USN | | | | | | | | | | |

Internal Assessment Test 3 – Nov. 2019

Sub:	Automation & Robotics					Sub Code:	17ME563	Branch:	Mech	
Date:	19.11.2019	Duration:	90 min's	Max Marks:	50	Sem / Sec:	V/A&B		OBE	

Answer any FIVE Questions

		MARKS	CO	RBT
1	Briefly explain the three levels of robot programming	[10]	CO5	L1
2	Discuss briefly about the various requirements of Robot programming language	[10]	CO5	L1
3	List and explain the problems pertaining to Robot Programming Languages	[10]	CO5	L1
4	Write short notes on OLP systems.	[10]	CO5	L1
5	Explain the various parameters to be considered while designing OLP systems.	[10]	CO5	L1
6	Write short notes on automatic subtasks in OLP systems	[10]	CO5	L1

Solutions Key

Q.No	Solution
1.	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 which the sequencer would read during playback. During the teach phase, the user would guide the robot by hand, or through interaction with a **teach pendant**. Teach pendants are hand-held button boxes which 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. **Explicit robot programming languages** With the arrival of inexpensive and powerful computers, the trend has has been increasingly toward programming robots via programs written in computer programming languages. Usually these computer programming languages have special features which apply to the problems of programming manipulators and so are called **robot programming languages** (RPLs). Most of the systems which come equipped with a robot programming language have also retained a teach-pendant style interface as well. Robot programming languages have taken on many forms as well. We will split them into three categories as follows:

control language, and 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, VAL II, now provides these features [2]. Another example of a specialized manipulation language is AL, developed at Stanford University [3].

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. Examples include AR-BASIC from American Cimflex [4] and Robot-BASIC from Intelledex [5], both of which are essentially subroutine libraries 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 [6].
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. An example of such a robot programming language is AML developed by IBM [7]. The Robot programming language KAREL, developed by GMF Robotics [8], is also in this category, although the language is quite similar to Pascal.

Task-level programming languages

The third level of robot programming methodology is embodied in **task-level programming languages**. These are languages which allow the user to command desired subgoals of the task directly, rather than to 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 which avoids collision with any surrounding obstacles, automatically choose a good grasp location on the bolt, and 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 which 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 exist yet but is an active topic of research [9], [10].

2.

World modeling

Since manipulation programs must by definition involve moving objects in three-dimensional space, 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 which are used to represent joint angle sets, as well as Cartesian positions, orientations, and frames. Predefined operators which 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 which 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.

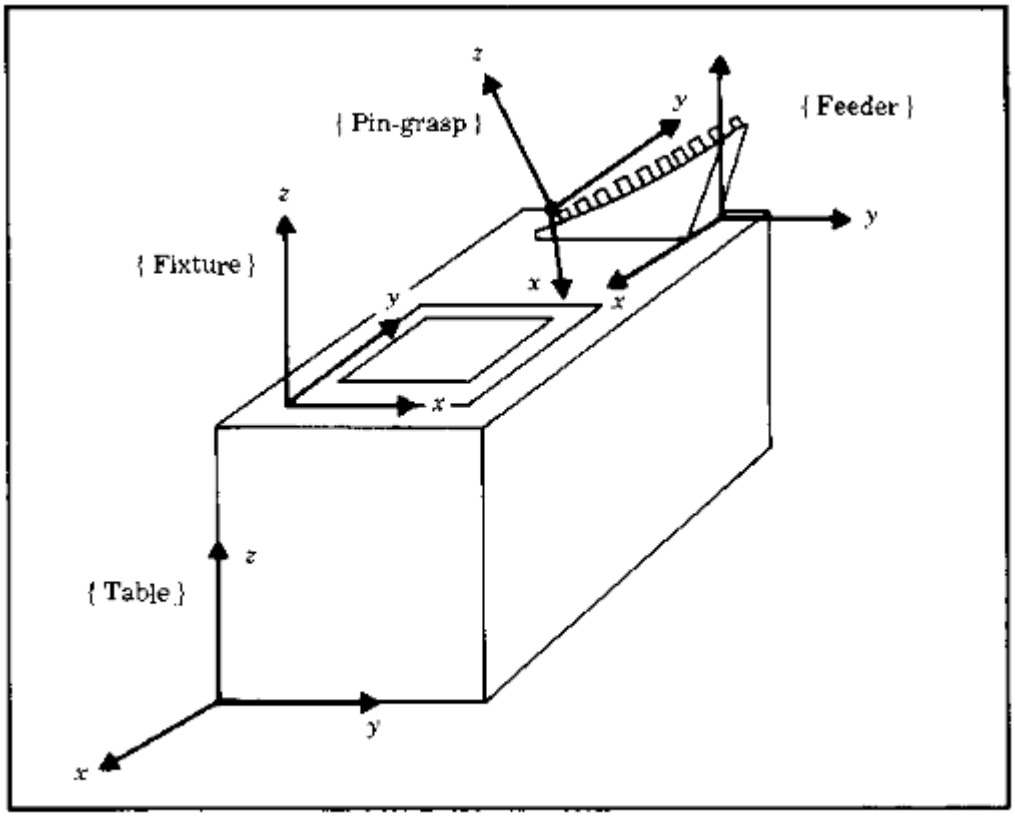

FIGURE 12.3 Often a workcell is modeled only by a set of frames which are attached to relevant objects.

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.

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 and the goal point, and whether to use joint-interpolated motion or Cartesian straight-line motion. Additionally, the user may 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
```

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. But even in single-robot applications such as the one shown in Fig. 12.2, there is other workcell equipment which 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 which are detected by the sensors. The ability easily to specify such **event monitors** is afforded by some robot programming languages [2], [3].

Programming environment

As with any computer languages, a good programming environment helps to increase programmers' 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. 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 the minimum capability to query touch and force sensors and 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. Since the camera's frame is known relative to the station frame, a desired goal frame for the manipulator can be computed from this information.

Some sensors may 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].

3.

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.

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 resulting in 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 are no assumptions concerning the context with which these program pieces execute. For manipulator progamming 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 may 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 may 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 cubic spline joint space paths studied in Chapter 7. While these effects might be dealt with by proper programming care, such problems may not arise until after the initial language statements have been debugged in isolation and are then joined with statements preceding them.

Error recovery

Another direct consequence of working with the physical world is that objects may not be exactly where they should be and hence motions that deal with them may 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, not being able to locate a hole, and so on.

4.

Off-line programming of robots offers other potential benefits which are just beginning to he appreciated by industrial robot users. We have discussed some of the prohlems 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. Thus, many of the prohlems peculiar to rohot 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 toward 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, etc.). Hence, OLP systems should be a useful tool in research as well as an aid in current industrial practice.

5. ## User interface

Since a major motivation for developing an OLP system is to create an environment that makes programming manipulators easier, the user interface is of crucial importance. However, the other 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, how can it he easier without the presence of physical device? This question touches upon the essence of the OLP design problem.

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 he readily available. Because of the strong desire for this kind of CAD integration from design to production, it makes sense for an OLP system 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.

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. Concerning inverse kinematics, the OLP system can interface to the robot controller in two distinct ways. First, the OLP system can 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.

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 robot controllers, and these 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 in predicting the cycle times of applications. When a robot is operating in a moving environment (for example, near another robot) accurate simulation of the temporal attributes of motion is necessary to accurately predict collisions, 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 modeling the dynamics of the manipulator and objects which it moves, as well as the control algorithm used in the manipulator controller. Presently 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 fruitfully pursued.

Multiprocess simulation

Some industrial applications involve two or more robots cooperating in the same environment. Even single robot workcells often contain a conveyor belt, transfer line, 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.

6.

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 which 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 perhaps by heuristic guided search techniques. Since most robots are mounted flat on the floor (or ceiling), and have their first rotary joint perpendicular to the floor, it is generally only necessary to search by tesselation 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 given an even stronger definition. A reasonable criterion to maximize might be some form of a *measure of manipulability* as discussed in Chapter 8. An implementation using a similar measure of manipulability has been discussed in [11]. 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 [12,13] and the planning of time-optimal paths [14-16] are natural candidates for inclusion in an OLP system. Some related problems which have a smaller scope, and 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 a certain relationship between the workpiece and the gravity vector to 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 in a coordinated fashion. In such a system there may be nine or more degrees of freedom to coordinate. Such systems are generally programmed today using teach 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. While some discrete time simulation systems offer abstract simulation of such systems [20], 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 errors 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.