BUILDING FAST, INTELLIGENT ROBOT SYSTEMS

Russell L. Andersson

Russell L. Andersson *is a member of technical staff in the Robotics Systems Research Department of AT&T Bell Laboratories in Holmdel, New Jersey. Mr. Andersson joined the company in 1981 and is researching high-performance robot systems, including the areas of sensing, intelligent control, and actuation. He has a B.S. in computer science and engineering, an M.S. in computer science, and a Ph.D. in computer science and robotics, all from the University of Pennsylvania.*

Conventional robot systems operate slowly and methodically, often playing back a pretaught sequence of positions. To investigate the construction of fast, intelligent robot systems, we have built a robot ping-pong player. We will examine the techniques required to cope with a dynamic environment, from the vision system to the robot's low-level control algorithms. We also describe an "expert controller," which executes loosely specified strategies in real time in spite of constraints imposed by the robot and by task geometry.

Introduction

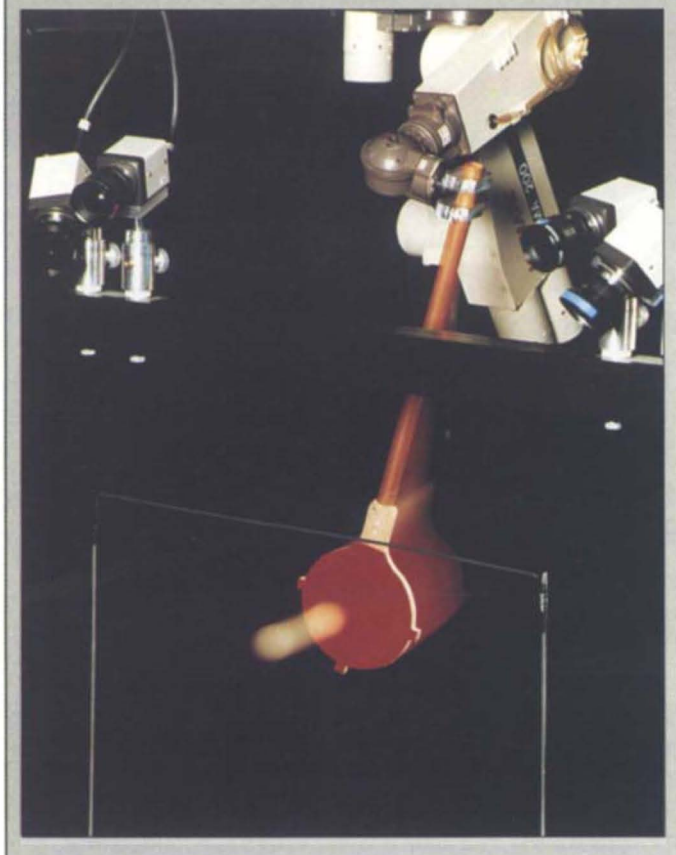
Robots can be designed for many tasks. How well a robot performs a task determines how cost-effective it is. Surprisingly, the computers currently used to control robots limit performance. Without accurate data, the controller must adopt excessively conservative estimates of the manipulator's performance when planning a motion. We can increase a robot system's speed and functionality by increasing the controller's knowledge of the manipulator's physical capabilities and by providing effective ways to use this information. We created the robot ping-pong experiment to investigate how to build robot systems that can display reasonably intelligent performance-enhancing behavior.

The most intelligent AI (artificial intelligence) programs can exhibit symbolic reasoning but are too slow to respond in the robot's dynamic environment. Speed is not the only issue, however, because AI systems concentrate on symbols to the exclusion of numbers; we need numbers to run robots. Robot controllers use directly coded numeric algorithms to attain the speed required to drive the robot; consequently, they do not display very interesting or complex behavior patterns.

We propose an intermediate level between the AI system and the robot controller, which we have dubbed an *expert controller*. The expert controller can do numeric and simple symbolic processing at the rates required for the robot's operation, while, hopefully, emulating the skilled behavior patterns of an athlete or assembly-line worker.

We use robot ping-pong as an experimental subject (Panel 1) because it is a stable, well-defined problem with many degrees of

Panel 1. The Robot Ping-Pong Problem



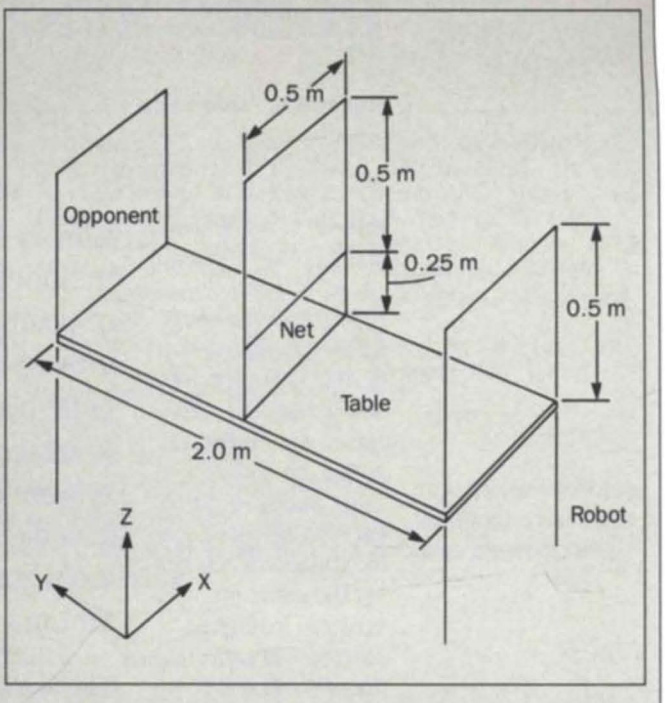


Figure 1. (left) The robot hangs upside-down just above the (1-m wide) horizontal camera support bar at the center of the image.
Figure 2. (above) Robot ping-pong table.

Our robot plays ping-pong according to rules proposed by Billingsley[3]. The rules place a premium on an accurate stroke. Wire frames limit the region the robot must cover. The ball must travel sequentially through the square frame at one end, through the center frame, then bounce once and travel through the frame at the table's other end. To simplify image processing, the table, the background, and opponent must be black.

The height of the net and the upper bound on the ball's height at the hitting end of the table (because of the wire frames) restrict the ball's speed to the approximate range of 4–8 m/s. The player must hit the ball 0.4–0.7 s from the time of the opponent's hit. An incoming shot's position and velocity substantially determine the range of possible returns; simple strategies, such as "aim for the middle of the opposite frame," do not work.

We use an upside-down Unimation PUMA™ 260 robot; it has a reach of 0.4 m. To create a workable robot configuration, we put the paddle at the end of a 0.45-m stick, mounted perpendicular to the axis of joint six. The configuration enhances reach and reduces the velocity required of the last (wrist) joint, which generates the striking velocity.

Ping-pong requires only five degrees of freedom because the paddle can be rotated about its normal vector; however, the direction of this degree of freedom varies. One challenge of the system is to take advantage of this redundant degree of freedom.

For comparison, a human table is 1.5 m wide and 2.7 m long; the net is 0.16 m high. The lower net and absence of end frames make the set of possible returns quite large. Ball speeds can reach 20 m/s. Human players increase their available reaction time by moving back from the table.

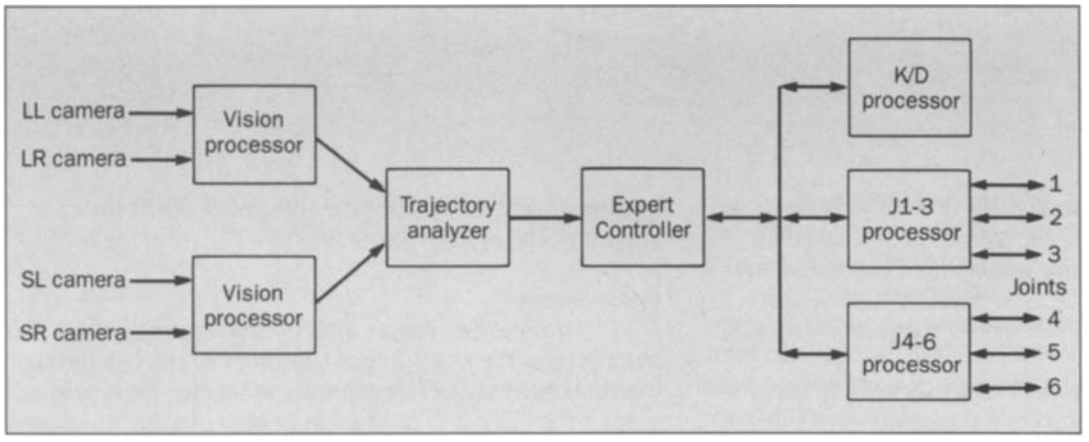


Figure 3. Simplified block diagram. The cameras [long-range left (LL), long-range right (LR), short-range left (SL), short-range right (SR)] feed the vision processors, Eye0 and Eye1. Data are sent to the trajectory analyzer, which produces predictions of the ball's path for the expert controller processor. A kinematics and dynamics slave processor (K/D) and two joint servo processors (J1-3, J4-6) drive the robot.

freedom and stringent performance requirements. Robot ping-pong has no best solution; best can only be defined in terms of some arbitrary evaluator. We know only one true evaluator: who won the point? Given an incoming trajectory, the robot controller must pick a suitable return that satisfies the constraints of the robot and the rules of ping-pong. Because the robot system must act before it can accurately sense, the robot must continually adapt its motion as additional, more accurate, sensor data arrive. In essence, the robot uses internal feedback to improve motion.

The techniques presented in this paper apply to robot systems with redundant degrees of freedom (such as robots with seven or more joints), with anthropomorphic hands, or to robot systems on movable bases. Such systems contain physical manipulator redundancies; task redundancies are useful as well. (Ping-pong contains both types.) We can exploit redundant degrees of freedom to increase speed and robustness.

Possible applications include combinations of task and manipulator redundancies such as:

- Manipulating objects on moving conveyor belts.
- Interacting with devices that have a fixed time cycle (machine tending).
- Scheduling multirobot systems.
- Conventional applications in which performance is at a premium.

Constructing a real-time intelligent system requires attention to every component's design, not just the expert controller. Accordingly, we have divided the paper into four major sections. First, we will describe the ping-pong player's *system engineering*. The remaining sections trace a single stroke from the *vision system* through the *expert controller* to the low-level *robot controller*.

System Design

The system operates on a network of Intel Corporation Multibus™ single-board computers containing Motorola 68020 microprocessors. They can communicate with high bandwidth and low latency across the S/Net, a local area network developed at AT&T Bell Laboratories.[1] (See Figure 3.) The Meglos multiprocessing and multitasking operating system, also developed at Bell Laboratories, supports real-time programming.[2]

There are two subsystems essential not only to the robot ping-pong player, but to any other fast, intelligent robot system as well. These are the *clock subsystem*, which enables the system to generate accurately timed motions, and the *data logging subsystem*, which we use to debug the system.

Achieving Accurate Timing. To get to the right place at the right time to make a hit, the robot system must recognize and compensate for system latencies. A sensor output statement such as "the ball is 0.5 meters (m) above the table" is useless, because it is inaccurate soon after it is generated. Any sensor reading of a time-varying signal must be stamped with the time at which it was taken; this defines the only time at which it has any validity. Sensor

data may be extrapolated forward in time to yield predicted sensor data. The processor can plan its action based on predicted sensor data. Taking into account processor and actuator latencies, it can start the actuator so that the actuator position will match the predicted sensor data at the same instant.

We must be able to measure precisely an event's time of occurrence. Microprocessor systems have timers that are readable by software (with substantial overhead). However, an approach that includes software in the measurement path is guaranteed to be inaccurate (a) because the processor may execute an arbitrary sequence of instructions between the event's time of occurrence and the time the software reads the timer, and (b) because cache misses, refresh, scheduling, and I/O (input/output) interrupts can occur.

Our sensors and actuators are distributed across multiple processors. Thus, we need a "wall clock," accessible to all processors at once, to maintain a consistent view of time across the system. We use a specialized board that resides in each processor; a specialized clock bus connects the boards.

Making the System Work. To have a fast, intelligent robot system, we must have a way to debug it. Techniques that add print statements or set breakpoints are impossible to use in real-time systems because they either slow it down or bring it to a complete halt.

The only fast way of logging a program's data is to store it in main memory (large buffers are currently possible). The log can be analyzed on line by learning programs, or written to disk for permanent storage and analysis by humans or programs.

The `printlog` routine emulates the UNIX® system `printf` subroutine call, but writes the data to the log instead of the UNIX I/O system. `Printlog` execution times range from 50–100 microseconds (μs), depending on the amount of data stored. On average, we generate 90 kilobytes (Kbyte) of data in a several-second volley.

The `rtd` program serves as the interface between the real-time system and the human controller. The user may request logged entries to be displayed or plotted, or alter parameters in running programs with a *remote procedure call* (RPC) facility.

We will use data and plots extracted from the data logging system throughout the remainder of the paper to show how the system performs.

Vision System

Processing begins in the vision system, which must predict the exact future trajectory of the ball through four-dimensional (4-D) space-time at 60 Hz. First, we extract the ball's 2-D position in several images, combine the 2-D positions using stereo to find the 3-D position, then process the 3-D position sequence to find the trajectory.

Image Transduction. Black-and-white video cameras convert the optical image to an electrical video signal. An analog-to-digital converter digitizes the analog video signal to eight bits of gray-scale. We use four cameras to achieve a large field of view; these serve as a short-range and a long-range stereo pair (Figure 3).

Unlike past experience in computer vision, a model of the camera's dynamic characteristics must be used to obtain accurate data. Images viewed by the television cameras vary rapidly with time. We need to understand what the camera will output as a function of the scene, so that we can define a self-consistent object position and time.

Television cameras constructed with vidicon (vacuum) tubes have a decay function such that the output at any time might be affected by a bright image many frames ago (an effect similar to the persistence of CRT displays). Each point of the image is sampled at a different time as the beam sweeps over it, so that a vertical bar moving horizontally results in a picture of a diagonal bar.

CCD (charge-coupled device) cameras operate as pipelined devices, integrating one image while reading out the previous one. Every pixel is integrated over the same finite time interval, so the bar stays vertical.

We use a third camera, an MOS (metal-oxide semiconductor) camera, which resets each pixel as it is being read out. The start of the (finite) sampling interval is different for each pixel. MOS cameras produce a diagonal bar image.

Image Processing. Conventional image processing systems either (a) convert the image to binary, then store a run-length-coded image, or (b) directly store the gray-scale data. However, to keep the system running at 60 Hz, we must process the data as they are generated.

To separate the ball from the background image, we use gray-scale thresholding (implemented by a look-up table):

$$pixel_{out} = \max(pixel_{in} - threshold, 0) \quad (1)$$

The ping-pong player's environmental design ensures that any nonzero pixel is part of the ball.

We locate the ball using moments, as found in physics and statistics:

$$x = \frac{\sum_{i,j} a_{i,j} i}{\sum_{i,j} a_{i,j}} \quad (2)$$

$$y = \frac{\sum_{i,j} a_{i,j} j}{\sum_{i,j} a_{i,j}} \quad (3)$$

where i ranges over the x axis and j ranges over the y axis; $a_{i,j}$ is the gray-scale intensity of that pixel.

A custom-integrated circuit performs the moment calculations in real time. The chips reside on a circuit board with support circuits; four of these cards plus four additional copies of an analog/digital converter card process data from all four cameras in real time. These video processing cards were designed for general application to robotic and inspection tasks, not just for ping-pong. They can determine the position, size, and orientation of an object.

Once the moments have been obtained, we can correct for several effects that occur earlier in the process. We compensate for lens distortions with a technique that analyzes the residual errors from the calibration of the camera's location in the workspace. (We watch a light at each of a large number of known locations.) This reduces the short-range cameras' root-mean-square error from 6.3 millimeters to 0.9 millimeters (mm), and the long-range cameras' from 1.6 mm to 1.1 mm.

Once all the corrections have been applied, the system's nominal static accuracy is approximately 1 mm, with repeatability better than that.

Finding the Sampling Time. If we take the moments of the smear flashed onto the camera by a single, moving surface patch, we get the average position of the patch during the interval of observation. If the patch's velocity remains constant during the sampling interval, the computed centroid corresponds to the temporal center of the frame's sampling interval. By induction, this result applies to the ball's entire image.

Because MOS cameras reset each pixel as it is being read, each pixel's sampling interval is different. At 3.0 meters per second (m/s), the 16-millisecond (ms) difference in the sampling interval between the top and bottom of the image causes a 5-centimeter (cm) shift in the ball's apparent position. The effect is especially noticeable when the ball jumps from the bottom of the long-range stereo pair to the top of the short-range pair. The sampling time can be computed from the ball's y coordinate.

Three-Dimensional Trajectory Analysis. Unlike vision research with general scenes, we do not have to worry about finding the correspondence between features in the left versus right camera images, because we track only one object. Instead, we have other problems. Stereo viewing usually means that we have two observations of the same body at the same position. Because the two cameras' effective sampling times differ, we violate this assumption. In practice, both y values are similar because we have oriented the cameras predominantly in the same plane; we use the average of the two y values.

The ball's 3-D location is computed from two 4 by 3 perspective transform matrices using closed-form least squares. Figure 4 shows the xyz trajectory from the middle of a volley. The human opponent has hit a reasonably hard 6.1-m/s shot at the robot, which the latter has returned.

The ball's trajectory must be computed each 1/60 second (when a new position becomes available) and the trajectory used to predict the future path. The need for prediction imposes a great demand on the trajectory fitting process. It is not enough to fit some high-order polynomial to the data. This will track the noise, and generate grossly unstable predictions. We must perform a noise reducing fit and use knowledge of the trajectories to perform the extrapolation.

Motion interpretation studies typically assume a constant velocity; because we need an accurate treatment of the motion model, we can not. Although not discussed

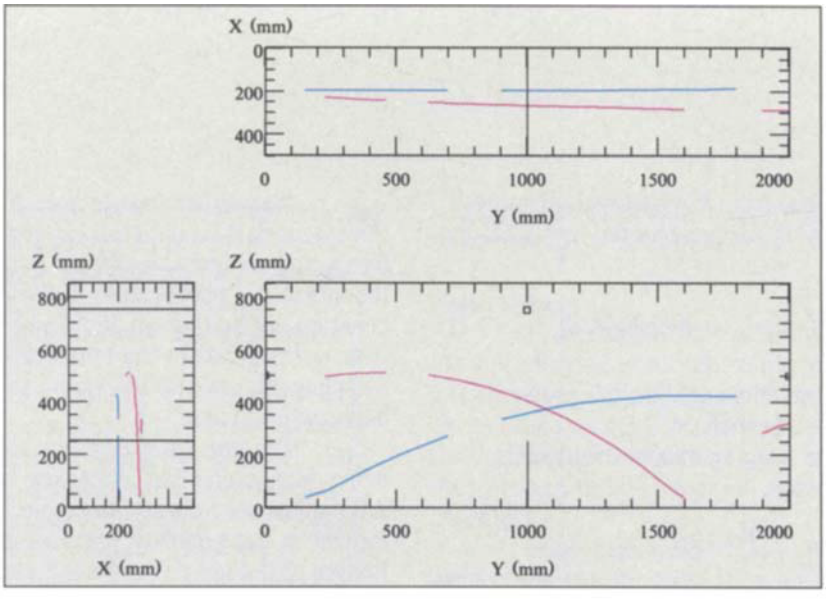


Figure 4. Trajectory of incoming volley. The human hits the first shot from right to left. The human's shot is shown in blue; the robot's in red.

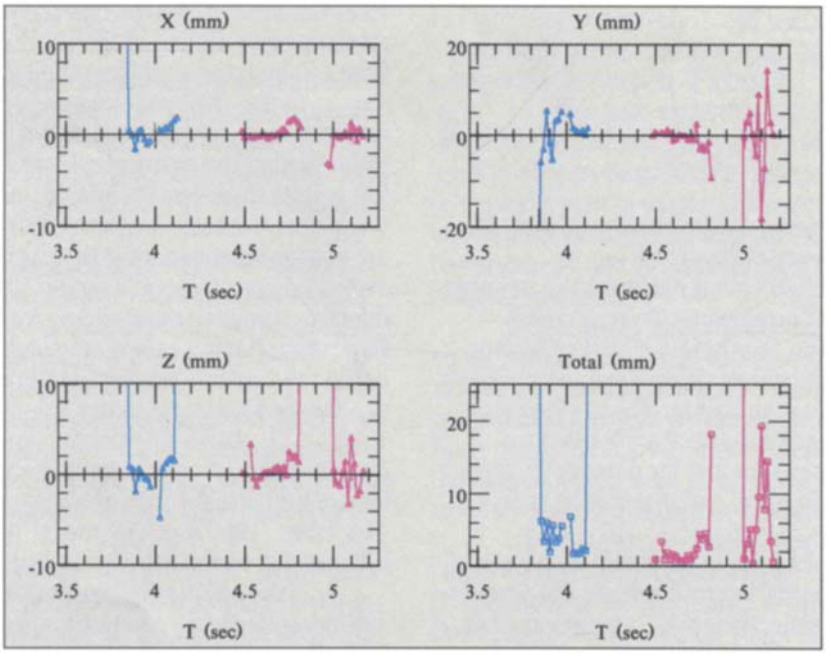


Figure 5. Prediction errors. As above, the human's shot is in blue; the robot's in red.

here, the trajectory analysis uses quadratic fits, but feeds forward corrections to compensate for higher-order terms.

The prediction error data in Figure 5 show the error on a moving ball. Subsequent to $T = 5$ s, the ball is 2 m away, and the depth error is predictably higher. The expert controller receives the time, position, velocity, and spin at which the ball will cross the robot's end of the table (where $y = 0$).

Real-Time Expert Controller

The real-time expert controller, the heart of the robot ping-pong player, revolves about a small number of "free variables" whose values are not directly determined from sensor data. When the ping-pong system's expert controller receives the first report of a ball's trajectory, it must generate initial values for the free variables including:

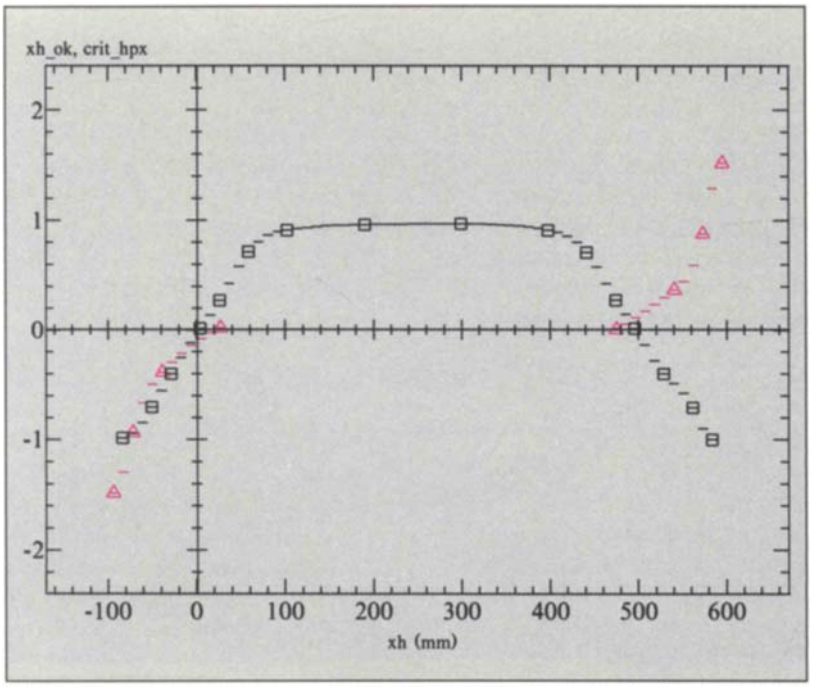


Figure 6. Evaluator for *x* hit position. The input to the model is the *x* position, `xh`, at which the ball might be hit. The central curve, marked with squares, generates a figure of merit for the position. The outer curves, marked with triangles, indicate that the position is too far to the side, including a numeric severity.

- How far from the table to hit the ball
- Where on the paddle to hit the ball
- The ball's return velocity
- The robot's configuration.

Having done that, it can compute values for the multitude of detailed variables that ultimately produce an arm trajectory.

We are concerned here not with the detailed numeric calculations, but with the process of controlling the free variables. Many free-variable selections result in an infeasible plan, but we do not know the constraints until the free variables are known. The expert controller must capture expert knowledge about how to generate feasible solutions rapidly.

To begin processing, the robot ping-pong player must choose the position along the ball's trajectory at which the robot should hit the ball. The *hit plane depth* (HPD) defines this location by the *y*-axis value of the point of contact. We consider three possible HPDs, corresponding to "up close," "nominal," and "far back."

We evaluate each possible HPD by:

- The time until the ball arrives there
- Its descent rate
- How hard the position is to reach.

A special data structure, called a *model*, contains the evaluating data. A model is a generalized look-up table that can return different output types. Figure 6 shows an evaluator for the ball's *x* position when it is hit. It returns either:

(`ok_to_hit`, *figure_of_merit*)
(`too_wide`, *how_far*)

Accordingly, the model returns both *symbolic* and *numeric* information. The model appears in the program as a C-language subroutine, although a support program generates the subroutine from the model's data.

Once each hit plane has been evaluated (pinpointing infeasible depths), the `decide` routine makes the final selection:

```
hpdno = decide (hpd_count, FOMt,
        FOMrate,FOMx,FOMz,FOMreach,OL)
```

where `FOM`s are arrays containing the figures of merit. `Decide` is a generic routine that forms a figure of merit from the sum of the evaluators, then selects the alternative with the highest figure of merit.

Table I shows an initial planning decision from the log system. Column headings show the time at which each message was logged (in microseconds). `Decide` has selected the deepest hit plane (#2) at −400 mm, largely on the basis of a favorable descent slope evaluation (`qth`).

A similar routine, `combine`, picks values for continuous variables. Instead of evaluators, the input to

Table I. Initial Planning Decision

Variables	Log Time (μs)			
	933671	935016	936360	937029
Time ball will cross this hit plane (t, sec)	1335310	1391525	1448597	
Position at crossing (p, mm)	146.0	139.4	132.9	
	−100.0	−250.0	−400.0	
	204.4	267.7	298.3	
Velocity at crossing (v, mm/sec)	−117.4	−115.7	−114.0	
	−2687.8	−2648.8	−2609.6	
	1421.9	832.1	247.6	
Evaluation of:				
Time until crossing (qt)	0.804	0.883	0.921	
X at crossing (qx)	0.949	0.943	0.943	
Z at crossing (qz)	0.986	0.873	0.758	
Ascent/descent angle (qth)	0.359	0.529	0.863	
Duration from loss of visual contact to hit (qcum)	0.990	0.967	0.934	
Reachability of this position (qo0)	−0.153	0.395	−0.265	
Reachability of this position (qo1)	−0.960	−0.978	−0.998	
HPD0 evaluation, –100 mm				3.604468
HPD1 evaluation, −250 mm				3.801466
HPD2 evaluation, −400 mm				3.859097
Best HPD				2

`combine` is possible values for the output variable; `combine` selects the (possibly weighted) average value.

For example, the return velocity must be planned in the *xy* plane. The first part of our strategy is to hit the ball directly across from where it is now, but displaced somewhat toward the middle of the opposing frame if the ball would be close to an edge horizontally. We could do this with two separate rules, but it is faster and easier to understand if we do it with only one model. The `aim_lr_al` model (see below) also varies the horizontal target, `aim_xc`, depending on the *xy* angle of the ball.

After testing, we discovered that additional semantics were required. If the ball is near one of the side wires, we must aim at the opposing corner to avoid running into the side wire at the robot's end of the table. To correct this, we modified the aim point to the opposite corner when the ball is near a side wire (Figure 7), and additionally, generated a weight (`w`) that causes the opposite-corner aim point to dominate the value determined from the *xy* angle:

```
w = Aim_lr_xh();
Aim_lr_al();
aim_xc = combinew
        (2, w,aim_lr_xh, 1.,aim_lr_al);
```

While creating the initial plan, the system continually evaluates its quality. Once the return velocity has been chosen, the initial planner simulates the ball's return flight in detail, using the full aerodynamic model. By `94826` in the example excerpted in Table I, the system determines that the ball's flight will last 0.46 s, crossing the table's far end at $x = 176$ mm, $z = 44$ mm. The simulation ensures that a valid velocity has been chosen. The planner stores a description of the trajectory as an aid to later processing.

The initial plan need not be perfect. Subsequent updating processes will improve the plan's success as new

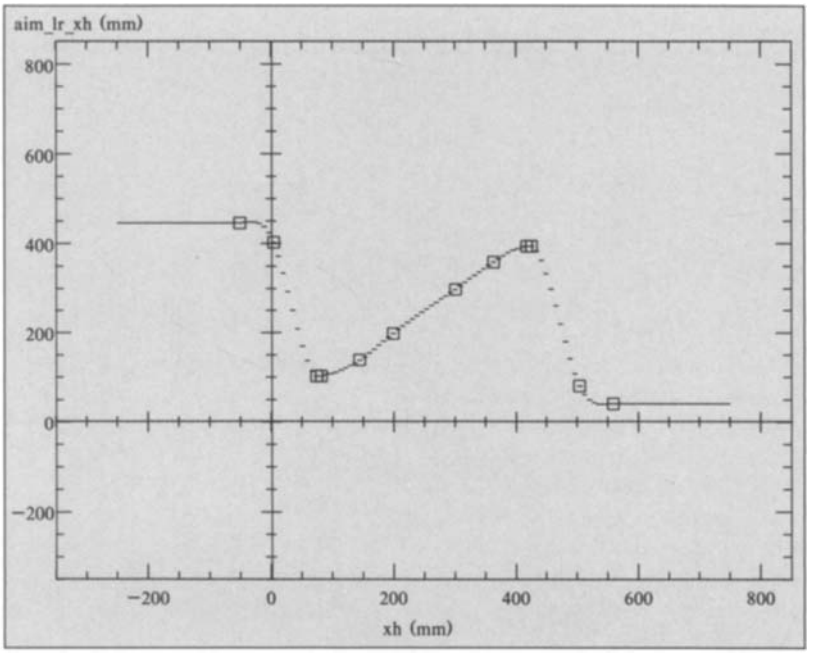


Figure 7. Horizontal aim point value. The model generates an *x* position at which to aim from the *x* position at which the ball will be hit.

sensor data arrive. It is much easier to solve problems during temporal updating, when all constraints are visible, than it is to solve problems during initial planning, when they are not. To further justify this approach, notice that the initial sensor data are flawed because not much of the ball's trajectory has been seen. We should not devote excessive effort and *time* (especially in the real-time context) to finding the optimal plan based on incomplete data (i.e., garbage in, garbage out).

Temporal Updating. Once we have formulated an initial plan and execution has begun, we must update the plan as additional sensor data become available. Temporal updating has two objectives:

- To change the plan in response to new sensor data
- To provide information about later stages of processing to earlier stages, giving a better solution to the problem.

For example, if a particular joint approaches its maximum torque output, the expert controller should begin to compensate for the problem in advance.

The free variables define the state of the system. We carry free variables from one plan to the next, modifying them each cycle to improve the plan. Routines called "tuners" perform the modifications; a tuner controls one free variable, or sometimes a small number of related variables. Although a tuner affects only one variable, its decision reflects the state of the entire system.

One ping-pong tuner, `tunealpha`, controls the direction of the return in the *xy* plane via the `alpha` free variable, complementing the `aim_xc` planner. (The initial planner converts `aim_xc` into an `alpha` value shortly after it is generated.) `Tunealpha` maintains the two initial planning objectives: (a) that the ball stay away from the table's edges, and (b) that the ball avoid the near side wire.

Our models evaluate the current plan by processing data stored when the plan was generated and then indicating preferred relative changes in the input. The `xfar_eval` model (Figure 8) evaluates the position at which the ball crosses the table's far end, proposing a correction to that crossing position. We can encode a time-varying strategy in the shape of the curve without any additional state, making it simpler and more robust.

The same evaluator can produce several evaluations, conveying symbolic as well as numeric information. The `xfar_eval` model creates a single bit of symbolic information during normal updating, indicating whether a correction is needed. The bit resides in a Boolean variable, which we call a *note* (`suspect_far_x`, in this case). A small code block proposes the correction to `alpha`:

```
if (visible (suspect_far_x))
{
    wei[nfact] = fabs (crit_far_x);
    val[nfact] = crit_far_x /
                 (2000. - chit_desc->P.y);
    nfact++;
};
```

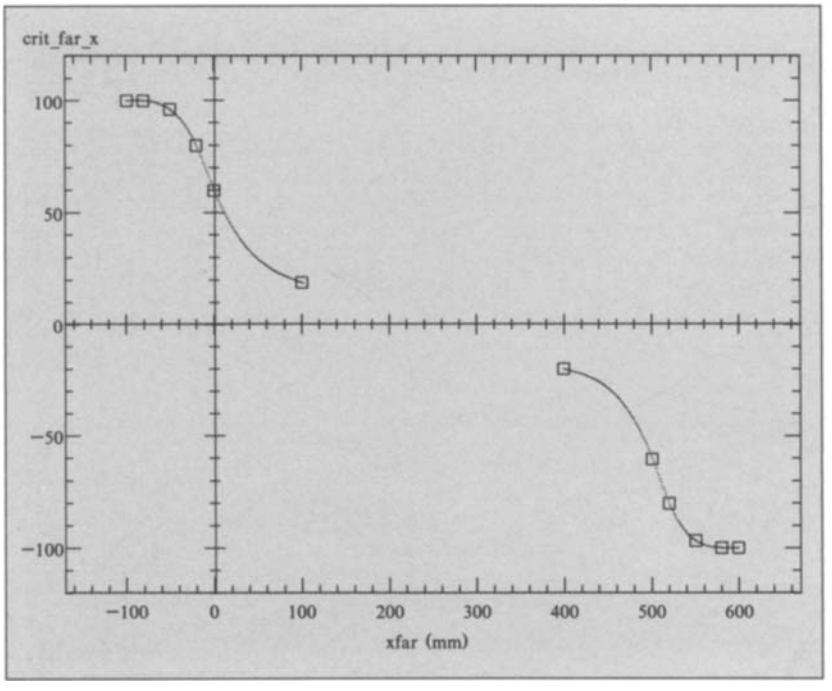


Figure 8. `Xfar_eval` **model. The model proposes a correction to the** *x* **position at which the ball will cross the table's far end.**

The symbolic information enables the system to spend time only on those conditions that are necessary at the time, speeding up execution.

After a similar code block to avoid the near wire, a `combine` routine makes the final update:


```
target_alpha += combinewv
          (nfact, wei, val);
```

More complicated tuners for the return velocity and robot configuration take into account not just problems in the ping-pong task, but in the robot system as well. These tuners combine the different proposed changes in the same fashion. The partial derivatives of the manipulator's position with respect to the joint angles (the Jacobian) and the derivative of the ball's return velocity with respect to the paddle orientation improve coordination between the task and robot domains. We can, for example, adjust the ball's return velocity when the paddle can not be accelerated fast enough.

When designing tuners, we must remember that we are not under any obligation to make a change at all, let alone an "optimal" correction. The tuner's behavior is loosely specified; we want to create tuners that exhibit the desired effects.

Exception Recovery. In a complex, dynamic environment, problems occur regularly as new sensor data demand responses of which the robot is incapable (despite temporal updating), or as a result of unexecutable initial plans. The expert controller has a centralized exception-recovery mechanism that attempts to solve the problem and return to the normal task as rapidly as possible. We use tuners to determine and execute the correction.

Once the initial planner or temporal updater has identified an error, it invokes the *centralized handler* (CEO, from "chief executive officer," showing a delegation of authority). (See Figure 9.) The CEO executes global checks on the recovery process. For example, it can break off processing to meet an upcoming deadline such as the contact time between ball and paddle. Otherwise, the CEO chooses a VP (from "vice president") capable of handling that particular class of problems, using symbolic information about the problem's classification. The software module `impVP` handles problems in the ball's return flight.

The VP must select a tuner to attempt a correction; the selection is based on symbolic information or model-generated factors. The two most important factors are that the problem be sensitive to the attempted correction, and the length of time required to effect a correction. The `tunealpha` tuner is the only means to prevent the ball from heading off the table's edge.

The `impVP` will call the tuner, then cause the execution to resume immediately following the generation of `alpha` in the normal initial planning or temporal updat-

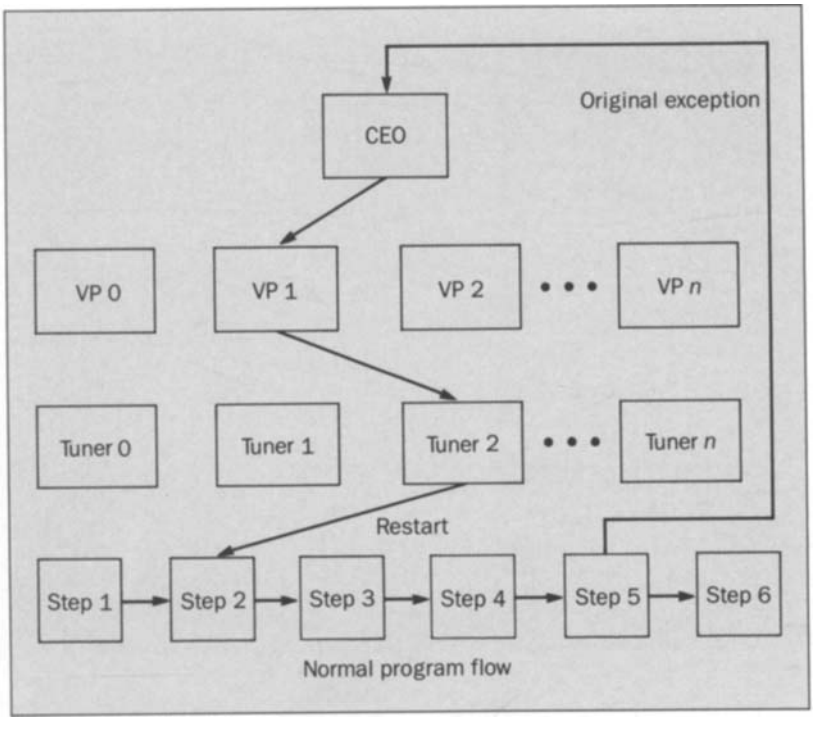


Figure 9. Exception handling architecture. An exception occurs in Step 5, causing CEO, VP1, and Tuner2 to be invoked in that order. After Tuner2 is run, VP1 adjusts the notes and does a `longjmp`; control passes to Step 2.

ing sequence. The overall program organization is such that the correct point of execution may be established simply by examining the symbolic *notes*. The UNIX `setjmp` and `longjmp` routines erase the subroutine calling sequence to the top level; when execution proceeds, the program rapidly falls through tests until the right code block is found (Panel 2).

Once a single exception has taken place, additional exceptions may occur, especially when the sensor data change late in a motion. Additional exceptions may also result from an exception's cure; with this in mind, we reevaluate the original exception handling path.

If the same problem has recurred, the problem's severity may be compared to the original value. If the problem has been mitigated, it is worthwhile to call the same tuner again; otherwise, a new strategy should be selected with that tuner excluded. A global algorithm, `worthwhile`, makes this decision, ensuring that the error recovery process will not result in an infinite loop, even across a collection of VPs.

Robot Controller

The robot ping-pong system must operate the actuator as close to its mechanical capabilities as possible. To take advantage of the robot's performance while still maintaining accuracy, we must compensate for the robot's dynamics, which become more significant as the robot's speed increases. Also, we must be able to predict the robot's performance to plan the motion.

Trajectory Representation With Polynomials. We must have a means of describing the trajectory the robot will follow. We plan each joint's trajectory [J1(t), J2(t), J3(t) . . .], rather than planning trajectories through space [$x(t)$, $y(t)$, $z(t)$], because the robot's joints are limited, not its spatial properties. It is the speed of the joints that limits the wrist's total speed. Similarly, the angular range of the joints limit the robot's position, while the torques of the joints limit acceleration.

Polynomials are a natural choice for a motion representation because their properties are well known, and closed-form results may often be derived. A single polynomial can represent the entire trajectory, so that the trajectory is inherently smooth and easy to evaluate. The derivatives are easy to compute, and may be represented in the same form. We use fifth-order, or quintic, polynomials to match the six degrees of freedom (positions, velocities, and accelerations at the beginning and end of the motion) required for trajectory planning. By evaluating the trajectory and its derivatives at a transition time, we can easily find the initial conditions of the next trajectory, ensuring a smooth transition. The wall clock system supplies the time input to all trajectories.

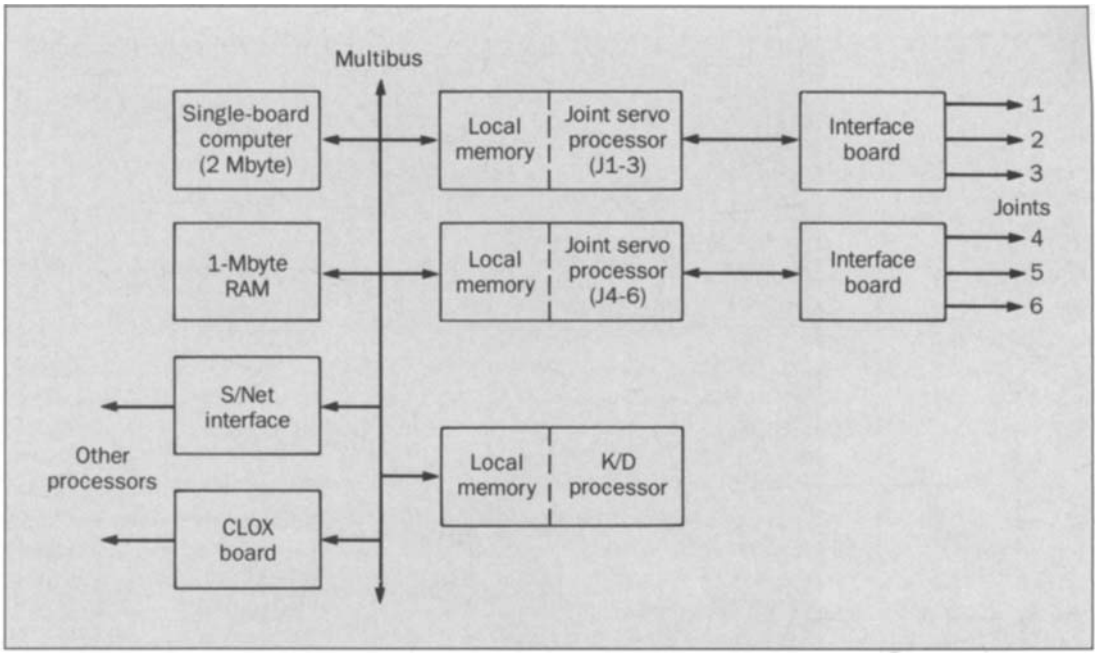


Figure 10. Robot controller architecture. The expert controller runs in the processor at top left. The K/D processor performs background kinematics and dynamics. The interface boards at the right contain the optical encoder circuits and digital-to-analog converters.

Dynamics and Control. Once a trajectory has been proposed, it must be evaluated to determine if it can be successfully executed by the robot, and then it must be executed as accurately as possible. We use the manipulator dynamics for both tasks.

The torque at a joint of a manipulator may be written as:[4]

$$T_i = \sum_{j=1}^{6} D_{ij} a_j + \sum_{j=1}^{6} \sum_{k=1}^{6} D_{ijk} v_j v_k + D_i - F_i v_i - S_i sgn(v_i) \quad (4)$$

where $D_{i,j}$ are the coupling inertias and D_{ii} includes the actuator inertia. The acceleration of joint j is a_j; D_{ijk} are centripetal and Coriolis terms; v_j is the velocity of joint j; D_i is the gravitational load on joint i; F_i is the viscous damping coefficient for joint i; and S_i is the intercept of the friction at rest. The function $sgn\ (x)$ equals:

- $+1$ for $x>0$
- 0 for $x = 0$
- -1 for $x<0$

We compute the joint inertias, gravity loads, and some coupling inertias. At present, only the coupling inertias to and from the wrist joints are computed. We do not feed-forward centripetal and Coriolis effects.

Panel 2. Program Flow Fragment

```
setjmp (main_jmp_buf); /*in TOPLEVEL */
if (invisible (did_task_1))
   task_1 ();
if (invisible (did_task_2))
   task_2 ();
   . . .
task_1 ()
{
   if (invisible (did_task_1a))
     task_1a ();
   . . .
   scribble (did_task_1);
}
```

During the planning process, we must establish whether a proposed trajectory is feasible. That is, can the actuators cause the robot to follow this trajectory accurately? The manipulator dynamics can be used to make this determination, but the entire motion must be simulated. Instead, we do a "quasi-worst-case" analysis with the maximum value of each parameter at the beginning and end of the trajectory. (We neglect the coupling torques to save time.) We are clearly throwing away system performance by this simplification, and furthermore, we run the

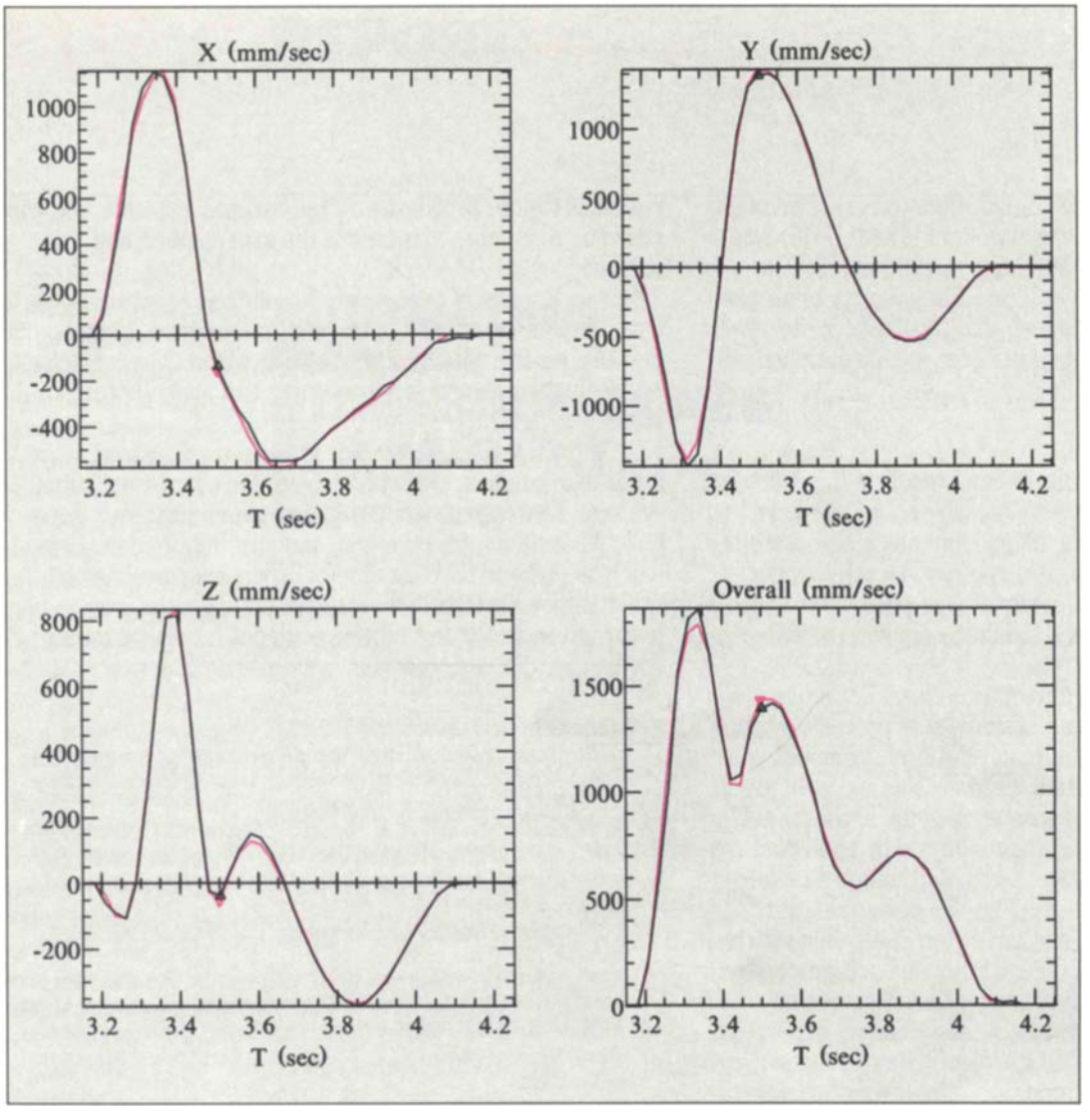


Figure 11. Cartesian velocities during motion. (Desired trajectory shown in red; actual trajectory shown in black.)

risk of not catching actual limits.

Implementation. Figure 10 shows the computing hardware of the robot controller. The computational burden demands multiple processors; the communications bandwidth requires shared memory. Timing analysis shows that the processing architecture will be able to handle the full dynamics, although we have not completed it yet.

The expert controller generates trajectory polynomials (including their exact starting time and duration) and queues them for execution on both the K/D (kinetics and dynamics) slave processor and the joint servo processor for the joint. The K/D processor computes the dynamics feed-forwards at the center of the next 26-ms window; the main processor transports them to the servo processors so that they may be used throughout that cycle.

Our servo control is based on a position-and-derivative (PD) controller. The joint servos execute the PD routine every 830 μs, computing the feed-forward due to the friction and acceleration (of that joint alone), then adding the feed-forward for that 26-ms cycle.

Analysis of a Hit. Figure 11 shows the Cartesian velocity of the paddle's center. The desired trajectory and the actual trajectory are overlaid. Most of the velocity is in the *y* direction toward the opponent. The peak paddle

acceleration is 2.5 g. At the contact time (3.5 s), the total position error from all sources was 3 mm [Joint 6 (J6) was 0.3° off], while the arm was moving at 1500 mm/s. The effective timing accuracy was 2 ms. The velocity error was 40 mm/s (5°/s on J6). We believe that flexibility in the motor-to-joint transmission contributes significantly to these errors.

Summary

The immediate experimental result of this work is a ping-pong system able to beat human beings. We have attained volleys of 21 hits by human and machine—a credit to the performance of both. It is too easy to see this performance and forget how and why it was done. The underlying design techniques should be applied to new robot system designs.

We described an architecture for constructing complex and robust real-time systems that process continuous streams of sensor data to solve poorly specified problems. Our most important lesson is that we have to view short-term task planning as an ongoing process, not something that is done once, then followed by blind plan execution. Every task contains redundant degrees of freedom that the controller may exploit. We have to identify these degrees of freedom, and invest the controller with the authority to alter them. The continuous self-perception of the robot's motion and the critical evaluation of that motion's quality seem much closer to the human experience than the blind plan/move cycles of today's robots.

To ensure an informed decision, the expert controller must be supplied with a continuous stream of external and internal sensor data. Our moment generator vision system illustrates one viable sensor system applicable to a variety of tasks. Interpreting a dynamic image sequence is certainly not just a matter of examining enough snapshots. Not only did we have to model the camera's spatial characteristics, including the lenses' distortion and illumination conditions, but in addition, we had to model the cameras' temporal characteristics.

Our low-level robot controller was designed to predict the robot's own limits, so that the robot could be operated as close as possible to those limits. We simplified the robot's dynamics to a form suitable for real-time execution. As more processor cycles become available, we will continue to be able to increase the arm's speed and accuracy.

A sense of time pervades our entire system. We have to view every task in the continuous-time domain, because we live in a continuous-time world. The wall clock system was essential for generating and applying temporal data throughout the system.

The expert controller enables us to exploit humanlike heuristics quickly enough for a real-time robot system. The system fills the gap between the slow, symbolic AI systems, and the fast, numeric robot controllers, much as humans have a range of processing levels from rational thought to trained reflexes. It leads the way toward even faster and more intelligent future systems. For further discussion of this system, see Reference 5.

References

1. S. R. Ahuja, "S/Net: A High Speed Interconnect for Multiple Computers," *IEEE Journal of Selected Areas in Communication*, Vol. SAC-1, No. 5, November 1983, pp. 751-756.
2. R. D. Gaglianello and H. P. Katseff, "Meglos: An Operating System for a Multiprocessor Environment," *Proceedings of the Fifth International Conference on Distributed Computing Systems*, May 1985.
3. J. Billingsley, "Machineroe Joins New Title Fight," *Practical Robotics*, May/June 1984, pp. 14-16.
4. R. P. Paul, "Robot Manipulators: Mathematics, Programming, and Control," The MIT Press, Cambridge, Massachusetts, 1981.
5. Andersson, R. L., *A Robot Ping-Pong Player: Experiment in Real-time Intelligent Control*, The MIT Press, Cambridge, Massachusetts, 1988.

(Manuscript received October 27, 1987)