Real-Time Trajectory Generation in Multi-RCCL

John Lloyd and Vincent Hayward
McGill Research Centre for Intelligent Machines
McGill University
Montréal, Québec H3A 2A7, Canada

Received April, 1992; accepted May 1992

This article describes the design of the trajectory generator for a robot programming system called Multi-RCCL, which is a package of "C" routines for doing real-time manipulator control in a UNIX environment. RCCL has been used successfully in developing robot control applications in numerous research and industry facilities over the last several years. One of its strongest features is the ability to integrate real-time sensor control into the manipulator task specification. RCCL primitives supply the trajectory generator with target points for motions in joint or Cartesian coordinates. Other primitives allow the code developer to specify on-line functions that can modify the target points, or possibly cancel motion requests, in response to various sensor or control inputs. The design requirements of the trajectory generator are that it be able to integrate these on-line modifications into the overall robot motion and provide a smooth path between adjacent motions even when sensor inputs make the future trajectory uncertain.

この発表では、UNIX環境でリアルタイムのマニピュレータ制御をおこなうC言語ルーチンのパッケージであるMulti-RCCLと呼ばれるロボット・プログラミング・システム用の軌道ジェネレータの設計について説明する。過去数年間に渡って、数多くの研究や工業用設備でのロボット制御アプリケーションの開発に、RCCLは使われてきた。最も強力な機能の一つは、リアルタイム・センサー制御をマニピュレータ・タスクの仕様に統合化したことである。その他のプリミティブによって、コード・デベロッパは、さまざまなセンサーや制御入力における応答として、ターゲット・ポイントの変更やモーション・リクエストのキャンセルができるオンライン関数を指定できる。これらのオンラインでの変更をロボットのすべての動きに反映し、センサー入力から生成される将来の不確定な軌道において隣接するモーション間のスムースなパスを生成することが、軌道ジェネレータの設計では要求される。

1. INTRODUCTION

Multi-RCCL is a library of "C" routines and data structures for writing robot control programs. It is an extension to the original RCCL (Robot Control C Library), which was written at Purdue University by Vincent Hayward.[1] Since then, various enhancements have been made to the system, including the abil-

Journal of Robotic Systems 10(3), 369–390 (1993)
 CCC 0741-2223/93/030369-022

ity to control multiple robots and distribute the trajectory control over several CPUs. Most of this work has been done at McGill University and at the Jet Propulsion Laboratory.[2-4]

RCCL's two fundamental features are:

- A comprehensive set of C language primitives for specifying robot motions and actions in a UNIX workstation environment.
- The ability to write application code that processes sensory information and adjusts the robot trajectory in real-time.

These two features have made RCCL a particularly useful tool for developing applications where manipulator actions are controlled by various sensor and control inputs. Because the system provides direct on-line control of the manipulator, special techniques have been used to obtain the required real-time performance from conventional UNIX workstations.[5]

Principal RCCL sites include the Jet Propulsion Laboratory, the General Electric Advanced Technology Laboratory (New Jersey), Columbia University, the University of Pennsylvania, the University of Southern California, NASA Goddard Space Flight Center, the University of Illinois, Universität Bielefeld (Germany), and several others. Specific applications for which it has been used include manipulator force control and telerobotics,[6,7] multiarm force control,[8] visual servoing and tracking,[9,10] active visual exploration,[11] and the development of robot programming interfaces.[12]

The system's trajectory generator, and its ability to integrate sensor inputs with preplanned actions, has thus been thoroughly demonstrated over the last couple of years. The purpose of this article is to present the design of the trajectory generator in detail.

It should be noted that responding to dynamic events requires that virtually all of the trajectory generation be done directly on-line in a manner that precludes some of the more optimal solutions that are possible if the trajectory is planned in advance.[13,14]

The reader is assumed to be familiar with the original book by Richard Paul[15] and all the notational conventions used there. For purposes of clarity, we have omitted describing certain details and special cases.

2. OVERVIEW

RCCL generates the trajectories necessary to move a manipulator through sequences of *target positions*, which are submitted to a trajectory generator for asynchronous execution. Target positions can be specified in either Cartesian or joint coordinates. This article will focus only on the former; joint target specifications can be handled in the same way. A Cartesian target is described by a *position equation*, which is a closed kinematic loop of 4×4 homogeneous transforms containing the manipulator's **T6** transform. The component transforms of such target equations may be time varying, and motions to such

targets will *track* the variations. For example, suppose we wish to have a manipulator pick up an object on a moving platform. The application program could define the following target equation:

$$\mathbf{Z}\ \mathbf{T6}(t)\ \mathbf{E} = \mathbf{W}\ \mathbf{P}(t) \tag{1}$$

and motions to this target will cause the manipulator to travel to a position where **T6** satisfies the equation (see Fig. 1). In this case, $\mathbf{P}(t)$ is time varying, and **T6** will track the changes. A principal means by which the application program may vary a component transform is to attach it to a function that is evaluated once every control cycle by the trajectory generator; such a function may modify the transform based on either some preprogrammed plan or sensor inputs. "Stopping" at a time-varying target implies that the manipulator will reach the target but continue to track its motions.

The fact that the target specification is dynamic makes the system quite versatile but means that virtually all of the trajectory computation must be done on-line. This is, in fact, the greatest constraint imposed on everything that we will discuss.

Internally, the trajectory generator divides the target position equation into a **TOOL** term and a **COORD** term: **TOOL** is the set of transforms between **T6** and whatever frame the application has designated as the *tool frame*, and **COORD** is "other half" of the equation. In general, these terms will be time varying if their components are. Assuming in the above example that the tool frame is designated by **E**, then RCCL would represent (1) internally as

$$\mathbf{T6}(t)\ \mathbf{TOOL} = \mathbf{COORD}(t)$$

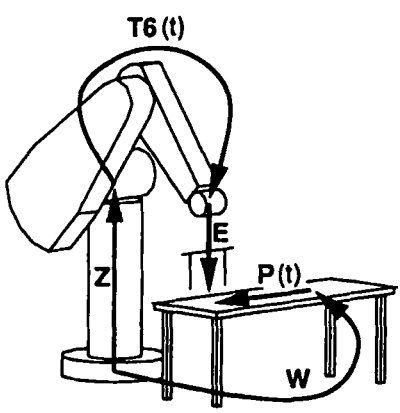

Figure 1. Kinematic loop for a manipulator target attached to a moving platform.

where

$$\mathbf{TOOL} = \mathbf{E} \tag{2}$$

$$\mathbf{COORD}(t) = \mathbf{Z}^{-1}\,\mathbf{W}\,\mathbf{P}(t) \tag{3}$$

This partioning is used for performing the *drive* computation (discussed alter).

The job of the trajectory generator is to compute paths between successive (possible time varying) target points. The application can specify whether this path is to be computed in Cartesian or joint coordinates and imposes velocity and acceleration constraints in the selected coordinate system. The velocity constraint determines the overall time of the path segment (σ), while the acceleration constraint determines the length of time (τ) spent doing a smooth *transition* between adjacent path segments.

Figure 2 shows (in one dimension) path segments connecting an initial point A, a via point B, and a stopping point C. The σ and τ values are shown for the path segment $\overline{BC}$. Notice that τ is actually one half the total transition time, which is symmetrical about the path segment endpoints. The action of "stopping" at C is also computed as a path segment (in effect another motion request to the same target) whose σ is specified explicitly. This technique is particularly useful because the target position we "stop" at might actually be in motion. Finally, notice that unless we actually do stop at a target point, the transitions are computed to "undershoot" it.

The trajectory computations are done in real-time at a fixed sample rate. The target position (i.e., the **COORD** and **TOOL** terms) is re-evaluated once every control cycle; this action will include the evaluation of any functions bound to the component transforms.

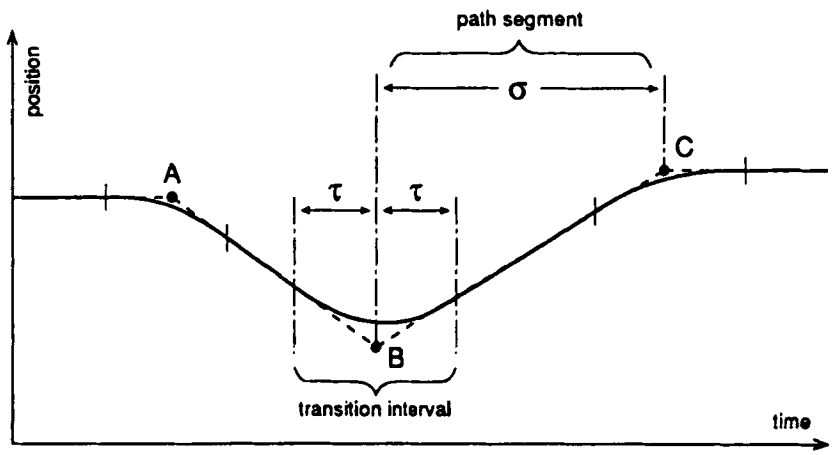

Figure 2. Path segments between target positions.

For convenience, this article will use a time scale where the start time of the path segment under discussion is given by t_b and its end time is t_c (such that $\sigma = t_c - t_b$). Moreover, the time corresponding to one control cycle will be defined as unity, and the time scale will be offset so that $t_b - \tau = 1$. Under these coordinates, τ and σ are equal to their respective number of control cycles.

As stated above, the path interpolation can be done in either Cartesian or joint coordinates. The latter is accomplished using a *drive* transform constructed in the manipulator tool frame. Individual path points are then computed from the equations

$$\mathbf{T6}(t) = \mathbf{COORD}(t)\ \mathbf{DRIVE}(t)\ \mathbf{TOOL}(t)^{-1}$$

$$\mathbf{j6}(t) = \wedge^{-1}(\mathbf{T6}(t))$$

where $\wedge^{-1}()$ indicates an inverse robot kinematics operation. At time t_b, the drive transform accommodates the initial difference between the current position and the target. Over the course of the path segment, **DRIVE**(t) is computed so as to trace a "straight line" from its initial value to the identity **I**. To do this, a function *trsfToDrive()* is used to convert **DRIVE** into a set of *drive parameters* (**dp**) that can be linearly scaled. Given a function *driveToTrsf()* that converts drive parameters back into transforms, and a normalized *motion scale* coordinate s, defined as

$$s = \frac{(t - t_b)}{\sigma}$$

the subsequent values of **DRIVE**(t) can be computed from

$$\mathbf{DRIVE}(t) = \mathit{driveToTrsf}\,((1 - s)\mathbf{dp})$$

The mechanics of the drive parameters are described in chap. 5 of ref. 15.

Computing the path in joint coordinates is similar except that in this case the drive interpolation is done in joint space. The corresponding computation is

$$\mathbf{TC}(t) = \mathbf{COORD}(t)\ \mathbf{TOOL}(t)^{-1}$$

$$\mathbf{j6}(t) = \wedge^{-1}(\mathbf{TC}(t)) + \mathbf{jdrive}(t)$$

where **TC** indicates the value of the motion target in the T6 frame. Instead of using a Cartesian **DRIVE** transform, we use a vector of joint coordinates **jdrive**. The drive parameters **jdp** are simply the initial values of **jdrive**, which means that

$$\mathbf{jdrive}(t) = (1 - s)\mathbf{jdp}$$

Notice that the path will still track time variations in the Cartesian target position.

Following from the above discussion, the rest of this article will describe, in general, how Multi-RCCL (1) computes transitions between path segments, (2) determines the drive parameters, and (3) computes σ and τ, under conditions where the target position may be changing. A detailed description of the specific Joint and Cartesian path generation algorithms is also included.

3. TRANSITIONS BETWEEN PATH SEGMENTS

When we abut adjacent path segments, we must generally allow a transition time τ during which one path segment can be smoothly blended into another. The magnitude of τ reflects the maximum amount of acceleration that can be tolerated by the manipulator; a longer value of τ is necessary for path changes involving larger velocity changes. The τ for any motion is computed dynamically (Section 5.1).

The path segment blending algorithm used by RCCL is quite independent of the other parts of the trajectory calculation, and so we discuss it first. For illustration, we will consider the blending of two 1-D path segments $S_0(t)$ and $S_1(t)$. Each path segment is the combined result of a *drive* toward a target plus variations in the target itself.

To smoothly connect the path segments, a blend function β (see Fig. 3) is applied to $S_1(t)$ over the interval $1 \leq t \leq t_b + \tau$ (recall that $t_b - \tau = 1$). The

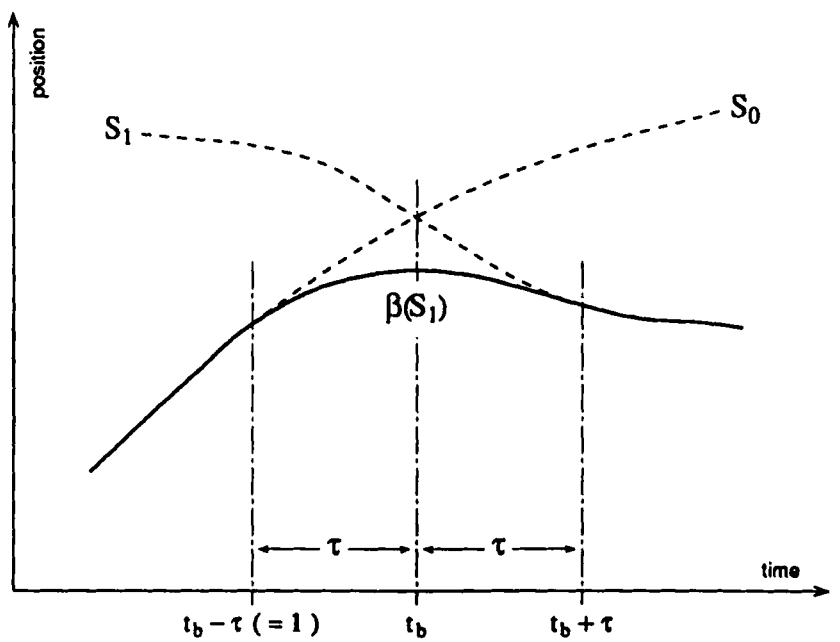

Figure 3. Blending two path segments together.

implication of this is that the computation for a path segment actually starts at time $t_b - \tau$ *before* the time t_b when it officially starts.

All the parameters needed to compute the blend function can be determined from the initial position, velocity, and acceleration of S_0 and S_1 at $t = 1$. Assume for the moment that the blend function can be applied simply by adding it to S_1 (although this is not true for rotations; see Section 3.2). It must then satisfy the following constraints:

$$\begin{aligned} S_1(1) + \beta(1) &= S_0(1) \\ S_1(1)' + \beta(1)' &= S_0(1)' \\ S_1(1)'' + \beta(1)'' &= S_0(1)'' \\ \beta(t_b + \tau) &= 0 \\ \beta(t_b + \tau)' &= 0 \\ \beta(t_b + \tau)'' &= 0 \end{aligned} \tag{4}$$

These can be satisfied by computing the blend function with a fifth-order Hermite polynominal:

$$\beta(h) = \alpha_5 h^5 + \alpha_4 h^4 + \alpha_3 h^3 + \alpha_2 h^2 + \alpha_1 h + \alpha_0 \tag{5}$$

where h is a time parameter normalized to span the interval [0,1] during the transition:

$$h = \frac{t - t_b + \tau}{2\tau}$$

Defining the difference between S_1 and S_2 to be $\delta(t) \equiv S_1(t) - S_2(t)$, the coefficients of (5) that satisfy (4) are

$$\begin{aligned} \alpha_0 &= \delta(0) \\ \alpha_1 &= \delta'(0) \\ \alpha_2 &= 0.5\ \delta''(0) \\ \alpha_3 &= -10\ \delta(0) - 6\ \delta'(0) - 1.5\ \delta''(0) \\ \alpha_4 &= 15\ \delta(0) + 8\ \delta'(0) + 1.5\ \delta''(0) \\ \alpha_5 &= -6\ \delta(0) - 3\ \delta'(0) - 0.5\ \delta''(0) \end{aligned}$$

Joint space paths, and the translational components of Cartesian paths, can be blended by the direct extension of this formulation to several dimensions. The blending of rotational trajectories, which is more complicated because rotations do not commute and therefore cannot be treated as vector quantities, is discussed in Section 3.2.

The blend function is actually just the "**DRIVE** transform" revisited, only here discontinuities are being removed in the derivative terms. We could dispose of the drive computation all together and simply make the transition interval cover the entire path segment. We do not do this, however, because applications often desire constant-velocity straight-line trajectories.

Notice that in Figure 3 S_0 and S_1 intersect at t_b. Generally, the trajectory generator sequences events so this is in fact the case, yielding an "undershoot" transition (of the sort described in ref. 15) that goes through the inside of the intersection point. However, the paths do not have to intersect at t_b; different blending behaviors will result depending on where (or even if) they intersect. Variations on the blend function described here have been suggested that allow the application to "tune" the extent of the via-point undershoot or specify an overshoot.[16]

The principal advantage of the blend technique is that it is robust: We do not need to know anything about the path S_0 after time $t = 1$ and we do not need to know anything about path S_1 in advance. S_0 and S_1 do not even have to be straight lines, as illustrated in the figure. This is important because the target may be time-varying and sensor-driven and hence not completely predictable.

3.1. Initializing the Blend Parameters

To initialize the blend function, it is necessary to know the values of S_0 and S_1 and their derivatives at time $t = 1$.

For computational reasons, RCCL does not presently worry about any initial difference in acceleration between the path segments: For purposes of computing the blend function, it is assumed that $S_0''(1) - S_1''(1) = 0$. We therefore need only $S_0(1)$, $S_0'(1)$, $S_1(1)$, and $S_1'(1)$. In general, however, we do not have $S_1'(1)$ because we have only started computing $S_1(t)$ during the first control cycle. To accommodate this, the trajectory generator waits for one more control cycle and then computes $S_1(2)$, from which $S_1'(1)$ may be estimated as $S_1'(1) = S_1(2) - S_1(1)$. The output setpoints for the first control cycle are simply set equal to $S_0(1)$.

Two interface functions are defined that compute the required derivatives and blend parameters from the values of S_0 and S_1 at times 1 and 2:

$$setBlend0(S_0(0),\ S_1(0))$$

$$setBlend1(S_0(1),\ S_1(1),\ 2\tau)$$

The third argument to *setBlend1()* gives the total length of the transition time, which is needed to compute h.

Because the blend function cannot always be applied by simple addition, its application to S_1 will be indicated in general as $\beta(S_1(t))$.

The computation of $S_0(1)$ and $S_0(2)$ is currently done by extrapolating the initial path:

$$S_0(1) = S_0(0) + S_0'(0)$$

$$S_0(2) = S_0(0) + 2S_0'(0)$$

There are two reasons for doing this instead of computing $S_0(t)$ directly. The first is simply to save computation. The second is that we do not wish (under the present implementation) to compute S_0 at the same time we are computing S_1 because some of the transforms that define the first target may also be used in the definition of the second target. If any of these transforms are bound to an application-defined function, then that function would have to be called twice during the same control cycle in two different motion contexts. Resolving this properly would require adjustments to the present application interface. The small errors introduced by extrapolating S_1 have not caused any difficulties to date.

3.2. Blending Rotations

Analogous to the previous discussion, we can consider the problem of blending rotations as computing a smooth transition between two rotational paths **R0**(h) and **R1**(h) over a normalized time interval [0,1]. Unlike the treatment above, however, we cannot do this using a simple Hermite polynominal because there is no vector-like representation for rotations that defines a proper metric for them.*

One obvious solution to this problem is to find a space that **does** form a metric for rotations and do the blending there. For distance, the unit sphere in $\mathcal{R}^4$ formed by the set of unit quaternions forms a metric for rotations and it is possible to construct curves on the surface of this sphere to smoothly connect different rotation states.[17] Unfortunately, this technique is too expensive (computationally) to be of use for one-line calculation.

Instead, we use a method that has some similarities to the work described in ref. 18. To begin with, the computation has been simplified by ignoring initial rotational accelerations. This has not caused any noticeable problems, and acceleration compensation could be added, if necessary, as a simple extension of the following method.

The blend function β takes the form of a blend rotation **RB**(h) that is applied to **R1**(h) over the transition interval, producing the output rotation **R**(h):

$$\mathbf{R}(h) = \mathbf{R1}(h)\ \mathbf{RB}(h)$$

*Although vector blending can work if the rotational distance between R0 and R1 is not large; for instance, interpolation of quaternions works quite nicely for displacements less than 45°.

Let the rotational velocities of **R0**, **R1**, and *RB* be given by Ω_0, Ω_1, and Ω_B. We then have the following boundary conditions:

$$\mathbf{RB}(0) = \mathbf{R1}(0)^{-1}\,\mathbf{R0}(0)$$

$$\mathbf{RB}(1) = \mathbf{I}$$

$${}^{R0(0)}\Omega_B(0) = {}^{R0(0)}\Omega_0(0) - {}^{R0(0)}\Omega_1(0)$$

$$\Omega_B(1) = 0$$

The velocities in the third equation are described with respect to the **R0**(0) coordinate frame.

Looking at the boundary conditions, we see that what we essentially want to do is remove the initial displacement and velocity of **RB**. The initial displacement **RB**(0) can be expressed as a single rotation about some axis. If this axis is parallel to $\Omega_B(0)$, then the whole problem reduces to a one-dimensional one in terms of the rotation parameter about that axis. While this is not in general true, it suggests resolving **RB** into separate rotations that compensate for velocity and displacement. Let the notation $rot(\theta, \mathbf{u})$ denote a rotation of θ about the axis **u**. Now, let $\omega_0 = \|\Omega_B(0)\|$, let u_v be a unit vector parallel to $\Omega_B(0)$, and define **RB** as the following product:

$$\mathbf{RB}(h) = \mathbf{R1}(0)^{-1}\,\mathbf{R0}(0)\,\mathbf{RC}(h)\,\mathbf{RV}(h)$$

The first two (constant) terms compensate for the initial displacement. The **RV**(h) term, which is initially equal to **I**, acts to bring $\Omega_B(0)$ to 0:

$$\mathbf{RV}(h) = rot(\omega_0\ g_v(h)/2,\ \mathbf{u}_v)$$

where g_v is the polynomial

$$g_v = (h^3 - 2h^2 + 2)h$$

defined so that **RV** satisfies the boundary conditions that require it to have an initial velocity of ω_0 and a final velocity of 0, and 0 acceleration at both ends.

Finally, the **RC**(h) term brings all displacements to 0, including those incurred by **RV**. Because **RB(1)** = **I**, we have that

$$\mathbf{RC}(1) = \mathbf{R0}(0)^{-1}\,\mathbf{R1}(0)\,\mathbf{RV}(1)^{-1}$$

Letting $\mathbf{u}_c$ be the axis of rotation for **RC**(1) and θ_c be the angle of rotation, **RC**(h) is computed from

$$\mathbf{RC}(h) = rot(\theta_c\ g_c(h),\ \mathbf{u}_c)$$

where g_c is the polynomial

$$g_c = (6h^2 - 15h + 10)h^3$$

which gives **RC** a velocity and acceleration of 0 at the transition endpoints.

4. COMPUTING THE DRIVE PARAMETERS

In this section, we continue to use the one-dimensional paths S_0 and S_1 to illustrate how drive parameters are computed, first for fixed targets and then for time-varying targets.

4.1. Fixed Targets

Refer to Figure 4. When the computation begins for path S_1 *at* $t = 1$, the manipulator is following path S_0 is on its way to some point B. If C_0 is the official target point for path S_0, and is constant, then $B = C_0$. If the $C_0(t)$ is not constant, then $B = C_0(t_b)$, which creates a minor problem because $C_0(t_b)$ is generally not known ahead of time. In this case, B can be estimated by extrapolation:

$$\hat{B} = S_0(0) + (\tau + 1)S_0'(0) \tag{6}$$

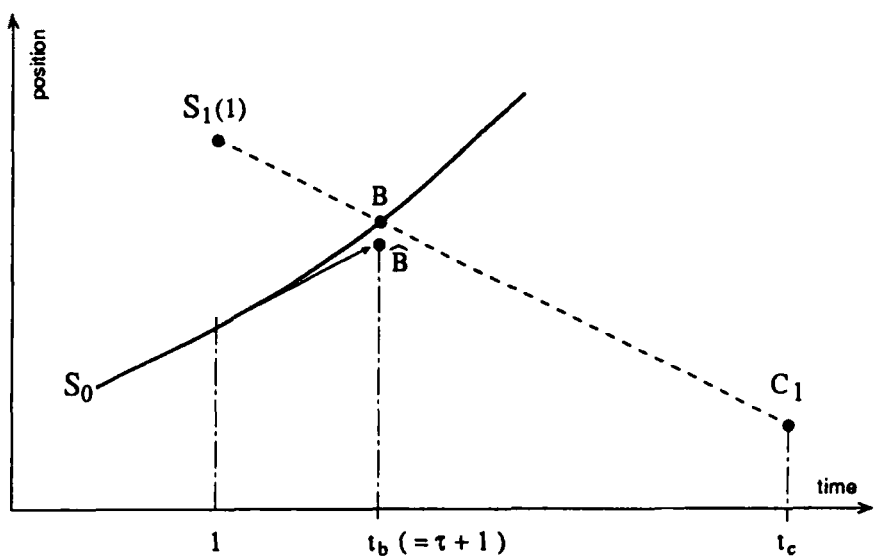

Figure 4. Computing the drive parameter for fixed targets.

Another instance where it becomes necessary to do this is when the motion along S_0 is interrupted prematurely, before it has had time to approach the official target.

Assume that the (constant) target for the next path segment is C_1. The drive parameter D for this motion is then simply

$$D = \hat{B} - C_1 \tag{7}$$

and S_1 can be computed by

$$S_1(t) = C_1 + (1 - s)D \tag{8}$$

4.2. Time-Varying Targets

If $C_1(t)$ is time varying, then there is another minor problem deciding how to compute the drive parameter. During the first cycle, the drive offset $O(1)$ can be computed from the initial target value as

$$O(1) = \hat{B} - C_1(1)$$

But, repeating the computation during the second cycle yields

$$O(2) = \hat{B} - C_1(2)$$

which may be different. What we really want is the drive offset for time t_b (see Fig. 5):

$$O(t_b) = \hat{B} - C_1(t_b)$$

This yields an equation for S_1 that looks like

$$S_1(t) = C_1(t) + (1 - s)\,(\hat{B} - C_1(t_b)) \tag{9}$$

Substituting in for time t_b, we notice that

$$S_1(t_b) = \hat{B}$$

which is correct.

While $O(t_b)$ cannot be known with certainty ahead of time, if it is assumed that $C_1(t)$ is roughly linear in the transition region then $O(t_b)$ can be estimated by extrapolation:

$$D \equiv \hat{O}(t_b) = O(1) + \tau(O(2) - O(1))$$

This requires postponing the computation of D until the second control cycle, but we already have to "coast" for the first cycle anyway for purposes of calculating the blend parameters.

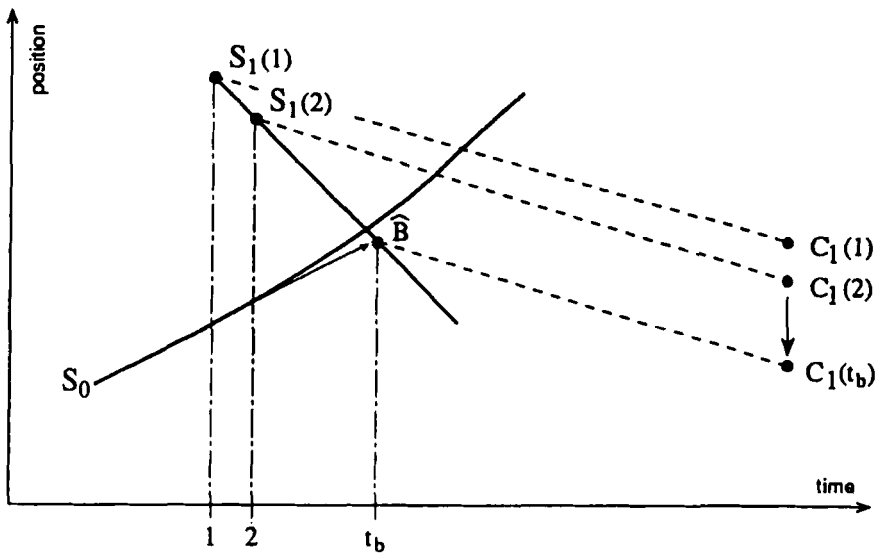

Figure 5. Computing the drive parameter for moving targets.

Once D has been computed, it is necessary to compute $S_1(2)$ and "backcompute" $S_1(1)$ to initialize the blend parameters. This can be done using (9), which yields, with the $1 - s$ term expanded,

$$S_1(1) = C_1(1) + \left(\frac{t_c - 1}{\sigma}\right) D$$

$$S_1(2) = C_1(2) + \left(\frac{t_c - 2}{\sigma}\right) D$$

4.3. Stopping

The path generation methods given above are versatile and can handle a variety of cases. For instance, if we switch path interpolation modes from the previous cycle then we only have to transform the initial velocity estimate associated with $S_0'(0)$. Because path points are computed in both Cartesian and joint coordinates anyway, this is easy. We can also handle motion interruption easily: If a motion is canceled before it has reached its anticipated target, its "virtual target" $\hat{B}$ can be computed using (6) and the rest of the computation is the same.

There is one particular case that is treated separately: stopping. Recall that this is implemented as a duplicate motion to the same target point. This implies some simplification in computing the trajectory paths. For one thing, there is no need to do a *drive* computation: by definition, the manipulator has already reached its target. It might be necessary to maintain a fixed *drive offset*, how-

ever: In cases in which the preceding motion was interrupted before completion, the most natural option is to stop at the point where the motion was interrupted rather than at a point that was never reached in the first place.

This is done in the following way. During the first computation cycle of the stop segment, the final scale factor s_f associated with the previous motion is determined. If the motion went to completion, then $s_f = 1$; otherwise, $s_f < 1$. The constant term $(1 - s_f)D$ is then used in (9) in place of the time-varying term $(1 - s)D$.

5. COMPUTING THE PATH SEGMENT AND TRANSITION TIMES

This section outlines briefly the calculation of τ and σ. It should be mentioned in passing that although these times are normally computed automatically, as described here, RCCL does permit the application to specify them explicitly. All operations described here are assumed to be normalized to the appropriate units.

5.1. Transition Times

RCCL computes its path segment transition times "on the fly," taking into consideration the current velocity, the distance to the next target, and the specified velocity and acceleration limits. While the method used is only a heuristic, it works reasonably well and is computationally tractable. The computation for Cartesian motions will be described here. The equivalent computation for joint motions is analogous.

Velocity and acceleration constraints for Cartesian motions are specified in terms of a maximum translational "cruising" speed $\bar{v}_t$, a maximum rotational speed $\bar{v}_r$, a maximum translational acceleration $\bar{a}_t$, and a maximum rotational acceleration $\bar{a}_r$. To determine τ, the system estimates the peak translational and rotational velocity ($\hat{\mathbf{v}}_t$ and $\hat{\mathbf{v}}_r$) for the upcoming path segment, subtracts these from the current velocities $\mathbf{v}_{t0}$ and $\mathbf{v}_{r0}$, and divides by the appropriate acceleration limit.

The path segment velocities are estimated as follows. Assume that the manipulator is currently heading toward a target point $\mathbf{C1}(t)$, and the target point for the next path segment is $\mathbf{C2}(t)$. The displacement $\mathbf{D}$ between these two targets can be computed as

$$\mathbf{D} = \mathbf{C1}(t)^{-1}\,\mathbf{C2}(t)$$

$\mathbf{D}$ is used to compute $\mathbf{u}_t$ (a unit vector parallel to the translation vector), d_t (the magnitude of the translation vector), $\mathbf{u}_r$ (a unit vector describing the equivalent axis of rotation), and d_r (the angle of rotation about $\mathbf{u}_r$).

The path segment is assumed to consist of an acceleration phase, a cruise phase, and a deceleration phase. Considering first the translational component, the system constructs estimates of the speed ($\hat{v}_t$) and path segment time ($\hat{\sigma}_t$).

Let $\delta_a = (\bar{v}_t^2/\bar{a}_t)$ be the nominal translational distance covered during the acceleration/deceleration phase. If $\delta_a \leq d_t$, then $\bar{v}_t$ and $\bar{\sigma}_t$ are given by

$$\hat{\sigma}_t = 2\bar{v}_t/\bar{a}_t + (d_t - \delta_a)/\bar{v}_t$$

$$\hat{v}_t = \bar{v}_t$$

Otherwise, if $\delta_a > d_t$, then the desired cruising velocity $\bar{v}_t$ will not be reached and $\hat{v}_t$ and $\hat{\sigma}_t$ are given by

$$\hat{\sigma}_t = 2\sqrt{d_t/\bar{a}_t}$$

$$\hat{v}_t = \hat{\sigma}_t \bar{a}_t/2$$

The same computation may be repeated for the rotational component to obtain $\hat{\sigma}_r$ and $\hat{v}_r$. Because $\hat{\sigma}_t$ and $\hat{\sigma}_r$ will generally not be equal, the actual estimate of the path segment time $\hat{\sigma}$ is set to the maximum of the two:

$$\hat{\sigma} = \max(\sigma_t, \sigma_r)$$

$\hat{\mathbf{v}}_t$ and $\hat{\mathbf{v}}_r$ can now be computed as

$$\hat{\mathbf{v}}_t = \left(\frac{\sigma_t \hat{v}_t}{\hat{\sigma}}\right) \mathbf{u}_t$$

$$\hat{\mathbf{v}}_r = \left(\frac{\sigma_r \hat{v}_r}{\hat{\sigma}}\right) \mathbf{u}_r$$

Finally, τ is determined from

$$\tau = \frac{1}{2} \max\left(\frac{\|\hat{\mathbf{v}}_t - \mathbf{v}_{0t}\|}{\bar{a}_t}, \frac{\|\hat{\mathbf{v}}_r - \mathbf{v}_{0r}\|}{\bar{a}_r}\right)$$

If the next path segment is a stop request, then $\hat{\mathbf{v}}_t$ and $\hat{\mathbf{v}}_r$ are 0 and may be eliminated from the computation altogether.

5.2. Motion Segment Time

The motion segment time σ was estimated during the computation of τ. However, this considered only the anticipated displacement for the path segment and was not updated to be consistent with the final computed value of τ.

During the first computation cycle of the path segment, σ is recomputed using the distance to the target and the known value of τ. This calculation is simple and is illustrated again using the translational component. This time, let $\delta_c = d_t - 2\tau\bar{v}_t$ represent the total distance covered during the cruise phase of

the path segment. If $\delta_c > 0$, then σ_t is computed from

$$\sigma_t = 2\tau + d_t/\bar{v}_t$$

Otherwise, the path segment does not contain a cruise phase and

$$\sigma_t = 2\tau$$

After repeating these computations for the rotational component to obtain σ_r, the final value of σ is computed from

$$\sigma = \max(\sigma_t, \sigma_r)$$

5.3. General Remarks

For joint-interpolated motions, the computations are similar to those above, with each joint value being treated as an independent constraint component.

It is emphasized again that the computation of τ is only approximate, with computation time being the chief limiting factor. In particular, a precise determination of $\hat{\mathbf{v}}_t$ and $\hat{\mathbf{v}}_r$ would have to also consider the initial velocities $\mathbf{v}_{t0}$ and $\mathbf{v}_{r0}$.

These calculations require that **C2**(t) be computed simultaneously with the current motion target **C1**(t). It was mentioned briefly in Section 3.1 that RCCL avoids computing two paths simultaneously so as to not have to evaluate a functionally bound transform twice in two different motion contexts. This rule is still adhered to; **C2**(t) is determined by multiplying the transforms of the associated position equation but **not** calling any functions that may be attached to them. This can cause inaccuracies if a transform's function introduces discontinuities to its value at the beginning of the motion, although such cases are rare.

Finally, it should be mentioned that $\bar{a}_t$ and $\bar{a}_r$ could be computed dynamically using the manipulator Jacobian and the joint acceleration limits, although this is not done at the present time.

6. CARTESIAN MOTIONS

This section presents the specific computations used by the trajectory generator to do Cartesian interpolated motions. Some things have been omitted, including calculations related to Sections 4.3 and 5.

6.1. Special Operations Used

A few of the operations used in the general discussion need to modified for Cartesian coordinates. In particular, it is not possible to simply extrapolate a transform matrix **M** directly by computing something like

$$\mathbf{M}(k) = \mathbf{M}(0) + k\mathbf{M}'(0) \tag{10}$$

An equivalent computation is possible, however. Suppose that a displacement is represented not by a transform but by a 7-tuple consisting of a position vector, an axis of rotation, and an angle of rotation about the axis.* RCCL maintains such objects and calls them DSPLs (for *displacements*). The functions *trsfToDspl()* and *dsplToTrsf()* are used to convert between them and transforms. A DSPL representation can be easily scaled by simply multiplying the position vector and the rotation angle. The combined operation of scaling a DSPL and converting it back to a transform is performed by the function *extrap TrsfByDspl()* (notice that this function returns a transform). Because a DSPL can represent a velocity as well as finite displacement, the equivalent to (10) can be constructed as

$$\mathbf{M}(k) = \mathbf{M}(0)\textit{extrap TrsfByDspl}(\mathbf{v}, k)$$

where $\mathbf{v}$ is a DSPL representing a one-cycle displacement. This can be easily computed from two successive transform values:

$$\mathbf{v} = \textit{trsfToDspl}(\mathbf{M0}^{-1}\,\mathbf{M1})$$

Note that, like drive parameters, DSPLs can be easily interpolated. DSPLs could be used in place of drive parameters, although the two-angle interpolation used for drive parameters offers some advantages in robot task specification.

The blending of Cartesian path segments is done in the manipulator T6 frame, which means **that** S_0 and S_1 correspond to values of **T6** that will be designated as **S0** and **S1**.

We are now ready to present the path computation algorithm.

6.2. First Path Segment Computation Cycle

It is assumed that **T6**(0) and $\mathbf{v}_0$ (a DSPL representing the unit velocity in the T6 frame) are initially available. We first compute the value of **T6** toward which we are heading either by setting it equal to the previous motion's T6 target (if it is constant and the previous motion was not interrupted) or by estimating it from

$$\widehat{\mathbf{TB}} = \mathbf{T6}(0)\ \textit{extrap TrsfByDspl}(\mathbf{v}_0, \tau + 1)$$

Next, an initial value drive offset is computed:

$$\mathbf{O}(1) = \mathbf{COORD}(1)^{-1}\,\widehat{\mathbf{TB}}\,\mathbf{TOOL} \tag{1}$$

*This forms a computationally convenient representation for a finite screw.

S0(1) and **S0**(2) are then computed by extrapolation:

$$\mathbf{S0}(1) = \mathbf{T6}(0)\ \textit{extrap}\ \textit{TrsfByDspl}(\mathbf{v}_0, 1)$$

$$\mathbf{S0}(2) = \mathbf{T6}(0)\ \textit{extrap}\ \textit{TrsfByDspl}(\mathbf{v}_0, 2)$$

and **S0**(1) is used as the output setpoint:

$$\mathbf{j6}(1) = \wedge^{-1}(\mathbf{S0}(1))$$

6.3. Second Path Segment Computation Cycle

This cycle is the busiest. A second drive offset is computed, and then the two offsets are extrapolated to obtain the real drive parameters **dp**:

$$\mathbf{O}(2) = \mathbf{COORD}(2)^{-1}\ \widehat{\mathbf{TB}}\ \mathbf{TOOL}(2)$$

$$\mathbf{ov} = \textit{trsfToDspl}(\mathbf{O}(1)^{-1}\mathbf{O}(2))$$

$$\mathbf{dp} = \textit{trsfToDrive}(\mathbf{O}(1)\ \textit{extrap}\ \textit{TrsfByDspl}(\mathbf{ov}, \tau))$$

The values of **DRIVE**(t) corresponding to cycles 1 and 2 are then computed:

$$\mathbf{DRIVE}(1) = \textit{extrap}\ \textit{TrsfByDrive}(\mathbf{dp}, (t_c - 1)/\sigma)$$

$$\mathbf{DRIVE}(2) = \textit{extrap}\ \textit{TrsfByDrive}(\mathbf{dp}, (t_c - 2)/\sigma)$$

These are used to compute **S1**(1) and **S1**(2) and set the blend parameters:

$$\mathbf{S1}(1) = \mathbf{COORD}(1)\ \mathbf{DRIVE}(1)\ \mathbf{TOOL}(1)^{-1}$$

$$\mathbf{S1}(2) = \mathbf{COORD}(2)\ \mathbf{DRIVE}(2)\ \mathbf{TOOL}(2)^{-1}$$

$$\textit{setBlend1}(\mathbf{S0}(1), \mathbf{S1}(1))$$

$$\textit{setBlend2}(\mathbf{S0}(2), \mathbf{S1}(2), 2\tau)$$

Last, the output setpoint is determined by applying the blend function to **S1**(2):

$$\mathbf{j6}(2) = \wedge^{-1}(\beta(\mathbf{S1}(2)))$$

6.4 Remaining Computation Cycles

The value of **DRIVE**(t) is computed:

$$\mathbf{DRIVE}(t) = \textit{extrap}\ \textit{TrsfByDrive}(\mathbf{dp}, 1 - s)$$

and used to compute the output setpoint directly. The application of the blend function β is omitted if $t \geq t_b + \tau$:

$$\mathbf{j6}(t) = \wedge^{-1}(\beta(\mathbf{COORD}(t)\ \mathbf{DRIVE}(t)\ \mathbf{TOOL}(t)^{-1}))$$

7. JOINT COORDINATES

This section presents the specific computations used by the trajectory generator to do joint-interpolated motions. Because joint coordinates can be treated as vector quantities, their associated path computations follow directly from the introductory discussion given earlier. Because the path segment blending is done in joint coordinates, the values of S_0 and S_1 correspond to values of **j6** that will be designated as **js0** and **js1**.

7.1. First Path Segment Computation Cycle

Assume that the initial joint angles **j6**(0) and velocities **jv**(0) are available. The value of **j6** toward which we are heading is computed either by setting it equal to the previous motion's j6 target (if it is constant and the motion was not interrupted) or by estimating it from

$$\widehat{\mathbf{jb}} = \mathbf{j6}(0) + (\tau + 1)\mathbf{jv}(0)$$

Next, the value of the motion target is computed in joint coordinates (**jc**) and used to find an initial value for the drive offset **jo**:

$$\mathbf{jc}(1) = \wedge^{-1}(\mathbf{COORD}(1)\ \mathbf{TOOL}(1)^{-1})$$

$$\mathbf{jo}(1) = \widehat{\mathbf{jb}} - \mathbf{jc}(1)$$

js0(1) and **js1**(2) are then computed by extrapolation:

$$\mathbf{js0}(1) = \mathbf{j6}(0) = \mathbf{jv}(0)$$

$$\mathbf{js0}(2) = \mathbf{j6}(0) = 2\mathbf{jv}(0)$$

and **j6**(1) is set equal to **js0**(1).

7.2. Second Path Segment Computation Cycle

A second value for the drive offset is computed and used with the first value to determine the drive parameters **jdp**:

$$\mathbf{jo}(2) = \widehat{\mathbf{jb}} - \wedge^{-1}(\mathbf{COORD}(2)\ \mathbf{TOOL}(2)^{-1})$$

$$\mathbf{jdp} = \mathbf{jo}(1) + \tau(\mathbf{jo}(2) - \mathbf{jo}(1))$$

js1(1) and **js1**(2) are then computed and used to set the blend parameters:

$$\mathbf{js1}(1) = \mathbf{jc}(1) + \left(\frac{t_c - 1}{\sigma}\right)\mathbf{jd}(\tau)$$

$$\mathbf{js1}(2) = \mathbf{jc}(2) + \left(\frac{t_c - 2}{\sigma}\right)\mathbf{jd}(\tau)$$

setBlend1(**js0**(1), **js1**(1))

setBlend2(**js0**(2), **js1**(2), 2τ)

Last, the output setpoint is computed by applying the blend function β to **js1**(2):

$$\mathbf{j6}(2) = \beta(\mathbf{js1}(2))$$

7.3. Remaining Computation Cycles

The output setpoint is just computed directly, with the blend function β being omitted if $t \geq t_b + \tau$:

$$\mathbf{j6}(t) = \beta(\wedge^{-1}(\mathbf{COORD}(t)\ \mathbf{TOOL}(t)^{-1})) + (1 - s)\mathbf{jdp})$$

8. COMPUTATION TIMES

How much CPU time do the trajectory computations described here require? On a microVAX II system (now somewhat obsolete), the trajectory generator can control one PUMA robot quite comfortably at a 30-ms sample rate. The trajectory computation itself, involving closed-form inverse kinematics, several transform multiples, the application of a blend function, and precomputation of the next motion request, typically requires no more than 10 ms. An additional 5 ms or so may be required for the extra computations during the first two cycles of a motion.

On a Sun4 Sparc system, one robot can be controlled easily at 5 ms, with the trajectory computations taking up about 1.5 ms of this time. Newer systems such as Silicon graphics workstations running the MIPS R3000 CPU can do all of the trajectory computations in as little as 0.5 ms.

To help achieve these speeds, most computations are done using single precision and the trigonometric and square root functions are computed using linearly interpolated lookup tables with a worst case error of about 5^{-7}.

9. CONCLUSION

The robot programming system RCCL has been demonstrated over the last several years to be a useful platform for developing a wide variety of robot control applications. The trajectory generation technique described here,

which uses first-order velocity extrapolation combined with path blending, has been shown to be effective in handling various situations where the robot's path is subject to on-line modification from a wide range of stimuli. Examples of these stimuli include operator control inputs from joysticks or hand controllers, readings from force/torque sensors used to implement impedance control, or various types of visual and range-finding information used to implement tracking.

References

1. V. Hayward and R. Paul, "Robot manipulator control under UNIX: RCCL, a robot control C library," *International J. of Robotics Research*, **5**, 94–111, 1986.
2. J. Lloyd, M. Parker, and R. McClain, "Extending the RCCL programming environment to multiple robots, and processors," *IEEE Conf. on Robotics and Automation*, Philadelphia, PA, 1988, pp. 465–469.
3. J. Lloyd and V. Hayward, "Multi-RCCL user's guide," Research Center for Intelligent Machines, McGill University, Montreal, 1989.
4. S. Hayati, T. Lee, E. Kahn, and J. Lloyd, "The JPL telerobot manipulator control and mechanization sub-system," *NASA Conf. on Space Telerobotics*, Pasadena, CA, 1989.
5. J. Lloyd, M. Parker, and G. Holder, "Real time control under UNIX for RCCL," *3rd International Symp. on Robotics and Manufacturing*, Vancouver, B.C., Canada, 1990, pp. 237–242.
6. J. S. Lee, S. Hayati, V. Hayward, and J. Lloyd, "Implementation of RCCL, a robot control C library, on a microVAX II," *Advances in Intelligent Robotics Systems, SPIE's Cambridge Symp. on Optical and Optoelectronic Engineering*, Cambridge, MA, 1986.
7. E. P. Kan, J. Lee, and M. Junod, "An integrated FRHC-PUMA-EE teleautonomous system," *3rd International Symp. on Robotics and Manufacturing*, Vancouver, B.C., Canada, 1990, pp. 397–404.
8. S. Hayati, T. Lee, K. Tso, P. Backes, and J. Lloyd, "A testbed for a unified teleoperated-autonomous dual-arm robotic system," *IEEE International Conf. on Robotics and Automation*, Cincinnati, OH, 1990
9. P. K. Allen, B. Yoshimi, and A. Timcenko, "Real-time visual servoing," *IEEE International Conf. on Robotics and Automation*, Sacramento, CA, 1991, pp. 851–856.
10. A. Castano and S. A. Hutchinson, "Hybrid vision/position servo control of a robotic manipulator," *IEEE International Conf. on Robotics and Automation*, Nice, France, 1992.
11. P. Whaite, and F. P. Ferrie, "From uncertainty to visual exploration," *IEEE Transactions on Pattern Recognition and Machine Intelligence*, **PAMI-13**, 1038–1050, 1991.
12. A. H. Fagg, M. A. Lewis, T. Iberall, and G. A. Bekey, "*R²AD*: Rapid robotics application development environment," *IEEE International Conf. on Robotics and Automation, Sacramento, CA, 1991, pp. 1420–1426.*
13. J. E. Bobrow, S. Dubowsky, and J. S. Gibson, "Time optimal control of robotic manipulators along a specified path" *International J. of Robotics Research*, **4**, 3–17 (1985).
14. K. G. Shin and N. D. Mckay, "A dynamic programming approach to trajectory planning of robotic manipulators," *IEEE Transactions on Automatic Control*, **AC-31**, 491–500, 1986.
15. R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*, MIT Press, Cambridge, MA, 1981.

16. V. Hayward, L. Daneshmend, and A. Nilakantan, "Model based trajectory planning using preview," Report TR-CIM-88-9, Research Center for Intelligent Machines, McGill University , Montreal, 1988.
17. K. Shoemake, "Animating rotation with quaternion curves," *Computer Graphics (SIG-GRAPH '85 Conf. Proceedings)*, San Fransisco, CA, 1985, pp. 245–254.
18. R. H. Taylor, "Planning and execution of straight line manipulator trajectories," *IBM J. of Research and Development*, **23**, 253–264 (1979).