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*.

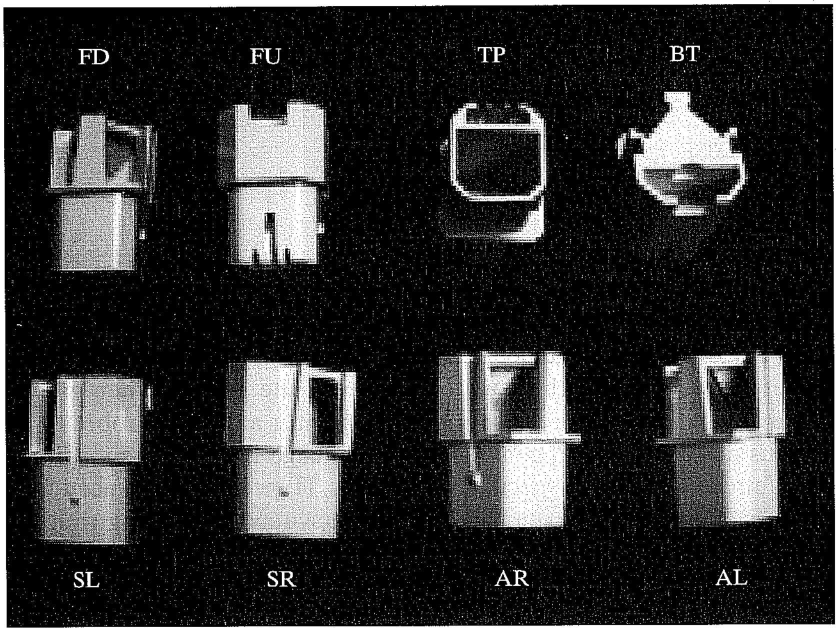

FIGURE 13.1: The eight stable states of the part.

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.

[14] R. Brooks, "Solving the Find-Path Problem by Good Representation of Free Space," *IEEE Transaction on Systems, Man, and Cybernetics*, SMC-13:190–197, 1983.

[15] J. Bobrow, S. Dubowsky, and J. Gibson, "On the Optimal Control of Robotic Manipulators with Actuator Constraints," *Proceedings of the American Control Conference*, June 1983.

[16] K. Shin and N. McKay, "Minimum-Time Control of Robotic Manipulators with Geometric Path Constraints," *IEEE Transactions on Automatic Control*, June 1985.

[17] J.J. Craig, "Coordinated Motion of Industrial Robots and 2-DOF Orienting Tables," Proceedings of the 17th International Symposium on Industrial Robots, Chicago, April 1987.

[18] S. Ahmad and S. Luo, "Coordinated Motion Control of Multiple Robotic Devices for Welding and Redundancy Coordination through Constrained Optimization in Cartesian Space," *Proceedings of the IEEE Conference on Robotics and Automation*, Philadelphia, 1988.

[19] M. Peshkin and A. Sanderson, "Planning Robotic Manipulation Strategies for Sliding Objects," IEEE Conference on Robotics and Automation, Raleigh, NC, April 1987.

[20] E. Russel, "Building Simulation Models with Simcript II.5," C.A.C.I., Los Angeles, 1983.

[21] A. Kusiak and A. Villa, "Architectures of Expert Systems for Scheduling Flexible Manufacturing Systems," IEEE Conference on Robotics and Automation, Raleigh, NC, April 1987.

[22] R. Akella and B. Krogh, "Hierarchical Control Structures for Multicell Flexible Assembly System Coordination," IEEE Conference on Robotics and Automation, Raleigh, NC, April 1987.

[23] R. Smith, M. Self, and P. Cheeseman, "Estimating Uncertain Spatial Relationships in Robotics," IEEE Conference on Robotics and Automation, Raleigh, NC, April 1987.

[24] H. Durrant-Whyte, "Uncertain Geometry in Robotics," IEEE Conference on Robotics and Automation, Raleigh, NC, April 1987.

EXERCISES

13.1 [10] In a sentence or two, define collision detection, collision avoidance, and collision-free path planning.

13.2 [10] In a sentence or two, define world model, path planning emulation, and dynamic emulation.

13.3 [10] In a sentence or two, define automatic robot placement, time-optimal paths, and error-propagation analysis.

13.4 [10] In a sentence or two, define RPL, TLP, and OLP.

13.5 [10] In a sentence or two, define calibration, coordinated motion, and automatic scheduling.

13.6 [20] Make a chart indicating how the graphic ability of computers has increased over the past ten years (perhaps in terms of the number of vectors drawn per second per $10,000 of hardware).

13.7 [20] Make a list of tasks that are characterized by "many operations relative to a rigid object" and so are candidates for off-line programming.

13.8 [20] Discuss the advantages and disadvantages of using a programming system that maintains a detailed world model internally.

PROGRAMMING EXERCISE (PART 13)

1. Consider the planar shape of a bar with semicircular end caps. We will call this shape a "capsule." Write a routine that, given the location of two such capsules, computes whether they are touching. Note that all surface points of a capsule are equidistant from a single line segment that might be called its "spine."
2. Introduce a capsule-shaped object near your simulated manipulator and test for collisions as you move the manipulator along a path. Use capsule-shaped links for the manipulator. Report any collisions detected.
3. If time and computer facilities permit, write routines to depict graphically the capsules that make up your manipulator and the obstacles in the workspace.