23/1/89.

MSc in Information Technology: Knowledge Based Systems

Intelligent Assembly Systems

Notes on On-Line Robot Programming Languages

Recommended Reading: Chapter 8 of Critchlow, *Introduction to Robotics.* Further reading: Chapters 8-10 of *Real Time Programming - Neglected Topics,* Caxton C Foster, Addison-Wesley Microbooks, 1982, and Chapter 8 of Craig's *Introduction to Robotics, Mechanics and Control.*

These notes are provided as an additional perspective to the material in Critchlow. By *on-line* robot programming languages I mean languages which are implemented in the on-line system, i.e., the robot controller. Some of these can be used for off-line programming, e.g., by using a PC with editor to construct the source code which will later be interpreted by the on-line system. In these lectures I reserve the term *off-line programming language* for those languages which require the use of off-line facilities of significant complexity to aid the construction of the robot program. This might be some kind of compiler, which takes a higher level specification of the assembly task, and compiles it into terms of the on-line language; or it might be the use of tools such as simulators to help debug the on-line program without the use of the on-line system.

In this course we are primarily concerned with progress towards *Intelligent Assembly Systems.* The earlier methods of robot control which did not provide a programming language are therefore only of historical interest, in so far as they have contributed to the evolution of programmable robots, in the sense of robots which can be programmed by a computer programming language, with appropriate robotic extensions.

Bottom up - programming electric motors

Consider the case of a robot driven by electric motors. The controlling computer has six output ports, connected to six D/A converters, connected to six amplifiers, which drive the six motors of the robot. We want to get this robot to do an assembly task. This is going to be difficult.

Computer science has a well established suspicion that the best way to tackle the design of complex computer systems is by a combination of top-down and bottom-up design. The problem is decomposed downwards from the top, in an elaborating sort of tree structure, where each node gives rise to several inferior nodes. At the same time the elementary programming language is usually unsuitably low in level as the final level of this expanding decomposition - that gives rise to too complex a system. So the level of the programming language is also raised in level by creating modules which are more suitable building bricks for the kind of system in question. This lowering of the top and raising of the bottom goes on until the two are close enough that it can be seen how to join them together. Two factors which are considered to be important in minimising the complexity of the final system are limiting the number of nodes resulting from each downwards decomposition to a maximum of about seven; and joining the bottom up modularisation to the downwards decomposition at the right level.

An obvious and appealing middle level in the case of an assembly robot is straight line motion between two points: it is clearly possible to express most if not all assembly tasks as a sequence of straight line motions between two points. Curves can be approximated by polygons.

Let us first consider the bottom level of this robot system.

An electric motor is an electromechanical device which produces an output torque which is proportional to the input current. A robot with six degrees of freedom will have six motors, and so it is driven, at the bottom level, by supplying six currents to the six motors, and varying these currents frequently enough, in proportion to the changing position of the robot, to get it to go to the required position. So the computer can control a robot by supplying output signals which are turned by amplifiers into currents for the motors. The computer must decide how strongly to drive each motor in order to get the robot to go to the desired position.

So the lowest level of control of the robot, to which all higher levels must eventually be decoded, is this level of supplying signals to the power amplifiers which drive the motors. In other words, the lowest level of programming the robot is in terms of a sequence of motor currents.

In a typical industrial assembly robot the decisions about what current to apply to each motor need to be taken about 50-100 times a second. And the power applied to each motor will have an effect on the the other motors, e.g., driving the shoulder one way will tend to drive the elbow the other way. And the dynamics of the arm changes radically as the robot changes its attitude, e.g., the inertia seen by the shoulder motor is very much greater with a fully extended arm than a fully retracted one.

If all the parameters were known sufficiently precisely, then the computations could be done in advance, and the whole sequence of currents fed into the robot controller. The robot program would then effectively consist of thousands of "motor current" steps, a step taking say 10 milliseconds to execute. In fact nobody even considered doing it this way: the size of the program turns out to be prohibitively large, well beyond the memory capacity of the computers used for robot control; and the parameters require to be known to an impractical precision - changes in oil viscosity and length of the robot due to temperature, for example, would introduce gross errors in position. On-line position feedback is essential in order to be able to handle these variations, and therefore on-line computation of required motor power, from the position signals, is essential.

It turns out that trying to solve the mathematics of such an inter-related system is very complex. It is so complex that the mathematics has only been solved for certain simplified cases, and there is as yet no known way of solving these cases within the computational power of the average robot controller in 10 or 20 milliseconds.

Servo control of electric motors

The first stage in the evolution of programmable control of robots was the development of motor servo control systems which involved a computer in the control loop. The basic motor servo system involves comparison of the position of the motor (or some device driven by the motor) with a goal position, to produce an error signal. From this error signal is then derived a voltage or current to supply to the motor, which will drive it in the direction required to minimise the error. The error signal could simply be fed into an amplifier to produce a voltage to apply to the motor, using the amplifier cut-off to ensure that the motor didn't burn out. If the gain was too large the system would overshoot the mark by a larger extent each time. If the gain was too small the system would fail to reach the goal. Setting the gain just right can only be done in such a simple system if other parameters, such as inertia and friction, are constant, and sufficient viscous damping exists to stabilise the system. This can't be arranged in the joint of a robot: inertia will vary considerably with the attitude of the whole arm, and what it is carrying; the fact that gravity sometimes assists and sometimes resists the motion will produce final errors that the system cannot null; and for reasons of efficiency viscous damping will be minimised by such measures as ball bearings.

If the output of the servomechanism is to be calculated from the error signal by a computer, then complex functions of the signal can be implemented in software. The kind of control most frequently used to control the actuators of servoed robots is known as PID control (Proportional Integral and Differential). This introduces a differential term into the computation (i.e. dependent on velocity) to provide a computed equivalent of viscous damping; and an integral term, which sums the errors over time, and so reduces errors to zero, despite the presence of perturbing factors such as gravity, friction, and stiction. Robust PID servo controls can be devised which will bring a motor reliably, and in a reasonable time, to the goal position, despite considerable variations in inertia, damping, and perturbing forces. And it so happens that a PID servo control loop can be run on the average 8 bit microcomputer well within the required 10-20 milliseconds.

Wrapping up an electric motor which is driving a joint inside an appropriate PID servo controller produces a device which can be given a command to go to a certain position, and which will reliably go there despite changing inertias, perturbing forces, and so on. And since each joint is controlled by a goal seeking servo, then the robot will finally end up in the desired position, despite the interactions between the motors. Some joints may initially be driven the wrong way by powerful forces exerted on them by other joints, but this increased error will increase the force exerted by that joint's motor, and the whole thing will settle down, in the end, in the right position.

Joint level robot programming

This is the simplest kind of control found in modern computer controlled robots, where all the joints are activated at once, and terminate at various times depending on how far they had to travel, and whether other motors, gravity, etc., was helping or hindering them. Critchlow calls it *Unco-ordinated Joint Control.* He also defines a simpler variant, where the joints are activated in sequence, to some extent minimising the interaction, and lessening the computer power required, calling this *Sequential Joint Control* - but this is rarely found today, even in "toy" robots.

While the control of one joint is within the computational scope of a simple microcomputer, the control of six is too much. Hence all modern industrial robot controllers are multiprocessor, with a processor devoted to each joint or two, and a controlling processor in charge of running them all.

The trouble with this level is that the path taken by the end effector changes direction every time one motor stops. This can give very erratic trajectories when the motion is large. By adding a little extra complexity to the motor control loops, we can arrange to control not only the position, but the time taken to reach it. This permits the implementation of what Critchlow calls *Terminally Co-ordinated Joint Control,* often referred to as *Joint-interpolated Motion,* in which it is arranged that all the motors will start and finish their motions together. This causes the end-effector of the robot to travel in a smooth curve between the start and finish positions - and the same curve, when travelling from A to B as from B to A. This smoother trajectory means that an assembly task requires the use of less positions to define it, because *unco-ordinated control* always requires the interpolation of extra positions to control the erratic trajectories when in narrow spaces. A further benefit is that, unlike *unco-ordinated control,* small *joint-interpolated motions* approximate to straight lines. Robots which are programmed at this level sometimes further simplify the programming task by employing special kinematics which ensure that the some of the end-effector rotations are kept constant with respect to the world, despite changes in the other joints. An anglepoise lamp with full parallelogram linkages is an example of this kind of kinematics.

While this is a practical way of controlling the motion of the robot, it is not efficient. Because no account is being taken of the changing dynamics of the robot, and reliance is placed on the robustness of the servos to keep the motions within limits, it is necessary to restrain the speed and acceleration of the robot within limits that the individual servos can cope with. It is also necessary to make the robot sufficiently stiff that it bends very little under the loads and accelerations it will be subject to. In practice this means that modern assembly robots are far more heavily and stiffly constructed, and have far more powerful motors, than would be necessary if we could implement more advanced control systems.

Manipulator level programming

Because small *joint-interpolated motions* approximate to straight lines, there is an obvious way of implementing *straight line motion,* in terms of *joint-interpolated motion,* which is for the computer to interpolate enough intermediate points along the straight line between the endpoints of the motion, that *joint-interpolated motion* between them is a good enough approximation to straightness. The problem that has to be solved here, in order to do this straight line interpolation, is known as the *reverse kinematics,* deducing the joint angles for which the end-effector will be in a certain position in space. The *forward kinematics,* deducing the position of the end effector from the joint angles, is easy for a serial robot, but the *reverse kinematics,* is hard. It is sufficiently hard that it has so far only been solved within the scope of the power of an economic robot controller for specially simplified cases. The most common simplification is to arrange the three rotations at the end of the robot arm, intersecting at a common point. This considerably decouples the rotations from the x,y,z translation motions of the arm, and permits the solution to be done in two simpler stages. [Note that for parallel arms (e.g. the Gadfly) the reverse kinematics is very easy, while the forward kinematics is particularly difficult, and still has unsolved problems.] It is also becoming popular to distribute the degrees of freedom, which also simplifies the control problem, by reducing the degrees of freedom in the robot, and adding them in again by holding the target assemblage in some kind of programmable vice or jig which has one or two degrees of freedom itself.

There is a lot of research going on in various ways of improving control of the robot. This can be by means of new mathematical simplifications, or new more tractable algorithms for solving existing mathematical formulations; and there are many ways of tackling it, such as solving the inverse dynamics of the robot, finding tractable and stable methods of adaptive control, or even by throwing away the joint-kinaesthetic approach, and deriving control from direct sensing of the end effector in cartesian space. This last method would permit the use of much lighter bendy robots.

In summary, the current generation of Cartesian-programmable robots (i.e., robots programmable at the so-called manipulator level, the positions of the end effector in terms of Cartesian space) have only been made practical by a number of simplifications and compromises in which engineering and metal are traded against computer power. As more sophisticated control algorithms are developed, and more powerful computers become more cheaply available, we can expect robots to become lighter, faster, more powerful, and more efficient in terms of energy use.

At the moment a single arm capable of the speed, power, and precision, of the human arm in assembly tasks (such as the Adept) is comparable in weight to a complete human - and that is leaving out the controller.

There is one more important mathematical discovery behind the current generation of Cartesian-programmable robots, and that is the use of homogeneous matrices to represent transformation of spatial co-ordinate systems. The mathematics was developed in 1955 by Denavit and Hartenberg as a way of describing the relationship between the links of a mechanism. The important feature of these homogeneous transformations is that they can perform both translation and rotation of an axis system expressed in this form. Thus expressing the relationship of each link to the next in this form permits the final position of the end-effector to be obtained by multiplying them together; and they similarly allow the chaining of transformations describing the relationships of the parts to be assembled to one another, pallets, jigs, feeders, and other features of the work cell. Thus this formalism permits the simple implementation and expression of indexing through six-dimensional space. In 1965 L.G. Roberts applied this to the definitions of end-points of motions in robot programming. On this basis, in 1972 D.E.Whitney developed a method of smoothly accelerated motion between several points without stopping, and R.P.Paul laid the theoretical foundations of trajectory control, some of which is still not implementable within the power of current robot controllers.

Programming sensors

The first computer controlled robots did have sensors - the joint position sensors - but these were wrapped up in the motor servo-controllers, and not accessible to the programmer. As far as the programmer was concerned, these sensors had been hidden from view in the transformation of a device controlled in terms of motor currents into a device controlled in terms of position. At first this position was expressed in terms of the joint angles of the robot, and hence the popularity of the robots with prismatic Cartesian joints, since these permitted a correspondence with a Cartesian description of the robot's position. The programming language of the robot was thus a simple list of positions through which to move, decorated with opening and closing the gripper, changing speed, and so on. It is possible at this level of programming to use dynamic sensing, i.e., sensors which require the motion of the robot, such as touch sensors. Thus the robot can perform a move either until it reaches the destination, or until a sensed condition, such as touching the object, occurs.

This is known as *guarded motion.* The robot program still consists of a sequence of positions through which to move, but *guarded motions* permit the early termination of some motions due to a sensed condition.

It is also possible to use these sensed conditions to take decisions about what to do next, i.e., to decide between alternative sequences of motions. In most on-line joint-level languages this facility is not provided, with MHI, the first, being an honoorable exception.

The solution of the reverse kinematics of the "anthropoid" type of arm (entirely revolute joints, but partitioned between position and orientation), and the use of homogeneous matrices permitted the programming of robot arms purely in terms of the Cartesian position of the end effector - the so-called manipulator level languages. This also created the possibility of indexing through space, and of using static sensors (i.e., which operate independently of robot motion, such as vision from a fixed camera) to discover the positions of objects to be grasped. Whereas guarded motions only permitted early termination of a motion along a predefined trajectory, this Cartesian level interface permitted a sensor to define a position for the robot to move to anywhere within its workspace. The kind of computations involved in this demanded the full power of a high level computer programming language, with its mathematical functions, and control structures. This in turn meant that sensors could also be used to take decision about what to do next, i.e., to decide between alternative tasks.

Some on-line robot programming languages

MHI (1961). This was developed to control the MIT Mechanical Hand One by Ernst, and had a variety of binary sensors. These could be used to terminate motions (guarded motions) or to choose between sequences. It lacked general purpose calculation and control commands.

VAL (1975), developed by Schienman for Unimation. Based on BASIC, it was limited by the absence of commands to decompose transformations, which made the properly general use of sensors impossible, This was remedied in the much improved VAL2 (1980). Although lacking a specific guarded move command, simple forms could be implemented using the sensed-value interrupt feature.

MCL (1978), McDonnell-Douglas, based on APT, the standard language for numerically controlled machine tools, it was limited by the primitive syntax of APT.

PAL (1981), developed by Paul by adding robotic capabilities to PASCAL. Very general and powerful transformation handling.

PASRO (1981), a similar development by Blume (of Karlsruhe), specifically for small stepper motor robots. This was basically a joint-level language for small micros, with manipulator level available as a rather slow extra.

LM (1981), developed by Latombe (Grenoble), a PASCAL-like language, and the only one of these on-line languages to provide complex guarded motions as an integral part of the motion commands.

RPL (SRI), a Forth-like language designed for non-programmers.

Experience had by this time made it clear that senseless robots would not be able to handle more than a fraction of industrial assembly tasks - uncertainty was too large a problem, and removing the uncertainty by means of hard automation techniques to the level of precision required by an assembly robot was inflexible and expensive enough to remove the motivation for the use of the robot. Robots were going to need sensors in order to handle the uncertainties of most industrial assemblies.

This meant that off-line robot programming and planning was going to have to incorporate the use of sensors to handle uncertainty. While most off-line languages from their earliest days did incorporate sensing commands, and the necessary generality in calculation, control, and transformation handling to exploit sense data fully, most of them were similar to the above on-line languages, but with enough extra generality and power in the handling of transformations and trajectories to put them beyond the power of the on-line controllers of the time. So while these languages permitted exploitation of the full power of sensors, it was still left to the programmer to decide when and how to use the sensors, and the necessary computations with which to link the sensor data into the Cartesian world of the robot had to be programmed explicitly at this level by the programmer.

----------O----------