# Distributed Real-Time Control of a Spatial Robot Juggler

**Alfred A. Rizzi, Louis L. Whitcomb, and Daniel E. Koditschek**
**Yale University**

**This robot uses a distributed network of transputers to process stereo camera data and control the torque of a three-degree-of-freedom arm to juggle a ball.**

In our continuing research on dynamically dexterous robots, we recently completed construction of a second-generation juggling machine. Its forebear was a mechanically trivial system that used a single motor to rotate a bar parallel to a near-vertical frictionless plane. The system could juggle one or two pucks, which it sensed through a grid of wires and batted into a specified stable periodic motion.[1] Simplicity of design notwithstanding, this robot required a four-processor network as its controller.

Figure 1 shows the second-generation system. It uses three motors to actuate a three-degree-of-freedom direct-drive robot arm. The arm bats an artificially illuminated ping-pong ball into a specified periodic vertical motion. It senses the ball through a field-rate stereo-vision system. The distributed real-time control network for this system has grown to 17 processors.

Both the one- and three-degree-of-freedom machines exhibit a surprising ability to contend with unfavorable initial conditions and to recover from significant unforeseen disturbances. The second-generation system is one of the few programmable robots to achieve a level of dexterity in any way comparable to that of a human. (See the sidebar "Why Juggling?" for further reading in this field.)

The introduction of multiple balls will necessitate even more system nodes and more sophisticated control algorithms, which will further increase the network size. Thus, pursuing the theory and practice of "intelligent machine" design has thrown us into the exciting confusion of building practical distributed real-time controllers of increasing complexity.

This article describes the Yale spatial juggler. Further, based on this experience, we describe an emerging set of working principles for the design and implementation of embedded real-time distributed controllers. We are convinced that complex, coordinated motion-control applications will increasingly incorporate medium-grained event-driven parallel processing. We hope to attract a larger community of researchers and practitioners to examine the novel issues that arise in this context.

In particular, we wish to call attention to a programming style that substitutes event-driven dynamical processes and geometrical transformations for a more syntactically oriented, if-then-else approach. We call this style "geometric programming." Its recourse to low-level feedback loops in the pursuit of medium- or even high-level goals — uncommon as this may be within the traditional computing community — typifies the control engineer's approach. The robust and (partially)


0018-9162/92/0500-0012$03.00 © 1992 IEEE

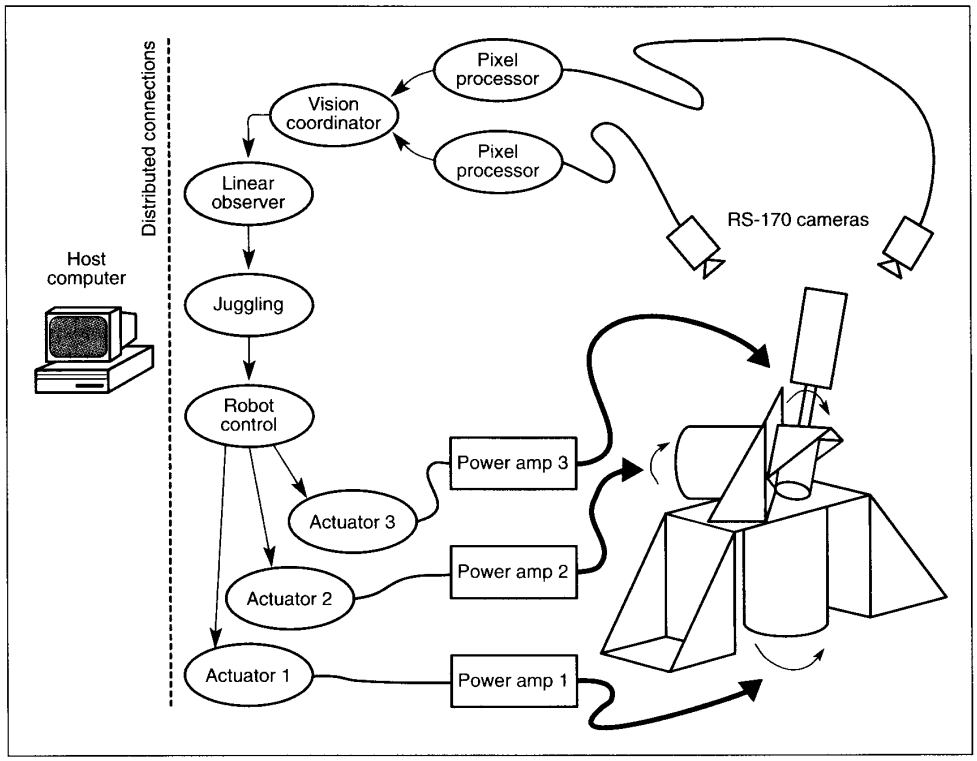



**Figure 1. The Yale spatial juggler.**

provably correct machine behavior that emerges may widen the attraction of such geometric programming techniques.*

## Overview of the juggling system

Our juggling algorithm is a direct extension of a new class of nonlinear feedback controllers, called "mirror laws," developed for use with our original planar juggling system[1] (that is, one where the ball is constrained to move in a plane). The algorithm takes the form of a mathematical expression that specifies robot position as a function of the ball's position and velocity. Thus, in the Yale spatial juggling system, data from the vision system flows directly through

*We use the word "correct" not only in the traditional sense of the relation between an algorithm and its hardware implementation but also in reference to the relation between an algorithm that may or may not have been correctly implemented in hardware and a behavior that the algorithm is supposed to elicit in the world. Our informal conviction is that geometric programming, when appropriate, will simplify formal analyses of correctness in both senses.

## Why juggling?

In brief, we have computers that play chess better than almost every human being, but haven't yet built a machine as capable of walking up the stairs or grabbing a cup as a toddler.

Starting with the pioneering work of Marc Raibert — whose book, *Legged Robots That Balance* (MIT Press, 1986), really got this field off the ground — a growing number of robotics researchers have begun to work in the general area of dynamically dexterous tasks. A forthcoming book, *Robot Control* (IEEE Press, 1992), edited by F.L. Lewis and M.W. Spong, contains a chapter devoted to the work of, among others, Russ Andersson on robot ping-pong and Tad McGeer on passive walking machines, as well as a lengthier introduction to the concerns and justification for this area of robotics by Daniel E. Koditschek, coauthor of this article.

These seemingly narrow investigations of machine dexterity raise problems that go to the heart of many open questions in autonomous-machine design. Further, the scope of these problems is narrow enough to admit of controlled and comparative experiments at the same time.

The complete treatment of our approach to juggling and the theoretical basis of its successes appears in a chapter titled "A Simple Juggling Robot: Theory and Experimentation," by M. Bühler, D.E. Koditschek, and P. J. Kindlmann, from *Experimental Robotics I*, edited by V. Hayward and O. Khatib, (Springer-Verlag, 1990). The exciting work of C.G. Atkeson and colleagues provides a contrasting view of the whys and hows of juggling (for example, see E.W. Aboaf, S.M. Drucker, and C.G. Atkeson, "Task-Level Robot Learning: Juggling a Tennis Ball More Accurately," *Proc. Int'l Conf. Robotics and Automation*, IEEE CS Press, Los Alamitos, California, 1989).

A broader technical understanding of the interdisciplinary field of robotics demands a working familiarity with a wide range of tools and concepts. A widely accepted introductory text, *Introduction to Robotics, Mechanics, and Control*, by John Craig (Addison-Wesley, 1986) provides a fine account of kinematics, dynamics, and elementary notions of control. There are many fine textbooks devoted to control itself, one of the more accessible and computationally oriented being *Digital Control of Dynamic Systems* by Gene Franklin, J. David Powell, and Michael Workman (Addison-Wesley, 1986).

the input channel of a memoryless nonlinear transformation whose output stream will, in turn, be treated as the input to a robot trajectory tracking scheme.

While there is by no means universal agreement in the field, a growing body of theory and number of working designs indicate that distributed processing represents the most viable solution to the expanding computational requirements of robotic systems. The relatively constant price of actuators and mechanisms contrasts dramatically with the advance of smaller, cheaper, and more powerful processors. In our view, this argues for the introduction of computational resources at ever lower levels of control as the most promising vehicle for increasing performance, even in motion-control applications that are less computationally demanding than juggling.

We believe that computational processes will inevitably be as distributed as the physical processes with which they are associated. Physical processes are coupled and their "communications" — that is, the effect of the state of one degree of freedom on another — are managed according to Newtonian dynamics. The control system designer's task amounts to providing complementary laws for managing communication between the computational processes. The Yale spatial juggler represents such an application of medium-grained processing power to rather simple local devices that are then "patched together" through the processor network to achieve a complex real-time function.

## Computational architecture for control

Our work in distributed real-time computation originated with the desire to build "merely" an easily reconfigurable and incrementally cheap family of control engines that would let us get on with the "real work" of robotics. We quickly discovered that the very ubiquity of design that facilitated interconnection led to a potentially bewildering variation in network behavior depending on apparently innocent changes in topology and communication protocols. This section explores some of the understanding we have derived from our experience in building computer networks for control.

**Computational models.** Having embraced a parallel controller implementation, we had to answer the question of what kind. Two widely accepted paradigms of concurrent computation are the shared memory (SM) model and the communicating sequential process (CSP) model. These models provide fundamentally different mechanisms for interprocess communication. In the CSP model, interaction between multiple processes takes place over private communication *channels*, while an SM-based system uses shared variables to *broadcast* information. Most multiprocessor systems are roughly structured according to one of these two popular concurrent computation models,[2] although systems that combine the two models at different levels of granularity are beginning to emerge.

In 1986 we designed a generic motion-control system using the CSP model. To avoid the cost associated with developing an architecture from scratch, we based the system on the SGS-Thomson/Inmos family of transputers. In addition to its commercial availability, the Inmos transputer family offered a stable and mature development environment, whose operation closely corresponded to the CSP model.[3]

Nearly six years later, we continue to benefit from this decision. In particular, the CSP model as instantiated by the Inmos transputer line has the following advantages:

- *Expandability.* The ease with which individual processors can be introduced to an overburdened segment of an existing network allows our computational capacity to grow with task complexity. Similarly, the simplicity with which existing mechanical or sensory subsystems can be integrated by hardware links and software channels has significantly accelerated the development of increasingly complex robotic systems in our laboratory.
- *Extendable interprocessor communication bandwidth.* Simple high-speed serial data links between multiple processors allow the total network data bandwidth to grow as the network grows — there is no fixed-bandwidth shared bus. The control structures we are developing do not require the broadcast of global information to every processor. Thus, the apparent weakness of being unable to broadcast messages to a large collection of processes is not a drawback for our applications.
- *Device I/O transmission and isolation.* Explicit control over the distribution of data reduces the required total network data rate and permits the use of lower bandwidth data links. These lower bandwidth data links are, in turn, easily "hardened" for use in communicating with robot I/O devices, which are often physically distant and electrically distinct.

**Controller hardware.** The Inmos product line supports a strategy that standardizes and places the burden of parallelism — interprocessor communications support, software, and development environment — on a commercial product, while customizing the computational identity of particular nodes by recourse to special-purpose hardware. For all our control experiments, we use the XP/DCS (transputer-based distributed control system) boardset, a line of I/O and memory customized transputer nodes originally developed within the Yale Robotics Laboratory and now commercially available from Evergreen Design of Branford, Connecticut.

The XP/DCS processor board is based on the Inmos T800 processor, a 32-bit scalar processor capable of 10 MIPS and 1.5 Mflops (sustained) with four bidirectional 20-MHz DMA-driven communication links, 4 Kbytes of internal (one-cycle) RAM, and hardware primitives for intraprocessor communication, and is capable of submicrosecond context switching.

This processor is augmented with an additional 1 to 8 Mbytes of dynamic RAM (three-cycle) and an I/O connector that presents the T800's bus to a daughterboard. The computational identity of a particular processor can then be customized by attaching different daughterboards to this connector.

Combining an I/O module daughterboard with an XP/DCS allows the processor to interface to custom hardware in a simple, standard fashion. The I/O module augments an XP/DCS by providing a 32-bit latched bidirectional data bus, six 4-bit-wide digital output ports, and eight digital input signals. All these ports are mapped directly into the T800's memory space. In practice, system-specific I/O devices (for example, digital-to-analog and analog-to-digital converters, quadrature decoders/counters, and watchdog circuits) are directly connected to the 32-bit bus with their control

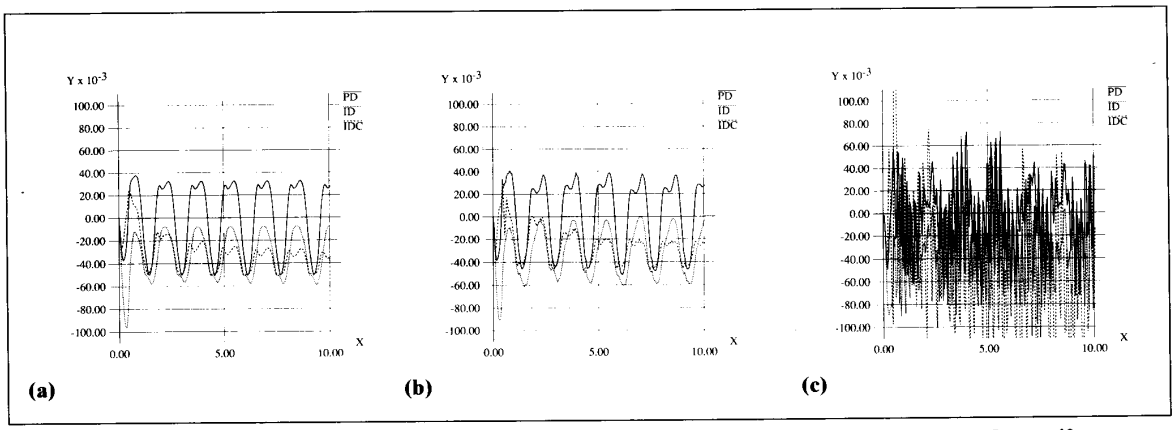



**Figure 2. The destabilizing effects of sampling versus latency: (a) update = 1 ms, latency = 1 ms, (b) update = 40 ms, latency = 1 ms, and (c) update = 1 ms, latency = 40 ms.**

signals driven by the digital outputs. This approach simplifies the custom hardware associated with a particular sensor or actuator at the cost of some additional software to perform low-level-device access.

Much like the I/O module, the Cyclops vision system boardset augments a set of XP/DCS motherboards in support of a particular sensing task. The vision system includes three major components:

- A digitizer that accepts an incoming RS-170 video signal and outputs it in digital form over a pixel bus.
- A filter board capable of performing real-time two-dimensional convolution on an image. This board is placed on the pixel bus. It accepts "commands" over a transputer link and manages image-filtering tasks, while introducing minimal additional latency.
- A memory daughterboard augments an XP/DCS with 128 kilobytes of video memory. By associating up to eight memory boards with a pixel bus, constructing a real-time parallel processing vision system becomes easy.

**Controller performance.** The gap between continuous plant and digital discrete controller models is familiar to every practicing control engineer. Certain new issues arise when the controller model allows concurrency.

The notions of *latency* and *update period* are familiar from their association with a simple sequential model of computation. In a simple concurrent model of computation, the system reads from $n$ inputs, computes an expression, and writes to $m$ outputs concurrently. For this class of computations, we refer to the interval between the system reading an input $i$ and seeing its effect at output $j$ as the $i-j$th *cross latency* of the computation. In general, such a system will have $n \times m$ cross latencies. If the computation is performed repetitively, we refer to the interval of time between successive results at the $j$th output to be the *update period* of the computation.

*Performance degradation due to untimely signals.* The critical dependence of closed-loop system stability on controller latency and update properties distinguishes the design of embedded computer control systems from their more familiar off-line counterparts. The latter need only to optimize some more abstract measure of cost (throughput, processor utilization, etc.). Delays in embedded control systems, whether in update rate or latency, will always degrade performance.

Figure 2 compares the effect of update rate and latency on robot tracking performance. These graphs show a robot's joint tracking error as a function of time for three different control algorithms with identical sinusoidal reference trajectories. (The algorithms — proportional derivative (PD) and two versions of computed torque (ID and IDC) — are described briefly in the section "Robot dynamics" and in detail elsewhere.[4]) A 40-millisecond degradation in update rate, graph (b), has relatively little effect on tracking performance, while a 40-millisecond degradation in latency, graph (c), severely compromises tracking performance and nearly destabilizes the system. Indeed, further degradation in latency resulted in controller instability.

Sampling theory for nonlinear dynamical systems is problematic,[5] and tools suitable for understanding the effects of network latency and update rate caused by asynchronous distributed processing on closed-loop control systems may now be available, but have yet to be applied.[6] In consequence, we do not really know when successive increments of cross latency built up along a network path will begin to destabilize an otherwise perfectly viable closed-loop robot system.

*Sensitivity of signal timeliness to communications schemes.* The foregoing discussion of how much latency is acceptable for system operation presumes an ability to design computational structures that meet latency specifications. Unfortunately, a body of theory capable of predicting the performance (cross latency) of distributed computational systems is only now emerging.[7] In any multiprocessor system the timeliness of computations throughout a network depends on both communication and computation costs. In turn, the communication costs critically depend on both the network topology and the interprocessor buffering paradigms employed.

As an example of the subtleties lurking in this arena, let's examine some of the data from our studies of cross latency on various transputer networks. One implementation had each servo processor communicating directly to two pro-

**Table 1. Cross latency measurements for three buffering schemes (μs).**

| Unbuffered | | | |
|---|---|---|---|
| Node | 1 | 2 | 3 |
| 1 | 817 | 706 | 823 |
| 2 | 982 | 832 | 930 |
| 3 | 860 | 731 | 829 |
| **Latest Buffering** | | | |
| Node | 1 | 2 | 3 |
| 1 | 995 | 1,320 | 1,378 |
| 2 | 1,166 | 1,021 | 1,270 |
| 3 | 1,422 | 1,296 | 967 |
| **Unbuffered and Latest Buffering** | | | |
| Node | 1 | 2 | 3 |
| 1 | 598 | 1,456 | 1,718 |
| 2 | 1,486 | 572 | 1,457 |
| 3 | 1,707 | 1,457 | 574 |

cessors: a dedicated computation processor and a dedicated communication server processor, all communicating in a ring topology. Identical code was distributed over this network for three cases that differed only in the type of internode buffering scheme employed. Table 1 shows matrices of actual cross-latency measurements for this implementation. The control network has three inputs and three outputs corresponding to the respective robot actuator processors. These data underscore the significance of buffering strategy in determining the communication costs of a given network topology.

*Current strategy for managing signal timeliness.* Precise control of cross latency in currently available distributed processing systems will no doubt remain a murky subject for some time to come. In the absence of a practicable theory, we have been forced to design our network topology and buffering schemes based on simple rules of thumb. Typically, we use unbuffered (blocking) communication within "local" modules that we presume to require synchronization and low latency. Communication between these modules, where latency is less of an issue, is asynchronous. This both allows for independent module development and protects against deadlock between multiple connected network elements. When we refer to asynchronous buffer processes, we mean a process that presents at its output the most recent information it has received at its input (latest buffering).

## Designing computational architectures

Both control and software engineers design systems by developing block diagrams that describe the flow of data in networks. Distributed systems require the additional design of a network topology suitable for instantiating this flow. Coordinating these design stages poses a significant problem for the system integrator. We have adopted the commonsense strategy of trying to match the network topology and code distribution as closely as possible to the control theoretic block diagram.

**Dataflow in a control system.** Transcending the specific details of any particular block diagram, three types of data appear to distinguish themselves in every control system. First are the data that pass between the designed controller and the physical plant under control — the internally bound sensor readings and the externally bound commands. Second, arising in the context of distributed implementations, are the signals passed from one portion of a network to another, required to complete computations of the first type. Finally, there is information concerning the system's state, which interests the user or performance analyst but has no intrinsic value with respect to the control task itself.

One might loosely associate update as a measure most relevant to the first data type, cross latency to the second type, and mere throughput to the last.

At the periphery of a control network are the I/O processes that handle the interface to the physical world. Each external device asserts an individual physical interface specification neces-

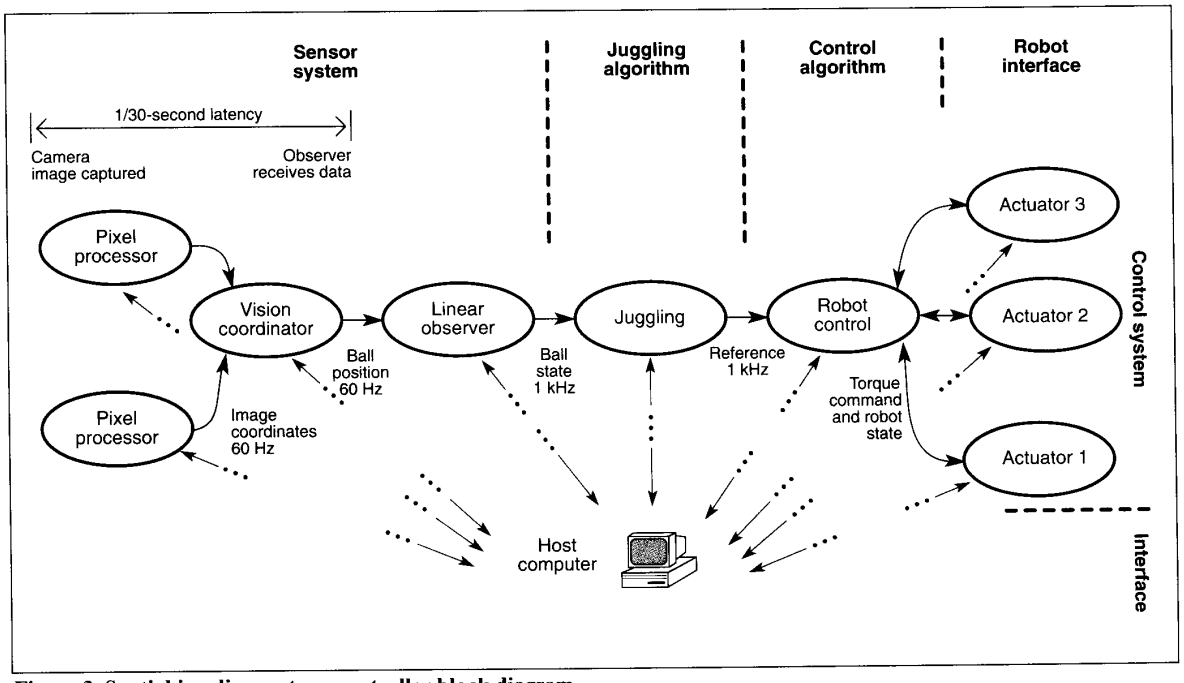



**Figure 3. Spatial juggling system: controller block diagram.**

sarily including some notion of sampling rate — that is, the frequency at which the device must be serviced. Our controllers are specifically constructed to permit isolating this externally driven dataflow in hardware separate from the remainder of the network. In particular, the I/O board is used to move moderate-bandwidth signals between the transputer's memory and the mechanical devices while the Cyclops boards acquire the high-bandwidth video signals from the sensing device.

From the viewpoint of the remainder of the system, an I/O node is simply another processor that has been customized to present an abstract version of the physical device to the system. This abstraction of an external device via processor customization is, of course, merely an extension to hardware of the familiar notion of limiting the scope of data within a process.

We further distinguish data types by how they are used: *Intrinsic* data are required as part of the control task itself, and *extrinsic* data are created and processed to facilitate system debugging and performance evaluation. Our hardware does not offer separate channels for these two data types. However, by paying proper attention to network topology and — in some cases — at the expense of adding more nodes simply to increase the available number of internode links, we have been able to use the transputer's on-board DMA engines to good effect in creating an intrinsic network that is effectively orthogonal to the coexisting extrinsic network.

The fundamental challenge for the system designer is to guarantee that intrinsic data is handled in a timely and efficient manner. As discussed in the previous section, the requirements for this are imperfectly understood, so we have resorted to commonsense strategies for network topology as well as buffering schemes.

Additionally, the design and development process must provide sufficient extrinsic data to monitor system performance. In particular, a more general-purpose communications network is needed for delivering operator commands and returning status/logging information to the host computer system for later analysis. Since there are no real-time issues involved, we find it practical to employ unbuffered blocking communications schemes. Because the real-time dataflow must meet this extrinsic network at some level, it is important to take precautions to minimize the impact on the real-time aspects of the system. Our solution to this dilemma is to provide a single "port" (implemented as a process) into each logical unit of the "control system" through which operation can be monitored and commands issued.

**Processor location and process assignment.** Given an engineering block diagram derived from the control viewpoint, the task remains of designing its network implementation and assigning processes to the various network processors. The most obvious strategy — to match processors (or groups of processors) and their interconnections to the engineering block diagram as much as possible — yields at least two benefits. First, the assignment of processes becomes equally obvious: The processors corresponding to a particular block in the engineering diagram get assigned the software processes associated with the block's function. Second, a natural abstraction of function results from details corresponding to the logic of the control scheme itself. In other words, each block may be construed as a distinct network with its own (hidden) lower level of intrinsic dataflow, contributing only via its I/O data to the intrinsic dataflow of the higher level network.

Figure 3 depicts the four major components of our juggling controller:

- a sensor system (stereo vision coupled with a signal processing system),
- a juggling algorithm (a memoryless nonlinear transformation),
- a robot control system (an inverse dynamics algorithm), and
- a robot interface.

Figure 4 portrays the hardware wiring diagram of the transputer network that implements the controller. The granularity of our transputer boards affords a matchup of roughly one or two pro-

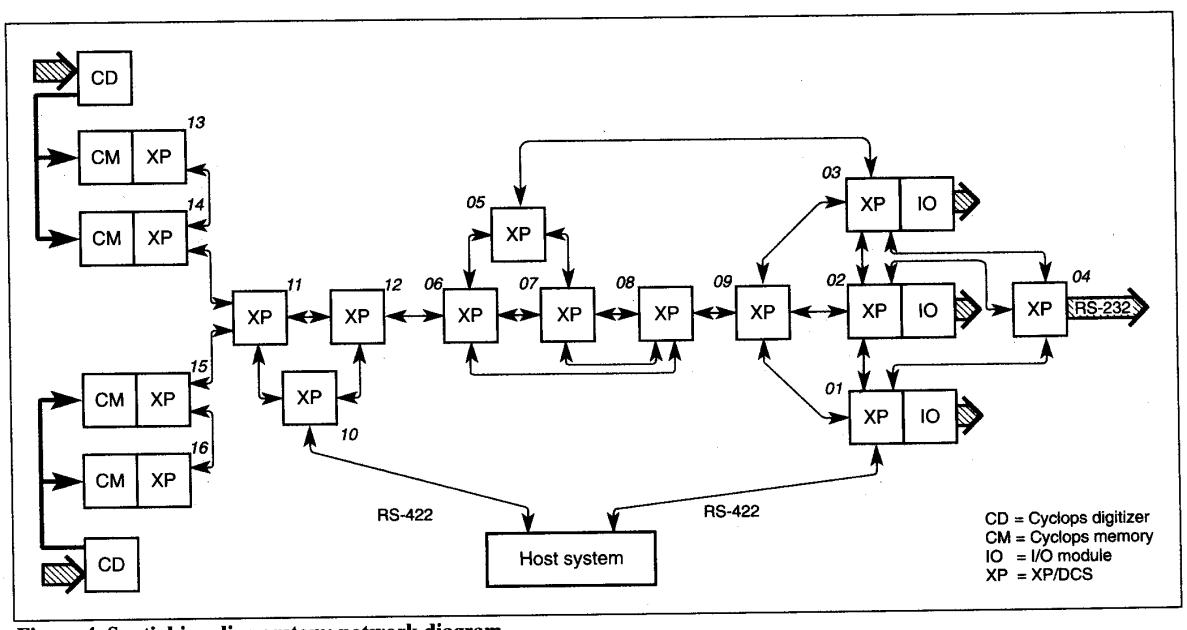



**Figure 4. Spatial juggling system: network diagram.**

cessors to each physical device. These devices, now represented as logically uniform nodes, are patched together via geometric and dynamical transformations implemented on intermediate nodes. The resulting network coordinates the reactions of the robot arm to the flight pattern of the ball so that the juggling behavior emerges reflexively.

The remainder of this article describes the manner in which the processes depicted in Figure 3 are associated with the processors depicted in Figure 4.

## Juggling environment: Striking a ball in flight

Our approach to juggling requires the controller to predict the behavior of the ball. Its two relevant properties are flight dynamics (how it behaves away from the paddle) and impact dynamics (how it interacts with the paddle/robot).

The mathematical models of these properties influence the design of both the sensing system and the juggling algorithm.

**Modeling the flight dynamics.** For simplicity, we chose to ignore the effects of friction, spin, and other aerodynamic properties and to model the ball's behavior as that of a point mass. Classical mechanics teaches us that a point mass under the influence of gravity obeys the following set of differential equations:

$$\ddot{b}_x = 0$$
$$\ddot{b}_y = 0$$
$$\ddot{b}_z = -\gamma$$

where $b_x$, $b_y$, and $b_z$ are the $x$, $y$, and $z$ position of the ball respectively and $\gamma$ is the acceleration due to gravity (9.8 meters/second/second). This model of the ball's behavior may also be expressed in *state space* notation as

$$\begin{bmatrix} \dot{\mathbf{b}} \\ \ddot{\mathbf{b}} \end{bmatrix} = \begin{bmatrix} 0 & I \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{b} \\ \dot{\mathbf{b}} \end{bmatrix} + \begin{bmatrix} 0 \\ \mathbf{g} \end{bmatrix} \qquad (1)$$

where $\mathbf{b}$ is a three-vector ($\mathbf{b} \in IR^3$) representing the position of the ball, $\dot{\mathbf{b}}$ (the ball's velocity vector) is the time derivative of $\mathbf{b}$, $\mathbf{g} = (0,0,-\gamma)^T$ is the acceleration vector experienced by the ball due to gravity, and $I$ is the identity matrix.

**Modeling the impact dynamics.** The ball's impact behavior is modeled by a simplistic (but standard) coefficient of restitution law, where the components of the ball's velocity tangent to the plane of impact are assumed to remain unchanged by the impact. For some positive *coefficient of restitution*, $\alpha$, this model can be expressed as

$$(\dot{b}'_n - v'_n) = -\alpha(\dot{b}_n - v_n) \qquad (2)$$

where $\dot{b}'_n$ and $v'_n$ denote the normal components of the ball and paddle velocities immediately after impact, while $\dot{b}_n$ and $v_n$ are the velocities prior to impact. Assuming that the paddle is much more massive than the ball (or that the robot has large torques at its disposal), we conclude that the paddle's velocity will remain constant throughout the impact ($v'_n = v_n$). It follows that the coefficient of restitution law can now be rewritten as

$$\dot{b}'_n = \dot{b}_n + (1+\alpha)(v_n - \dot{b}_n) \qquad (3)$$

## Vision: Sensing the environment

In our present implementation we have structured the visual environment to make the task of locating the ball conceptually trivial and computationally tractable. In particular, the vision system needs only to extract the three-

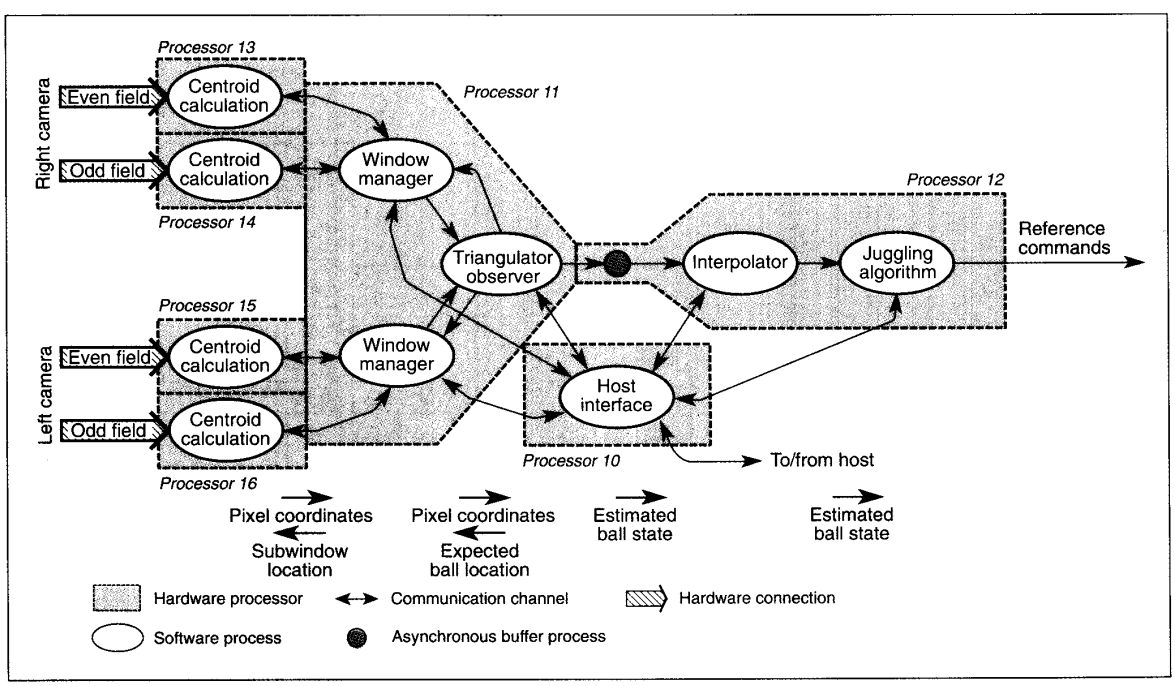



**Figure 5. Process diagram for the Cyclops vision system.**

dimensional position of a single brightly illuminated white ball against a black background.

To perform even this apparently trivial task in real time requires two cameras, one for each "eye," as shown in the Figure 5 diagram of our Cyclops vision system. Each of these systems utilizes a single video digitizer that services two video memory boards, each with an associated XP/DCS processor. These four processors perform the low-level vision processing at field rate (60 Hz) as follows.

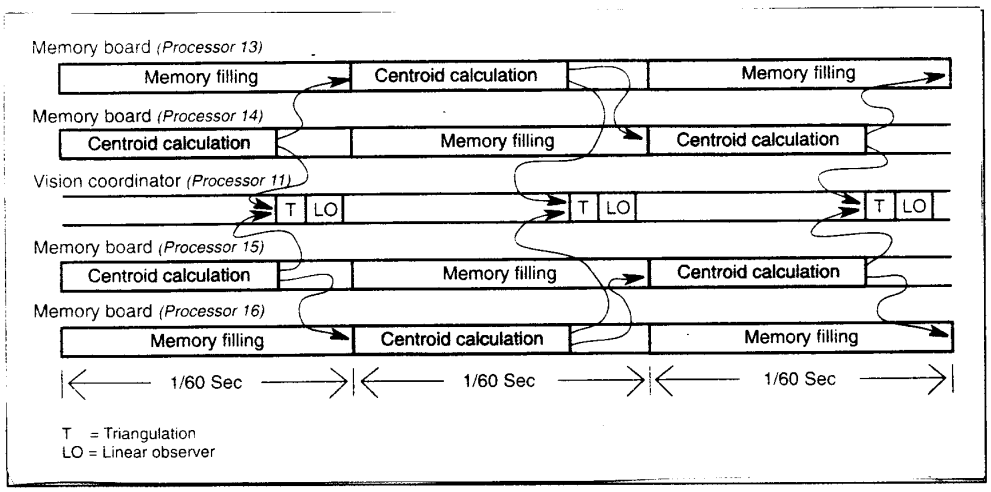



**Figure 6. Detailed timing diagram for the stereo Cyclops vision system.**

At each camera, one processor extracts data from an image while the digitizer fills the frame memory associated with the other processor. The data-extraction procedure includes computing the centroid (in pixel coordinates) of the ball in the image via a simple first-order moment computation. Despite the simplicity of this computation, we do not have sufficient computational capacity to examine the entire image within the 16 milliseconds before the next field arrives. Therefore, only a moderate-sized region of the image is actually processed. Computational limitations currently allow the system to process windows of fewer than 3,000 pixels from an image of 123,392 pixels (a 241 × 512 image).

Once the low-level pixel processors have determined the ball location in a pair of images, the system uses stereo triangulation to locate the ball's position in space with respect to a fixed reference coordinate system. Having extracted this location, it is necessary to interpret the data and present it in a way the remainder of the system can use. In this context, "interpret" means producing smooth and timely estimates of the ball's position and velocity. This is accomplished by passing the camera system's output through a filter that implements a standard linear observer and some less orthodox prediction functions concerning the discontinuities introduced by impacts, as modeled by Equation 3.

**Linear observers.** An observer is a filter specially designed to estimate the unknown state of a known linear dynamical system from I/O measurements only. A time-sampled model of the free-flight ball dynamics (Equation 1) is given by

$$\begin{aligned} &\mathbf{x}_{i+1} = A\mathbf{x}_i + \mathbf{b} \quad A = \begin{bmatrix} I & tI \\ 0 & I \end{bmatrix} \\ &\mathbf{b}^T = \begin{bmatrix} \frac{t^2}{2}\mathbf{g}^T & t\mathbf{g}^T \end{bmatrix} \\ &\mathbf{y}_i = \mathbf{c}^T\mathbf{x}_i \qquad \mathbf{c}^T = \begin{bmatrix} I & 0 \end{bmatrix} \\ &\mathbf{g}^T = \begin{bmatrix} 0 & 0 & -\gamma \end{bmatrix} \end{aligned} \tag{4}$$

where $\mathbf{x} \in IR^6$ (the position and velocity of the ball), while $\mathbf{y} \in IR^3$ is only the position, and $t$ is the sampling rate (1/60-second for our vision system). Passing $\mathbf{y}_i$ through a filter defined by

$$\begin{aligned} \hat{\mathbf{x}}_{i+1} &= A\hat{\mathbf{x}}_i + \mathbf{b} + \mathbf{k}(\mathbf{y}_i - \hat{\mathbf{y}}_i) \\ \hat{\mathbf{y}}_i &= \mathbf{c}^T\hat{\mathbf{x}}_i \end{aligned} \tag{5}$$

leads to an "error" system given by

$$\mathbf{e}_{i+1} = (A + \mathbf{k}\mathbf{c}^T)\mathbf{e}_i \tag{6}$$

where $\mathbf{e}_i \triangleq \hat{\mathbf{x}}_i - \mathbf{x}_i$. It can be shown that if the pair $(A,\mathbf{c}^T)$ satisfies a particular algebraic property (observability), then a vector of gains, $\mathbf{k}$, can be found such that all the eigenvalues of $(A + \mathbf{k}\mathbf{c}^T)$ lie within the unit circle. Thus, the system described by Equation 3 will have solutions that asymptotically approach zero, and we conclude from the definition of $\mathbf{e}_i$ that $\hat{\mathbf{x}}$ approaches $\mathbf{x}$. Note that the well-known Kalman filter is a linear observer whose gain matrix, $\mathbf{k}$, has been chosen to guarantee a particular type of optimal behavior.

**Timing synchronization.** Figure 6 presents a timing diagram that shows the vision system adding an unavoidable 1/30-second delay between the time an image is captured and the time a spatial position measurement has been formed. Since the ball's flight model is a sufficiently simple dynamical system, its future behavior can easily be predicted with reasonable accuracy by simple integration. Thus, the unavoidable delay introduced by the camera can be compensated for by integrating the delayed estimates forward in time.

Finally, our juggling algorithm (presented later in the section titled "Juggling") is based on continuous ball information. Therefore, we must step up the slow data rate of the vision system (60 Hz) to a rate well over the robot system's bandwidth. Integrating the current-state estimate of the ball forward over small time steps allows an increase in the data rate from 60 Hz to 1 kHz.

Note that the synchronous nature of these calculations, all of which are carried out on processor 11 in Figure 4, ends at the buffer process located between the observer and the interpolator. This transition from purely synchronous processing allows the data rates through the remainder of the system to be adjusted independent of the camera.

# Plant: A three-degree-of-freedom direct-drive robot

At the heart of the juggling system resides a three-degree-of-freedom direct-drive robot — the Bühgler arm — equipped at its end effector with a paddle. Three properties of a robot directly influence the design of its controller:

- kinematics, the geometric relations between the structural members of the machine;
- dynamics, the temporal behavior of the machine in response to actuator inputs; and
- its physical interface.

**Robot kinematics.** A robot's kinematics describe the position and orientation of the end effector, or gripper, as a function of the positions of its joints. The geometric properties are specified for the entire mechanism — that is, the position and orientation of each *link* in the three degrees of freedom are specified with respect to a fixed *world* frame of reference. Using the conventions of Craig, frames of reference are attached to each link of the robot, as shown for the Bühgler's arm in Figure 7.

These frames of reference are used to derive the Denavit-Hartenburg parameters as a formal description of robot kinematics. Table 2 presents the Denavit-Hartenburg parameters for the Bühgler arm. These parameters appear in a corresponding set of frame transformations that define the relationships between the frames as a function of the joint variables ($q_1$, $q_2$, and $q_3$).

$$ {}^0H_1 = \begin{bmatrix} \cos q_1 & -\sin q_1 & 0 & 0 \\ \sin q_1 & \cos q_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^1H_2 = \begin{bmatrix} \cos q_2 & -\sin q_2 & 0 & 0 \\ 0 & 0 & 1 & l \\ -\sin q_2 & -\cos q_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^2H_3 = \begin{bmatrix} \cos q_3 & -\sin q_3 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \sin q_3 & \cos q_3 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

For the geometrically simple (flat) paddle on the Bühgler arm, it is convenient to define two additional "joint variables," $s_1$ and $s_2$, associated with the unactuated degrees of freedom created by the paddle surface. The position of the robot's "virtual gripper" can then be expressed in the third link's frame of reference as

$$ {}^3\mathbf{p}_g = \begin{bmatrix} 0 \\ s_1 \\ s_2 \\ 1 \end{bmatrix} $$

and in the world frame as shown in Figure 8 (Equation 7).

In a similar fashion the unit normal to the paddle may be expressed as

$$ {}^3\mathbf{n}_g = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} $$

which results in

$$ {}^0\mathbf{n}_g = \begin{bmatrix} \cos q_1 \cos q_2 \cos q_3 - \sin q_1 \sin q_3 \\ \cos q_2 \cos q_3 \sin q_1 + \cos q_1 \sin q_3 \\ -(\cos q_3 \sin q_2) \\ 0 \end{bmatrix} \quad (8) $$

In practice, we generate all these expressions by specifying the parameters in Table 2 and using a commercially available symbolic math package (Mathematica from Wolfram Research) to derive and simplify the descriptions for ${}^0\mathbf{p}_g$ and ${}^0\mathbf{n}_g$.

The *inverse kinematics* of a robot arm specify the values of the joint variables required to realize some specific grip-

**Table 2. Denavit-Hartenburg parameters for the Bühlger arm of Figure 7.**

| Link No. | Length ($a_{i-1}$) | Twist ($\alpha_{i-1}$) | Height ($d_i$) | Variable ($\theta_i$) |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | $q_1$ |
| 2 | 0 | $-\pi/2$ | $l$ | $q_2$ |
| 3 | 0 | $\pi/2$ | 0 | $q_3$ |

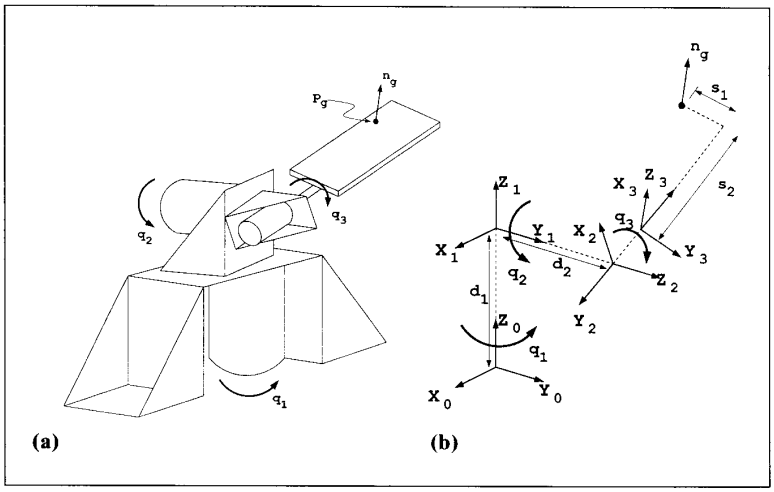



**Figure 7. The Bühgler arm (a) and its kinematics (b).**

$$ {}^0\mathbf{p}_g = \begin{bmatrix} -(l\sin q_1) + (-(\cos q_3 \sin q_1) - \cos q_1 \cos q_2 \sin q_3)s_1 + \cos q_1 \sin q_2 s_2 \\ l\cos q_1 + (\cos q_1 \cos q_3 - \cos q_2 \sin q_1 \sin q_3)s_1 + \sin q_1 \sin q_2 s_2 \\ \sin q_2 \sin q_3 s_1 + \cos q_2 s_2 \\ 1 \end{bmatrix} \quad (7) $$

**Figure 8. Position of robot's "virtual gripper" in the world's frame of reference.**

per position and orientation. By deciding that we will use only the middle of the paddle ($s_1 \equiv 0$), we can easily construct the inverse kinematics for Equation 7 as

$$\begin{bmatrix} \phi \\ \xi \\ \psi \\ \zeta \end{bmatrix} \triangleq p^{-1}(\mathbf{b}) = \begin{bmatrix} -\arccos\left(\frac{l}{\sqrt{b_1^2+b_2^2}}\right) + \arctan(b_2/b_1) \\ -\arctan\frac{\sqrt{\mathbf{b}^T\mathbf{b}-l^2}}{b_3} \\ 0 \\ \sqrt{\mathbf{b}^T\mathbf{b}-l^2} \end{bmatrix} \quad (9)$$

**Robot dynamics.** The generally accepted Lagrangian rigid-body model of robot arm dynamics takes the form

$$M(\mathbf{q})\ddot{\mathbf{q}} + C(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}} + g(\mathbf{q}) = \tau \quad (10)$$

where $\mathbf{q} = (q_1, q_2, q_3)^T$ is the vector of joint variables, $M(\mathbf{q})$ arises from the inertial properties of the links, $C(\mathbf{q},\dot{\mathbf{q}})$ arises from the coriolis and centripetal forces of motion, $g(\mathbf{q})$ arises from the effect of gravity, and $\tau$ represents the forces applied to the joints by the motors. (See the sidebar for more information on the computational burden of robot dynamics.)

It can be shown that if the robot were to track exactly the "reference signal" (see the section, "Juggling"), then collisions with the ball would occur in such a fashion that the desired periodic motion is asymptotically achieved.[1] Thus, it falls to the robot controller to ensure that the joint angles, $\mathbf{q}(t)$, track the reference signal, $\mathbf{q}_d(t)$.

*Nonlinear feedback controllers for robot trajectory tracking.* Given a desired robot trajectory $(\mathbf{q}_d, \dot{\mathbf{q}}_d)$, one traditional approach to robot control employs a proportional-derivative (PD) feedback control law

$$\tau = K_d(\dot{\mathbf{q}} - \dot{\mathbf{q}}_d) + K_p(\mathbf{q} - \mathbf{q}_d) \quad (11)$$

The PD control law corrects the inputs to a plant based upon the differences between the desired and actual positions and velocities, then obtains actuator torque commands based on amplifying these differences by the fixed gain matrices, $K_p$ and $K_d$, for positions and velocities, respectively. All Lagrangian dynamical systems, including robot Equation 10, fall within the class of nonlinear systems for which PD controllers are guaranteed to produce asymptotically small errors[8] whose magnitude is inversely proportional to that of the gains. Because of the computational simplicity obtained when $K_p$ and $K_d$ are diagonal, the PD control law (often with an additional integral term) is employed, almost without exception, in all existing industrial robots.

This control law, however, ignores the complicated dynamics of the robot arm and may deliver poor high-speed performance. It is never practically possible to raise the gains past some empirically determined level. Thus, the theoretically guaranteed "small" steady-state errors corresponding to stable gain magnitudes may not be small enough with respect to the job at hand.

As of this writing, the most widely discussed algorithm in the robot control literature employs a variant of a second traditional approach to trajectory tracking — the so-called "inverse dynamics" point of view — that has become known as the "computed torque" algorithm.[9] Examination of one version of the computed torque controller

$$\tau = C(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}} + M(\mathbf{q})[\ddot{\mathbf{q}}_d] + K_d(\dot{\mathbf{q}} - \dot{\mathbf{q}}_d) + K_p(\mathbf{q} - \mathbf{q}_d) \quad (12)$$

reveals that the technique proposes to *exactly cancel* the complicated nonlinear terms in Equation 10 and to use the traditional engineering technique

## Computational burden arising from robot dynamics

The simplicity of Equation 10 belies the underlying complexity of the model. For the Bühgler arm, for example, the (1,1) term of the matrix $M(\mathbf{q})$ takes the form

$$1.365 + 0.1408\cos(q_2)^2 + 0.0456\cos(q_3) + 0.0317\cos(q_3)^2 + 0.00507\cos(q_2)^2\cos(q_3)^2 + 0.02099\cos(q_2)\sin(q_2) + 0.00373\cos(q_2)\cos(q_3)\sin(q_2) + 0.7242\sin(q_2)^2 + 0.0105142\sin(q_3) - 0.0747\cos(q_2)\sin(q_2)\sin(q_3) + 0.0236\cos(q_3)\sin(q_2)^2\sin(q_3) + 0.00507\sin(q_3)^2 + 0.0317\cos(q_2)^2\sin(q_3)^2$$

while the (1,3) term of the matrix $C(\mathbf{q},\dot{\mathbf{q}})$ is

$$(0.0105\cos(q_3) - 0.0747\cos(q_2)\cos(q_3)\sin(q_2) + 0.0236\cos(q_3)^2\sin(q_2)^2 - 0.0456\sin(q_3) - 0.0532\cos(q_3)\sin(q_3) + 0.0532\cos(q_2)^2\cos(q_3)\sin(q_3) - 0.0037\cos(q_2)\sin(q_2)\sin(q_3) - 0.0236\sin(q_2)^2\sin(q_3)^2\,\dot{q}_1 + (-0.001866\cos(q_2)\cos(q_3) - 0.0228\cos(q_3)\sin(q_2) - 0.02663\cos(q_3)^2\sin(q_2) + 0.03737\cos(q_2)\sin(q_3) - 0.00525\sin(q_2)\sin(q_3) - 0.04731\cos(q_3)\sin(q_2)\sin(q_3) + 0.02663\sin(q_2)\sin(q_3)^2)\,\dot{q}_2 + (0.005257\cos(q_2)\cos(q_3) - 0.03737\cos(q_3)\sin(q_2) - 0.0228\cos(q_2)\sin(q_3) - 0.001866\sin(q_2)\sin(q_3))\,\dot{q}_3$$

As in the case of the kinematics, we generate these expressions using Mathematica.

These expressions are so complicated that one might imagine them representing merely second-order effects influencing the robot's behavior only slightly. On the contrary, they represent coupling forces that typically vary (reasonably quickly) over several orders of magnitude in the joint variables. This accounts for our recourse to advanced control techniques that ensure adequate tracking of the challenging reference signals generated by the juggling algorithm.

To understand the practical impact this has on controller design, consider a typical servo motor — a device that has a time constant on the order of 10 milliseconds. While the Nyquist sampling rate suggests running the controller at a rate that merely doubles the fastest frequency of the system to be controlled, the more common engineering practice is to run at a rate that exceeds the fastest by an order of magnitude. Thus, a 1-kHz sampling frequency is considered reasonable for robot control.

Expressions of the form presented above incur roughly $10^3$ floating-point operations per second (flops) for each degree of freedom. Presuming each degree of freedom might be allocated a processor, as argued in the introduction, we conclude that the baseline grain size of 1 Mflops is appropriate.

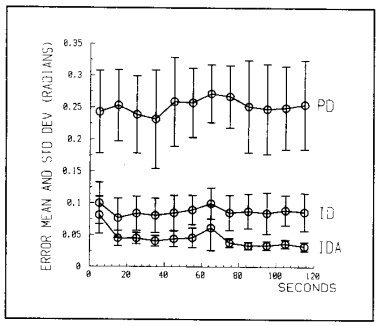



**Figure 9. Bühgler position-error mean, plus/minus one standard deviation, of 10 different reference trajectories computed at 10-second intervals for three controllers.**

of pole placement on the resulting error dynamics.

This algorithm has the advantage of promising excellent tracking performance — one can show that the tracking errors eventually vanish. It has two disadvantages. First, since it includes a full model of the robot dynamics (Equation 10), it is computationally intensive and has been only recently implemented exactly in real time. Second, the algorithm requires exact knowledge of both the robot kinematic and inertial parameters. Unfortunately, inertial parameters are usually unknown and difficult to obtain. Without exact robot inertial parameters, the promised advantage of this controller is compromised. To correct deficiencies of the computed torque control algorithm in the presence of unknown inertial parameters, several researchers[10] have proposed algorithms that "learn" the robot's inertial parameters.

We have conducted experiments to determine the worth of these model-based and adaptive-model-based computed torque algorithms in comparison to the computationally much simpler PD law.[4] Figure 9 summarizes the typical pattern of our results. This graph presents the means and variances in tracking errors resulting from 10 very different reference trajectory inputs that were switched abruptly (at 60 seconds) from one qualitative type (that is, fast and at the top of the work space) to a very different type (that is, slow and in the middle of the work space).

The statistics suggest that the model-based controller in Equation 12 (labeled ID in the plot) improves performance over the PD law by at least a factor of two and that the adaptive version (labeled IDA in the plot) improves steady-state performance by about a factor of two over its fixed-parameter counterpart. Moreover, this relative ordering is not destroyed during the transient period of adjustment to the new reference signal. We conclude that there is good reason to introduce the more computationally intensive model-based controllers.

*Implementation of the model-based robot controllers.* Currently, all physical experiments use a robot control subsystem that has five processors, as shown in Figure 10. Small input buffer processes implement nonblocking communication between these processors. The buffer processes, situated at the inputs to each computational process, allow the various controller elements to receive data from each other and other elements of the system asynchronously. In point of fact, the present controller block topology represents a convenient developmental benchmark rather than our best understanding of how to organize this critical portion of the overall system. Earlier off-line studies of various controller subnetwork topologies, discussed briefly in the earlier subsection on controller performance, have shown that this kind of code distribution scheme often exhibits relatively poor latency properties. Future versions of the controller block are under development and will employ the more efficient subnetwork topology described in Bühler et al.[11]

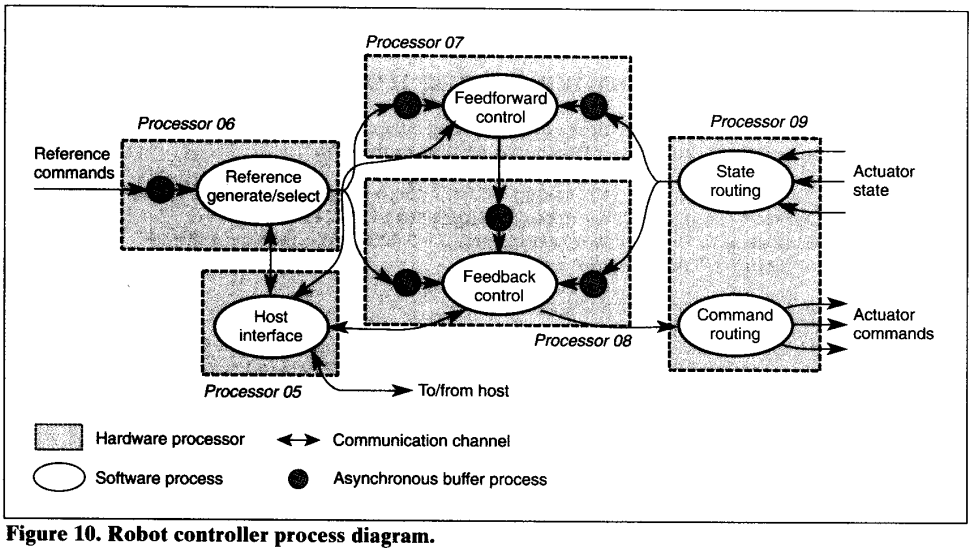



**Figure 10. Robot controller process diagram.**

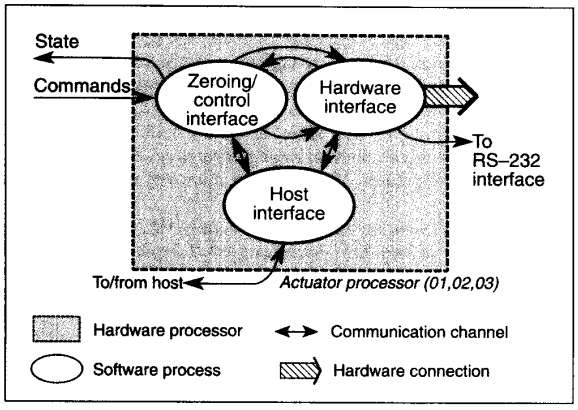



**Figure 11. Internal structure of the actuator management nodes.**

**Physical interface.** Of course, actually operating a motor invariably differs from the abstraction represented by Equation 10 for commanding torque and monitoring position and velocity. Safety issues dictate continuous low-level system monitoring to ensure operation within predefined limits. Physical properties of the motors often require more action to deliver torques, and motor initialization often requires special procedures such as the location of a "home" position from which to measure relative positions. The details associated with these considerations combine to make the reliable operation of a particular actuator system nontrivial.

Although the particular motors used in the juggling system are not individually difficult to operate, we have chosen to associate one processor with each motor (processors 01, 02, and 03 in Figure 4), while one additional processor (processor 04) services an RS-232 interface to the power amplifiers. This simplifies the system overall at the price of some additional hardware. Figure 11 shows the structure of the software used to service each actuator, which is divided into three processes.

## Juggling

Our juggling strategy causes the paddle to respond to the motions of the ball in four ways:

(1) The paddle tracks under the ball at all times.

(2) The paddle "mirrors" the ball's vertical motion as expressed by the original planar mirror law.[1]

(3) The ball's radial motion causes the paddle to raise and lower, resulting in adjustment of the normal to correct for radial deviation in the ball position.

(4) Lateral ball motion causes the paddle to roll, again adjusting the normal to correct for lateral position errors.

In particular, we first transform joint space coordinates according to Equation 9, then compute

$$\eta = \gamma b_z + \frac{1}{2}\left(\dot{b}_z^{\,2}\right) \qquad \rho = \sqrt{b_x^2 + b_y^2 - l^2} \tag{13}$$

the ball's total vertical energy (modulo its mass) and its radial position. The complete mirror law combines these two measures with a set point description

$$\mathbf{q}_d = m(\mathbf{w}) \triangleq \begin{bmatrix} \overbrace{\underbrace{-\frac{\pi}{2} - (\kappa_0 + \kappa_1(\eta - \bar{\eta}))\left(\theta_b + \frac{\pi}{2}\right)}_{(2)} + \underbrace{\kappa_{00}(\rho_b - \bar{\rho}_b) + \kappa_{01}\dot{\rho}_b}_{(3)}}^{(1)\ \phi_b} \\ \underbrace{\kappa_{10}(\phi_b - \bar{\phi}_b) + \kappa_{11}\dot{\phi}_b}_{(4)} \end{bmatrix} \tag{14}$$

**Figure 12. The juggling algorithm, where (1), (2), (3), and (4) correspond to the four points in the juggling strategy.**

($\bar{\eta}$, $\bar{\rho}$, and $\bar{\phi}$) to form the function shown in Figure 12 (Equation 14), where (1), (2), (3), and (4) correspond to the four points of the juggling strategy, $\mathbf{q}_d$ is the vector of reference commands for the robot joints, and the $\kappa$'s are various gains adjusted to ensure the stability of the system. A reference velocity, $\dot{\mathbf{q}}_d$, and acceleration, $\ddot{\mathbf{q}}_d$, are also generated by evaluating the symbolic derivatives of this expression. Note that this does not represent an implementation difficulty since $\dot{\mathbf{b}}$ and $\ddot{\mathbf{b}}$ are both available.

The implementation of this algorithm consists of a single process that evaluates $\mathbf{q}_d$, $\dot{\mathbf{q}}_d$, and $\ddot{\mathbf{q}}_d$ whenever new ball information is received from the interpolator. Setting the interpolation time constant dictates the rate at which this computation is carried out and, thereby, the rate at which the reference commands to the robot controller are updated.

The ultimate proof of concept lies in successful function. By this measure, our algorithms and their arrangement on this embedded controller are hugely successful. We typically log thousands and thousands of impacts — a continuous vertical one-ball juggle that lasts several hours — until some random imperfection in the paddle knocks the ball out of the robot's work space and the juggling stops. Moreover, the robot generally recovers from human-induced midflight disturbances, assuming that the ball remains in the work space.

We are encouraged to believe that there is something empirically valid about "geometric programming" — that is, encoding higher level goals in terms of mathematical functions — for specifying behavior. Not accidentally, this style of programming is well adapted to implementation in a message-passing environment. In turn, the CSP model of computation fits naturally into the medium-grained processor framework required to distribute computation down to the layer of physical processes that we consider most effective.

As more systematic process-distribution and communication-management techniques become available, it should become possible to extend the theoretical performance guarantees associated with our control algorithms to their implementation in embedded real-time distributed computer networks. ■

## Acknowledgments

Suggestions by Geoffrey Brown, Bud Mishra, and John Stankovic have helped improve the focus and presentation of this article.

Martin Bühler was largely responsible for the original conception and design of the XP/DCS system as well as the Yale Bühgler robot. We thank him for his advice and assistance in the present study.

This work was supported in part by SGS Thomson-Inmos Corporation, the Superior Electric Corporation, GMF Robotics Corporation, and the National Science Foundation under a Presidential Young Investigator Award held by Daniel E. Koditschek.

## References

1. M Bühler, D.E. Koditschek, and P.J. Kindlmann, "A Family of Robot Control Strategies for Intermittent Dynamical Environments," *IEEE Control Systems Magazine*, Vol. 10, Feb. 1990, pp. 16-22.
2. G.R. Andrews and F.B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, Vol. 15, No. 1, Mar. 1983, pp. 3-43.
3. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, N.J., 1985.
4. L.L. Whitcomb, A.A. Rizzi, and D.E. Koditschek, "Comparative Experiments with a New Adaptive Controller for Robot Arms," *Int'l Conf. Robotics and Automation*, IEEE CS Press, Los Alamitos, Calif., Order No. 2163, pp. 2-7.
5. P. Hsu and S. Sastry, "The Effect of Discretized Feedback in a Closed-Loop System," *Proc. 26th Conf. Decision and Con-*

*trol*, IEEE Press, Piscataway, N.J., 1987, pp. 1,518-1,523.

6. D.P. Bertsekas and H.N. Tsitsiklis, *Parallel and Distributed Computation*, Prentice-Hall, Englewood Cliffs, N.J., 1989.

7. K. Arvind, K. Ramamritham, and J.A. Stankovic, "A Local Area Network Architecture for Communication in Distributed Real-Time Systems," *J. Real-Time Systems*, Vol. 3, 1991, pp. 115-147.

8. D.E. Koditschek, "Robot Control Systems," *Encyclopedia of Artificial Intelligence*, S. Shapiro, ed., John Wiley & Sons, New York, 1987.

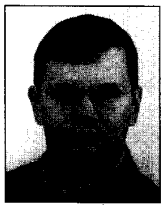
9. J.Y.S. Luh, M.W. Walker, and R.P. Paul, "Resolved Acceleration Control of Mechanical Manipulators," *IEEE Trans. Automatic Control*, Vol. AC-25, 1980, pp. 468-474.

10. J.-J.E. Slotine and J.J. Craig, "Adaptive Trajectory Control of Manipulators," in *Robotics Review*, O. Khatib, J.J. Crag, and T. Lozano-Perez, eds., MIT Press, Cambridge, Mass., 1989, pp. 367-377.

11. M. Bühler et al., "A New Distributed Real-Time Controller for Robotics Applications," *Proc. 34th Compcon*, IEEE CS Press, Los Alamitos, Calif., Order No. 1909, 1989, pp. 63-68.

**Alfred A. Rizzi** is a research associate and PhD candidate at Yale University. He received the ScB degree in electrical engineering from MIT in 1986. From 1986 to 1988 he worked as a designer in the aerospace industry. In 1988 he returned to graduate school at Yale, where he received the MSc in electrical engineering.

His research interests include the study of robot systems capable of dynamical dexterous manipulation, and the theory and implementation of distributed real-time robot controllers. He is a member of the IEEE.



**Louis L. Whitcomb** is a research associate and PhD candidate at the Yale Robotics Laboratory. He obtained his BS in mechanical engineering from Yale in 1984. From 1984 to 1986 he was a member of the research and development staff of GMFanuc Robotics, Detroit, Michigan.

His research interests are in theory and implementation of distributed computational architectures for real-time robot control; theory, implementation, and experimentation with adaptive control of robot manipulators; and automated assembly via artificial potential functions. He is a member of the IEEE.

**Daniel E. Koditschek** was appointed assistant professor of electrical engineering at Yale in 1984, where he is presently an associate professor. He received his BS in 1977 in engineering and applied science and his PhD in 1983 in electrical engineering from Yale University.

His research interests include the application of dynamical systems theory to autonomous-machine design, nonlinear control theory, and the application of computational theory and hardware to feedback control of physical processes. He is a member of the AMS, SIAM, ACM, and IEEE.

Readers may contact Daniel E. Koditschek at the Center for Systems Science, Dept. of Electrical Eng., Yale Univ., PO Box 1968 Yale Station, New Haven, CT 06520-1968; e-mail koditschek@cs.yale.edu.