A Parallel Inverse Kinematics Solution for Robot Manipulators Based on Multiprocessing and Linear Extrapolation

Hong Zhang, *Member, IEEE,* and Richard P. Paul, *Fellow, IEEE*

Abstract—**In this paper, we present a method to compute inverse kinematics in parallel for robots with a closed form solution. We distinguish a pipelined solution from a parallel solution by that although both increase the system throughput, only the parallel solution reduces the computational latency. The computational task of computing inverse kinematics is partitioned with one subtask per joint and all subtasks are computed in parallel. The intrinsic dependency among subtasks is removed by linear extrapolation through the gradient of the inverse kinematic functions and joint velocity information. The simplicity of the solution makes it easily applicable to any robot manipulator with a closed form solution. Examples are used to illustrate the effectiveness and the efficiency of the algorithm. Implementation of the algorithm on a multiprocessor system is also described.**

I. Introduction

The fundamental problem of robot kinematics deals with mappings between vectors in two spaces: joint space θ and Cartesian space $\mathbf{x}$, where θ represents positions of the joints of a robot manipulator and $\mathbf{x}$ represents the position and orientation of the robot end effector. The mapping from joint space to Cartesian space is referred to as *direct kinematics* and mapping from Cartesian space to joint space is referred to as *inverse kinematics* [1]. Inverse kinematics is more interesting from the robot control point of view since in most robot applications, tasks are specified in Cartesian coordinates and inverse kinematics is computed to arrive at the joint coordinates to be used by lower level joint controllers.

Although direct kinematics can always be expressed analytically in a closed form, i.e.,

$$\mathbf{x} = f(\theta) \tag{1}$$

this is not always true for inverse kinematics. There must be at least as many joins in a manipulator as there are degrees of freedom to be controlled in Cartesian space in order for the inverse kinematics to yield a solution in general. If such a solution exists in a closed form, it can be expressed analytically by

$$\theta = g(\mathbf{x}). \tag{2}$$

Computation of the inverse kinematics in this case becomes the evaluation of (2).

Control of robot manipulators is a real-time process. Once a task is defined in terms of the initial and final Cartesian configuration[1] of the robot end effector, it is usually necessary to compute the joint trajectories with inverse kinematics to carry out the task. If one is concerned only with the initial and final configurations of a motion trajectory, the inverse kinematics need be evaluated only for those two configurations. On the other hand, if one is also concerned with all the intermediate Cartesian configurations along the trajectory, in cases such as a *straight-line* motion, the inverse kinematics must be evaluated repeatedly every sampling period Δt, where Δt is at the millisecond level [1]. It is strongly desirable to minimize the time of execution of the inverse kinematics, thereby minimizing Δt. Not only does a minimized Δt assure a stable and smooth motion, but, perhaps more importantly, it also enables the control system to react to external changes quickly, a characteristic that is critical in many robot tasks such as sensor-driven operations and compliant motion.

Various attempts have been made to speed inverse kinematics. Off-line computation relieves the control computer of computing kinematics in real time but is feasible only if a trajectory is predictable. Constructing special purpose hardware offers another solution but requires major efforts in design and fabrication [3]. Parallel processing supported by a multiprocessor architecture has been known to be effective in increasing the performance of computing systems. A pipelined architecture based on coordinated rotation digital computers (CORDIC) was described in [4] for robot manipulators with closed-form solutions. Harber *et al.* [5] further revised the architecture and presented its implementation on a bit-slice VLSI chip. The architecture divides the inverse kinematics into 25 subtasks, each being performed by a CORDIC computer. Since inverse kinematics is highly sequential [4], most of the 25 subtasks were executed sequentially and parallelism was achieved in the form of pipelining.

Although the architecture such as [4] drastically increased

Manuscript received April 27, 1989; revised January 21, 1991. This work was supported in part by the National Science Foundation under Grant ECS-8411879 and by the National Science and Engineering Research Council of Canada under Grant OGP0042194. Portions of this paper were presented at the IEEE International Conference on Robotics and Automation, Cincinnati, OH, May 13-18, 1990.

H. Zhang is with the Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1.

R. P. Paul is with the Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104.

IEEE Log Number 9101260.

[1] We define a Cartesian *configuration* by both the position and the orientation.

the throughput of the system roughly by the number of processing elements, it failed, however, to reduce the latency of the inverse kinematics, which is defined as the time between when a desired $\mathbf{x}$ is available and when the corresponding joint positions $\boldsymbol{\theta}$ are computed. Latency in a pipelined architecture is usually determined by the number of processing elements along the critical path, the path that contains the most processing elements from the entry node to the exit node in the task graph. As pointed out in [5], a pipelined architecture does not lead to any significant reduction in computational delay.

In this paper, we present an algorithm for computing inverse kinematics in parallel on a general-purpose multiprocessor system with the objective of reducing the latency significantly. The algorithm calls for one processor per joint. Unlike a pipeline that exploits parallelism in space, a parallel architecture exploits parallelism in time and consequently effectively reduces computational latency [6]. The amount of reduction is by a factor proportional to the number of joints. For a six-joint robot, for example, the delay to compute inverse kinematics is reduced by a factor of roughly 3. Since the algorithm distributes the computations by joints, no scheduling is necessary. The simplicity of the algorithm makes implementation a straightforward matter.

In Section II, we describe the algorithm development in two steps. In Section III, the parallel solution is applied to a realistic example, the Stanford arm, to demonstrate the feasibility and the effectiveness of the algorithm to reduce the latency. Section IV examines the algorithm in quantitative terms to study the sensitivity of the algorithm to robot trajectory requirements such as desired Cartesian velocities and sampling time. An implementation of the algorithm on a multiprocessor is described in Section V. Finally, in Section VI, we make a few concluding remarks.

II. Algorithm Formulation

If the desired Cartesian position and orientation $\mathbf{x}$ at time $t = t_k$ is specified by a 4×4 homogeneous transformation $\mathbf{T}_n$ [7]:

$$\mathbf{T}_n(t_k) = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

where the 3×3 orthogonal matrix formed by the $[\mathbf{n}, \mathbf{o}, \mathbf{a}]$ vectors represents the orientation of the end effector and the $\mathbf{p}$ vector represents the position of the end effector, both with respect to the base of the robot, the computation of the inverse kinematics with a closed-form solution essentially amounts to the evaluation of the following set of equations:

$$\theta_i(t_k) = g_i(\mathbf{T}_n(t_k), \theta_1(t_k), \cdots, \theta_{i-1}(t_k)) \quad \text{for } i = 1, 2, \cdots, n. \tag{4}$$

To evaluate (4), g_1 through g_n are evaluated in the ascending order of the joint number since g_i depends on the solutions of g_1 through g_{i-1}. In other words, the evaluation of the inverse kinematics is intrinsically a serial process, as illustrated in Fig. 1 where

$$\boldsymbol{\theta}^j(t_k) = [\theta_1(t_k), \theta_2(t_k), \cdots, \theta_j(t_k)]. \tag{5}$$

The sequential nature of the inverse kinematics thus makes it difficult to exploit the parallelism offered by a multiprocessor system. A pipelined architecture with one processing element computing for a joint cannot reduce the latency between when $\mathbf{T}_n$ is available and when the corresponding joint angles are generated.

A. An Approximate Parallel Solution

Observing that the joint positions do not change substantially from one sampling period (t_{k-1}) to the next (t_k) due to the continuity of the joint trajectories, and observing that the inverse kinematics is evaluated repetitively, one can remove the dependency among the joint solutions g_i by approximating joint positions, $\boldsymbol{\theta}^{i-1}(t_k)$, used in solving for joint i, with $\boldsymbol{\theta}^{i-1}(t_{k-1})$, joint positions in the previous sampling period [9]. Analytically, (4) can be approximated by

$$\hat{\theta}_i(t_k) = \hat{g}_i(\mathbf{T}_n(t_k), \theta_1(t_{k-1}), \hat{\theta}_2(t_{k-1}), \cdots, \hat{\theta}_{i-1}(t_{i-1})) \quad \text{for } i = 1, 2, \cdots, n \tag{6}$$

where $\hat{\theta}_i$ represents the approximated joint position. By this reformulation all $\hat{g}_i$ in (6) may begin evaluating *simultaneously* in parallel once $\mathbf{T}_n(t_k)$ becomes available. The reduction in computational delay can be substantial on a parallel processor. In fact, assuming a balanced computational cost for all the joints and ignoring the overhead introduced by the algorithm, the solution reduces the latency of inverse kinematics by a factor of n, the number of joints of the robot manipulator.

The problem of (6), however, is the errors it introduces due to the approximation. In general, given a desired Cartesian configuration $\mathbf{x}$, the exact serial inverse kinematics g produces $\boldsymbol{\theta}$ so that

$$f(\boldsymbol{\theta} = g(\mathbf{x})) = \mathbf{x}. \tag{7}$$

In contrast, the approximate solution $\hat{g}$ produces $\hat{\boldsymbol{\theta}}$, which, when used to solve direct kinematics, leads to

$$f(\hat{\boldsymbol{\theta}} = \hat{g}(\mathbf{x})) = \hat{\mathbf{x}} \neq \mathbf{x}. \tag{8}$$

If we define errors caused by $\hat{g}$ as the Euclidean norm of the difference between the correct configuration and the approximate configuration, then in joint space the error is

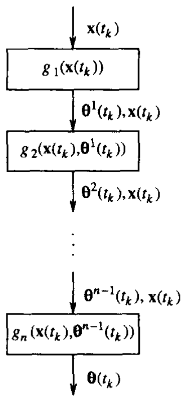
$$e_\theta = |\hat{g}(\mathbf{x}) - g(\mathbf{x})| \tag{9}$$

and in Cartesian space it is

$$e_{\mathbf{x}} = |f(\hat{g}(\mathbf{x})) - \mathbf{x}|. \tag{10}$$

Furthermore, the error $e_{\mathbf{x}}$ can be separated into a position component e_p and an orientation component e_r [8]. $e_{\mathbf{x}}$ is of more significance than e_θ as far as the task execution is concerned, although given one error the other can be easily found.

It should be pointed out that the Cartesian errors caused by (6) do not accumulate or deteriorate over time, as the parallel



$x(t_k)$
$g_1(x(t_k))$
$\theta^1(t_k), x(t_k)$
$g_2(x(t_k), \theta^1(t_k))$
$\theta^2(t_k), x(t_k)$
$\theta^{n-1}(t_k), x(t_k)$
$g_n(x(t_k), \theta^{n-1}(t_k))$
$\theta(t_k)$

Fig. 1. Serial computation of inverse kinematics.

solution is repeatedly computed over the course of a trajectory using approximated joint positions. To show this informally, we consider the following steps:

1) θ_1 is always computed exactly as no approximation is involved.

2) Error in θ_2 at t_k is caused by approximating $\theta_1(t_k)$ by $\theta_1(t_{k-1})$. Although the error results from the approximation each period, error in θ_2 at t_k is unrelated to that at t_{k-1}, as computation of $\hat{\theta}_2(t_k)$ is nonrecursive, i.e., it contains no reference, explicit or implicit, to $\hat{\theta}_2(t_{k-1})$. Only the approximation of θ_1 contributes to the error.

3) Two sources contribute to error in θ_3, approximations of a) $\theta_1(t_k)$ by $\theta_1(t_{k-1})$ and b) $\theta_2(t_k)$ by $\hat{\theta}_2(t_{k-1})$. Similar to θ_2, computation of $\hat{\theta}_3$ at t_k contains no reference to $\hat{\theta}_3$ at t_{k-1}. Therefore, an error in θ_3 does not propagate from one period to the next. From 1) and 2), since disturbance to the computation of θ_3 due to either approximation does not accumulate over time, neither will the error in $\hat{\theta}_3$.

4) An exact same argument as 3) can be applied to θ_4 through θ_6 to show that errors in all joints do not accumulate over time.

5) Cartesian errors are determined by joint errors and are therefore not accumulative.

Finally, it is easy to show that static errors in (6) disappear within n sampling periods when the manipulator comes to a stop and $\mathbf{T}_n$ stops changing.

On the other hand, errors in joint space increase with the number of intervening links from the base. The larger the joint number, the more approximation terms in (6) are involved in the computation of the joint variable, and the less accurate the solution becomes. However, this is not very significant as we are interested primarily in errors in Cartesian space.

The computation delay required by each processor to provide its contribution to the solution does not affect the accuracy of the solution. However, the rate at which the solution is computed, along with the velocity of the manipulator, does. These two factors, taken together, affect the relevance of the joint angles used in the computation of each joint solution due to the delay resulting from the propagation of the results through the various stages of the computation, even though the stages are computed in parallel. The effects of both sampling rate and manipulator velocity will be described fully in Section IV.

B. Algorithm Improvement by Linear Extrapolation

The error in the initial parallel solution summarized in this section is caused by the strong and incorrect assumption that $\theta^{i-1}(t_{k-1})$ has not changed when $t = t_k$. It is therefore reasonable to expect to improve the algorithm by making use of velocity information. Specifically, when a general inverse kinematic function g_i is expressed as

$$\theta_i(t_k) = g_i(\mathbf{T}_n(t_k), \theta^{i-1}(t_k)) \tag{11}$$

to compute g_i in parallel, we would like to obtain an expression of the form

$$\theta_i(t_k) = g_i(\mathbf{T}_n(t_k), \theta^{i-1}(t_{k-1})) + \Delta_i \tag{12}$$

where the first term is the same as (6) and Δ_i represents a compensation term that depends on the rate of change of both θ and $\mathbf{T}_6$ so as to make (12) an equality. Obviously, in order to maintain the parallel nature of (6), (12) is meaningful only if Δ_i is determined solely from information available at $t = t_{k-1}$ and $\mathbf{T}_n(t_k)$.

If a first-order approximation is accurate for θ_i and g_i when the sampling period is small, we have

$$\theta_i(t_k) = \theta_i(t_{k-1}) + \dot{\theta}_i(t_{k-1})\, dt \tag{13}$$

and

$$g_i(\mathbf{T}_n(t_k), \theta^{i-1}(t_{k-1})) = g_i(\mathbf{T}_n(t_{k-1}), \theta^{i-1}(t_{k-1})) + \frac{\partial g_i}{\partial \mathbf{T}_n}(t_{k-1})\, d\mathbf{T}_n \tag{14}$$

where (13) implies a linear joint motion for infinitesimally small dt and (14) simply represents a linear extrapolation (or projection) of g_i along the gradient direction. In addition, since

$$\theta_i(t_{k-1}) = g_i(\mathbf{T}_n(t_{k-1}), \theta^{i-1}(t_{k-1})) \tag{15}$$

we can substitute (15) into (13) first and then (14) into (13) to obtain a new expression for $\theta_i(t_k)$

$$\begin{aligned} \theta_i(t_k) &= \theta_i(t_{k-1}) + \dot{\theta}(t_{k-1})\, dt \\ &= g_i(\mathbf{T}_n(t_{k-1}), \theta^{i-1}(t_{k-1})) + \dot{\theta}_i(t_{k-1})\, dt \\ &= g_i(\mathbf{T}_n(t_k), \theta^{i-1}(t_{k-1})) \\ &\quad - \frac{\partial g_i}{\partial \mathbf{T}_n}(t_{k-1})\, d\mathbf{T}_n + \dot{\theta}_i(t_{k-1})\, dt. \end{aligned} \tag{16}$$

Removing intermediate steps in (16), we obtain the fundamental equation of the parallel inverse kinematics:

$$\theta_i(t_k) = g_i(\mathbf{T}_n(t_k), \theta^{i-1}(t_{k-1})) - \frac{\partial g_i}{\partial \mathbf{T}_n}(t_{k-1})\, d\mathbf{T}_n + \dot{\theta}_i(t_{k-1})\, dt. \tag{17}$$

Comparing (12) and (17), we notice that

$$\Delta_i = -\frac{\partial g_i}{\partial \mathbf{T}_n}(t_{k-1})\, d\mathbf{T}_n + \dot{\theta}_i(t_{k-1})\, dt. \tag{18}$$

As we hoped, Δ_i in (12) can indeed be computed exactly from expressions available at t_{k-1} and $\mathbf{T}_n(t_k)$, when (9) and (10) are perfect, leading to a parallel solution computable on a multiprocessor of which the latency is the worst of the n processors. Dependencies among g_i in the serial inverse kinematics are removed by linear extrapolation in both joint space (13) and Cartesian space (14). The parallel solution is illustrated in Fig. 2.

A few questions arise upon the first observation of (17). First, how does one compute the extrapolation term

$$\frac{\partial g_i}{\partial \mathbf{T}_n}\, d\mathbf{T}_n? \tag{19}$$

An examination of inverse kinematic solutions for many robot manipulators reveals that although θ depends on $\mathbf{T}_n$, each inverse kinematic function may depend only on one of the four column vectors of $\mathbf{T}_n$, i.e., $\mathbf{n}$, $\mathbf{o}$, $\mathbf{a}$, and $\mathbf{p}$ of (3). If, for example, the manipulator is built with a spherical wrist, then the joints before the wrist determine the $\mathbf{p}$ vector, or their solution depends only on the $\mathbf{p}$ vector. Consequently, we can rewrite the general inverse kinematic function as

$$\theta_i(t_k) = g_i(v, \theta^{i-1}(t_k)) \tag{20}$$

where $v \in \{\mathbf{n}, \mathbf{o}, \mathbf{a}, \mathbf{p}\}$. Equation (18) can then be rewritten as

$$\begin{aligned}\Delta_i &= -\frac{\partial g_i}{\partial v}\, dv + \dot{\theta}_i(t_{k-1})\, dt \\ &= -\nabla g_i \cdot dv + \dot{\theta}_i(t_{k-1})\, dt \end{aligned} \tag{21}$$

where ∇g_i represents the gradient of g_i consisting of three partial differentials and, in general, has the form

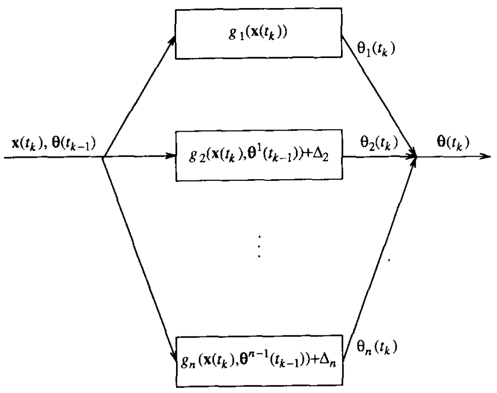
$$\nabla g_i = \left[\frac{\partial g_i}{\partial v_x}, \frac{\partial g_i}{\partial v_y}, \frac{\partial g_i}{\partial v_z}\right]. \tag{22}$$

In this case, (19) simply becomes a dot product between two three-vectors, the gradient of g_i, and a vector of differential change, of which g_i is a function. Partial differentials of a multivariate function can be readily derived and, consequently, (21) can be readily computed. In case g_i depends on more than one vector of $\mathbf{T}_n$, (19) can still be evaluated just as easily, but at a higher computational cost, by

$$\Delta_i = -\frac{\partial g_i}{\partial v_1}\, dv_1 - \frac{\partial g_i}{\partial v_2}\, dv_2 + \dot{\theta}_i(t_{k-1})\, dt \tag{23}$$

where v_1 and v_2 are, for example, the two vectors that solution g_i depends on. Two dot products are required in this case to compute extrapolation.

Two other concerns with regard to the computation of (19) are 1) since in reality (13) and (14) are not perfect for finite Δt, how do the second- and higher order effects affect the solution? and 2) will the additional computational cost which (18) consumes still make the parallel solution (17) computationally attractive? We address both issues in the next sections.



$g_1(\mathbf{x}(t_k))$
$\theta_1(t_k)$
$\mathbf{x}(t_k), \theta(t_{k-1})$
$g_2(\mathbf{x}(t_k),\theta^1(t_{k-1}))+\Delta_2$
$\theta_2(t_k)$
$\theta(t_k)$
$g_n(\mathbf{x}(t_k),\theta^{n-1}(t_{k-1}))+\Delta_n$
$\theta_n(t_k)$

Fig. 2. Parallel computation of inverse kinematics.

Finally, unlike the approximate parallel solution (6), errors in (17) that arise when Δt is finite can propagate to the next sampling period by way of $\dot{\theta}_i$, which has to be obtained by the difference between inexact positions. We will however use examples to show numerically that this propagation does not cause an accumulation of errors.

III. The Stanford Arm

In this section, we demonstrate the feasibility of the parallel solution on the Stanford arm [7] by first deriving the gradients of its inverse kinematic functions using the general equation (22). We then calculate the speed-up factor in this case. We choose only one of the multiple solutions that exist, as this does not affect the result.

A. Inverse Kinematic Functions and Their Gradients for the Stanford Arm

The inverse kinematics for the Stanford arm is computed with the following set of equations.

Joints One: An inverse Kinematic function of joint one of the Stanford arm is defined by[2]

$$\theta_1 = g_1(\mathbf{p}) = \tan^{-1}\frac{p_y}{p_x} - \tan^{-1}\frac{d_2}{\sqrt{p_x^2 + p_y^2 - d_2^2}}. \tag{24}$$

Joint Two: One of the two inverse kinematic functions of joint two is

$$\theta_2(\mathbf{p}, \theta^1) = \tan^{-1}\frac{g_{21}}{p_z} \qquad \text{where} \qquad g_{21} = C_1 p_x + S_1 p_y. \tag{25}$$

The gradient of g_2 can be computed recursively as follows:

$$\nabla g_2(\mathbf{p}) = \left[\frac{p_z C_1}{g_{22}}, \frac{p_z S_1}{g_{22}}, -\frac{g_{21}}{g_{22}}\right] \qquad \text{where} \qquad g_{22} = p_z^2 + g_{21}^2. \tag{26}$$

[2] $\tan^{-1}(y/x)$ here denotes atan2(y, x).

Joint Three: Inverse kinematic function g_3 computes the position of the prismatic joint by

$$d_3 = S_2(C_1 p_x + S_1 p_y) + C_2 p_z. \tag{27}$$

∇g_3 is computed by

$$\nabla g_3(\mathbf{p}) = [C_1 S_2, S_1 S_2, C_2]. \tag{28}$$

Joint Four: Inverse Kinematic function g_4 is defined by

$$\theta_4 = g_4(\mathbf{a}, \theta^3) = \tan^{-1}\frac{g_{41}}{g_{42}} \tag{29}$$

where

$$g_{41} = -S_1 a_x + C_1 a_y \quad \text{and}$$
$$g_{42} = C_2(C_1 a_x + S_1 a_y) - S_2 a_z. \tag{30}$$

Gradient of g_4 is computed by

$$\nabla g_4 = \left[\frac{S_1 S_2 a_z - C_2 a_y}{g_{43}}, \frac{C_2 a_x - C_1 S_2 a_z}{g_{43}}, \frac{S_2}{g_{43}}\right] \tag{31}$$

where

$$g_{43} = g_{41}^2 + g_{42}^2. \tag{32}$$

Joint Five: Inverse kinematic function g_5 is computed by

$$\theta_5 = g_5(\mathbf{a}, \theta^4) = \tan^{-1}\frac{g_{51}}{g_{52}} \tag{33}$$

where

$$g_{511} = (C_1 C_2 C_4 - S_1 S_4), \qquad g_{512} = (S_1 C_2 C_4 + C_1 S_4),$$
$$g_{513} = -S_2 C_4$$
$$g_{51} = g_{511} a_x + g_{512} a_y + g_{513} a_z$$
$$g_{521} = C_1 S_2, \qquad g_{522} = S_1 S_2$$
$$g_{52} = g_{521} a_x + g_{522} a_y + C_2 a_z. \tag{34}$$

The gradient of g_5 is

$$\nabla g_5(\mathbf{a}) = [g_{511} g_{52} - g_{521} g_{51}, g_{512} g_{52} - g_{522} g_{51}, g_{513} g_{52} - C_2 g_{51}]. \tag{35}$$

Joint Six: In the case of the inverse kinematic function g_6,

$$\theta_6 = g_6(\mathbf{o}, \theta^5) = \tan^{-1}\frac{g_{61}}{g_{62}} \tag{36}$$

where g_{61} and g_{62} are computed by

$$g_{611} = -C_5(C_1 C_2 C_4 - S_1 S_4) + S_5 C_1 S_2,$$
$$g_{612} = -C_5(S_1 C_2 C_4 + C_1 S_4) + S_5 S_1 S_2$$
$$g_{613} = S_2 C_4 C_5 + C_2 S_5$$
$$g_{61} = g_{611} o_x + g_{612} o_y + g_{613} o_z$$
$$g_{621} = -C_1 C_2 S_4 - S_1 C_4, \qquad g_{622} = -S_1 C_2 S_4 + C_1 C_4,$$
$$g_{623} = S_2 S_4$$
$$g_{62} = g_{621} o_x + g_{622} o_y + g_{623} o_z. \tag{37}$$

The gradient of g_6 is

$$\nabla g_6(\mathbf{o}) = [g_{611} g_{62} - g_{621} g_{61}, g_{612} g_{62} - g_{622} g_{61}, g_{613} g_{62} - g_{623} g_{61}]. \tag{38}$$

B. Speed-up Factor Calculation

From (24) through (38), Tables I and II summarize the number of arithmetic operations required to evaluate the inverse kinematics and their gradient functions, respectively. Note in Table I that d_3 is a prismatic joint and requires no sine and cosine computation. Joint six is not used by any other joints and its sine and cosine are not computed either. Joint position θ_1 is always computed exactly and requires no ∇_1 in Table II.

To make a meaningful comparison, we use the execution times on a floating-point processor (Intel 8087 [10]) shown in Table III. Tables I and II can now be interpreted in terms of real time and the results are shown in Table IV, which displays the time of execution of inverse kinematics, the time of the extrapolation term computation, and the sum of the two.

Therefore, in the case of Stanford arm, the weighted cost of the serial execution is the sum of g_i,[3] which totals 7343, whereas that of the parallel execution with linear extrapolation is the worst of $g_i + \Delta_i$, which turns out to be 2766 (θ_6). This represents a speed-up factor of 2.65. In other robot manipulators we have studied, such as the PUMA 560, the speed-up factor is higher—around 3.0—because of the absence of a prismatic joint [11].

IV. Performance Evaluation

In this section, we examine the performance of the parallel algorithm in terms of the errors that are not completely removed by the compensation term Δ_i when, as mentioned previously, dt and $d\mathbf{T}_6$ are not infinitesimally small. Since infinitesimal dt and $d\mathbf{T}_6$ are an unrealistic assumption, it is only natural to study the errors they cause with respect to the extent to which this assumption is violated. Specifically, we will evaluate the algorithm for various finite sampling times Δt at various Cartesian linear and angular velocities (v_x and ω_x).

A. Method

Unfortunately, it is extremely difficult—if not impossible—to obtain an analytical evaluation based on the error definition (10). Instead, we opt for a numerical approach similar to [9]. We use the Stanford arm as an example, randomly generate a large number of test trajectories, and collect statistics of the tests under various conditions. The test conditions we consider include sampling period Δt, Cartesian linear velocity v_v, and Cartesian angular velocity ω_x, which is defined as a constant rotational velocity about a fixed axis in space [7]. For each chosen set of conditions, 200 trajectories in the robot work space are randomly generated (a virtually identical result is obtained when the number of

[3] We ignore the fact that joints may share some intermediate expressions when the inverse kinematics is computed serially, for it will only make a marginal difference.

TABLE I
ARITHMETIC OPERATIONS FOR INVERSE KINEMATIC FUNCTIONS

	+ or −	× or /	$\tan^{-1}$	*sqrt*	sin/cos
θ_1	2	2	1	1	1
θ_2	1	2	1	0	1
d_3	2	4	1	0	0
θ_4	3	6	1	0	1
θ_5	6	14	1	0	1
θ_6	11	23	1	0	0

TABLE II
ARITHMETIC OPERATIONS FOR LINEAR EXTRAPOLATION Δ_I

	+ or −	× or /
Δ_2	5	9
Δ_3	4	5
Δ_4	7	12
Δ_5	7	9
Δ_6	7	9

TABLE III
TIME OF EXECUTION OF DIFFERENT OPERATIONS

Operation	Time of exec (μs)
Adds/subs	40
Multiply/divide	53
Inverse trig	350
Sin/cos pair	360
Square root	100

TABLE IV
TOTAL TIME OF EXECUTION OF THE PARALLEL SOLUTION

	g_i (μs)	Δ_i (μs)	$g_i + \Delta_i$ (μs)
θ_1	1346	0	1346
θ_2	856	677	1533
d_3	292	425	717
θ_4	1148	916	2064
θ_5	1692	757	2449
θ_6	2009	757	2766

test trajectories is increased to up to 1000). From each trajectory we obtain two critical values—the maximum Cartesian position error (e_p) and the maximum Cartesian orientation error (e_r) as defined by (10) that are produced by the parallel inverse kinematics (17). For that set of test conditions, therefore, we collect 200 maximum e_p and e_r, respectively. To facilitate easier interpretation, the maxima are sorted first and then plotted. A typical result is displayed in Fig. 3. In this case the test conditions consist of a sampling period of 1 ms, a Cartesian linear velocity at 1 m/s, and a Cartesian angular velocity at 1.2 rad/s. The logarithmic y axis is used because of large disparity among the test trajectories.

This numerical technique allows us to characterize the performance of the parallel algorithm in several ways. We will use the technique to study the effectiveness of the compensation term and the sensitivity of the algorithm to changing motion parameters.

B. Comparison Between With and Without Δ_i

Fig. 4 shows the results of the comparison with respect to position error distributions under the given conditions. The

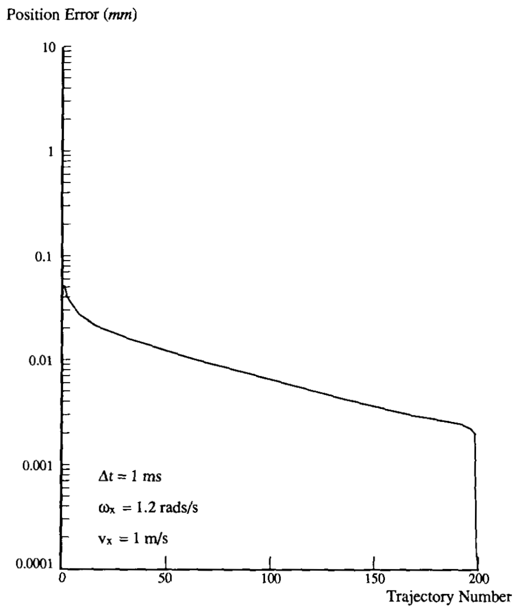


Fig. 3. Typical position error distribution with 200 random trajectories.

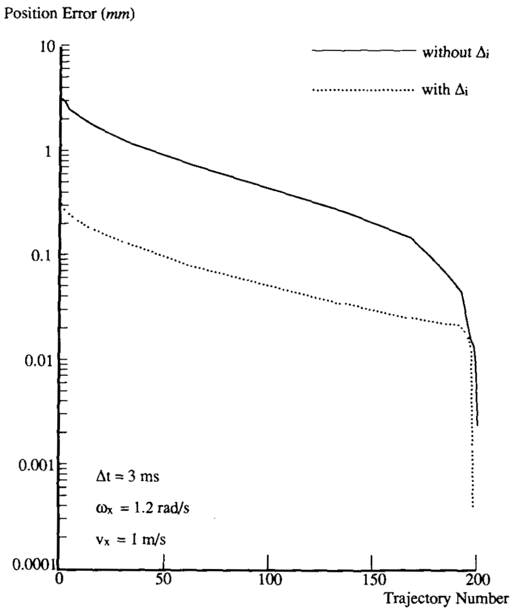


Fig. 4. Effect of compensation term on position error.

effect of the compensation term is clearly shown to reduce the position error of the algorithm at least by an order of magnitude. More significant is the fact that even under rather harsh test conditions of a long sampling period and high Cartesian velocities, the worst position error ever observed is 0.3 mm and the majority of the test trajectories (>75%) experience maximum position errors of less than 0.1 mm, a level negligible for all practical considerations.

The effect of the compensation term in removing errors is even more dramatic in the case of orientation errors as shown in Fig. 5 where more than an order of magnitude of reduction is achieved and the majority of the maximum orientation errors of the test trajectories are below 0.1°.

C. Position Errors

To further study the performance of the algorithm on the Stanford arm, we first examine the relationship between position errors and sampling period and Cartesian velocities. Fig. 6 displays the results corresponding to four different sampling periods. As we can see, each time the sampling period is reduced by 1 ms, the maximum error on the average drops by a factor of 2.

A similar effect is observed when the Cartesian linear velocity is changed (Fig. 7). Each time v_x is reduced by 50%, the position error is reduced by a factor of 4. In the case of the slowest motion tested of 0.25 m/s for a relatively long sampling period of 3 ms, the worst error ever recorded falls below 0.05 mm. This is equivalent to the quantization error resulting from the use of 16-bit integers to represent joint positions.

Fig. 8 shows the effect of Cartesian orientation velocity on position error distribution. Interestingly, no significant effect is observed. This can be explained by the fact that the kinematics of the Stanford arm, like many other robot manipulators built with a spherical wrist, forces a null upper right 3×3 block in its Jacobian matrix and hence in the inverse Jacobian. Any change in angular velocity, therefore is, achieved only by the change in the joint velocities of the wrist, which makes no contribution to the position error. The slight differences among the four distributions in Fig. 8 are caused merely by the randomness of the test trajectories.

D. Orientation Errors

Effects of the sampling period, Cartesian linear, and angular velocities on orientation error distributions are seen in Figs. 9, 10, and 11. We can conclude similarly to position error distributions. The only major difference is that Fig. 11 shows a strong correlation between Cartesian angular velocity and orientation errors.

V. Multiprocessor Implementation

A multiprocessor system based on off-the-shelf single board computers was constructed to control a PUMA 260 robot manipulator [12]. The parallel solution (6) *without* the compensation term was implemented as we were more concerned with speed than accuracy of the system. The architecture of the controller is illustrated in Fig. 12 where each joint employs an 8086-based single-board computer with an 8087 coprocessor (iSBC 86/30 [10]), as does the supervisor.

The supervisor performs trajectory planning in the Cartesian space. In addition, it synchronizes all the processes running in the system. Its master clock initiates the interrupt to the real-time loop on the supervisor, which in turn interrupts the joint processors and sends necessary data (such as T_6 and sines and cosines of the joints) to the joints for the next iteration of the trajectory computation.

Once the joints are interrupted, their real-time loops start in parallel simultaneously. Each joint waits for the data arrival from the supervisor and then computes, among other things, inverse kinematics at 250 Hz. Between motion segments, joints perform transitions also in parallel to remove any discontinuities in position and velocity between the current and next segment of motion. Dynamics is computed on a separate processor (not shown) running in the background at a slower rate. At the servo level, the system runs a proportional integral and differential (PID) control loop at a higher

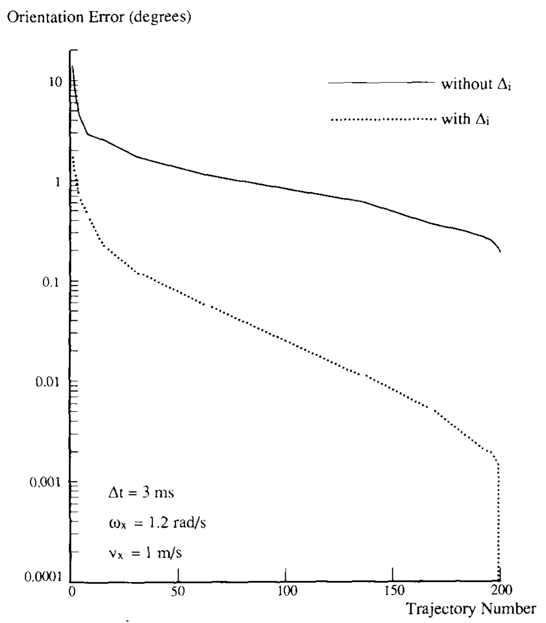


Fig. 5. Effect of compensation term on orientation error.

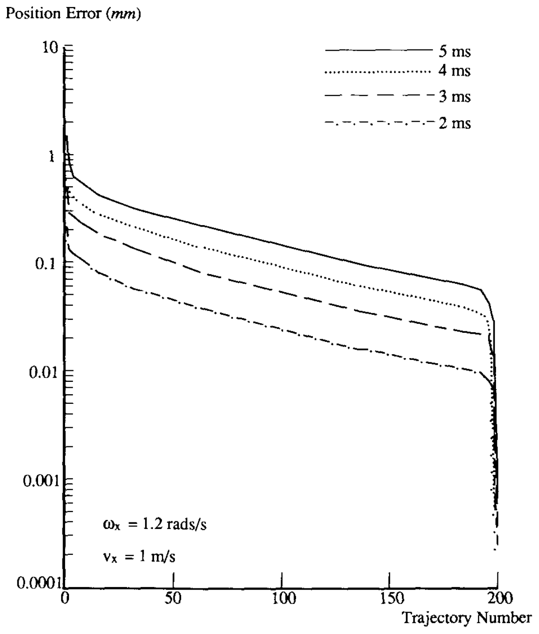


Fig. 6. Effect of sampling period on position error.

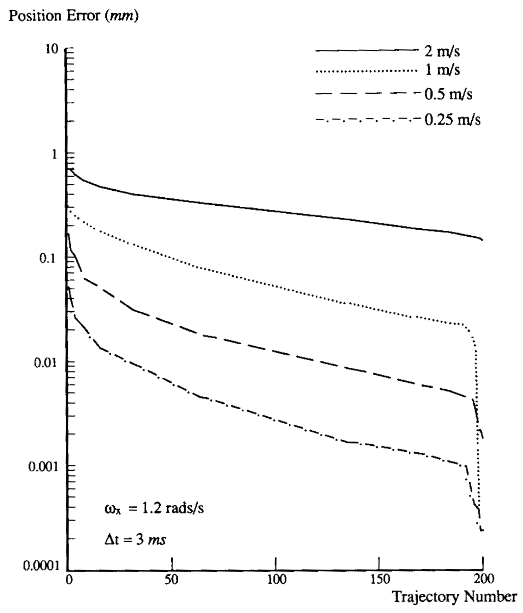


Fig. 7. Effect of linear Cartesian velocity on position error.

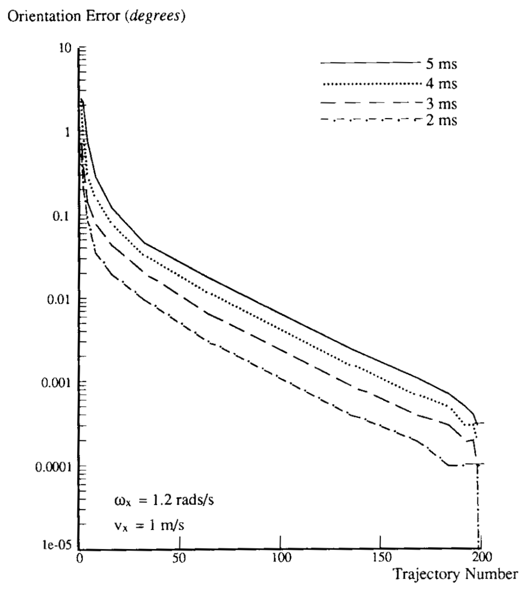


Fig. 9. Effect of sampling period on orientation error.

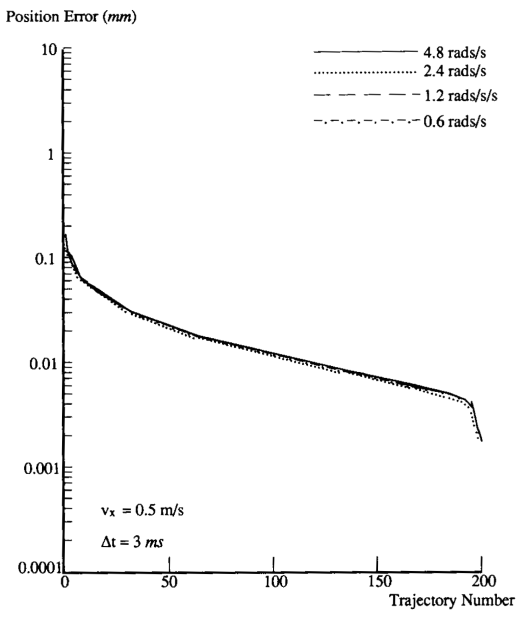


Fig. 8. Effect of angular Cartesian velocity on position error.

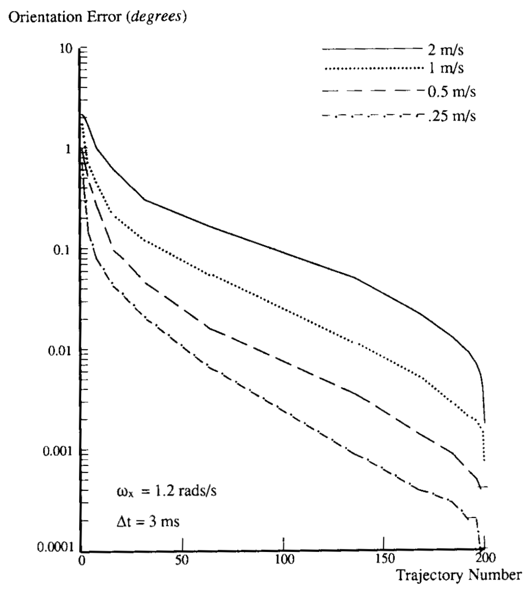


Fig. 10. Effect of linear Cartesian velocity on orientation error.

rate than the set-point generation. The servo derives the intermediate command positions by linearly interpolating set-points computed by the joint inverse kinematics process.

Joints and supervisor communicate via the multibus through shared memory. At the end of each sampling period, each joint stores the computed results in mailboxes, which then are collected and distributed to other joints by the supervisor. The joint processors depend on each other for the evaluation of the sines and the cosines of joint angles. A copy of current desired $\mathbf{T}_6(t_k)$ must also be sent to all the joints every sampling period. All the information is exchanged among the processors prior to each iteration of the inverse kinematics evaluation. The overhead is therefore dominated by such data collection and distribution among supervisor and the joints. Given the speed of the multibus at 1 mbyte/s, data communication costs roughly half a millisecond to perform every sampling period. The situation however is improved by ordering the writing of information according to the loads of the joints and by using a binary semaphore on each joint so that a joint can begin evaluation as soon as the necessary data arrive. Furthermore, efficiency is obtained in our system by

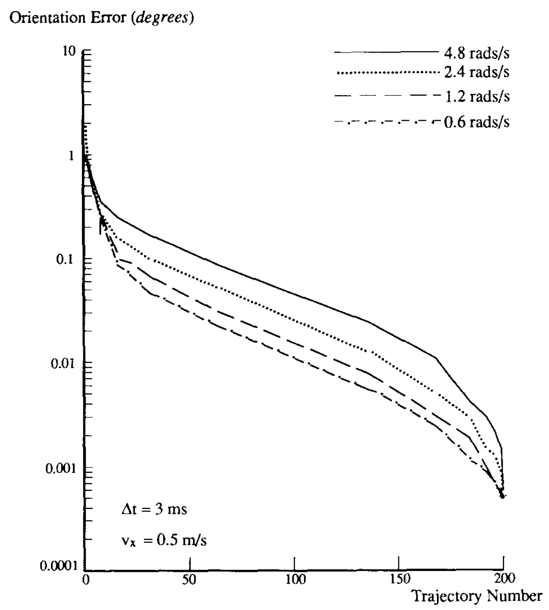


Fig. 11. Effect of angular Cartesian velocity on orientation error.

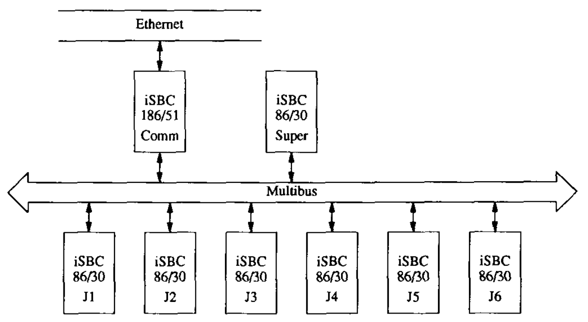


Fig. 12. System implementation.

performing many other operations, such as joint servos, completely in parallel.

The controller communicates with the external world through an Ethernet communication processor iSBC 186/51 [10]. Task definitions are first sent to the controller through this processor. While the task is being executed, it can be modified through this communication process.

The simplicity of the task partioning according to (17) facilitates the implementation of the architecture in Fig. 12. The architecture, in turn, allows easy integration of the kinematics computation with the rest of the control system. Servoing of the joint actuators, for example, is achieved by a special parallel interface attached to each joint processor and is performed in parallel. Finally, with the development of more advanced microprocessors, the joint processors can simply be superseded without any major software overhaul.

VI. Discussion and Conclusion

We have introduced a formalism by which robot inverse kinematics with closed-form solutions can be computed in parallel. The formalism removes the dependencies among joints by extrapolation in joint space through joint velocity information and in Cartesian space by the gradient of the inverse kinematics. Since the algorithm is parallel, it effectively reduces the latency of the inverse kinematics computation. In general the algorithm requires an n-processor system where n is the number of joints and a factor of speed-up proportional to n is expected. The proportional constant is roughly 0.5 for the examples we have studied. The simplicity of the algorithm makes it easy to implement on a parallel architecture such as the one in Section V.

The algorithm introduces errors that are negligible for all practical purposes. The errors, however, increase with the nonlinearity of the joint motions. An attempt was made to use a motion model of a higher order (a quadratic, for example) for joint space extrapolation to reduce the residual error but was unsuccessful since higher order joint parameters such as acceleration and jerk obtained by differencing joint positions tend to be too noisy to bring about any significant improvement to the algorithm.

The evaluation of (17) requires the estimation of a joint velocity from joint positions generated by the inverse kinematics. A conventional approach is differencing the two neighboring positions and dividing it by the sampling period. In our experiments, however, such a solution is less accurate than a least squares solution that estimates velocity at t_{k-1} from joint positions at t_{k-1}, t_{k-2}, and t_{k-3}. Given the three joint positions separated pairwise by Δt in time, the least squares solution yields a velocity estimate that is the average of the last two sampling periods, which has the form:

$$\dot{\theta}_i(t_{k-1}) = \tfrac{1}{2}(\theta_i(t_{k-1}) - \theta_i(t_{k-3}))/\Delta t. \qquad (39)$$

In practice, the parallel solution described above can be used with or without the compensation term. When accuracy is crucial, linear extrapolation is used to minimize the errors, but when it is not, the approximate parallel solution is used for efficiency. Finally, even though our implementation uses a number of general-purpose computers tightly coupled by a data bus, it is also possible to execute the algorithm on an array processor with n-processing elements and a memory unit where a single copy of global variables shared by joints is stored so that all data accesses by the processing elements are local and the parallel system overhead is thus minimized.

References

[1] J. J. Craig, *Introduction to Robotics: Mechanics and Control.* Reading, MA: Addison-Wesley, 1986.
[2] R. P. Paul and H. Zhang, "Computational efficient kinematics for manipulators with spherical wrists based on homogeneous transformation representation," *Int. J. Robotics Res.*, vol. 5, no. 2, pp. 30-42, Summer 1986.
[3] Y. Wang and S. Butner, "A new architecture for robot control," in *Proc. 1987 IEEE Int. Conf. Robotics Automat.* (Raleigh, NC), 1987, pp. 664-670.
[4] C. S. G. Lee and P. R. Chang, "A maximum pipelined CORDIC architecture for inverse kinematics position computation," *IEEE J. Robotics Automat.*, vol. RA-3, pp. 445-458, Oct. 1987.
[5] R. Harber, J. Li, X. Hu, and S. Bass, "The application of bit-serial CORDIC computational units to the design of inverse kinematics processors," in *Proc. 1988 IEEE Int. Conf. Robotics Automat.* (Philadelphia, PA), 1988, pp. 1152-1157.
[6] R. Y. Kain, *Computer Architecture: Software and Hardware.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

[7] R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control.* Cambridge, MA: MIT Press, 1981.
[8] R. H. Taylor, "Planning and execution of straight line manipulator trajectories," *IBM J. Res. Devel.*, vol. 23, no. 4, July 1979.
[9] H. Zhang and R. P. Paul, "A parallel solution to robot inverse kinematics," in *Proc. 1988 IEEE Int. Conf. Robotics Automat.* (Philadelphia, PA), 1988, pp. 1140-1145.
[10] Intel Corporation, *iAPX 86/88, 186/188 User's Manual*, Intel Corporation, Santa Clara, CA, 1985.
[11] H. Zhang and R. P. Paul, "A parallel inverse kinematics solution for robot manipulators based on multiprocessing and linear extrapolation," in *Proc. 1990 IEEE Int. Conf. Robotics Automat.* (Cincinnati, OH), 1990, pp. 1140-1145.
[12] —, "A robot force and motion server," in *Proc. 1986 ACM/IEEE Computer Soc. Fall Joint Conf.* (Dallas, TX), Nov. 1986, pp. 178-184.

Hong Zhang (M'88) received the B.S. degree in electrical engineering from Northeastern University, Boston, MA in 1982, and the Ph.D. degree in electrical engineering from Purdue University, West Lafayette, IN in 1986. His Ph.D. dissertation was in the area of force control of robot manipulators.

He subsequently spent 18 months as a postdoctral fellow at the University of Pennsylvania in Philadelphia, conducting research in the GRASP laboratory in the Department of Computer and Information Science. In January 1989, he joined the University of Alberta, Edmonton, Alberta, Canada, where he is now an Assistant Professor in the Department of Computing Science. His main research interests are force control and kinematic analysis of robot manipulators, robot fine manipulation, and estimation and decision theory as applied to robotics.

Richard P. Paul (S'60-SM'78-F'89) received the Ph.D. degree in computer science in 1972 from Stanford University, Palo Alto, CA.

He was associated in research and engineering with Stanford University from 1963 to 1974, where he developed the WAVE robot language and demonstrated the first use of programmable robots for assembly. In 1976, he joined the faculty of Purdue University, West Lafayette, IN, where he was a Professor of Electrical Engineering and the Ransburg Professor of Robotics. He joined the University of Pennsylvania in 1984 where he is currently a Professor of Computer and Information Science with a joint appointment in mechanical engineering. He is the author of *Robot Manipulators: Mathematics, Programming and Control* (MIT Press, 1981) and over 100 articles, conference presentations, films, and technical reports on robotics, kinematics, manipulation, and artificial intelligence. He has made major contributions to robot programming languages and has consulted for most major U.S. robot manufacturers.

Dr. Paul was one of the founding editors of the *International Journal of Robotics Research*. He served as a President of the IEEE Council on Robotics and Automation.

scription prices.