Parallel Stiffness Controller for Stanford/JPL Hand

King Shaw
Roderic A. Grupen

COINS Technical Report 90-48

June 5, 1990

Laboratory for Perceptual Robotics
Department of Computer and Information Science
A305, Graduate Research Center
University of Massachusetts
Amherst, MA 01003

Abstract

This report presents the design and implementation of a transputer-based, parallel stiffness controller for the Stanford/JPL hand.

In active stiffness control, each finger is modeled by a virtual spring with a dimensionality that depends on the contact type between the finger and the object as well as the degrees of freedom of the finger. The stiffness controller controls the finger-joint torques in such a way that the finger behaves like the virtual spring being modeled. It takes joint positions as inputs and sends out motor torque commands to motors. Feedbacks are taken from strain gauges and motor position encoders.

The controller is implemented in the parallel language Occam 2 to fully utilize the inherent parallelism in multifinger control. The whole system is a set of concurrent processes that communicate among each other by synchronous, symmetric, indirect message-based IPC. Efforts have been made in the design to ensure that the allocation of processes to different topology of network is easy and flexible. One of the chief design goals is to have the finger servo loop operate at or above 500 Hz.

In order to meet this real time constraint, the controller is distributed onto a network of 5 transputers hosted by a Sun workstation. Finally, a UNIX interface to communicate with the controller is also provided such that the controller can be accessed through a set of C function calls.

Contents

1 Introduction **4**

2 Transputer Networks and Occam Programming **6**

2.1 Transputers 6

2.2 Occam Programming 8

3 Controller Architecture **12**

3.1 Finger Servo Loop 12

3.2 Dispatcher 13

3.3 Root 14

3.4 UNIX Interface 14

4 Controlling the Finger **16**

4.1 Impedance Control 16

4.2 Finger Modeling and Stiffness Control 16

5 Hand Kinematics **19**

5.1 Mechanical Structure Description 19

5.2 Tendon Tension Control 20

5.3 Motor Torque Control 21

5.4 Joint Position from Motor Encoder Readings 21

5.5 Hand Constants 23

5.6 Servo Loop Design 23

6 Conclusion **26**

6.1 Performance 26

6.2 Real-time scheduling 27

A Source Code Details **28**

A.1 Location 28

A.2 Description of the Files 29

1 Introduction

This project implements a parallel stiffness controller for the Stanford/JPL hand on a network of transputers hosted by a Sun workstation.

In active stiffness control, a finger can be modeled as a virtual spring with a dimensionality based on the contact type and the degrees of freedom of the finger. By controlling the joint torques in response to the displacement of the finger tip, we can make the finger behave like the spring being modeled. Although this virtual spring model is very easy to understand, the actual implementation is not so straightforward because of the design of this particular hand. Firstly, since the Stanford/JPL hand has taken the $(n+1)$ tendons approach and that tendons can only be pulled but not pushed, the 3 joint torques of a finger can only be controlled by pulling 4 tendons connected to that finger. Also, since the relationship between joint torques and tendon tensions is not one-to-one in general, there is an issue of choosing the tendon tensions that is both feasible and optimal, subject to the minimal tension constraint and a particular objective function. Secondly, since there is no joint position sensor, the joint position will have to be calculated using motor encoder counts.

This implementation is designed with the following goals:

- The finger servo loop must meet the real-time constraint, which is set at 500 Hz. The original hand controller did not come up to this standard because of insufficient processing power.

- This software system must be easy to modify and expand, subject to hardware upgrades and/or control law changes. This flexibility requirement has a profound influence on the choosing the underlying software and hardware architecture.

- Not only must the servo loop be fast enough, it also has to be as predictable as possible. It must be predictable in the sense that:

1. Each iteration of the servo loop is finished in a certain amount of time.
2. The frequency at which the servo loop will be run should be a constant.

This predictability requirement is necessary since we plan to incorporate other sensor inputs in the future. In a complex control system, it is highly desirable that each task is periodic and the bound of its computation time can be determined off-line. These characteristics make optimal real-time scheduling possible.

- It should be easy to modify to include some reflexive, local manipulator control in the future.

Taking these considerations into account, our design is a transputer-based, parallel control system which is fast, predictable and very flexible. It controls the hand through the I/O board[1] designed by Eugene Hertz and can be accessed through a set of C function calls from SUN.

An introduction to transputer-based architecture and parallel programming in Occam is given in Section 2. Section 3 discusses the implementation of the controller. Section 4 briefly explains the details of finger modeling. Section 5 contains the kinematics of the finger and derives necessary transformation functions. Section 6 concludes with the performance measurement and discusses future expansions. Appendix A contains details of the source code.

[1] Currently under construction.

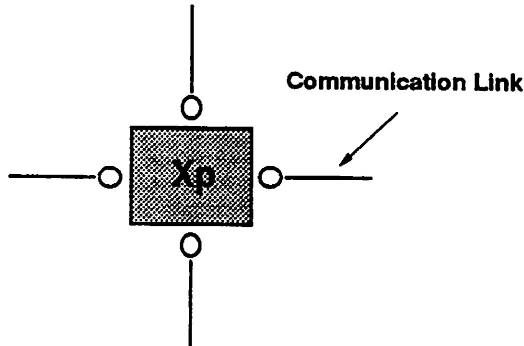

Figure 1: A transputer

2 Transputer Networks and Occam Programming

In this section, we will briefly review the architecture of transputer network and the programming model of the language Occam. A complete definition and discussion on both of these subjects may be found in [4] and [5] . For a tutorial on programming in Occam, see [14].

2.1 Transputers

Transputers are high performance microprocessors that support parallel processing through on-chip hardware. Each transputer has a number of high speed communication links that allow transputers to be connected to each other. For the INMOS T800, the tranputer we are using, there are four such links. Figure 1 shows a schema for the transputer.

A transputer network can be built easily by using transputers as building blocks. The network topology is software configurable in two levels:

- It can be set up by first defining the **softwire** file in a specific directory (in our system, it is */usr2/Xputer/b014*), and then invoking

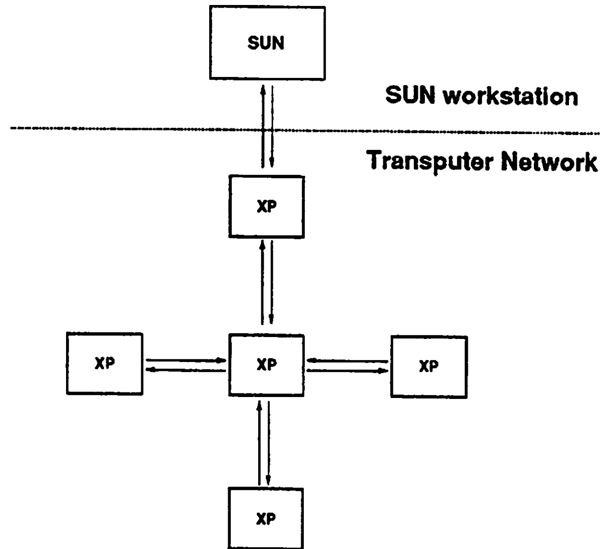

Figure 2: A transputer network

the Module Motherboard Software tool (MMS)which references the softwire file.

- The network configurer **iconf** is used to select a sub-network, allocate Occam processes to processors and assign channels to links in the sub-network.

Figure 2 shows a network of transputers with a SUN workstation as a host. Notice that each transputer has four links and each link can be allocated as an input/output channel. The mapping of channels to links is done in the second stage configuration time, by **iconf**. This allows programmers to design and test the software on a single transputer and distribute the final product onto a network of transputers without any modification to the software except the configuration specification.

As an example for this process allocation problem, Figure 3 shows four processes that are mapped to three transputers. This is not the only alloca-

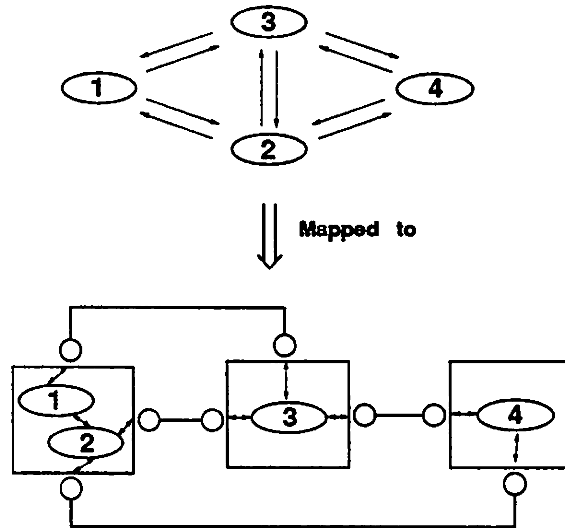

Figure 3: Processes allocation

tion possible. These four processes can be mapped onto a network of one, two, three or four processors. On the other hand, the best allocation for a fixed network can only be determined by looking into each process's load as well as traffic flows on the links. A good software design should map to general network configurations.

2.2 Occam Programming

Occam is a language designed to minimize the problems that constantly arise in programming concurrent systems. It bears a special relationship with the transputer architecture. Together they make design and implementation of parallel systems much easier. Here, we only discuss some of the most important Occam language features that have profoundly influenced the way the control system is designed.

In the Occam programming model, processes synchronization and mu-

tual exclusion is done through the use of message-based interprocess communication (IPC). In particular, Occam's IPC is

- Indirect: Since a sender process does not send messages to the receiver process directly. It sends to a channel from where the receiver will pick up the messages.

- Symmetric: A channel takes input from exactly one process and sends output to exactly another process. This mapping of a channel to the pair of (sender, receiver) processes is done in compile time.

- Synchronous: When a sender sends a message to a channel, it waits until the message is received by the receiver. A receiver will not proceed when it is expecting a message from a channel.

An example of a set of parallel Occam programs is shown in Figure 4. Channels can be either soft channels (those between processes on the same processor) or hard channels (those mapped to hard links, between processes on different processors). However, it is a configuration time decision to designate channels to be soft or hard channels.

Common variables among parallel processes are allowed, as long as they are used in a read-only fashion. Violation of this rule will be reported as a syntax error by the compiler.

A form of deadlock can still very easily occur. However, the tool set provides little help in debugging problems like this. For example, there are potential deadlocks between Dispatcher and Servers in the code shown in Figure 4. Consider the following scenario:

1. Dispatcher received a command that was to be forwarded to Server 1. Dispatcher then placed this command on channel **To.Svr.1**.

2. At the same time, the **ALT** process got no incoming message. It did some computations and then exited.

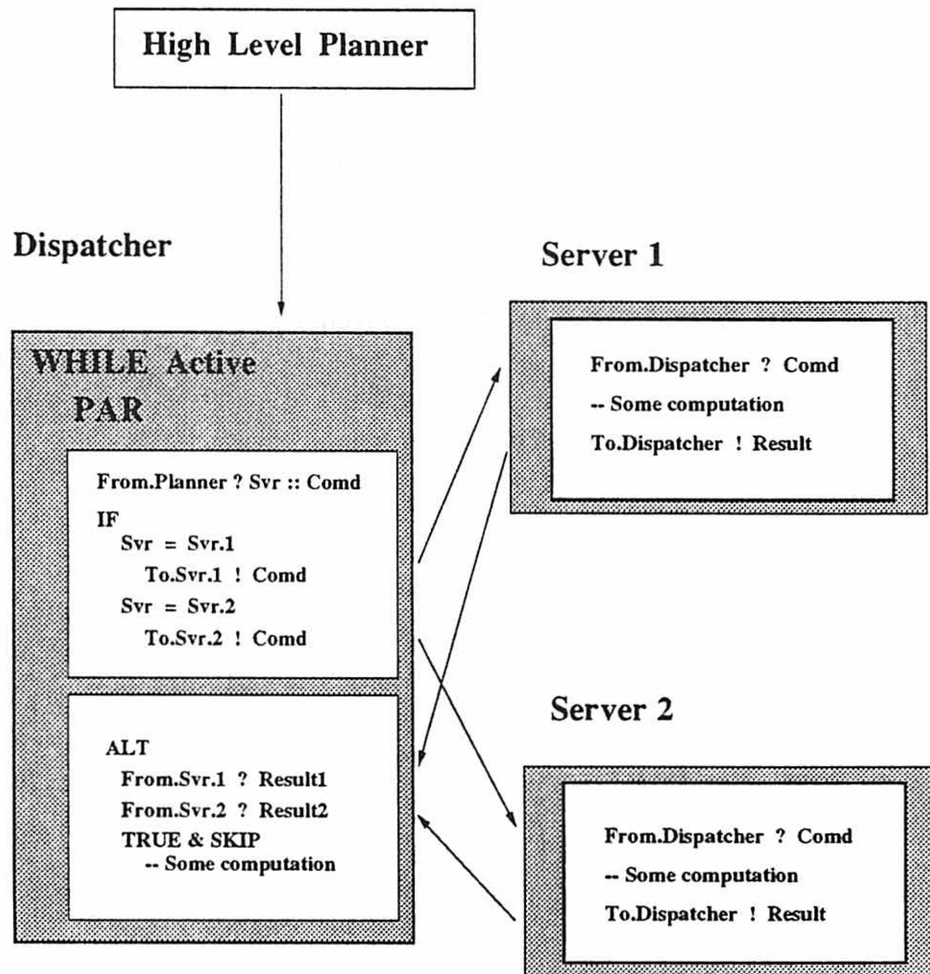

Figure 4: An example of Occam processes

3. Server 1 received the command and started its computation.

4. The **PAR** exited since both its subprocesses had terminated. Since **Active** is still true, the loop started again.

5. Dispatcher received another command that was, again, to be forwarded to Server 1. Dispatcher then placed this command on channel **To.Svr.1**.

6. At the same time, the **ALT** process still did not get any incoming message. It did some computations and then exited.

7. Server 1 finished its computation for the first command and placed the result on channel **To.Dispatcher**

8. Now, both Dispatcher and Server 1 are waiting for the other to pick up the messages. The system is in deadlock.

3 Controller Architecture

The run time of the finger servo loop must satisfy the real-time constraint, which is set at 500 Hz. With this constraint, we need to have at least three dedicated transputers to run the servo loops for three fingers, respectively. We will also need a transputer (the **root** transputer) to connect to the host, a SUN 3. Thus four transputers are the minimal requirement. In order to make the whole system more flexible for future expansion, a task decomposition transputer is added between the root transputer and three servo loop transputers. With this design, the system can be divided into four parts:

- Finger servo loop: 3 transputers run control loops for each finger.
- Dispatcher: The task decomposition transputer. Random tactile event logger and dispatcher.
- Root: The root transputer that connects to the SUN.
- UNIX interface: A UNIX interface that talks to the transputer network.

Figure 5 shows the allocation of processes to processors. We describe these parts in the following subsections.

3.1 Finger Servo Loop

This process implements the control loop. It receives input commands from the Dispatcher (task decomposition transputer), collects feedback from the I/O board and generates output motor current commands to the I/O board. Every transform function in the control loop is implemented as a process, including the Σ function. Although this does not make the control loop faster, it does provide a nice and clean modular design that is very easy

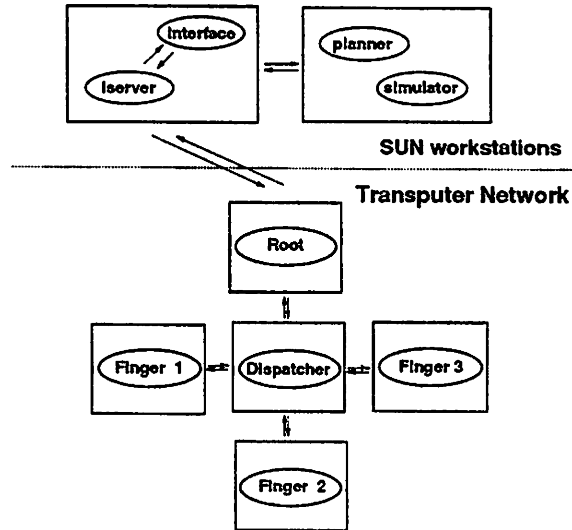

Figure 5: System Overview

to update. And it also makes it possible to run the servo loop on several transputers. The servo control loop is shown in Figure 9 after we derive all the transform equations in the loop in Section 5.

3.2 Dispatcher

The task decomposition transputer is responsible for taking commands from root transputer and delivering them to appropriate servo transputers. It must also collect reports from fingers and forward them to the Root transputer. Experiment shows that this is a very lightly loaded processor. In the future, we expect to use its spare cycle time to receive tactile sensors' data. However, since random tactile events can severly degrade the real-time performance of the system, these events should logged and scheduled by a real-time scheduler.

3.3 Root

The Root transputer talks to the UNIX interface on SUN. It receives commands from the UNIX interface and send the decoded commands to the Dispatcher. It also reports the status of fingers back to the higher level planner. A debugging utility is also built-in in this process.

3.4 UNIX Interface

This is a C process running on SUN. When called, it sets up a pipe and forks out a child process which in turn initiates the **iserver** that loads the network programs. This interface provides the following function calls for the higher level planner:

1. OpenHand(): Initiate the hand.
2. CloseHand(): Stop all the transputer processes. Kill the **iserver** UNIX process.
3. SendPos(i, Pos): Send the desired joint position to finger i.
4. SendKx(i, Kx): Set the stiffness matrix for finger i.
5. GetPos(i, Pos): Get the current position of finger i.

Figure 6 shows an example of how these functions are called from a C program.

```
#include Sun2Xp.c

main( )
{
  OpenHand( );
        ⋮
  SendPos(1, Pos);
  SendKx(1, Kx);
        ⋮
  GetPos(1, Pos);
        ⋮
  CloseHand( )
}
```

Figure 6: Calling the Hand from a C program

4 Controlling the Finger

4.1 Impedance Control

A passive device composed of springs and dampers can be useful in compliant motion control. Drake[1] shows that peg-in-hole insertions can be facilitated by appropriately introducing low lateral and rotational stiffness in the grasping mechanism. Based on this principle, a passive mechanical device known as Remote Center Compliance (RCC) is suggested for this kind of assembly operation[15].

Passive devices such as RCC are typically capable of quick responses, and are relatively inexpensive. However, their applications are limited to specific tasks. Controlling an active manipulator, e.g., a finger, will allow for more general purpose manipulation. While **position control** and **force control** have been heavily studied, they have been shown to be inadequate in many cases. An alternative is to control the dynamic behavior of the manipulator. An example of this approach is **impedance control**[3].

There are only two types of physical systems: admittances and impedances. This concept is best illustrated by Figure 7. When two physical systems (e.g., a finger and an object) have dynamic interactions, one of them must be an impedance and the other must be an admittance.

Since for most tasks the environments are properly described as admittances, the manipulator should behave like an impedance. The lowest-order term in any impedance is the static relation between output force and input displacement. Impedance control of this type is called **stiffness control**.

4.2 Finger Modeling and Stiffness Control

A finger can be modeled as a virtual Cartesian spring. The dimensionality of this spring depends on the type of contact between finger tip and the

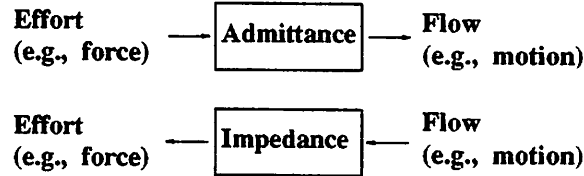

Figure 7: Impedance vs. Admittance

object as well as degrees of freedom of the finger. For example, for a **hard finger with friction** contact, the finger can be modeled as a 3 dimensional spring. For a **soft finger** contact, it can be modeled as a 4 dimensional spring.

After the dimensionality of the spring is determined, the stiffness of this spring can be represented by a D by D Cartesian stiffness matrix K_x. K_θ, the stiffness matrix in joint space, can be derived in the following way. The relation between joint torques τ and finger tip force f is:

$$\tau = J^T f. \tag{1}$$

Stiffness of a finger can be expressed in both Cartesian space and joint space:

$$f = K_x \delta x, \tag{2}$$

$$\tau = K_\theta \delta \theta. \tag{3}$$

Since, by definition,

$$\delta x = J \delta \theta, \tag{4}$$

we can relate K_θ and K_x by combining above equations:

$$\tau = K_\theta \delta \theta = (J^T K_x J) \delta \theta. \tag{5}$$

Therefore,

$$K_\theta = J^T K_x J, \tag{6}$$

which is the equation we need to calculate K_θ from input K_x. This equation also shows how we can build a pseudo Cartesian stiffness controller[8][11].

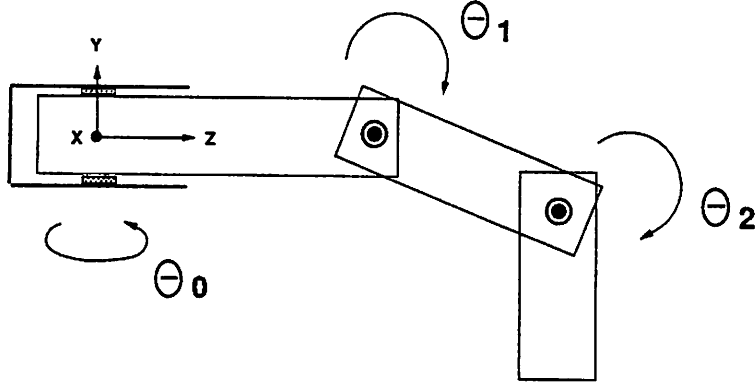

Figure 8: Finger frame

5 Hand Kinematics

5.1 Mechanical Structure Description

The Stanford/JPL hand is an articulated dexterous hand consisting of three fingers, each with three joints. The Jacobian of a finger, based on the coordinate system in Figure 8, can be derived as

$$\begin{bmatrix} \dot{x} \\ y \\ z \end{bmatrix} = J\dot{\theta}, \tag{7}$$

$$J = \begin{bmatrix} (l_0 + l_1C_1 + l_2C_{12})C_0 & (-l_1S_1 - l_2S_{12})S_0 & (-l_2S_{12})S_0 \\ 0 & -l_1C_1 - l_2C_{12} & -l_2C_{12} \\ (l_0 + l_1C_1 + l_2C_{12})(-S_0) & (-l_1S_1 - l_2S_{12})C_0 & (-l_2S_{12})C_0 \end{bmatrix}, \tag{8}$$

where l_i's are link lengths and

$$S_i = \sin(\theta_i), \tag{9}$$

$$C_i = \cos(\theta_i), \tag{10}$$

$$S_{ij} = C_i S_j + S_i C_j, \tag{11}$$

$$C_{ij} = C_i C_j - S_i S_j. \tag{12}$$

Each finger of the hand is actuated by four Teflon coated steel tendons, which in turn are actuated by four DC motors. The contribution of tendon j to the torque on joint i is directly related to the radius r_{ij} of the pulley/shaft at joint i around which tendon j runs. Then the relation between joint torques τ and tendon tensions t is described by:

$$\tau = Rt, \tag{13}$$

where

$$R = \begin{bmatrix} -r_{11} & r_{12} & r_{13} & -r_{14} \\ r_{21} & r_{22} & -r_{23} & -r_{24} \\ 0 & r_{32} & -r_{33} & 0 \end{bmatrix}. \tag{14}$$

5.2 Tendon Tension Control

In order to make R invertible, and to obtain a unique mapping relationship between joint torques and tendon tensions, we add one more row to R and create a "ghost joint" on the finger. Equations 14 and 13 now become

$$R = \begin{bmatrix} -r_{11} & r_{12} & r_{13} & -r_{14} \\ r_{21} & r_{22} & -r_{23} & -r_{24} \\ 0 & r_{32} & -r_{33} & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}. \tag{15}$$

$$\begin{bmatrix} \tau \\ \tau_g \end{bmatrix} = Rt. \tag{16}$$

The problem we will be facing in controlling the finger is:

Given τ, find t such that $t \geq t_{min} > 0$ and $|\ t\ |$ is minimum.

This can be solved by properly choosing the value of τ_g as shown below. First, let $Q = R^{-1}$ and

$$Q = R^{-1} = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ q_{41} & q_{42} & q_{43} & q_{44} \end{bmatrix} = \begin{bmatrix} Q_1 & q_{14} \\ Q_2 & q_{24} \\ Q_3 & q_{34} \\ Q_4 & q_{44} \end{bmatrix}. \qquad (17)$$

Let t_{min} be the required minimum tendon tension, then the following equations hold:

$$t = Q \begin{bmatrix} \tau \\ \tau_g \end{bmatrix}, \qquad (18)$$

$$t_i = Q_i\tau + q_{i4}\tau_g \geq t_{min_i}, \;\; 1 \leq i \leq 4. \qquad (19)$$

So,

$$\tau_g = max\{\frac{(t_{min_i} - Q_i\tau)}{q_{i4}} \mid 1 \leq i \leq 4\}. \qquad (20)$$

5.3 Motor Torque Control

Let M be a diagonal matrix with M_{ii} being the equivalent shaft radius of the motor i that controls tendon i, the relation between tendon tension t and motor torque m_τ is

$$t = Mm_\tau. \qquad (21)$$

Hence

$$m_\tau = M^{-1}t. \qquad (22)$$

5.4 Joint Position from Motor Encoder Readings

From Equations 16 and 21, we have

$$\begin{bmatrix} \tau \\ \tau_g \end{bmatrix} = RMm_\tau. \qquad (23)$$

By the principle of virtual work, the following equation holds:

$$\delta \begin{bmatrix} \theta \\ g_\theta \end{bmatrix}^T \begin{bmatrix} \tau \\ \tau_g \end{bmatrix} = \delta m_\theta^T m_\tau. \tag{24}$$

Substituting in Equation 23, we get

$$\delta \begin{bmatrix} \theta \\ g_\theta \end{bmatrix}^T RM = \delta m_\theta^T. \tag{25}$$

Finally, by transposing both sides, we have

$$\delta \begin{bmatrix} \theta \\ g_\theta \end{bmatrix} = (RM)^{-T} \delta m_\theta. \tag{26}$$

If we set $m_\theta = 0 \Leftrightarrow \begin{bmatrix} \theta \\ g_\theta \end{bmatrix} = 0$, then

$$\begin{bmatrix} \theta \\ g_\theta \end{bmatrix} = (RM)^{-T} m_\theta. \tag{27}$$

The relation between m_θ and the motor encoder reading m_c is

$$m_\theta = \frac{2\pi(m_c - m_0)}{2000}, \tag{28}$$

where m_0 is the motor counter value when $\theta = 0$. Note that the motor encoder emits 2000 pulses per rotation. Combining Equations 27 and 28 together, we have

$$\begin{bmatrix} \theta \\ g_\theta \end{bmatrix} = \frac{2\pi}{2000}(RM)^{-T}(m_c - m_0). \tag{29}$$

This shows how to calculate the current joint positions from motor encoder readings.

5.5 Hand Constants

The hand constants used in this section are listed below, with all lengths measured in *cm*.

1. Link lengths:

$$l_0 = 3.556 \text{ cm}$$

$$l_0 = 5.080 \text{ cm}$$

$$l_0 = 4.000 \text{ cm}$$

2. R matrix:

$$R = \begin{bmatrix} -1.09 & 0.62 & 0.62 & -1.09 \\ 1.09 & 0.62 & -0.62 & 1.09 \\ 0 & 0.62 & -0.62 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

3. M matrix:

$$M = \begin{bmatrix} 0.0475 & 0 & 0 & 0 \\ 0 & 0.0475 & 0 & 0 \\ 0 & 0 & 0.0475 & 0 \\ 0 & 0 & 0 & 0.0475 \end{bmatrix}$$

5.6 Servo Loop Design

Figure 9 shows the current design of the servo control loop. Each box in the loop is implemented as a subprocess with each arrow implemented as a channel. All the equations needed have been derived in previous sections. The following is a description of the transfer functions and their corresponding equations.

- A^{-1}: The transfer function that implements inverse kinematics. Not included in current implementation.

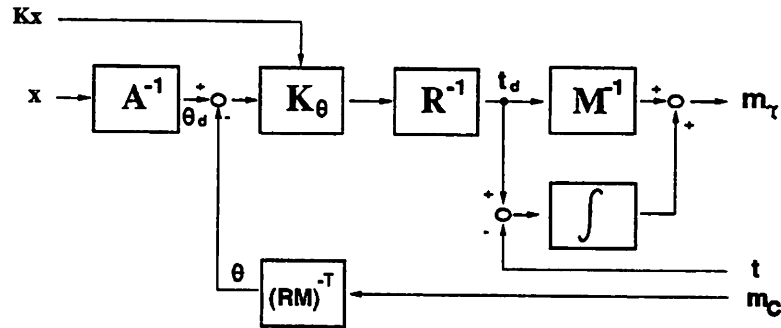

Figure 9: Servo loop

- K_θ: The transfer function that transforms joint position errors to desired joint torques, using Equations 3 and 6.
- $(RM)^{-T}$: The transfer function that derives joint positions from motor counts, by Equation 29.
- R^{-1}: The transfer function the calculates the optimal tendon tension, by Equations 20 and 18.
- M^{-1}: This transfer function that transform tendon tensions to motor torques, using Equation 22.
- $\int$: The integral term which removes steady state errors.

The integral term in the loop could be a source of instability. An alternative, as shown in Figure 10, is suggested by Salisbury in [11]. It includes a damper and a proportional and integral term. Notice that, in his notation, R_m is the M^{-1} we have discussed, $K_T + \frac{K_{TI}}{s}$ is the proportional and integral term, and $K_{\dot{\theta}}$ is the damping constant. As another alternative, Figure 11 shows an adaptive control scheme which adapts the model of the motor. In this

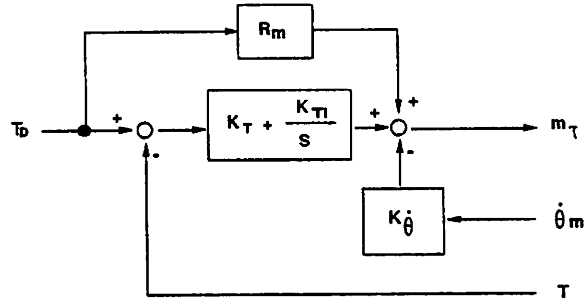

Figure 10: Tendon control system

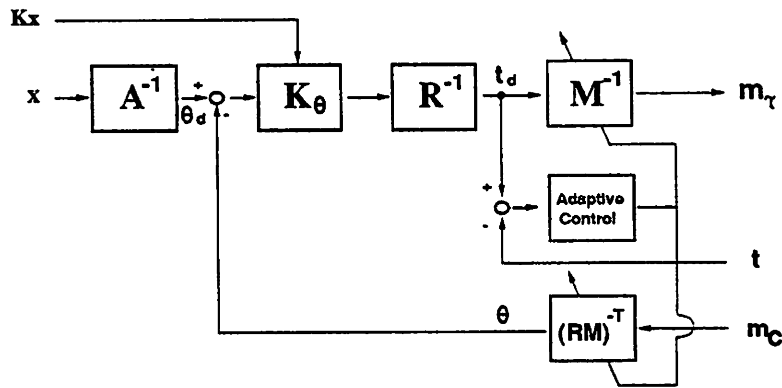

Figure 11: Adaptive Servo Loop

scheme, we can adjust the value of the matrix M by comparing the desired tension and the actual tension. Since M is no longer a constant matrix now, this design will need more computation time for each iteration. However, since a closed form can be obtained for M^{-1} and $(RM)^{-T}$, this should not introduce too much delay.

6 Conclusion

6.1 Performance

At the time of writing this report, we have only two transputers available. We allocate the **Root** process on the root transputer and both **Dispatcher** and one finger's **Servo Loop** on another transputer. We have tested the speed of the **Servo Loop** by two methods:

1. By using the transputer's internal timer:
   For the T800 transputer, one tick on the clock equals to 64 μs. So we can calculate the computation time by counting the number of ticks passed on the clock, minus the time it needs to "get the value of the timer."

2. By using an oscilloscope[2]:
   Every time one iteration of the **Servo Loop** is done, we can toggle a bit in the **Hand Shaking** register designed for controlling I/O devices. This forms a square wave in the oscilloscope reading. The wave's frequency corresponds to the **Servo Loop**'s frequency.

In our testing, both methods give results close to 700 Hz. Considering that the **Dispatcher** is sharing the transputer with the **Servo Loop**, it is clear that the **Servo Loop** can satisfy the real-time constraint, which has been set at 500 Hz.

Three computations in the loop take up most of the time. A matrix multiplication for two 3×3 matrices takes 190 μs. Computing a Jacobian needs 130 μs. Multiplication between a 4×4 matrix and a 4×1 vector takes 47 μs. The whole loop takes 1430 μs for one iteration.

[2]This idea is from Eugene Hertz.

6.2 Real-time scheduling

In the future, this system is expected to incorporate sensors of various types, including tactile sensors, vision, and so on. Proper utilization of this finite resource, the transputer network, thus becomes a very important issue. Reading these sensing devices and servoing fingers can all be treated as tasks with hard or soft real-time constraints. With proper design, they can be implemented as periodic tasks with a fixed amount of computation time. In such static systems with fixed task allocations, an optimal scheduling algorithm could be used to determine sequence and timing for executing this set of tasks in such a way that all tasks can be guaranteed to meet their real-time constraints[12]. Most importantly, real-time scheduling of a static system can be done off-line, therefore, the on-line scheduler can be very fast.

The servo loop could be designed to run faster, for example, the Jacobian may not need to be recalculated for minor changes in joint positions. However, this will compromise the predictability of the system and make it very hard for the future expansion to remain static.

A Source Code Details

A.1 Location

All the source codes are in the directory:

/usr/kanga/users/shaw/hand/source

which includes the configuration file, Occam programs, C programs, header files, makefiles, and example files. The executable controller, **Hand.btl** is in:

/usr/kanga/users/shaw/hand/control

A.2 Description of the Files

The following is a list of file names in **/usr/kanga/users/shaw/hand/source** and their contents.

File	Contents
Hand.pgm	Configuration file for a two-transputer system.
Hand.occ	Main program for a one-transputer system. Can be used with the simulator **isim**.
Hand.net	Makefile for a two-transputer system.
Hand	Makefile for a one-transputer system.
Sun2Xp.h	Header file for **Sun2Xp.c**.
Sun2Xp.c	Unix interface for the controller.
Hand.inc	System constants and channel protocals.
Finger.inc	Local constants and channels for **Servo Loop**.
Root.occ	Program for the **Root** process.
Task.occ	Program for the **Dispatcher** process.
Finger1.occ	Program for the **Servo Loop** process.
Get.occ	Utility program.
Matrix.occ	Codes for matrix operations.
Jacobian.occ	Codes for computing Jacobian.
High.c	An example program on calling the controller.

References

[1] Drake, S.H., Using Compliance In Lieu of Sensory Feedback for Automatic Assembly, Doctoral dissertation, Department of Mechanical Engineering, (1977), Massachusetts Institute of Technology.

[2] Hanafusa, H., Asada, H., A Robot Hand with Elastic Fingers and Its Application to Assembly Process, IFAC Symposium on Information and Control Problems in Manufacturing Technology, Tokyo, (1977), 127-138.

[3] Hogan, N., Impedance Control: An Approach to Manipulation, *Journal of Dynamic Systems, Measurement, and Control*, Vol. 107, (1985).

[4] INMOS Limited, *Occam 2 Reference Manual*, (1988), Prentice Hall.

[5] INMOS Limited, *Occam 2 toolset User Manual*, (1989).

[6] Kerr, J., Roth, B., Analysis of Multifingered Hand, *International Journal of Robotics Research*, Vol. 4, No. 4, (1986), 3-17.

[7] Salisbury, J.K., Articulated Hands: Force Control and Kinematic Issues, *International Journal of Robotics Research*, Vol. 1, No. 1, (1982), 4-17.

[8] Mason, M.T., Salisbury, J.K., *Robot Hands and the Mechanics of Manipulation*, (1985), Mit Press.

[9] Nguyen, V., Constructing Force-Closure Grasps, *International Journal of Robotics Research*, Vol. 7, No. 3, (1988), 3-16.

[10] Nguyen, V., Constructing Force-Closure Grasps, *International Journal of Robotics Research*, Vol. 8, No. 1, (1989), 26-37.

[11] Salisbury, J.K., Craig, J.J., Articulated Hands: Force Control and Kinematic Issues, *International Journal of Robotics Research*, Vol. 1, No. 1, (1982), 4-17.

[12] Stankovic, J.A., Ramamritham, K., *Tutorial: Hard Real-Time systems*, Computer Society Press of IEEE.

[13] Venkataraman, S.T., *Task Dependent Dextrous Hand Control*, Ph.D Thesis, Department of Electrical and Computer Engineering, (1988), Umass, Amherst.

[14] Pountain, D., May, D., *A Tutorial Introduction to Occam Programming*, (1988), BSP Professional Books.

[15] Whitney, D.E., Nevins, J.L., What is Remote Center Compliance and What Can It Do? Proceedings, Ninth ISIR, Washington, D.C., (1979).