc., "AIM Manual," Available from Adept Technology Inc., San Jose, CA, 2002.

[12] B. Nelson, K. Pedersen, and M. Donath, "Locating Assembly Tasks in a Manipulator's Workspace," IEEE Conference on Robotics and Automation, Raleigh, NC, April 1987.

[13] plus 1.67pt minus 1.11pt T. Lozano-Perez, "A Simple Motion Planning Algorithm for General Robot Manipulators," *IEEE Journal of Robotics and Automation*, Vol. RA-3, No. 3, June 1987.