Russell H. Taylor

Planning and Execution of Straight Line Manipulator Trajectories

Recently developed manipulator control languages typically specify motions as sequences of points through which a tool affixed to the end of the manipulator is to pass. The effectiveness of such motion specification formalisms is greatly increased if the tool moves in a straight line between the user-specified points. This paper describes two methods for achieving such straight line motions. The first method is a refinement of one developed in 1974 by R. Paul. Intermediate points are interpolated along the Cartesian straight line path at regular intervals during the motion, and the manipulator's kinematic equations are solved to produce the corresponding intermediate joint parameter values. The path interpolation functions developed here offer several advantages, including less computational cost and improved motion characteristics. The second method uses a motion planning phase to precompute enough intermediate points so that the manipulator may be driven by interpolation of joint parameter values while keeping the tool on an approximately straight line path. This technique allows a substantial reduction in real time computation and permits problems arising from degenerate joint alignments to be handled more easily. The planning is done by an efficient recursive algorithm which generates only enough intermediate points to guarantee that the tool's deviation from a straight line path stays within prespecified error bounds.

Introduction

Over the past ten years, general purpose automation machines consisting of a sequence of motor driven links, or "joints," operating under control of a computer have begun to appear in industry. These "manipulators" generally terminate in a gripper-like hand or other tool and can be used for assembly, parts handling, welding, and many other applications.

The development of these devices requires both provision of suitable formalisms for describing the motions to be made and implementation of suitable control strategies for carrying them out. The simplest approach is to record the values of the joint parameters which place the hand at particular desired points and then to move the joints independently from one set of parameters to the next. More sophisticated motion execution schemes (*e.g.*, [1-3]) frequently include an open loop trajectory component that generates intermediate target values for the joints. These schemes are often accompanied by more sophisticated means of describing the desired motion to be made.

Of particular interest has been the development of programming languages [4-7] in which manipulator target points are described by transformations relating the coordinate system of the hand or tool to the coordinate system of the work station. Motions in these languages are specified as sequences of "knot" points through which the controlled frame is to pass. The joint vectors corresponding to each Cartesian knot point are computed by solving the link equations for the manipulator, which are then used by the motion execution programs. One drawback to this scheme is that it leaves undefined the precise path taken by the manipulator between knot points. This makes programming more difficult, since it is hard to predict just how many intermediate points should be added to a motion statement and complicates the de-

Copyright 1979 by International Business Machines Corporation. Copying is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract may be used without further permission in computer-based and other information-service systems. Permission to *republish* other excerpts should be obtained from the Editor.

velopment of model-based program automation tools [5, 7].

One obvious motion strategy is to cause the hand or tool to move along a straight line path between knot points. Whitney [8] achieved differential straight line motions in 1968 by multiplying the inverse Jacobean of the manipulator's joint to Cartesian space transformation by a desired motion increment vector. By repeated evaluation of the inverse Jacobean, this technique can be used to produce long straight line motions. In 1974, Paul [9] implemented a rather more straightforward technique, in which intermediate Cartesian space goals are evaluated every 100 ms during motion execution. The manipulator link equations are then solved to produce intermediate joint goals.

This paper discusses two approaches to the problem of straight line motion between knot points. The first approach is similar to Paul's. However, a number of refinements are presented which permit the path functions to be evaluated somewhat more efficiently, provide more uniform rotational motions, and allow "tracking" of real time changes in the knot points. Although the method is flexible and conceptually straightforward, it requires considerable real time computation and is vulnerable to degenerate configurations of the manipulator's joints. In the second approach, a motion planning phase adds enough intermediate points so that the manipulator may be controlled by linear interpolation of joint values without allowing the hand to deviate more than a prespecified amount from a straight line Cartesian path. This substantially reduces the amount of real time computation required to drive the machine and permits problems arising from degenerate joint alignments to be handled more easily.

Notational conventions

We assume that the manipulator's motion is specified by a sequence of "frame" transformations giving the location and orientation of the hand with respect to the coordinate system of the work station. Each such frame F_i consists of a rotation R_i followed by translation by a displacement vector $\mathbf{p}_i$. Frames may be composed by "multiplication" on the left, where

$$\text{rotation part } (F_1 \circ F_2) = R_1 \circ R_2,$$

$$\text{translation part } (F_1 \circ F_2) = R_1 \circ \mathbf{p}_2 + \mathbf{p}_1.$$

For instance, if F_A gives the location and orientation of an object A with respect to the work station and $\mathbf{p}_f$ is the location of a point with respect to the origin of A, then the location of the point with respect to the work station is given by

$$\mathbf{p}_w = R_A \circ \mathbf{p}_f + \mathbf{p}_A.$$

Similarly, if R_f gives the orientation of a feature with respect to A, then its orientation with respect to the work station is

$$R_w = R_A \circ R_f.$$

We frequently express a rotation R as a "right-handed" twist by an angle θ about an axis $\mathbf{n}$:

$$R = Rot(\mathbf{n}, \theta).$$

From this definition, it follows that

$$R^{-1} = Rot(\mathbf{n}, -\theta) = Rot(-\mathbf{n}, \theta).$$

Frame transformations are commonly represented as 4 × 4 matrices:

$$\mathbf{F} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & \mathbf{p}_1 \\ R_{21} & R_{22} & R_{23} & \mathbf{p}_2 \\ R_{31} & R_{32} & R_{33} & \mathbf{p}_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

This representation is easy to understand and use, since frames may be composed using the ordinary rules for matrix multiplication. Also, since rotation matrices have the property that their inverse is the same as their transpose,

$$R^{-1} = R^{\mathrm{T}},$$

the inverse transformation for a frame is easily computed.

On the other hand, there are several disadvantages to the use of matrix representations. The matrices are moderately expensive to store, and computations on them require more operations than for some other representations. Also, since the representation of rotations is highly redundant, numerical inconsistencies can be a problem, so that occasional renormalization may be necessary.

Quaternions [10, 11] offer another convenient representation for rotations and have been applied extensively to the analysis of kinematic linkages [12]. The quaternion corresponding to a rotation by angle θ about axis $\mathbf{n}$ is

$$Rot(\mathbf{n}, \theta) = \left[\cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right) \cdot \mathbf{n}\right].$$

For our purposes, this representation is rather more efficient than the matrix representation. Storage requirements are reduced, and calculations involving rotations can be done with fewer primitive operations (adds and multiplies) than are required if matrices are used. Appendix A provides a brief review of quaternions and includes a table comparing the computational requirements of quaternion and matrix representations for rotations.

Table 1 Computational costs.

Operation	*Quaternion representation*	*Matrix representation*
Segment traversal, Eq. (2)	13 adds, 24 multiplies, 1 sine-cosine pair	29 adds, 44 multiplies, 1 sine-cosine pair
Segment traversal, Eq. (2), with R_1 fixed	12 adds, 12 multiplies, 1 sine-cosine pair	17 adds, 19 multiplies, 1 sine-cosine pair
Segment traversal, Eq. (3), with R_0 fixed (Ref. [6])	—	35 adds, 57 multiplies, 2 sine-cosine pairs
Segment transition, Eq. (4)	26 adds, 52 multiplies, 2 sine-cosine pairs	58 adds, 90 multiplies, 2 sine-cosine pairs
Segment transition, Eq. (4), with R_1 fixed	25 adds, 40 multiplies, 2 sine-cosine pairs	49 adds, 66 multiplies, 2 sine-cosine pairs
Segment transition, Eq. (5), with R_0, R_1, R_2 fixed (Ref. [6])	—	53 adds, 79 multiplies, 2 sine-cosine pairs
Pursuit solution, Eq. (6)	27 adds, 44 multiplies, 1 arctangent, 1 sine-cosine pair, 1 square root	57 adds, 78 multiplies, 2 square roots, 1 arctangent, 1 sine-cosine pair
Pursuit traversal, Eq. (7)	52 adds, 92 multiplies, 2 sine-cosine pairs, 2 arctangents, 2 square roots	112 adds, 156 multiplies, 2 square roots, 3 sine-cosine pairs, 2 arctangents
Joint solution for manipulation in Fig. 1	23 adds, 35 multiplies, 5 inverse trigonometric functions, 3 square roots, 2 sine-cosine pairs	*19 adds, 35 multiplies 5 inverse trigonometric functions, 3 square roots

*See Ref. [14].

Cartesian path control

The basic algorithm of Cartesian path control is quite simple:

loop: wait for next control interval

$t := t + \Delta t$,

$F(t)$:= where hand should be at time t,

$\mathbf{j}(t)$:= joint solution corresponding to $F(t)$

[$\mathbf{j}(t)$ is now the new "open loop" goal],

go to loop.

The computation thus consists of a "path function," $F(t)$, followed by a "joint solution," $\mathbf{j}(F(t))$. The path functions developed in this paper resemble those developed in 1974 by R. Paul [2, 9, 13]. However, they handle rotations more efficiently and uniformly, provide a cleaner mechanism for transition between segments, and permit the manipulator to chase down changes to knot points during execution of the path. Table 1 summarizes the computational costs associated with the different path functions. The computational requirements of joint solution algorithms are ignored throughout most of this discussion, since they depend quite strongly on manipulator geometry and represent fixed overhead to the various path algorithms.

- *Single segment motion*

We wish to move the manipulator's tool frame along a "straight" path from frame F_0 to frame F_1 in time T. We envision this path as consisting of translation of the tool frame's origin from $\mathbf{p}_0$ to $\mathbf{p}_1$, coupled with rotation of the tool frame orientation part from R_0 to R_1. Let $\lambda(t)$ be the fraction of the motion segment still to be traversed at time t. For *uniform* motion we pick

$$\lambda(t) = \frac{T - t}{T}.$$

The displacement and rotation parts of the tool frame at time t are given by

$$R(t) = R_1 \circ Rot[\mathbf{n}, -\theta \cdot \lambda(t)],$$

$$\mathbf{p}(t) = \mathbf{p}_1 - \lambda(t)(\mathbf{p}_1 - \mathbf{p}_0). \quad (1)$$

Here, $Rot(\mathbf{n}, \theta)$ is a rotation by θ about axis $\mathbf{n}$ required to reorient R_0 into R_1:

$$Rot(\mathbf{n}, \theta) = R_0^{-1} \circ R_1.$$

The computational requirements of this algorithm vary somewhat, depending on the representation chosen for rotations, what elements are kept constant, and details of coding. A typical implementation is shown below:

$$\lambda := \frac{(T - t)}{T},$$

$$\mathbf{p}(t) := \mathbf{p}_1 - \lambda \cdot \Delta\mathbf{p},$$

$$R(t) := R_1 \circ \left[\cos\left(\frac{\lambda\theta}{2}\right) + sin\left(\frac{\lambda\theta}{2}\right) \cdot \mathbf{n}\right]. \quad (2)$$

Here, $\Delta\mathbf{p} = (\mathbf{p}_2 - \mathbf{p}_1)$, $\mathbf{n}$, and θ are assumed to remain constant throughout the motion, and rotations are assumed to be represented by quaternions. If R_1 is known to be fixed throughout the segment execution, it is possible to precompute many of the intermediate expressions required to produce $R(t)$, with a corresponding reduction in the real time calculation.

In contrast, Paul [9] defined a "straight line" motion from F_0 to F_1 as consisting of a uniform translation of the tool frame's origin from $\mathbf{p}_0$ to $\mathbf{p}_1$ coupled with a nonuniform rotation from R_0 to R_1:

$$t := t + 1,$$

$$\eta := t/T,$$

$$\mathbf{p}(t) := \mathbf{p}_0 + \eta\Delta\mathbf{p},$$

$$R(t) := [R_0 \circ Rot(\mathbf{k}, \eta\Theta] \circ Rot(\mathbf{z}, \eta\Phi). \quad (3)$$

The rotation consists of a uniform twist by an amount Φ about the tool frame **z** axis, coupled with another rotation of amount Θ about an axis **k** reorienting the tool's **z** axis from its initial to its final orientation. Since the composition of these two rotations will usually produce some angular acceleration, this method may be less desirable than that of (2) for applications where uniform motion on segment traversal is important. The computational cost, assuming that R_0, $\mathbf{k}$, $\Delta\mathbf{p}$, Θ, and Φ are held constant, is roughly twice that of (2) under comparable assumptions, since rotations about *two* axes must be computed.

In return for the nonuniform rotation and the higher computation costs involved, a "decomposed" rotation technique such as that of (3) may offer compensating advantages in some cases. Paul describes his rotation strategy as easy to visualize and relatively insensitive to changes in the final orientation of the tool's z axis. It may well be worthwhile to consider such factors when designing a motion strategy.

Note that the method of (2) *subtracts* a shrinking increment from the destination point, F_1. Thus, if the value of F_1 changes during the motion, then the path function "tracks" the change. So long as the change is sufficiently gradual, the discontinuity introduced whenever F_1 is changed should cause little difficulty. Later we describe a form of the path algorithm which may be applied when such discontinuities may be significant. Important uses of this tracking ability include tasks in which television images or other sensor data are used to locate objects in real time, in which the motion destination is modified with forces encountered along the way, or in which the objects being manipulated are being transported on a conveyor belt.

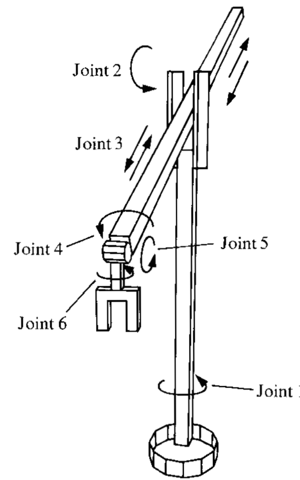

Figure 1 Typical manipulator geometry.

Paul used an alternative technique to achieve tracking of conveyor belts. Essentially, his approach is to define each knot point by its transformation with respect to a base coordinate system (the conveyor). Intermediate Cartesian goals are generated with respect to this coordinate system. The intermediate goals are then transformed into the manipulator's base coordinate system before joint goals are computed. This method, which is compatible with any of the path functions described above, is probably better suited than target point tracking for cases where a program developed for stationary objects must be executed with the objects on a conveyor, but it seems less well adapted for cases where individual object positions are updated sporadically or are computed from real time sensory information.

- *Transition between path segments*

Assume we are moving along a segment from F_0 to F_1, as discussed in the previous section. On the next segment we wish to go with uniform velocity from F_1 to F_2. If acceleration is to be limited, then the transition must start before the knot point F_1 is reached. The cornering method described here starts turning at a precomputed time τ be-

fore the scheduled arrival at F_1 and completes the transition to the new segment at time τ after the scheduled departure through F_1. When the transition is completed, the manipulator will be at the same point and moving at the same velocity as if it had gone all the way to F_1 and then made an instantaneous transition to the next segment.

For the moment, consider only the position part of the motion. The "boundary conditions" for the segment transition are

$$\mathbf{p}(T_1 - \tau) = \mathbf{p}_1 - \frac{\tau \Delta \mathbf{p}_1}{T_1},$$

$$\mathbf{p}(T_1 + \tau) = \mathbf{p}_1 + \frac{\tau \Delta \mathbf{p}_2}{T_2},$$

$$\frac{d}{dt}\mathbf{p}(t)\,|_{t=T_1-\tau} = \frac{\Delta \mathbf{p}_1}{T_1},$$

$$\frac{d}{dt}\mathbf{p}(t)\,|_{t=T_1+\tau} = \frac{\Delta \mathbf{p}_2}{T_2},$$

where $\Delta \mathbf{p}_1 = \mathbf{p}_1 - \mathbf{p}_2$, $\Delta \mathbf{p}_2 = \mathbf{p}_2 - \mathbf{p}_1$, and T_1 and T_2 are the "constant rate" traversal times for the two segments.

During the transition, we apply a constant acceleration:

$$\frac{d^2}{dt^2}\mathbf{p}(t) = \mathbf{a}_p.$$

Integrating this twice and applying the boundary conditions gives

$$\mathbf{p}(t') = \mathbf{p}_1 - \frac{(\tau - t')^2}{4\tau T_1}\Delta \mathbf{p}_1 + \frac{(\tau + t')^2}{4\tau T_2}\Delta \mathbf{p}_2, \qquad (4)$$

where $t' = T_1 - t$ is the time from the knot point.

The path equation for rotation transitions is similar:

$$R(t) = R_1 \circ Rot\left[\mathbf{n}_1, -\frac{(\tau - t')^2}{4\tau T_1}\theta_1\right] \circ Rot\left[\mathbf{n}_2, \frac{(\tau + t')^2}{4\tau T_2}\theta_2\right],$$

where

$$Rot(\mathbf{n}_1, \theta_1) = R_0^{-1} \circ R_1,$$

$$Rot(\mathbf{n}_2, \theta_2) = R_1^{-1} \circ R_2.$$

This formula produces a smooth transition between the two segments, although the angular acceleration will not be quite constant unless the axes $\mathbf{n}_1$ and $\mathbf{n}_2$ are parallel or unless one of the spin rates

$$\phi_1 = \frac{\theta_1}{T_1}, \qquad \phi_2 = \frac{\theta_2}{T_2},$$

is zero.

The computational costs for an implementation which keeps $\Delta \mathbf{p}_1$, $\Delta \mathbf{p}_2$, θ_1, θ_2, $\mathbf{n}_1$, and $\mathbf{n}_2$ constant are shown in Table 1. Again, note that the knot point F_1 is tracked throughout the transition. Further savings are possible if F_1 is known to be fixed.

An alternative transition method [9] is to modify segment path function (3) by accelerating the spin rates about tool $\mathbf{z}$ and $\mathbf{k}$ while rotating $\mathbf{k}$ to a new direction.

$$t := t + 1,$$

$$\mathbf{p}(t) := \mathbf{p}_0 + f(\Delta \mathbf{p}_1, \Delta \mathbf{p}_2, T, \tau, t),$$

$$\mathbf{k} := R_k \circ \mathbf{k},$$

$$R(t) := (R_0 \circ Rot[k, f(\Delta\Theta_1, \Delta\Theta_2, T, \tau, t)] \circ Rot[\mathbf{z}, f(\Delta\Phi_1, \Delta\Phi_2, T, \tau, t)], \qquad (5)$$

where $f(\Delta_1, \Delta_2, T, \tau, t)$ is an appropriate interpolation function. However, this technique requires more computation and is rather more involved than that given above.

- *Pursuit formulation*

Earlier, we pointed out that the path functions would track changes to the knot points, but that any sudden change would produce a parallel displacement in the command frame $F(t)$. This section describes how these discontinuities can be averaged out over the remaining segment time.

Straight line motion

As before, assume that we are moving along a straight line segment which is to reach F_1 at time T. At time t, we wish to compute the target frame for the next sample interval $t + \Delta t$, given $F(t)$ and $F_1(t)$. To do this, we compute the displacement and rotation required to move from $F(t)$ to $F_1(t)$ and then compute the correct fractional step to take, based on the time remaining:

$$\mathbf{p}(t + \Delta t) = \mathbf{p}_1(t) - \lambda(t)[\mathbf{p}_1(t) - \mathbf{p}(t)],$$

$$R(t + \Delta t) = R_1(t) \circ Rot[\mathbf{n}, -\nu \cdot \lambda(t)], \qquad (6)$$

where

$$\lambda(t) = \frac{T - (t + \Delta t)}{T - t} \text{ for } t < T, \qquad \lambda(t) = 0 \text{ for } t \geq T,$$

$$Rot(\mathbf{n}, \nu) = R(t)^{-1} \circ R_1(t).$$

Notice that any errors introduced into the calculation of $F(t)$ at one iteration will tend to be canceled out in subsequent iterations. Thus, rather crude approximations may be used for the trigonometric functions without serious harm to the performance of the algorithm.

Much of the additional computational cost of this method, as compared with that developed earlier, is incurred

in recomputing the rotation parameters, **n** and ν. In many cases, therefore, it may be worthwhile to adopt a mixed approach in which the pursuit form is only used to chase translational changes to the knot points, and any rotational changes are assumed to be small enough so that the other form of tracking will suffice.

Transition between segments

The transition between one segment, $F_0 \rightarrow F_1$, and its successor, $F_1 \rightarrow F_2$, may be thought of as consisting of two simultaneous motions. The first motion chases down a knot point F_k so that at the end of the transition ($t' = \tau$) the hand is at F_k and at rest with respect to it. The second motion accelerates F_k away from F_1 toward F_2, so that at $t' = \tau$, F_k is at the proper place and moving with the proper speed on the new segment.

$$\mathbf{p}(t') = \mathbf{p}_k(t') - \lambda(t')[\mathbf{p}_k(t') - \mathbf{p}(t')], \quad (7)$$

where

$$\mathbf{p}_k(t') = \mathbf{p}_1(t') + \mu(t')[\mathbf{p}_2(t') - \mathbf{p}_1(t')],$$

$$\lambda(t') = \left(\frac{\tau - (t' + \Delta t)}{\tau - t'}\right)^2,$$

$$\mu(t') = \frac{(\tau + t')^2}{4\tau T_2}.$$

Similarly,

$$R(t') = R_k(t') \circ Rot[\mathbf{a}, -\alpha \cdot \lambda(t')],$$

where

$$R_k(t') = R_1(t') \circ Rot[\mathbf{b}, \beta \cdot \mu(t')],$$

$$Rot(\mathbf{a}, \alpha) = R(t')^{-1} \circ R_k(t'),$$

$$Rot(\mathbf{b}, \beta) = R_1(t')^{-1} \circ R_2(t').$$

Again, it may be worthwhile to consider a mixed strategy, in which translations are handled as shown here and rotations are handled in the old way.

• *Discussion*

Figure 2 illustrates straight line motion for a typical manipulator similar to the Stanford arm [1] used in a number of research laboratories.

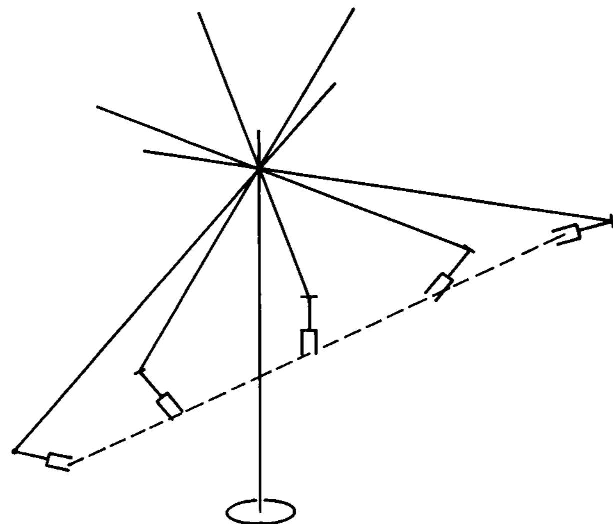

Figure 2 Straight line motion.

One of the principal advantages of the method is that the trajectory followed by the manipulator between segment endpoints is readily predictable. This predictability greatly simplifies programming and is especially important for the development of good program automation tools, such as model driven "collision avoidance" packages. Similarly, since straight line paths frequently correspond to the desired motion of the manipulator, the number of intermediate points which a user must specify in order to achieve a desired result may be significantly reduced. Since the motion is uniform, there will be no inertial forces on objects in the hand during segment traversal, and they will be constant or nearly constant during segment transitions.

On the other hand, there are a number of disadvantages. First, the calculations involved may be rather time consuming, especially if the "pursuit" form is used. For instance, the estimated time requirements using an IBM Series/1 with floating point hardware range from 3 to 8 milliseconds per segment traversal step and 5 to 10 milliseconds per transition step, depending on the particular form of path equations used. Paul [9] attacks this difficulty by slowing down the generation of Cartesian path points to a rate his computer can handle, and then interpolating additional joint space targets. This approach solves the problem of limited computer power at the expense of additional program complexity. However, it does not avoid the other difficulties associated with Cartesian paths.

A number of other difficulties arise from the fact that the hand position can only be controlled indirectly, through the joints of the manipulator. With many manipulators, there may be "degenerate" hand positions which are nearby in Cartesian space, but which are widely separated in joint parameter space. If the path passes through such points, then the joints of the manipulator may be unable to keep up with their targets unless the motion is very slow.

This problem may be solved at the expense of additional computation and programming complexity by using the current joint velocities to recompute at each sample interval how large a fraction, $\lambda(t + \Delta t) - \lambda(t)$, of the segment is to be covered in the next sample interval [9]. The path followed will still be a straight line, but the hand will not move at uniform speed, so that objects in the hand will experience various inertial forces, which will be difficult to predict without "simulating" the motion in advance. Also, coordinated motions involving several manipulators are more difficult to plan, since it is difficult to predict just where each manipulator will be at a given time.

Physical stops or limits on individual joints of the manipulator introduce complications similar to those introduced by degenerate axis alignments. In order to move the hand to a nearby point, it may be necessary to "unwind" a joint through 359° rather than advance it through 1°. Such discontinuities cannot be handled well unless they have been anticipated. Simple schemes which merely adjust speeds based on joint velocities cannot do this.

Finally, the fact that there are generally several possible sets of joint parameters which can place the hand at a desired target frame $F(t)$ introduces still further complications. Unless a certain amount of care is taken in the joint solution procedure, discontinuities in the joint target values may be introduced at awkward moments. Indeed, it may frequently be very important to select the solution which minimizes the total motion required through a many segment motion, or to avoid a joint stop during a critical segment. It is unclear just how this can be done without a certain amount of preplanning which pays attention to joint space trajectories.

The conclusion is that, although Cartesian path interpolation offers a number of significant advantages, the joint space behavior of the manipulator itself cannot be ignored. The joint space strategy described in the next section preserves many of the advantages of Cartesian path control, while requiring somewhat lower computational effort and allowing joint space considerations to be handled more easily.

Bounded deviation joint paths

The principal disadvantages of Cartesian path control are the amount of real time computation required and the difficulty of dealing in real time with constraints on the joint space behavior of the manipulator. These problems may be avoided or at least greatly reduced by *preplanning* the motion before it is executed. Sometimes this preplanning can be performed well in advance. Often, however, the values of the knot points are not known until just before the motion is executed. In such cases, it is important that the time the manipulator spends waiting for planning to be completed be kept as short as possible.

As an extreme case, the real time algorithm could be simulated and used to precompute the joint parameters for every sample interval. Motion execution would then be trivial: the joint parameter values would simply be read from memory and used as local goals for the servoing algorithm. Such a policy, however, is rather wasteful, since the amount of data that would have to be stored is quite large and since the computation time needed may approach that required by the motion itself.

One possible way out would be to precompute the joint solutions for every nth sample interval and then to perform interpolation on the joint parameters to generate real time goals. The difficulty with this method is that the number of intermediate points needed to keep the manipulator acceptably close to a straight Cartesian path depends on the particular motion being made. Any "standard" interval small enough to guarantee low deviations everywhere will require a wasteful amount of precomputation for many motions.

This section presents a simple algorithm which generates only enough intermediate points to guarantee that the manipulator's deviation from a straight path on each motion segment stays within prespecified error bounds.

• *Joint space motion strategy*

Suppose we compute joint parameter vectors $\mathbf{j}_k$ corresponding to the knot points F_k of our desired motion. We can then use these $\mathbf{j}_k$ as the knot points for a joint space interpolation strategy analogous to that used for the position part of our Cartesian space paths. For motion from $\mathbf{j}_0$ to $\mathbf{j}_1$, we have

$$\mathbf{j}(t) = \mathbf{j}_1 - \frac{T_1 - t}{T_1} \Delta\mathbf{j}_1,$$

and for transition between $\mathbf{j}_0 \rightarrow \mathbf{j}_1$ and $\mathbf{j}_1 \rightarrow \mathbf{j}_2$,

$$\mathbf{j}(t') = \mathbf{j}_1 - \frac{(\tau - t')^2}{4\tau T_1} \Delta\mathbf{j}_1 + \frac{(\tau + t')^2}{4\tau T_2} \Delta\mathbf{j}_2,$$

where $\Delta\mathbf{j}_1 = \mathbf{j}_1 - \mathbf{j}_0$, $\Delta\mathbf{j}_2 = \mathbf{j}_2 - \mathbf{j}_1$, and T_1, T_2, τ, and t' have the same meanings as for the Cartesian path motion. Here, the *joints* of the manipulator move at uniform velocity between the knot points and make smooth transitions with constant acceleration between segments [15].

The hand frame, however, deviates from a straight line Cartesian path, as shown in Fig. 3. Let $F_j(t)$ be the manipulator hand frame corresponding to the joint target $\mathbf{j}(t)$,

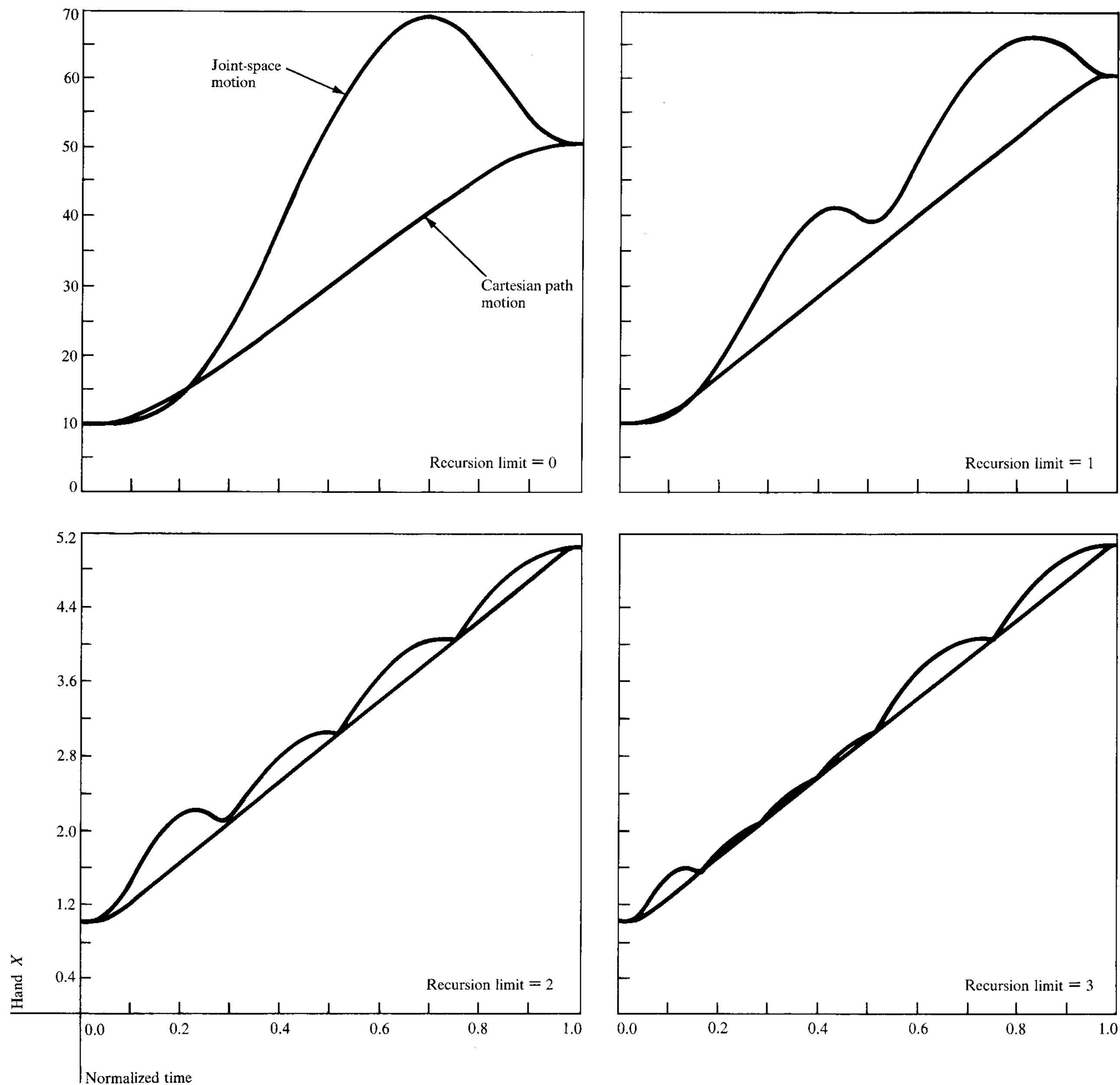

Figure 3 X coordinate of hand for typical motion segment, with midpoint interpolation limited to 0, 1, 2, and 3 levels of recursion ($\delta_p^{max} = 0.5$ cm, $\delta_R^{max} = 5°$).

and let $F_c(t)$ be the frame target for the corresponding Cartesian space path. Then, the *displacement deviation* $\delta_p(t)$ and the *rotation deviation* $\delta_R(t)$ are defined as

$$\delta_p(t) = |\mathbf{p}_j(t) - \mathbf{p}_c(t)|,$$

$$\delta_R(t) = |\text{angle part of } R_c(t)^{-1} \circ R_j(t)|.$$

- *Point interpolation method*

Consider an arbitrary motion segment, $F_0 \rightarrow F_1$, for which we have specified the maximum acceptable deviations:

$$\delta_p \leq \delta_p^{max}, \qquad \delta_R \leq \delta_R^{max}.$$

We must specify enough intermediate points along the Cartesian path so that the path deviations introduced by straight line interpolation between the corresponding joint solutions stay within these bounds. The generation of an "optimal" set of intermediate points requires a good characterization of the path deviation functions δ_p^{max} and δ_R^{max}. These functions depend on the particular manipulator being used and can be quite complicated. For many

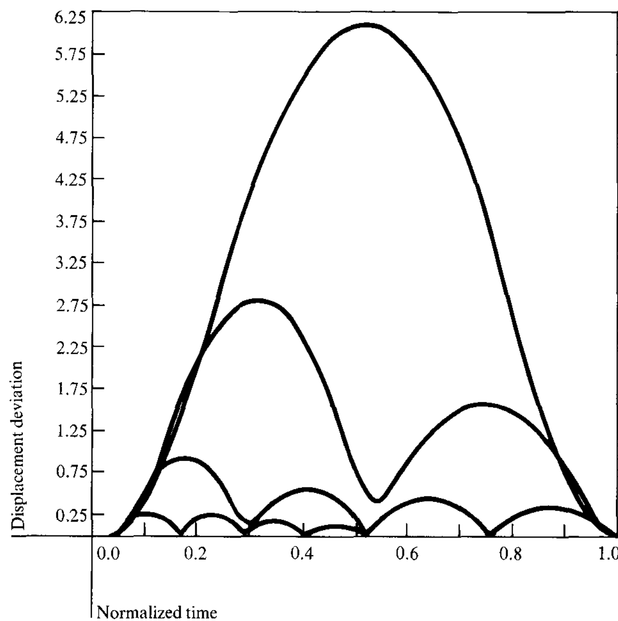

Figure 4 Displacement error for joint space execution of typical segment with $\delta_p^{max} = 0.5$ cm, $\delta_R^{max} = 5°$, and point interpolation recursion limited to 0, 1, 2, and 3 levels.

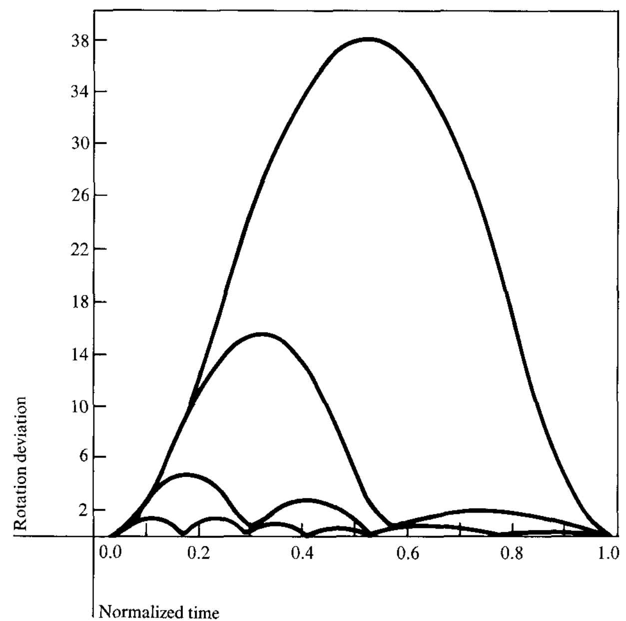

Figure 5 Rotation error for joint space execution of typical segment with $\delta_p^{max} = 0.5$ cm, $\delta_R^{max} = 5°$, and point interpolation recursion limited to 0, 1, 2, and 3 levels.

common manipulator geometries, however, the maximum deviations occur at or near the segment midpoint. This property provides the basis for the following simple interpolation algorithm, which converges rapidly to produce a good, though not necessarily minimal, set of intermediate points.

Step 1 Compute the joint parameters $\mathbf{j}_0$ and $\mathbf{j}_1$ corresponding to F_0 and F_1, respectively.

Step 2 Compute the joint space midpoint,

$$\mathbf{j}_m = \mathbf{j}_1 - \frac{1}{2}\Delta\mathbf{j}_1.$$

Use $\mathbf{j}_m$ to compute

F_m = frame corresponding to joint values $\mathbf{j}_m$.

Also, compute the Cartesian path midpoint F_x,

$$\mathbf{p}_x = \frac{\mathbf{p}_0 + \mathbf{p}_1}{2}, \qquad R_x = R_1 \circ Rot\left(\mathbf{n}, \frac{-\theta}{2}\right),$$

where

$$Rot(\mathbf{n}, \theta) = R_0^{-1} \circ R_1.$$

Step 3 Compute the deviation between F_m and F_x,

$$\delta_p = |\mathbf{p}_m - \mathbf{p}_x|,$$

$$\delta_R = |\text{angle part of } R_x^{-1} \circ R_m|.$$

Step 4 If $\delta_p \le \delta_p^{max}$ and $\delta_R \le \delta_R^{max}$, then we are done. Otherwise, compute the joint solution $\mathbf{j}_x$ corresponding to the Cartesian space midpoint F_x, and apply steps 2–4 recursively for the two segments $F_0 \rightarrow F_x$ and $F_x \rightarrow F_1$.

The application of this algorithm to the motion segment shown in Fig. 2 is illustrated in Figs. 3 through 5, which show the effects of limiting the recursion to zero, one, two, and three levels. Similar results were obtained for other motions of this test manipulator and for a number of other manipulator geometries, including those shown in Fig. 6.

Convergence of the method is quite rapid, with the maximum deviation usually being reduced by approximately a factor of four for each level of midpoint interpolation. The reason for this may be understood, informally, by considering the motion of the simple manipulator shown schematically in Fig. 7, which has one revolute and two translational joints. Suppose that we move this manipulator from position A to position B, as shown in the figure. The displacement deviation at the midpoint of this motion is given by

$$\delta_p = r \cdot [1 - \cos(\theta)].$$

If we invent an intermediate target point here, then the deviation at the midpoint of each of the new segments is

$$\delta_p = r \cdot \left[1 - \cos\left(\frac{\theta}{2}\right)\right].$$

For $\theta = 180°$, *i.e.*, for a complete 360° turn, the deviation is reduced by a factor of two. For the more common cases where $\theta \leq 90°$, the reduction ratio is at worst about 1:3.4 and rapidly approaches 1:4 as θ gets small.

Where there are several revolute joints, the situation is considerably more complicated. However, so long as degenerate joint configurations are avoided, the algorithm will behave much the same for most common manipulator geometries.

• *Degeneracies, alternate solutions, and other refinements*

Convergence of the interpolation algorithm may be rather slow near degenerate points, since a small error in the Cartesian space midpoint may lead to large variations in the joint angles and, consequently, in the manipulator's behavior on the new joint space segments.

Techniques for improving the algorithm's behavior near such points depend somewhat on the kinematics of the particular manipulator in question. However, several "geometry independent" strategies seem generally applicable. These techniques are discussed below.

Degenerate joint solutions occur when two (or more) axes of the machine are lined up, thus destroying a degree of freedom for the hand. In such cases, the lined-up joints are under-determined. The behavior of the algorithm may be improved appreciably if the values of the under-determined joints in degenerate midpoint joint solutions are adjusted to correspond as closely as possible to their joint space midpoint values.

For instance, if the angle of joint 5 of the manipulator shown in Fig. 1 is 0, then joints 4 and 6 may be modified as follows without changing the position of the hand:

$$\mathbf{j}(4)' = \mathbf{j}(4) + \nu,$$

$$\mathbf{j}(6)' = \mathbf{j}(6) - \nu.$$

More generally, if the value of joint 5 is some small angle β, the orientation error introduced by making the above transformation of joints 4 and 6 is given by

$$\beta \text{ sqrt } (2 - 2 \cos \nu).$$

This result may be turned around to tell us how large a correction to make near a "degenerate" midpoint. First,

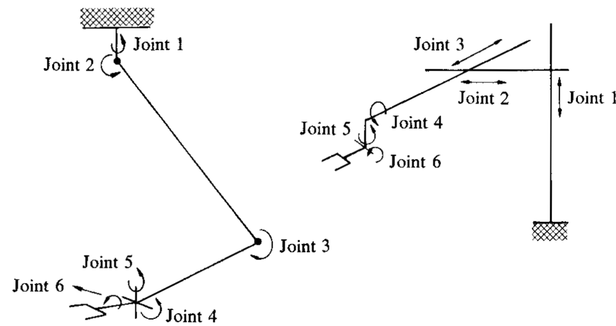

Figure 6 Other manipulator geometries used in tests of the point interpolation algorithm.

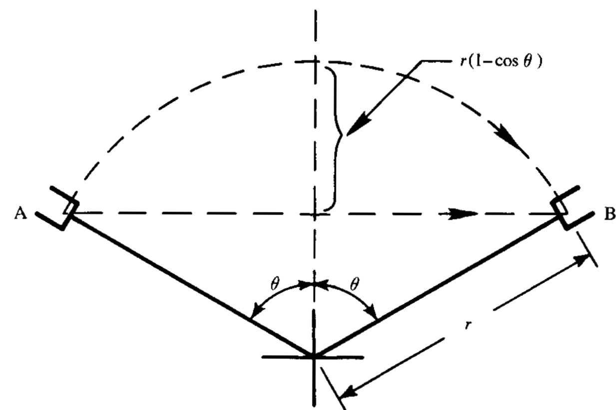

Figure 7 *XYθ* manipulator.

we compute the maximum error that can be tolerated in the hand orientation:

$$\delta = \min\left(\delta_R^{max}, \frac{\delta_p^{max}}{hand\ length}\right).$$

Then, given a joint solution $\mathbf{j}_x$ with a small value for joint 5,

$$\mathbf{j}_x(5) = \beta,$$

compute the maximum correction μ which may be applied to $\mathbf{j}_x(4)$ and $\mathbf{j}_x(6)$ without violating the error bounds and the correction ν required to bring joint 4 to the midpoint value $\mathbf{j}_m(4)$:

$$\mu = \cos^{-1}\left[\max\left(-1, 1 - \frac{\delta^2}{2\beta^2}\right)\right],$$

$$\nu = \mathbf{j}_m(4) - \mathbf{j}_x(4).$$

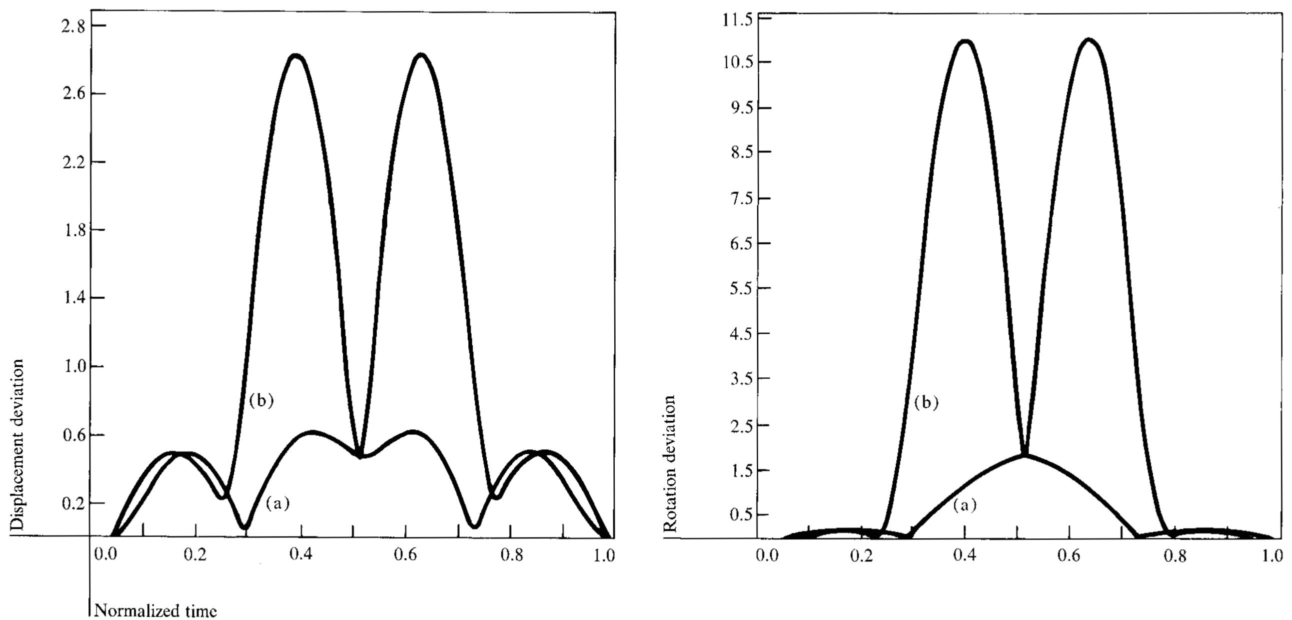

Figure 8 Path deviations for segment whose midpoint is near a degeneracy (a) with and (b) without fixup of nearly degenerate solutions ($\delta_p^{max} = 0.5$ cm, $\delta_R^{max} = 5°$, recursion limit = 2).

The smaller of these two corrections may now be applied to $\mathbf{j}_x$:

$$\mathbf{j}_x(4)' = \mathbf{j}_x(4) + \max[-\mu, \min(\mu, \nu)],$$

$$\mathbf{j}_x(6)' = \mathbf{j}_x(6) - \max[-\mu, \min(\mu, \nu)].$$

The importance of this refinement is illustrated in Fig. 8, which shows the displacement and rotation deviations for a segment whose midpoint is near a degeneracy.

The joint solution for many manipulators may be redundant, in the sense that there may be several sets of joint parameters which will put the hand at a particular point. For instance, the gimbal of our example manipulator has two solutions, characterized by

$$\mathbf{j}(4)' = \mathbf{j}(4) + 180°,$$

$$\mathbf{j}(5)' = -\mathbf{j}(5),$$

$$\mathbf{j}(6)' = \mathbf{j}(6) - 180°.$$

If the "polarities" of the solutions chosen at the end points of a segment do not match properly, the midpoint error may be substantially increased. In general, it is important to consider all possible joint solutions at each knot point and select the one which provides the best behaved trajectory. Figure 9 illustrates how a typical trajectory may be degraded by failure to do this.

Conclusions

This paper has discussed two somewhat different approaches to straight line motions. Each has certain strengths and weaknesses.

The first approach, "Cartesian path control," is conceptually straightforward and lends itself readily to applications where the segment knot points may change while the motion is being made or where perturbations to the motion, such as those required for force accommodation, are conveniently described in Cartesian space. On the other hand, the method is rather expensive computationally and is rather vulnerable to unexpected difficulties where degeneracies or joint limits are encountered.

The second method, "bounded deviation joint paths," relies on a planning phase to interpolate enough intermediate points so that the manipulator may be driven in joint space without deviating more than a prespecified amount from the desired path. This method greatly reduces the amount of computation that must be done at every sample interval and permits joint space constraints to be treated in a natural manner. If the motion planning cannot be done in advance of program execution, then the latency between a MOVE command and the start of the motion may be increased somewhat. Convergence of the interpolation algorithm is sufficiently rapid, however, so

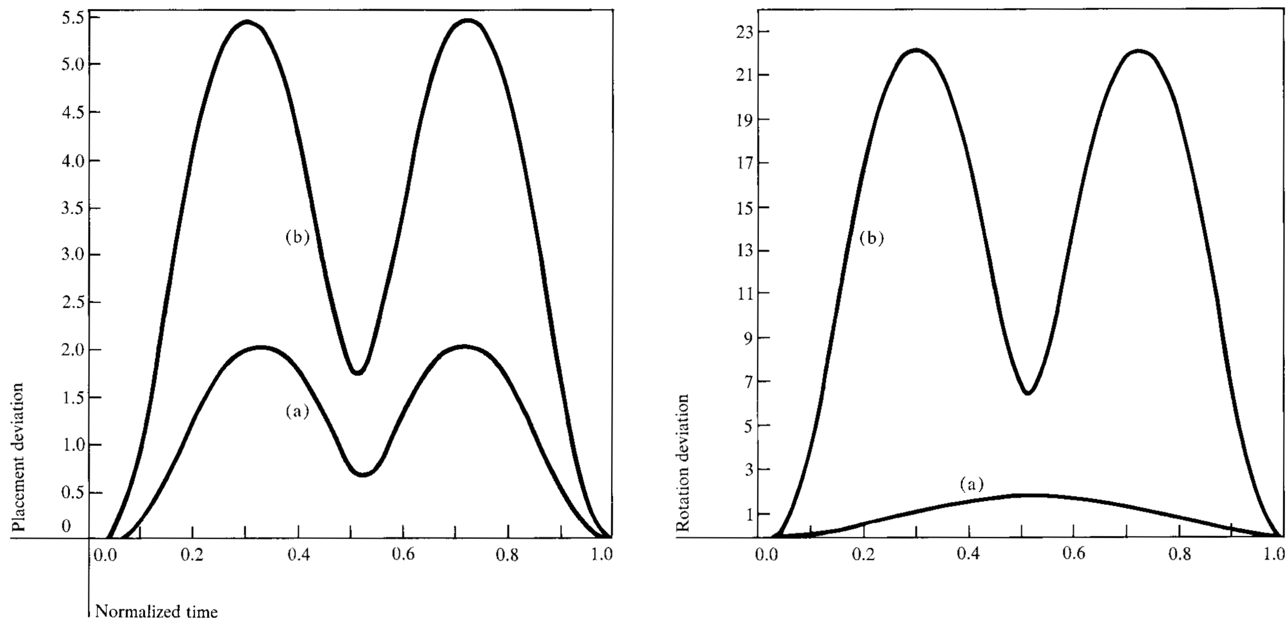

Figure 9 Path deviations for segment whose midpoint is near a degeneracy (a) with and (b) without checking for alternate solutions (δ_p^{max} = 0.5 cm, δ_R^{max} = 5°, recursion limit = 1).

that the extra time is small, at least for the deviation bounds required for many applications. In cases where shorter latencies are required, it should be possible to start moving along the first part of a trajectory while finishing planning the remainder of the motion.

Acknowledgments
I would like to thank David Grossman and Larry Lieberman for their assistance with the geometric modeling and graphics facilities used to prepare the figures. Also, I am grateful to John Griffith and Peter Will for reading the manuscript and making a number of helpful editorial suggestions.

Appendix A: Quaternions
In general, a quaternion Q consists of a scalar part s and a vector part $\mathbf{v}$ and here is written

$$Q = [s + \mathbf{v}].$$

The multiplication rule is

$$Q_1 \circ Q_2 = [s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2 + s_2\mathbf{v}_1 + s_1\mathbf{v}_2 + \mathbf{v}_1 \times \mathbf{v}_2].$$

From this definition, it follows that if

$$s = \sin\left(\frac{\theta}{2}\right) \text{ and } c = \cos\left(\frac{\theta}{2}\right),$$

then

$$[0 + Rot(\mathbf{n}, \theta) \circ \mathbf{u}] = [c + s \cdot \mathbf{n}] \circ [0 + \mathbf{u}] \circ [c + -s \cdot \mathbf{n}].$$

In other words, we can represent a rotation $Rot(\mathbf{n}, \theta)$ by a quaternion

$$\left[\cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right) \cdot \mathbf{n}\right].$$

The "null" rotation would be given by

$$Rot(\mathbf{n}, 0) = [1 + 0 \cdot \mathbf{n}] = [1 + \mathbf{0}].$$

Notice that if a rotation R is represented by a quaternion Q, then the quaternion corresponding to R^{-1} can be obtained trivially by negating the vector part of Q. Similarly, it is easy to see that if rotations R_1 and R_2 are represented by quaternions Q_1 and Q_2, respectively, then the rotation $R_1 \circ R_2$ will be represented by the quaternion $Q_1 \circ Q_2$.

The computational requirements of some common operations involving rotations, using quaternion and matrix representations, are given in Table 2. In addition to requiring generally fewer operations, quaternions have the advantage of being a less redundant representation than 3 × 3 rotation matrices, thus simplifying problems associated with machine roundoff errors.

Table 2 Computational requirements of common rotation operations.

Operation	*Quaternion representation*	*Matrix representation*
$R_1 \circ R_2$	9 adds, 16 multiplies	15 adds, 24 multiplies
$R \circ v$	12 adds, 22 multiplies	6 adds, 9 multiplies
$R \rightarrow (\mathbf{n}, \theta)$	4 multiplies, 1 square root, 1 arctangent	8 adds, 10 multiplies, 2 square roots, 1 arctangent
$(\mathbf{n}, \theta) \rightarrow R$	4 multiplies, 1 sine-cosine pair	10 adds, 15 multiplies, 1 sine-cosine pair
Renormalize	3 adds, 8 multiplies, 1 square root	7 adds, 18 multiplies 2 square roots
Convert to other representation	19 adds, 9 multiplies	7 adds, 5 multiplies, 3 sine-cosine pairs, 1 arctangent

References and notes

1. T. Binford *et al., Exploratory Studies of Computer Integrated Assembly Systems*, NSF Progress Reports, Stanford Artificial Intelligence Laboratory, Stanford University, Stanford, CA, September 1974, November 1975, July 1976 (AIM 285).
2. C. Rosen *et al., Machine Intelligence Research Applied to Industrial Automation*, NSF Report, Grant APR75-13074, Stanford Research Institute, Stanford, CA, November 1976.
3. P. M. Will and D. D. Grossman, "An Experimental System for Computer Controlled Mechanical Assembly," *IEEE Trans. Computers* **C-24,** No. 9, 879–888 (1975).
4. R. Finkel, R. Taylor, R. Bolles, R. Paul, and J. Feldman, *AL, A Programming System for Automation*, Stanford Artificial Intelligence Laboratory Memo AIM-177, Stanford Computer Science Report STAN-CS-74-456, Stanford University, Stanford, CA, November 1974.
5. R. H. Taylor, "The Synthesis of Manipulator Control Programs from Task-Level Specifications," Ph.D. Thesis, Stanford Artificial Intelligence Laboratory Memo AIM-282, Stanford Computer Science Report STAN-CS-76-560, Stanford University, Stanford, CA, July 1976.
6. R. Paul, "Modelling, Trajectory Calculation, and Servoing of a Computer Controlled Arm," Ph.D. Thesis, Stanford Artificial Intelligence Laboratory Memo AIM-177, Stanford Computer Science Report STAN-CS-72-311, Stanford University, Stanford, CA, November 1972.
7. L. I. Lieberman and M. A. Wesley, "AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly," *IBM J. Res. Develop.* **21**, No. 4, 321–333 (1977).
8. D. Whitney, "Resolved Motion Rate Control of Manipulators and Human Prostheses," *IEEE Trans. Man Mach. Syst.* **MMS-10,** 47–53 (1969).
9. Richard Paul, "Manipulator Path Control," *Proc. 1975 International Conference on Cybernetics and Society*, IEEE, New York, September 1975.
10. M. Beeler, R. W. Gosper, and R. Schroeppel, *Hakmem*, MIT Artificial Intelligence Laboratory Memo No. 239, Massachusetts Institute of Technology, Cambridge, MA, February 1972.
11. W. R. Hamilton, *Elements of Quaternions*, Third Edition, Chelsea Publishing Co., New York, 1969.
12. A. T. Yang and F. Freudenstein, "Application of Dual-Number Quaternion Algebra to the Analysis of Spatial Mechanisms," *J. Appl. Mech., Trans. ASME* **86,** 300–308 (1964).
13. C. Rosen *et al., Exploratory Research in Advanced Automation*, NSF Report, Grant GI-38100X1, Stanford Research Institute, Stanford, CA, January 1976.
14. B. Shimano, "Improvements in the AL Runtime System," in *Exploratory Study of Computer Integrated Assembly Systems*, Stanford Artificial Intelligence Laboratory Memo AIM-285, Stanford University, Stanford, CA, August 1976.
15. Revolute joints introduce a number of minor complications into the definition of $\Delta\mathbf{j}$, especially where the joint limits allow complete revolutions. In the examples given with this section, $\Delta\mathbf{j}$ was chosen so as to cause each joint to move through the smallest of two possible placements. For instance, if $\theta_0 = 30^\circ$ and $\theta_1 = 330^\circ$, then $\Delta\theta = -60^\circ$.
16. R. H. Taylor, "Planning and Execution of Straight-Line Manipulator Trajectories," *Research Report RC6657*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1977.

Received July 10, 1978; revised March 1, 1979

The author is located at the IBM Information Records Division laboratory, P.O. Box 1328, Boca Raton, Florida 33432.