Learning Maps Between Sensorimotor Systems on a Humanoid Robot

by

Matthew J. Marjanović

S.B., Massachusetts Institute of Technology (1993)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Author . . . .
Department of Electrical Engineering and Computer Science
May 12, 1995

Certified by . . . .
Rodney A. Brooks
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Learning Maps Between Sensorimotor Systems on a Humanoid Robot

by

Matthew J. Marjanović

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 1995, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

The cerebellum has long been known to be associated with the coordination of the human motor system. Contemporary research indicates that this is one product of the cerebellum's true function: the generation of dynamic models of systems both within and without the body. This thesis describes the instantiation of one such model on the humanoid robot Cog, developed at the MIT Artificial Intelligence Laboratory. The model takes the form of an adaptive mapping of head movements into anticipated motion in the visual field. This model is part of a visual subsystem which allows Cog to detect motion in the environment without being confused by movement of its own head. The author hopes that this work will be the first step in creating a generalized system for generating models between sensorimotor systems, and that such a system will be the first step in the development of a fully-functional artificial cerebellum for Cog.

Thesis Supervisor: Rodney A. Brooks
Title: Professor of Electrical Engineering and Computer Science

Learning Maps Between Sensorimotor Systems on a Humanoid Robot

by

Matthew J. Marjanović

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 1995, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

The cerebellum has long been known to be associated with the coordination of the human motor system. Contemporary research indicates that this is one product of the cerebellum's true function: the generation of dynamic models of systems both within and without the body. This thesis describes the instantiation of one such model on the humanoid robot Cog, developed at the MIT Artificial Intelligence Laboratory. The model takes the form of an adaptive mapping of head movements into anticipated motion in the visual field. This model is part of a visual subsystem which allows Cog to detect motion in the environment without being confused by movement of its own head. The author hopes that this work will be the first step in creating a generalized system for generating models between sensorimotor systems, and that such a system will be the first step in the development of a fully-functional artificial cerebellum for Cog.

Thesis Supervisor: Rodney A. Brooks
Title: Professor of Electrical Engineering and Computer Science

Ack.

This is not my Life's Work.

But, if it's all the same to you, I'd like to thank those people who made this possible, or at least plausible.

First and foremost is **Matt Williamson**, because the mystic visions behind this project were the outcome of a particularly inspired conversation we had, months ago. Second and foremost is, of course, **Rod Brooks**. Rod has made Cog possible, has made my graduate career plausible, and has signed this document.

Thanks to those Usual Suspects: **Cindy** and **Yoky**, officemates par excellent, who made the PleasureDome possible (and that is only the beginning); **Scaz** and **Robert**, who have shared these last few late nights with me, albeit in disparate parts of the building.

Mike Bolotski is still a nice guy. He would be even if he didn't drive me to hockey games.

Maja and **Rich**, the happy couple, deserve mention. Rich is a dude; Maja incessently offered to help with proofing this thesis. Maja also got me involved in all this; send all complaints to her.

(Dear Reader: Listen to your Jesus Jones CD's again. Nothing has changed.)

I'd like to thank **Jose**. You gotta give that guy some credit, so I just did.

My apologies to **Liana**; this thesis is not a choose-your-own-adventure novel. I'll save that spark for later, I suppose.

Mike Binnard — he built Cog, you know. Solid, bulletproof. Too bad he and **Gina** have swept each other away to the Sunshine.

Eric Jordan, my dear old friend Eric Jordan, deserves a bullet in the head for no other reason than his constant comment of "Give it up." Of course, he did it for me; look, I finished, didn't I? Lucky I didn't punch him out is all I can say. Thanks. I do appreciate it.

Roommates **Diane** and **Sinan**, on the other hand, gave me the more traditional words of encouragement. Will the cleaning ever end?!?

Farewell to **Lisa T**, although she's not going so far. The connection will always be there.

Greetings to **Evan R**. He is a friend, despite his worries to the contrary.

Hi-5 to **Dave B**. The week of Sunshine he gave to the Matts did wonders for my complexion.

Charles Isbell. What a man. Need I say more?

Did I mention that all the cheesecakes were very delicious? So was the jello. Green and red.

Finally, of course, **Mom** and **Tash**, because I wouldn't be me without the two of them.

Welcome to *ThesisLite*. Sit back, pop the top, and enjoy.

—Matto

Contents

List of Figures

Chapter 1

Background

This thesis describes some preliminary work I have done with the visuomotor system of Cog, a robot currently under development at the MIT Artificial Intelligence Laboratory. Cog is a humanoid robot, designed to be able to interact with the world in a human-like fashion, hopefully with human-like capacities. The purpose of the Cog project is to learn about human cognition by attempting to simulate or replicate it, with the underlying hypothesis that cognition is shaped by interaction with the world.

As one of the first toddling steps towards that goal, I implemented a system which learns a model of head movement in terms of the visual motion it induces by moving the camera. This is one stab at what I hope will eventually be many such models linking many (all) of Cog's sensorimotor systems together. I have implemented a single form of such a model, based on a network of linear approximators; many alternatives exist. In the future, I would like to have a system which automatically generates these functional maps between systems, picking the appropriate format or using a sufficiently general and powerful template.

This idea of a phalanx of internal models is inspired by the cerebellum. Another higher level goal of mine is to implement cerebellar functions on Cog; a large part of that task involves figuring out just what the cerebellum does. One current view is that the cerebellum makes dynamic models of interactions of the body with the world, as well as models of internal systems and their interactions. Along these lines, generating models of sensorimotor systems is an implementation of cerebellar function.

From a more pragmatic perspective, as Cog becomes more developed, it will have more and more specialized computational systems which will need to cooperate with each other. Models which translate the actions of one system into the native parameters of another will facilitate such cooperation. Once the number of subsystems on Cog exceeds a handful, an infrastructure which automatically generates such adaptive models will make the entire programming experience much more manageable.

Cerebellum

The primary inspiration for this project comes from the cerebellum, an organ which is well-known but little-understood. In humans, the cerebellum is a corrugated golf ball of neurons tucked into the back of the skull between the occipital lobe and the brainstem (Figure 1-1). Its cortex is about 1mm thick, but deeply folded, and comprises about half the neurons in the brain. Unlike the cerebral cortex, it has no central fissure. The microscopic structure of the cerebellar cortex is crystalline compared to the cerebral cortex; of the five major cell types, four are oriented and located in well-defined perpendicular planes. The cerebella of other species have roughly the same structure; all vertebrates have a cerebellum. A full discussion of the intricacies of cerebellar physiology can be found in (Ito 1984), or (Eccles et al. 1967).

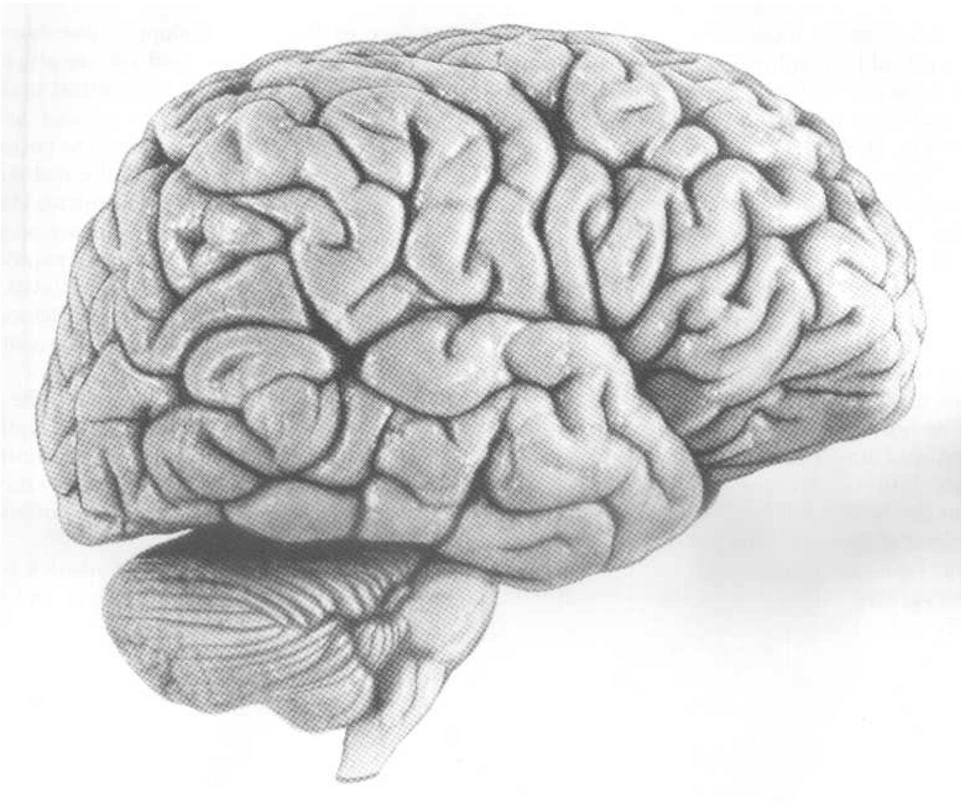


Figure 1-1: Sagittal view of the brain. The cerebellum is the finely folded structure in the lower left, tucked under the cerebrum and behind the brainstem (Noback 1981, p.3)

The first major step in unraveling the functioning of the cerebellum was made by Holmes (1917), a military doctor who carefully documented the motor deficits of WWI soldiers suffering from head wounds. This became the first volume of a great body of evidence pointing towards the human cerebellum's involvement in motor control.

So far, via such lesion studies, we know that the cerebellum is involved in both autonomic and voluntary motor activity. The medial cerebellum controls reflexes such as the vestibular ocular response (VOR). The paramedial cerebellum helps with posture and stance. The lateral cerebellum plays a part in coordinating voluntary motor activity. Lesions of the lateral cerebellum cause well-known motor deficits (Dichgans & Diener 1985):

- diminished tendon reflexes (*hypotonia*)
- weak, easily tired muscles (*asthenia*)
- decomposition of movements into component parts
- inability to correctly gauge distance of movements (*dysmetria*)
- inability to perform smooth alternating, repetitive movements (*dysdiadochokinesia*)
- oscillation in voluntary motion (*intention tremor*)
- drunken-like walking (*ataxic gait*)
- slow, explosive, slurred speech (*scanning speech*).

Note that in no case does cerebellar damage prevent voluntary movement; it just degrades the quality.

The outcome of such lesion data was the general hypothesis that the cerebellum's function is to learn motor programs for common, repeated motions, such as reaching out to different points in space. These programs would be used by the cerebral motor cortex to create smooth, well-controlled voluntary motions. Deficits such as intention tremor become a result of the cerebral cortex being forced to use its own slow feedback loops to control activity which should have been directed by a feedforward motor program. Marr (1969) and Albus (1971) introduced the first of the modern theories that try to explain how the cerebellum accomplishes this learning and control. Both theories mapped the cerebellar microstructure into a three-layer network of perceptrons which acted as an associative memory system.

Modelling

In the last twenty-five years, the field has evolved to a more elegant hypothesis, that the cerebellum's function is to create and learn dynamic models of the body and the body's interaction with the world. These models are not restricted to the motor systems, but encompass sensory systems as well, and may even include models of cognitive subsystems within the brain.

Paulin (1993*b*) cites examples of vertebrates which have large and highly developed cerebella, but primitive motor skills. The Mormyrid fish, one such example, are electric fish which use electrolocation to navigate and feed, and the enlarged portions of their cerebella are tied to electrodetection. In mammals which rely on echolocation, such as bats and dolphins, certain lobules of the cerebellum which react to input from the auditory system are very highly-developed, compared to lobules related to wings or fins. In all these cases the cerebellum plays a greater role in processing sensory information than in regulating motor output.

Daum et al. (1993) cite recent reports in which cerebellar lesions in people result in:

- impairments in visuospatial recall and in three dimensional manipulation
- deficits in anticipatory planning
- deficits in verbal learning
- changes in visuospatial organizational abilities.

Furthermore, a study by Akshoomoff & Courchesne (1992) gives evidence that the cerebellum plays a role in quickly shifting attention between different sensory modalities.

All of these are hard to explain if the cerebellum's function is restricted to motor programming; however, these and motor programming can all be explained by viewing the cerebellum as a dynamic systems modeller. Historically, it may just be that the motor deficits were the most obvious results of cerebellar dysfunction in humans.

Modelling the Modeller

This brings us back around to my own project. Many theories have been advanced to try to explain how the cerebellum can learn and maintain such models; no one has yet answered that question. Paulin thinks that the cerebellum acts as a Kalman filter (Paulin 1993*a*). Miall, Weir et al. (1993) advocate the idea that the cerebellum is a Smith predictor, a system which models time delays in the control loop as well as kinematics. Keeler (1990) returns to the Marr-Albus models, but recasts them as prediction generators rather than motor program memories. Kawato et al. (1987) model the cerebellum as networks of non-linear elements which learn both forward and inverse dynamics from each other.

All of this has inspired the simple system implemented on Cog, which learns a kinematic model of the head in terms of vision. The learning is performed by a network of linear experts (Section 3.2.2). This system is the first solid step towards an implementation of cerebellar functions on Cog.

Chapter 2

Apparatus

This project was implemented on the robot Cog; one purpose of the project was to see just what Cog was capable of at this early stage in its electromechanical development. This chapter gives an overview of Cog's mechanical and electrical features, followed by a more detailed look at the framegrabbers, motor control software, and IPSmacrolanguage. These are hardware and software systems which I designed myself and which are a integral parts of this project.

2.1 Cog in General

Cog is a robotic humanoid torso (Figure 2-1). Mechanically, Cog is an approximation of the human frame from the waist up, with all the major degrees of freedom [dof] accounted for, except for the flexible spine. Electronically, Cog is a heterogeneous, open architecture, multi-processor computer.

Body

Cog's body consists of a torso, a head, and two arms. The torso is mounted to a large, sturdy machine base via 2-dof "hips"; Cog has no legs. The head is mounted on a 3-dof neck, and the neck and arms are connected to a 1-dof shoulder assembly which rotates about the vertical axis of Cog's torso. Cog's arms are 6-dof mechanisms with compliant (ie. spring-coupled) actuators (Williamson 1995). Hands for Cog have also been designed and are currently being tested (Matsuoka 1995). (Figure 2-2, A, B)

All of Cog's actuators use electric motors with optical encoders to measure joint position. The motors are controlled by dedicated 68HC11 microprocessors, one processor per motor. The 68HC11's regulate motor power via a pulse-width modulation (PWM) circuit, and can measure motor current, temperature, and position.

Cog's head holds two 2-dof eyes, each composed of two miniature black and white CCD cameras. One camera has a wide-angle lens with a 115° field of view to cover full-field vision. The other camera has a 23° narrow-angle lens to approximate foveal vision. The cameras interface to Cog's brain via custom-built frame-grabbers (see Section 2.2). At this moment, the cameras are the only sensors on Cog's head, however Cog will soon have hearing (Irie 1995) and a vestibular apparatus as well.

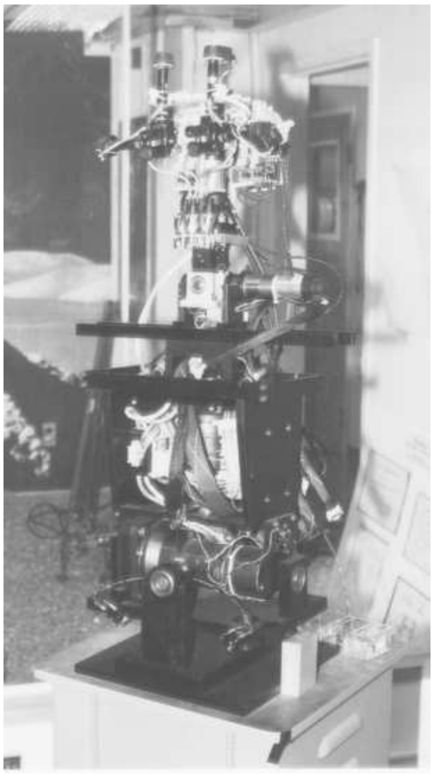


Figure 2-1: Front view of Cog, a humanoid robot from the waist up. The arms have been taken off for table-top testing.

Cog will eventually also have tactile sensors around its body.

Brain

The architecture of Cog's brain is based on a backplane containing multiple independent processor boards (Kapogiannis 1994). Processors can be sparsely connected one-to-another via dual-ported RAM (DPRAM). The backplane supplies the processors with power and a common connection to a front-end processor (FEP), which provides file service and tty access. In the current scheme, up to 16 backplanes can each support 16 processors.

We are currently using Motorola 68332 microcontrollers as the main processors. These are 16 MHz variants of the 68020, with a 16-bit data bus and on-chip serial I/O hardware. Each processor is outfitted with 2 Mb of RAM and each runs its own image of `L`, a multitasking dialect of Lisp written by Rod Brooks (1994). The processors have six DPRAM ports which can be used for communication with other processors, or for interfacing to framegrabbers and video display boards. They also have a 2 Mbit/s synchronous serial port through which they connect to the peripheral motor control boards.

The brain is described as an open architecture system because it could conceivably

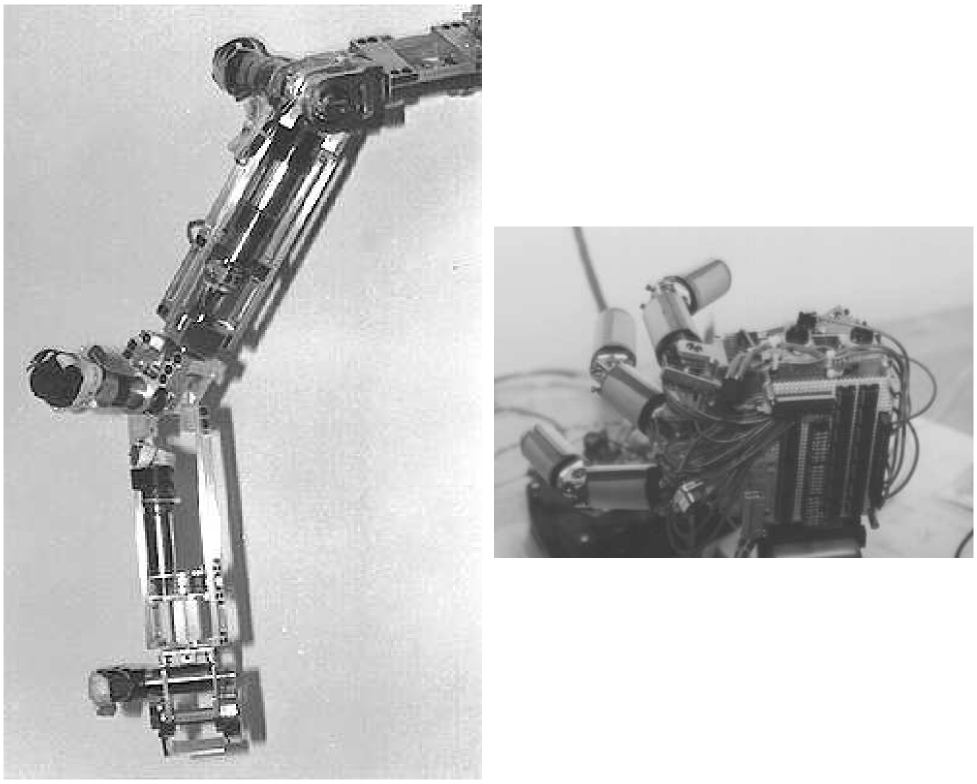


Figure 2-2: One of Cog's arms and one of Cog's hands, currently disembodied.

contain any processor which uses the same FEP and DPRAM interfaces. A C-40 based DSP board with DPRAM interface has been built and will soon be integrated into Cog for use with audition and vision.

2.2 Frame Grabbers

Cog's vision system consists of four black and white commercial CCD cameras and four custom-built framegrabbers (Figure 2-3), which were designed and built in-house by me. The primary motivation for designing our own was the issue of interfacing to Cog's brain. Any over-the-counter framegrabber would have needed an add-on DPRAM interface. Monochrome framegrabbers are not very complex; it was cheaper and easier to build our own grabbers which stored images directly into DPRAM's.

The framegrabbers operate in real-time, dumping one 128×128 pixel frame to DPRAM every $^1/_{30}$ second. Each pixel is 8-bit grayscale; 128×128 is the largest image array that will fit into a DPRAM. After a frame is complete, the DPRAM generates an interrupt request on the 68332 processor to indicate that the frame is ready for processing. Each framegrabber has slots for six DPRAM cards, so that six processors can simultaneously and independently access the same camera image. Each grabber also has a video output for debugging which can display the raw digitized image or

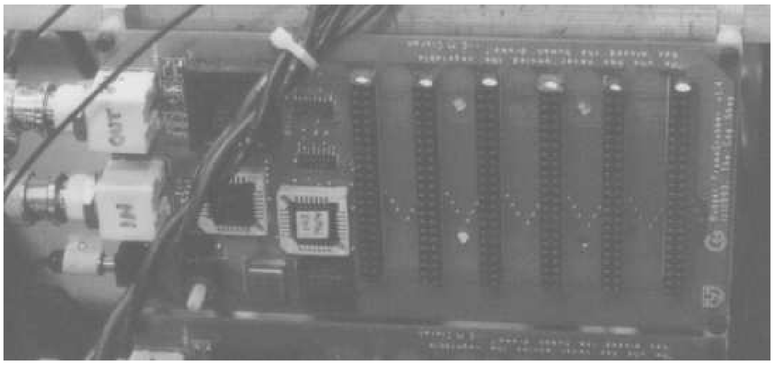


Figure 2-3: One of Cog's home-built real-time framegrabbers. The circuit board is dominated by the six ports for DPRAM cards.

the contents of a DPRAM.

The basic outline of a framegrabber is shown in Figure 2-4 (the complete schematic can be found in Appendix A). This circuit is a conglomeration of single-chip solutions (which explains why it is relatively cheap and easy to build). A sync separator chip extracts timing signals from the video input. A Brooktree Bt218 analog-to-digital converter (Bro 1993) digitizes the video input and puts the pixel data onto a bus. Two counter chips count pixel column and row, and control addressing of the DPRAM's. A Brooktree Bt121 videoDAC generates a video output of whatever is on the data bus for debugging. All of the support logic is contained in two programmable array logic (PAL) chips. The Decode PAL controls read/write to the data bus and handles the debugging switches. The Clock PAL generates the pixel clock, various blanking pulses, and the write pulses for the DPRAM's. A 19.66 MHz crystal oscillator drives the Clock PAL.

The 19.66 MHz oscillator is divided down to 2.46 MHz for the pixel clock by the Clock PAL — this yields a 128-pixel horizontal line. The video signal has 525 lines vertically; we get 128-pixel vertical resolution by digitizing only every other line of every other field. Each video frame is transmitted as two interlaced fields (Roberts 1985), one of which is completely ignored by the framegrabber.

Each of the video fields is $^1/_{60}$ of a second long. The grabber writes to a DPRAM during the even field, but ignores the DPRAM during the odd field. This gives a processor 17 ms to respond to the image-ready interrupt and to use the data before it gets clobbered. With the current system software, the frame is immediately copied from DPRAM into local memory. Thus, there is still a complete 33 ms latency (one video frame) between when the image is captured and when the brain sees it.

Note that the circuit does not contain a phase-locked loop (PLL). The sytem clock (the oscillator) is not directly synchronized with the video input signal. However, the pixel clock divider is reset with every horizontal line. So, there is at most a $^1/_8$ pixel jitter between lines.

The framegrabbers have gone through a few revisions, mostly via PAL firmware

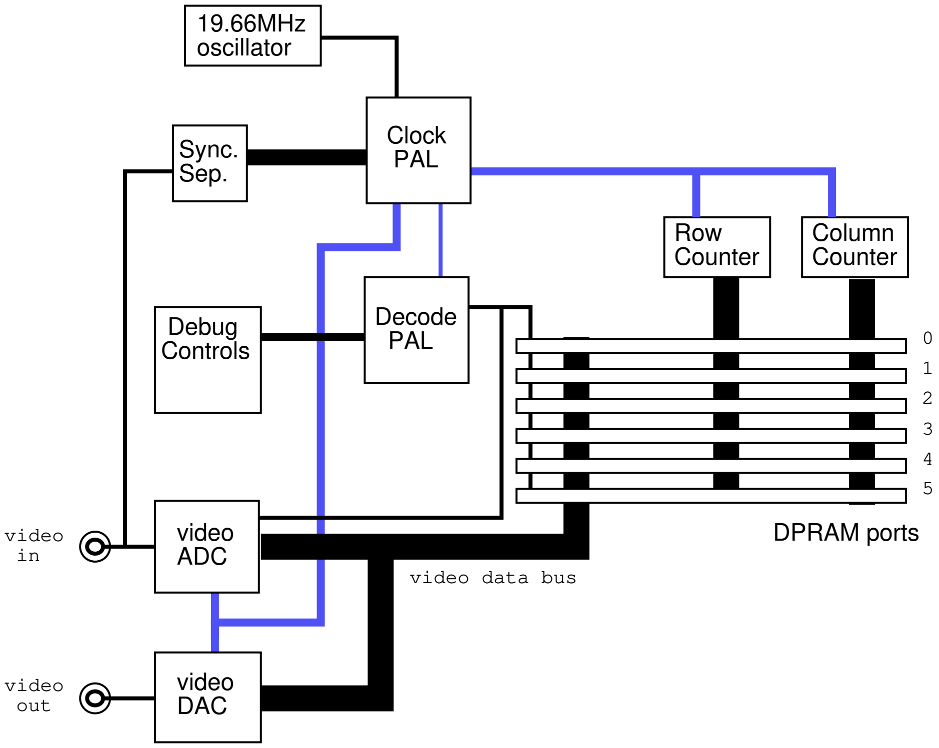


Figure 2-4: Functional diagram of a framegrabber.

upgrades. Since almost all of the control logic is routed through the PAL's, overhauls of the system are much easier and cheaper than if the logic had been hardwired. Framegrabbers v1.4.3 were used for this project; a new set of PC boards with extra circuitry for improved timing and image stability (v.1.5) are almost ready to be put into service.

2.3 InterProcess Socks

L is a multi-tasking dialect of Lisp, which allows multiple procedures to execute with simulated concurrency. However, L has very little in the way of built-in process control, and no provisions for interprocess communication. To fill these gaps, I wrote IPS.

IPS, or InterProcess Socks, is a macro package which provides for simple but very convenient data sharing and process control via two abstractions, the `svariable` and the `sprocess`.

A single `svariable` (for *shared*-variable) consists of two parts, a datum and a flag. A lone `svariable` is no better than a regular variable; the fun starts when many svariables are `connected` together to form a connection group. Connected svariables share the same data, via an indirected pointer, but have independent flags.

When the common data is changed by access through one svariable, the flags on all connected svariables are set. Svariables can also be `disconnect`ed, or separated from their groups.

A `sprocess` (for *sharing*-process) is a kind of execution-independent process object with input and output ports. A sprocess contains executable code which can be spawned to create a real, running process; this process can be killed and respawned without obliterating the sprocess. A sprocess also has *port variables*, variables which are lexically bound within the sprocess body but can be accessed outside of the sprocess. Port variables can contain any data structure, but by default they are svariables, which means that the ports of different sprocesses can be connected together. The body of a sprocess is specially parsed so that the data fork of a shared port variable can be accessed by name like any other variable.

Shared port variables are the cornerstone of IPS. They allow the programmer to create individual lexically-scoped processes and to easily specify data connections between them. All of the code for this thesis is written using IPS. Each subsystem on Cog is composed of individual sprocesses —motor handlers, video frame handlers, calibration processes, etc.— which are linked together by connected svariables.

A full description of IPS can be found in Chapter 2 of the *HumOS Reference Manual* (Brooks, Marjanović, Wessler et al. 1994); a quick overview of the command syntax is given in Appendix B.

2.4 Motor Control

Each of Cog's motors is driven by a dedicated peripheral controller board. These boards contain a 68HC11 8-bit microprocessor, a pulse-width modulation (PWM) circuit, a high power H-bridge for motor current control, amplifiers for current and temperature sensors, and an encoder chip to keep track of joint position. Each microcontroller communicates with a 68332 processor via a synchronous serial line by exchanging 16-byte packets containing 13 bytes of data, tag bits, and a checksum. A single 68332 can service eight controller boards with a total throughput of up to 400 packets per second. In the case of the neck, with three motors and controllers hooked up to one processor, the data rate is a comfortable 100 Hz apiece.

The synchronous serial protocol specifies that the 68332 is the master port, initiating byte transfers between it and a 68HC11. In general, communication from the 68332 side is driven by a motor handler coded as an sprocess using IPS. The handler has svariables for each of the command parameters and regularly stuffs the data into the appropriate packets and ships them off. Return packets are parsed and the sensor data is placed into other svariables. Any other sprocesses which want to use this data, or want to command the motor, simply need connections to the right ports in the motor handler.

The motor controllers havea couple of built-in safety features. If communication with the master 68332 processor is lost for more than 250 ms, or if the 68HC11 hangs, it will automatically reset itself and shut down its motor. Furthermore, whenever a controller is restarted, it waits to handshake with the master processor before starting

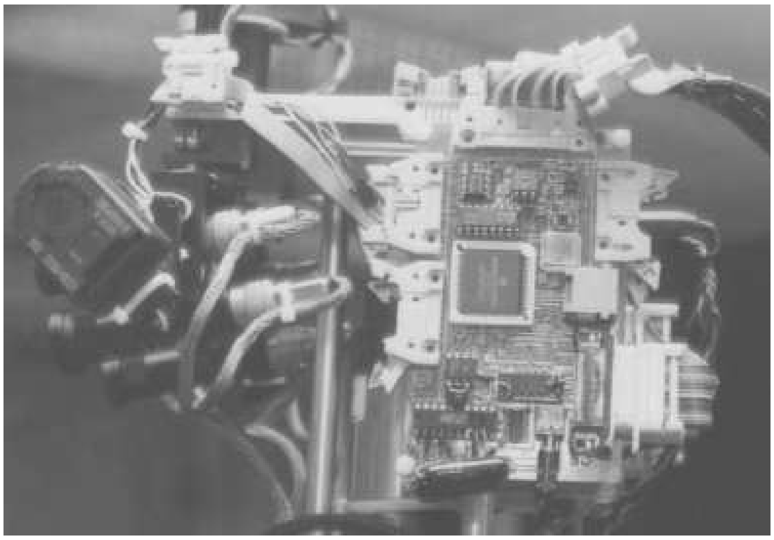


Figure 2-5: A 68HC11 peripheral motor controller board, in this case mounted on the head to control an eye motor. The various ribbon cables are for power, sensor and encoder input, motor output, and communication with a master 68332 processor.

its main program. This *startup-interlock* involves the exchange of three query packets and three acknowledgement packets to make sure that the computers on both ends of the serial link are ready and willing to start processing. During the interlock, the controller also identifies what type and version of code it is running, and gives a guess as to why it was last reset.

The bulk of the code on a motor controller is for servocontrol. The servo code decides just what kind of muscle a motor looks like to the rest of Cog's brain. After reviewing some literature on theories of biological motor control (Bizzi, Hogan et al. 1992), and equilibrium point control (McIntyre & Bizzi 1993) in particular, I decided to implement a basic proportional-derivative (PD) controller for the neck motors. The PD controller has adjustable gains so it is possible to execute equillibrium point guided movements from the 68332, and as a fallback, it always has plain position control.

Other actuators on Cog use different servocontrol code. The eye controllers have proportional-integral-derivative (PID) controllers optimized to perform unassisted position-directed saccades. The arm controllers have torque feedback, and take PD control to the point of simulating virtual springs in the joints.

Chapter 3

Task

The task at the root of this project is to accurately determine motion in the visual scene in spite of any movement of the head. The basic set-up is illustrated in Figure 3-1. The head has two degrees of freedom, pan θ_p and tilt θ_t.[1] When the head moves, the visual field of the camera is swept across the scene. If Cog is looking for motion in the scene, it will mistakenly determine that the whole world has moved, unless the motion detection processes receive notice of head movement and can compensate accordingly.

This task has three components:

- detecting motion
- moving the head
- learning the transfer function between head motion and scene motion.

The transfer function depends on the head and camera geometries. This function could be hard-coded, but learning it yields a more robust system. Continuous, slow learning allows this system to adapt to inevitable changes such as motor shaft slippage and mechanical disalignment. Learning can also be easier: why waste time measuring parameters which are due to change if the system can figure out the parameters for itself?

The solution to this task is embodied in the control loop illustrated in Figure 3-2. A motion detection system outputs vectors which show motion velocity in the scene. At the same time, the head moves because the neck motors are commanded by some motion generator. Neck positions and differential movement are fed into a network which produces a velocity correction for the motion detector. Assuming that most of a visual scene is still and that most movement is not correlated to head motion, the corrected detector output should be zero across the scene. So, the corrected output itself can be used as the error signal to train the network.

Any real movement in the scene will generate noise in the network training input. If the learning rate is low, this noise will not significantly disturb the network. The

---

[1] Cog's neck has a third degree of freedom, roll, which is held fixed in this project, along with the eyes. Integrating three more degrees of freedom is another project altogether (See Chapter 5).

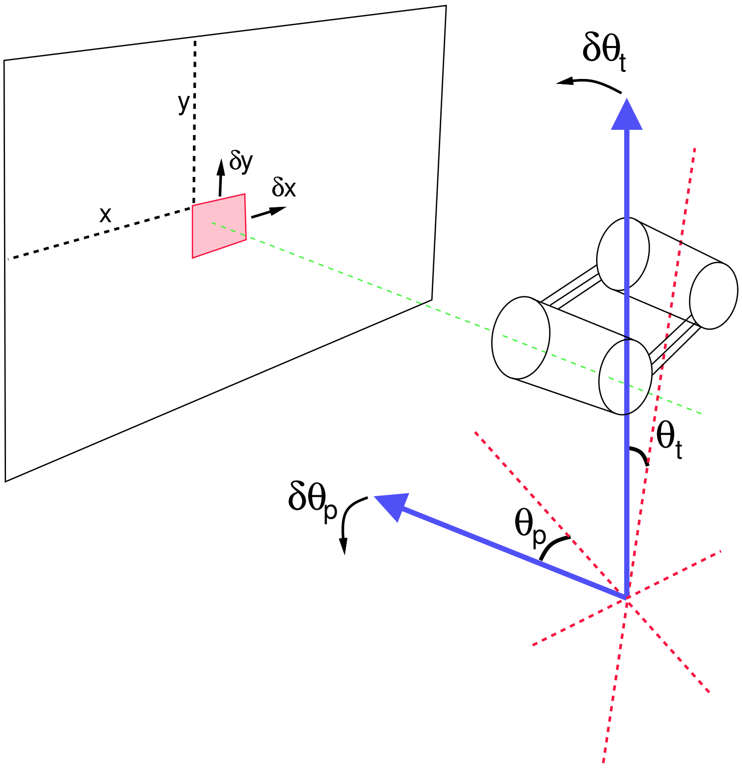


Figure 3-1: Parameterization of the motion detection task. (x, y) is a coordinate in the visual field; (θ_p, θ_t) represents the orientation of the head. When the head turns by $(\delta\theta_p, \delta\theta_t)$, the shaded patch appears to move by $(\delta x, \delta y)$.

entire system will give correct velocity outputs for real motion in the scene, while continuously learning to mask out false motion induced by a moving head.

3.1 Sensorimotor Subsystems

My goal is to learn maps between sensorimotor systems, and I chose neck motor control and motion detection as the first pair because they were available. When I started on this project (and actually continuing to now), the only reliably functioning parts on Cog were the neck motors, the eye motors, and the cameras and frame grabbers. The eye motors and their servo code were the subject of some experimentation by a fellow graduate student (Cynthia Ferrell), so I effectively had only the neck and cameras to work with — one sensor system and one actuator system.

What I came up with is a simple but extensible motion detection system, and a rather degenerate neck motion generator. The motion detection system will probably be a permanent feature on Cog, in a more refined form. The motion generator will hopefully be unnecessary later on.

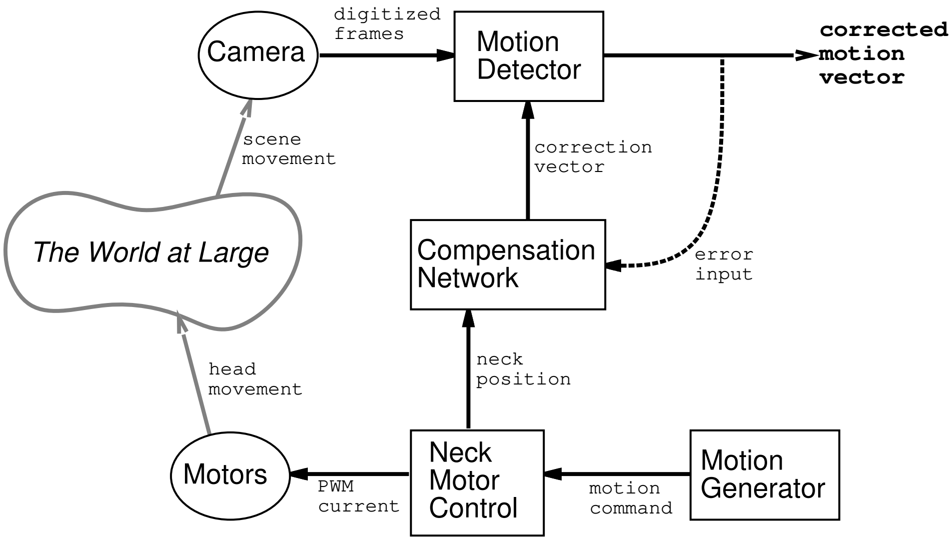


Figure 3-2: The basic control and training loops of the motion detection task. The compensation network learns the effect of head movement on scene movement in order to cancel it out.

3.1.1 Motion Detection

The motion detection system consists of a number of independent detector processes, each of which analyzes the motion of a small patch of the visual field. The output of each detector is a motion vector indicating the velocity of the patch. Taken together, the resulting vector field yields a coarse approximation of the optical flow in the scene (Horn 1986, p. 278ff). Due to processing limitations, only a single detector process was ever used at once, though.

A detector operates on a 16×16–pixel patch located at some position (x, y) in the visual field (recall that the full field is 128×128). The detector operates as follows (refer to Figure 3-3):

1. It grabs and stores a 24×24 patch centered over the 16×16 patch at (x, y).
2. From the next successive video frame, it grabs the 16×16 patch.
3. It searches for the best correlation of the 16×16 patch within the 24×24 patch. The offset of the best match from the center yields the motion vector.

The best correlation is found by calculating the Hamming distance between the 16×16 patch and its overlap at each of 81 positions within the 24×24 patch. The position with the lowest distance wins.

To make even a single detector operate in real-time on a 68332 processor, each 8-bit grayscale pixel in a patch is reduced to 1-bit by thresholding with the average

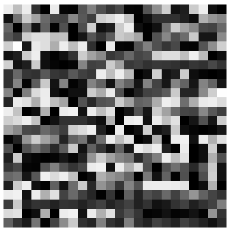


Figure 3-3: Illustration of the motion detection algorithm. Image patches from two video frames are grabbed, thresholded, and then correlated. The offset of the best correlation from the center gives the motion vector.

pixel value of the entire patch.[2] This allows an entire row of pixels to be packed into a single machine word, and then two rows can be correlated using a single XOR machine instruction.

To speed up the process even more, the averaging, thresholding, and correlating routines were coded in assembly language. An assembly-coded correlation is twice as fast as an L-coded version, taking 11.8 ms versus 25.6 ms.

Compensation

To be useful in this project, a motion detector needs the capacity to compensate for head motion error. To achieve this, each motion detector takes a correction vector $\vec{v}_c$

---

[2] The average value of the first 24×24 patch is used to threshold both it and the subsequent 16×16 patch.

as input. This vector is a measure of how far the image is expected to have moved between video frames. The 16 × 16 patch from the second frame in the sequence is grabbed from a location offset by this amount (Figure 3-4). This effectively subtracts $\vec{v}_c$ from the motion vector $\vec{v}$ determined by the correlation, yielding a corrected output $\vec{v} - \vec{v}_c$. By giving the detector a hint where to look for the image, the limited dynamic range of the motion detector is not made any worse by head movement.

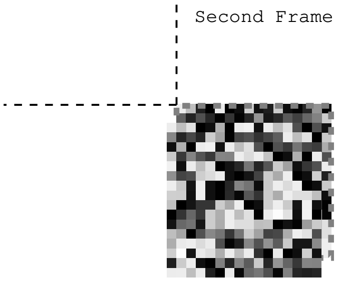


Figure 3-4: Illustration of error compensation in the motion detector. Since the scene is expected to have shifted by $\vec{v}_c$, the second patch is grabbed from an offset of $\vec{v}_c$.

Performance

By itself, a single motion detector runs in real-time on a 68332, processing every frame at 30 frames/second. With the network and learning code included, this drops to 3 updates per second. Since the search space for the correlation is 9×9, the motion vector output is in the range ± 4, which translates into $\pm 120°$/sec at full-speed, or $\pm 11°$/sec as part of the complete system. The 3-bit precision is rather paltry, and the dynamic range is pretty small, but the system works for the regular head movement it faces in this project. To improve either range or precision would require more processing power than available on a single 68332.

Due to the 1-bit thresholding, this motion detection scheme works best on high-contrast objects which occupy the entire 16×16 patch area. This is not so good for registering hands waving in front of the camera. On the other hand, this is perfectly adequate for analyzing the motion of the background scene, which is mostly what the detector needs to do.

3.1.2 Neck Motion Control

To learn to correlate head movement with visual motion, Cog's head needs to move around. In the future, Cog will have processes that will want to move its head, to track motion or to fixate on a sound source, for example. Right now, unfortunately, Cog's head has no particular reason to move at all.

For the purposes of this project, I concocted a random motion generator. This process chooses random targets in the head's range of motion and directs the head to each target in a smooth motion via position control. The motion can be constrained to vertical or horizontal directions only in order to test the learning algorithm.

The speed of generated motion along each axis is nominally 8°/sec, tuned to fit nicely in the range of the motion detectors.

3.2 Learning the Transform

3.2.1 Geometric Analysis

Without too many simplifying assumptions, one can find an analytic solution for the position (x, y) of an object in the camera field, given the head orientation and the position of the object in space.

Let us assume that the neutral point of the camera optics is centered on the pivot point of the neck; this is not the case, but I am deferring the corrections until later. Refering to Figure 3-5, let $\vec{r}$ be the target position relative to the camera. When the head is in full upright and locked position, the camera orientation is described by axial unit vector $\hat{c}_{||0} = (0, 0, 1)$; the camera horizontal is described by perpendicular unit vector $\hat{c}_{\perp 0} = (1, 0, 0)$.

When the head moves, tilt angle θ_t specifies the elevation of the camera's horizontal plane off the horizon, and pan angle θ_p specifies the rotation of the camera in that plane. The camera orientation is modified by a rotation matrix $\mathbf{R}$, yielding $\hat{c}_{||} = \mathbf{R}\hat{c}_{||0}$ and $\hat{c}_{\perp} = \mathbf{R}\hat{c}_{\perp 0}$, where

$$\mathbf{R} = \begin{pmatrix} \cos\theta_p & 0 & -\sin\theta_p \\ \sin\theta_p \sin\theta_t & \cos\theta_t & \cos\theta_p \sin\theta_t \\ \sin\theta_p \cos\theta_t & -\sin\theta_t & \cos\theta_p \cos\theta_t \end{pmatrix}.$$

In order to work out the optics, we need to express the target position in terms of θ_{or} and $\theta_{o\phi}$, where θ_{or} is the angle betweeen $\hat{c}_{||}$ and $\vec{r}$, and $\theta_{o\phi}$ is the angle of $\vec{r}$ relative to the camera horizontal. Let $\vec{r}_{||} = (\vec{r} \cdot \hat{c}_{||})\hat{c}_{||}$, the projection of $\vec{r}$ onto the camera axis. Then we get:

$$\cos\theta_{or} = \frac{\vec{r} \cdot \hat{c}_{||}}{r} = \hat{r} \cdot \hat{c}_{||}$$

and

$$\cos\theta_{o\phi} = \frac{\vec{r} \cdot \hat{c}_{\perp}}{||\vec{r} - \vec{r}_{||}||} = \frac{\vec{r} \cdot \hat{c}_{\perp}}{r \sin\theta_{or}} = \frac{\hat{r} \cdot \hat{c}_{\perp}}{\sin\theta_{or}}.$$

Note that both of these are independent of the object depth r.

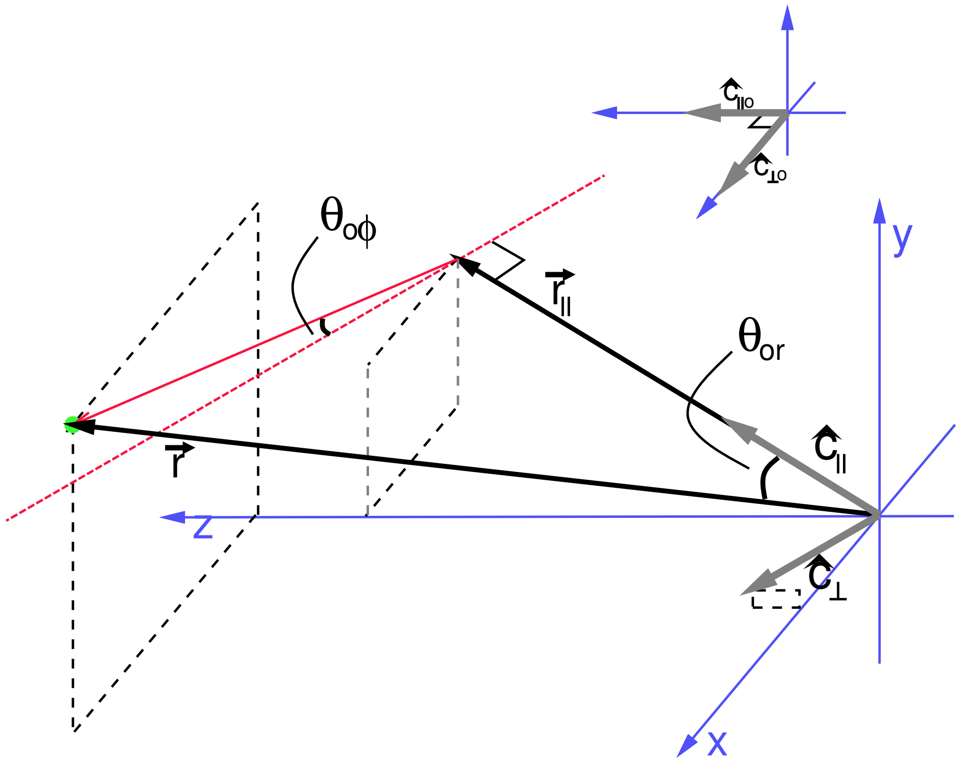


Figure 3-5: Simplified geometry of the head and camera. $\vec{r}$ is the object position; $\hat{c}_{||}$ is the axial vector of the camera; $\hat{c}_{\perp}$ determines the horizontal plane of the camera.

A ray of light from position $\vec{r}$ enters the center of the camera optics with angles $(\theta_{or}, \theta_{o\phi})$. It exits at $(\theta_{ir}, \theta_{i\phi}) = (m\theta_{or}, \theta_{o\phi})$ — that is, $\theta_{o\phi}$ is unchanged; θ_{or} is mulitplied by some constant dependent on the optic configuration. The resulting pixel (x, y) illuminated by this ray is $(d \sin\theta_{ir} \cos\theta_{i\phi}, d \sin\theta_{ir} \sin\theta_{i\phi})$, where d is another constant of the optics, related to the focal length.

From all of this, we can try tosolve for (x, y) in terms of $(\theta_p, \theta_t, \vec{r})$. An explicit solution is not very pretty, and fortunately we don't really want an explicit solution. Since the motion detection algorithm (Section 3.1.1) looks at motions of small patches of visual field for small head displacements, we can locally linearize the solution by finding the Jacobian $\mathbf{J}$ of (x, y) over (θ_p, θ_t), such that

$$\begin{pmatrix} \delta x \\ \delta y \end{pmatrix} = \mathbf{J} \begin{pmatrix} \delta\theta_p \\ \delta\theta_t \end{pmatrix}$$

where $\mathbf{J}$ is a function of $(\theta_p, \theta_t, x, y)$.

The general solution for $\mathbf{J}$ is unforgiving, too — to find it requires finding $\hat{r}$ in terms of $(\theta_p, \theta_t, x, y)$, the inverse of the original optical projection. So, we will just let our network learn it. However, it is relatively painless to solve $\mathbf{J}$ for the special case where (x, y) is fixed at $(0, 0)$, that is, for a point centered in the camera's field of view. Working through yields

$$\mathbf{J}_{(0,0)} = \begin{pmatrix} -md & md\sin\theta_p\cos\theta_p \\ 0 & -md\cos^2\theta_p \end{pmatrix}.$$

This will be used in Chapter 4 to double check what the network has learned.

Parallax Error

The above calculations were made assuming that the center of the camera optics coincides with the center of rotation. In reality, it isn't, and this introduces a parallax error, or depth dependence. Let the camera be offset from the pivot by some vector $\vec{p} = \mathbf{R}\vec{p}_o$ (see Figure 3-6), and let $\vec{r}_o$ be the position of the object point relative to the pivot. Now, $\vec{r}$ is still the position relative to the camera, so the expressions for $\cos\theta_{or}$ and $\cos\theta_{o\phi}$ remain the same. However, $\vec{r}$ varies with head orientation: $\vec{r} = \vec{r}_o - \vec{p}$. This introduces an error term roughly proportional to $\frac{p_o}{r_o}$:

$$\begin{aligned} \hat{r} = \frac{\vec{r}}{r} = \frac{\vec{r}_o - \vec{p}}{\|\vec{r}_o - \vec{p}\|} &\approx \frac{\vec{r}_o - \vec{p}}{r_o} \\ &\approx \hat{r}_o - \frac{p_o}{r_o}\hat{p}. \end{aligned}$$

The exact contribution is also affected by the orientation of $\hat{p}_o$; if $\hat{p}_o$ is collinear with the pan axis, then $\hat{p}_o \cdot \hat{c}_{||0} = \hat{p}_o \cdot \hat{c}_{\perp 0} = 0$, and the offset will not cause any problems. Otherwise, there will be an error which is inversely proportional to depth. Luckily, on Cog most of the learning is done off of a background scene consisting of the far walls of a large room, where $\frac{p_o}{r_o}$ is on the order of $\frac{1}{25}$.

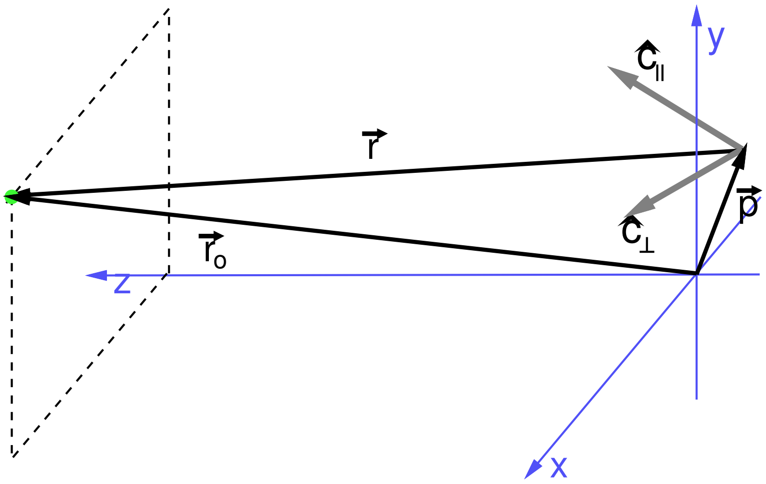


Figure 3-6: Modification of the head and camera geometry when the camera is offset from the pivot point of the neck by $\vec{p}$.

3.2.2 Network Architecture

The learning problem comes down to learning the function $(\theta_p, \theta_t, x, y, \delta\theta_p, \delta\theta_t) \mapsto (\delta x, \delta y)$, which tells us how far a point in the visual scene moves when the head moves. Any way the problem is expressed, it has eight degrees of freedom — the solution set is an eight dimensional manifold. We could try to learn the explicit functional solution of $(\delta x, \delta y)$, which will be a map $\mathcal{R}^6 \rightarrow \mathcal{R}^2$. Or, we could use the aforementioned linearized solution and learn the Jacobian $\mathbf{J}$, which gives us a map $\mathcal{R}^4 \rightarrow \mathcal{R}^4$.

I chose to use the latter approach. The volume of the input space, which our learning system has to model, is exponential in its dimension. Although we need to learn twice as many outputs for the Jacobian over the explicit solution, the input dimensionality is reduced by two. For a given complexity of learning technique (i.e. number of nodes in a network), this increases the density with which the modeller can cover its input space, which should mean a better model. Also, for purely aesthetic reasons, the latter approach is closer to the heart of the problem. We are really interested in the effect of position differentials, not absolute positions, so the differential approach is more natural.

$\mathbf{J}$ could be learned via a number of different techniques, ranging from learning unknown parameters of the exact functional form, to maintaining a table of examples to interpolate from; Atkeson (1989) gives a review of the possibilities. I chose a function approximation scheme consisting of a hybrid neural network of linear experts, similar to a network used by Stokbro et al. (1990). This network is a type of *mixture of experts model*, composed of a set of output nodes $\{u_{pqi}\}$ and a set of gating nodes $\{g_i\}$. It generates an interpolated, piecewise linear approximation of $\mathbf{J}$. The network architecture is illustrated in Figure 3-7. In this diagram, and for convenience in the rest of this discussion, $(x_1, x_2) = (x, y)$ and $(z_1, z_2) = (\theta_p, \theta_t)$.

Each output node u_{pqi} is a linear function of $\vec{z}$ and $\vec{x}$, corresponding to an element $\mathbf{J}_{pq}$ of the matrix $\mathbf{J}$:

$$u_{pqi} = \sum_j a_{pqij} x_j + \sum_j b_{pqij} z_j + c_{pqi} \tag{3.1}$$

Each gating node g_i — one for every set of output nodes — is a radial gaussian unit characterized by a mean $\vec{\mu}_i$. The receptive field of a node g_i determines a region of expertise for its associated output nodes u_{pqi}. That is, for some input $\{\vec{x}, \vec{z}\}$, the weight given to the linear approximation u_{pqi} in the final output is:

$$h_i = \frac{g_i}{\sum_j g_j}, \quad g_j = e^{-(\{\vec{x},\vec{z}\} - \vec{\mu}_j)^2} \tag{3.2}$$

The final output of the network is the weighted sum of all the linear approximations contained therein:

$$\mathbf{J}_{pq} = \sum_i h_i u_{pqi} \tag{3.3}$$

Each output node u_{pqi} is thus an expert for inputs in a neighborhood about the mean $\vec{\mu}_i$; the output of the network is the sum of the experts' opinions, weighted by their expertise.

This network is trained in two phases. First, competitive learning — an unsupervised learning technique — is used on the gating network. A new input $\{\vec{x}, \vec{z}\}$ is presented to the gating nodes and the "fittest" node — the one with maximal h_i — is jostled closer to the input point via:

$$\Delta\vec{\mu}_i = \rho(\{\vec{x}, \vec{z}\} - \vec{\mu}_i), \tag{3.4}$$

where ρ is an appropriately chosen (*i.e.* small) learning rate constant. The overall effect of this jostling is to distribute the gating nodes' receptive fields over the input space. Although the input vectors are represented as members of some real space $\mathcal{R}^n$ (in this case, $\mathcal{R}^4$), they may lie in a lesser dimensional manifold or be restricted to some bounded subset. With competitive learning, the gating nodes find this subspace, concentrating the efforts of the linear expert nodes where they are actually needed.

In the second phase of training, the output nodes $\{u_{pqi}\}$ are trained using gradient descent with a least-mean-squares (LMS) learning rule, a standard supervised learning technique (Hertz, Krogh & Palmer 1991). As mentioned earlier, the error signal is simply $\Delta\vec{x}_c$, the compensated output from a motion detector, since the system is being trained to report zero motion. After propagating $\Delta\vec{x}_c$ back through a sum-of-square-error cost function, weight updates take the form:

$$\begin{aligned} \Delta a_{pqij} &= \eta h_i(\delta z_q)(\Delta x_{cp})x_j && (3.5) \\ \Delta b_{pqij} &= \eta h_i(\delta z_q)(\Delta x_{cp})z_j && (3.6) \\ \Delta c_{pqi} &= \eta h_i(\delta z_q)(\Delta x_{cp}) && (3.7) \end{aligned}$$

where η is another learning rate constant.

Both learning phases are applied to the network with every new data point, collected in real-time from the motion detectors. Since each datum is a fresh, new piece of information from the world-at-large, there is no danger of overtraining the network, as there might be if the network were being trained from some fixed sample data set.

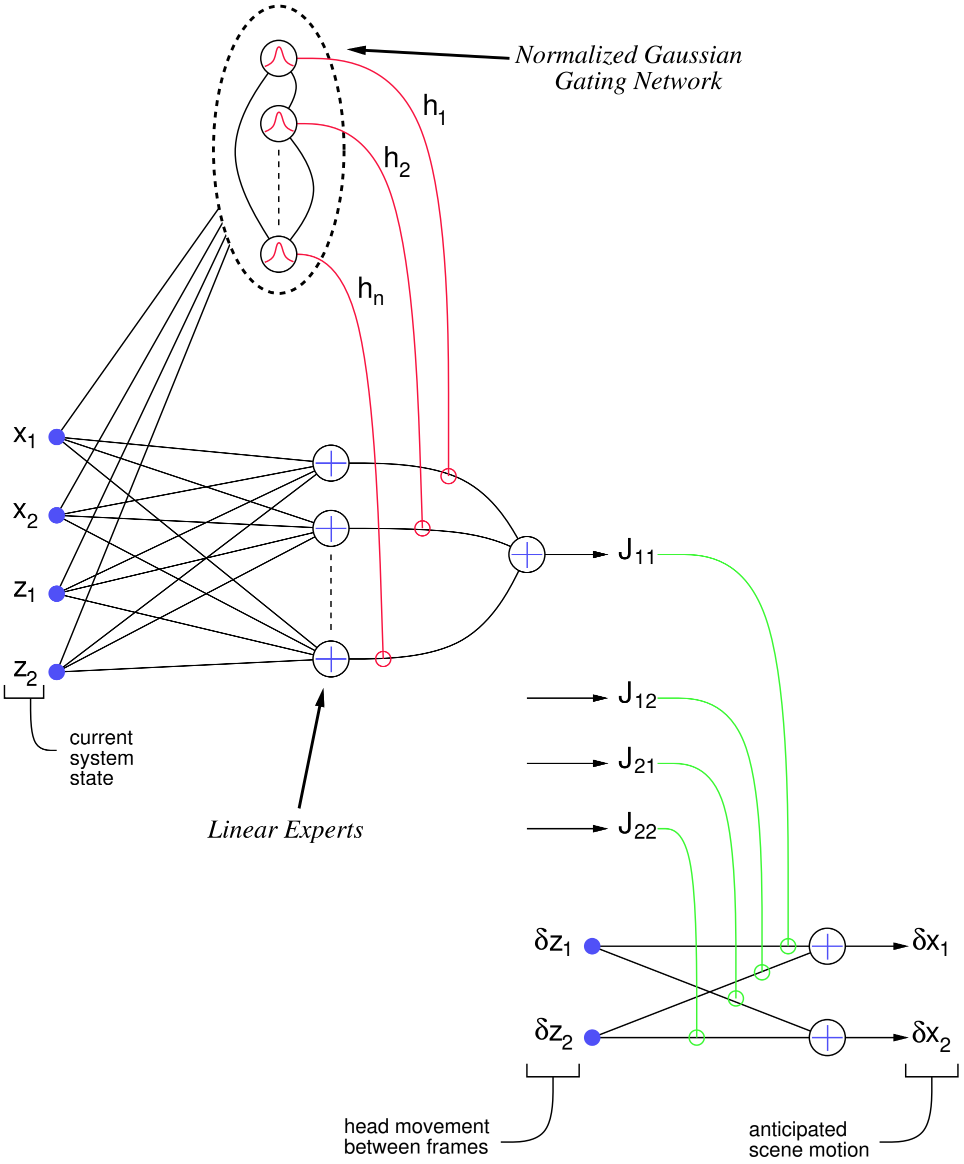


Figure 3-7: A schematic diagram of the network used to approximate the Jacobian **J** and generate a correction vector for the motion detectors. Note that $\vec{x} = (x, y)$ and $\vec{z} = (\theta_p, \theta_t)$.

Chapter 4

Performance

The algorithms described in the previous chapter have been implemented on Cog, and they actually work, more or less. Several experimental runs were made to test the performance of the system. All of these runs were with a network of 25 experts. The network code uses `L`'s software floating point, which is slow. Twenty-five experts translates into 600 parameters to be adjusted at every time step, which translates into a single detector sytem throughput of 3 motion vector updates per second.

In the experimental runs, I was interested in three aspects of the system performance:

- *RMS motion vector output vs. time.* Recall that we assume that the world is mostly still (especially at 4am) and that the motion vector output is used as the training signal for the network. The average motion vector output is thus a measure of the network error over time.

- *Direct network output.* The network is supposed to be learning the Jacobian $\mathbf{J}$. It is enlightening to compare the function it actually learns to the special-case analytical solution from Section 3.2.1.

- *Positions of the linear experts vs. time.* The competitive learning phase forces the experts to move around in the input space. Hopefully they behave in a rational way.

The system performance is determined by three sets of parameters:

- *the learning rates ρ and η.* The learning rates determine how much a network weight is changed at each learning step, affecting the speed and stability of the system.

- *the number of linear experts.* The number of experts determines how finely the input space is partitioned by the network and the level of detail in the resulting model. The trade-off is between accuracy and speed.

- *the dimension of the input space.* The input space is basically four dimensional $(\theta_p, \theta_t, x, y)$, but for the sake of comparison some runs were made with the motion detector fixed at center screen, constraining the input space to two dimensions.

4.1 Overall Error

The primary performance measure for this system is the network training error. Figure 4-1 gives graphs of the root-mean-square (RMS) error for four different values of the expert learning rate η. Each data point in these graphs corresponds to the value of the x or y component of the corrected motion vector, averaged over 1000 updates. In each run, Cog's head moved around its entire range of motion, and the target patch of the motion detector moved to a different location in the video frame every two seconds. The compensation network had to model the entire four dimensional input space, and the error is averaged over that entire space.

In each case the error decayed to a non-zero base level. I fit the data to the function

$$\epsilon = \epsilon_o + ke^{-\frac{t}{\tau}}$$

to find the baseline error ϵ_o and the time constant τ for the convergence of the network. Figures 4-2(a) and 4-2(b) summarize these results.

The time constants τ varied roughly inversely with η, as expected. The y-error always converged faster than the x-error. This might be because values in the y components of the network have a larger effect than in the x components, due to the scaling and normalization of θ_p and θ_t. Those components of $\mathbf{J}$ did not have to grow as large, so they converged faster.

The baseline error ϵ_o converged to roughly the same amount for all values of η. I had expected ϵ_o to be larger for larger η, because of enhanced noise susceptibility in the network. It is not, and this leads me to suspect another, larger source of noise in the system which limits how much the network can learn. This noise amounts to only 1 bit, however, so it could just be a limitation of the 3-bit motion detector.

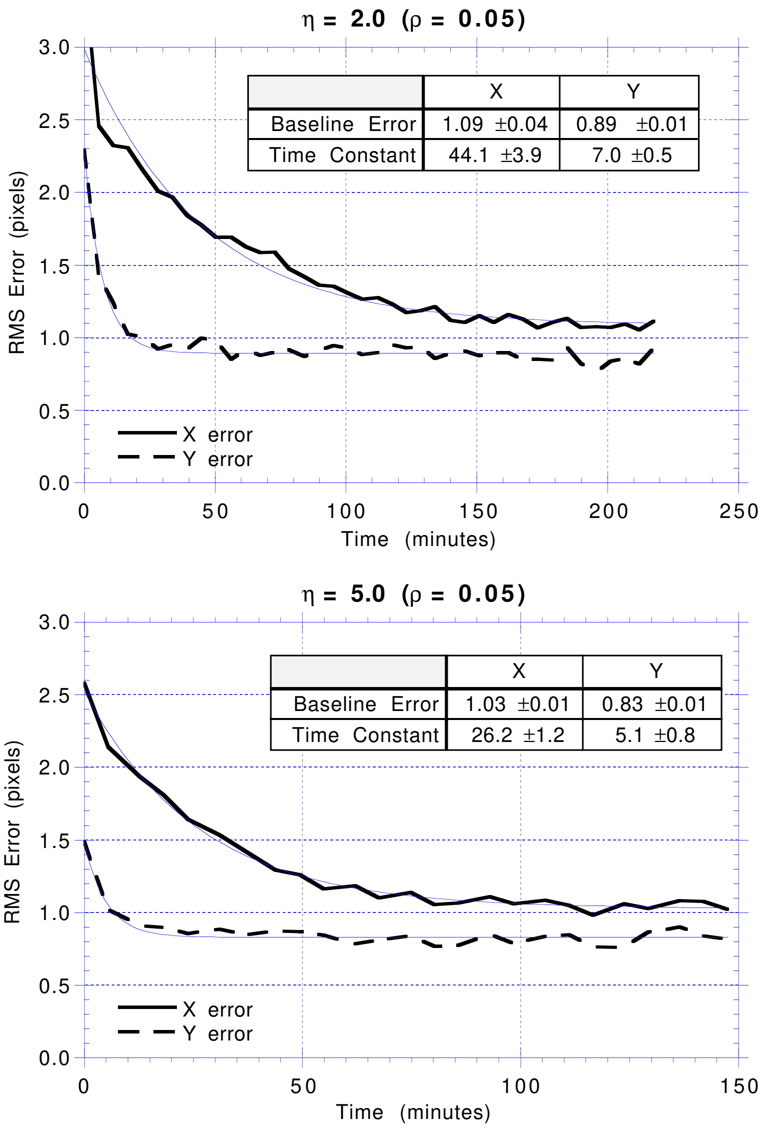
For another comparison, I repeated the $\eta = 5.0$ run with the motion detector target fixed at center screen. This constrained all the input data to lie on a single plane in $(\theta_p, \theta_t, x, y)$-space, and the twenty-five experts confined themselves to that plane instead of exploring the entire space. Surprisingly, the network performance was not much improved, if improved at all. Figure 4-3 shows the error results from this run, and Figure 4-4 compares the curve fits of this run with original full 4-dimensional run.

I also tried an additional run with $\eta = 500$, but the results were erratic and the network never converged.

4.2 Network Output

The network does appear to converge, but what does it converge to? Figure 4-5 shows a plot of the analytical solution of $\mathbf{J}$ vs. $(\theta_p, \theta_t, x, y)$ for (x, y) fixed at $(0, 0)$. This is not the exact solution for Cog, but an illustration of the basic functional form, taking into account the scaling of θ_p and θ_t. Figure 4-6 shows the actual network output (after training with $\eta = 5.0$ and $\rho = 0.05$) for the same case. They don't look very much alike.

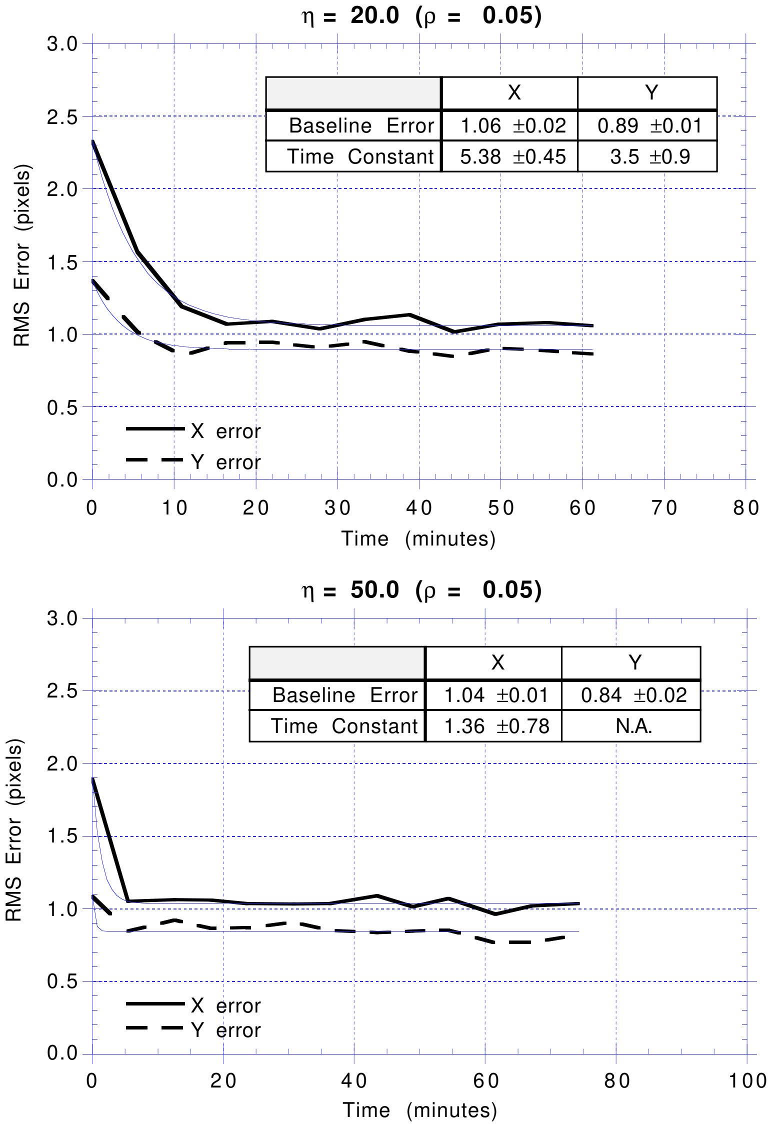
This strange outcome is most probably a result of the parallax error discussed at



	X	Y
Baseline Error	1.09 ±0.04	0.89 ±0.01
Time Constant	44.1 ±3.9	7.0 ±0.5

	X	Y
Baseline Error	1.03 ±0.01	0.83 ±0.01
Time Constant	26.2 ±1.2	5.1 ±0.8

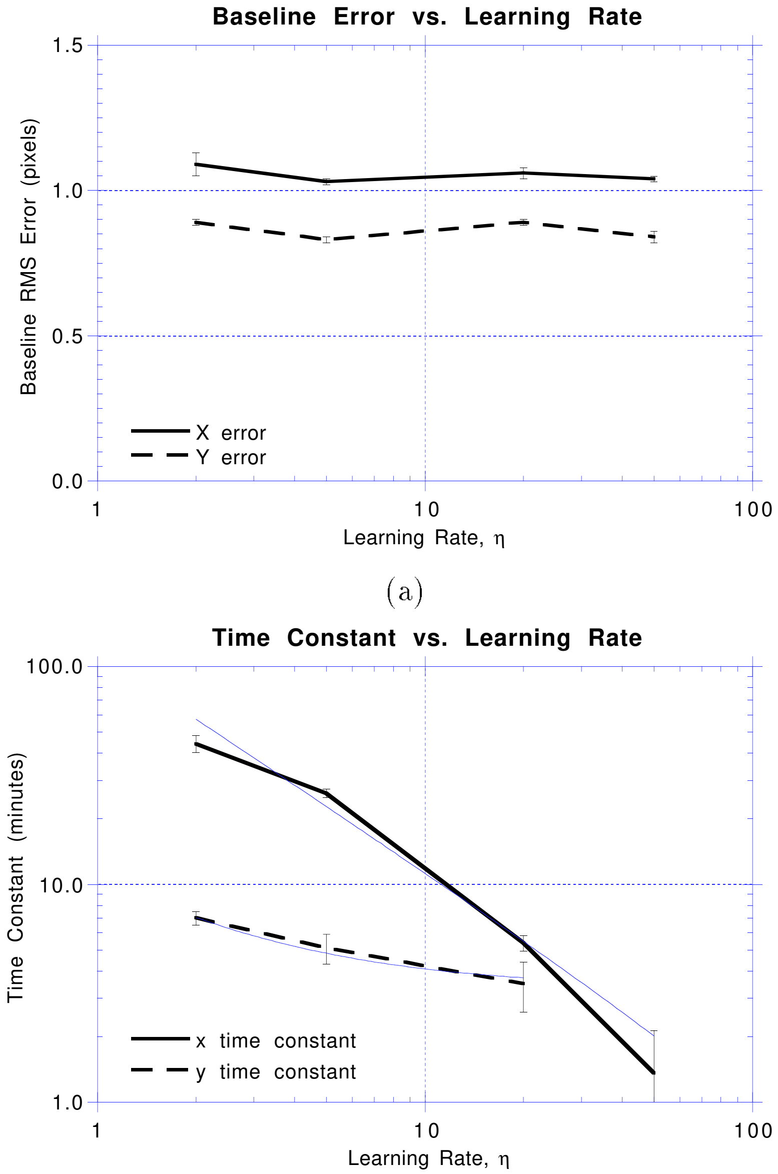
Figure 4-1: Motion detector error, which is also the corrected motion detector output, for $\eta = 2.0$ and $\eta = 5.0$. Data has been averaged over 1000 updates. The gray line is a curve fit of $\epsilon = \epsilon_o + k \exp(-t/\tau)$, which yields the baseline error ϵ_o and the time constant τ.



	X	Y
Baseline Error	1.06 ±0.02	0.89 ±0.01
Time Constant	5.38 ±0.45	3.5 ±0.9

	X	Y
Baseline Error	1.04 ±0.01	0.84 ±0.02
Time Constant	1.36 ±0.78	N.A.

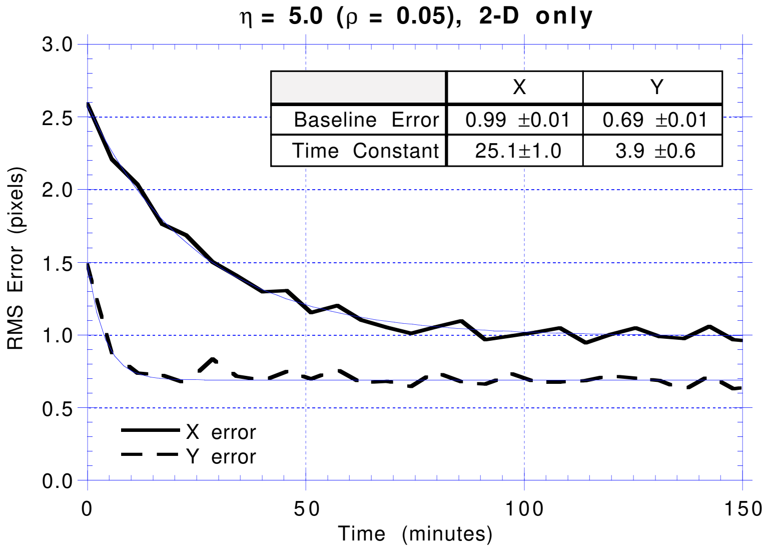
Figure 4-1: (cont.) Motion detector error for $\eta = 20$ and $\eta = 50$.



(a)

(b)

Figure 4-2: Graphs of baseline error ϵ_o (a) and time constant τ (b) as a function of learning rate η.



	X	Y
Baseline Error	0.99 ±0.01	0.69 ±0.01
Time Constant	25.1±1.0	3.9 ±0.6

Figure 4-3: Motion detector error for $\eta = 5.0$, with the input space constrained to $(x, y) = (0, 0)$.

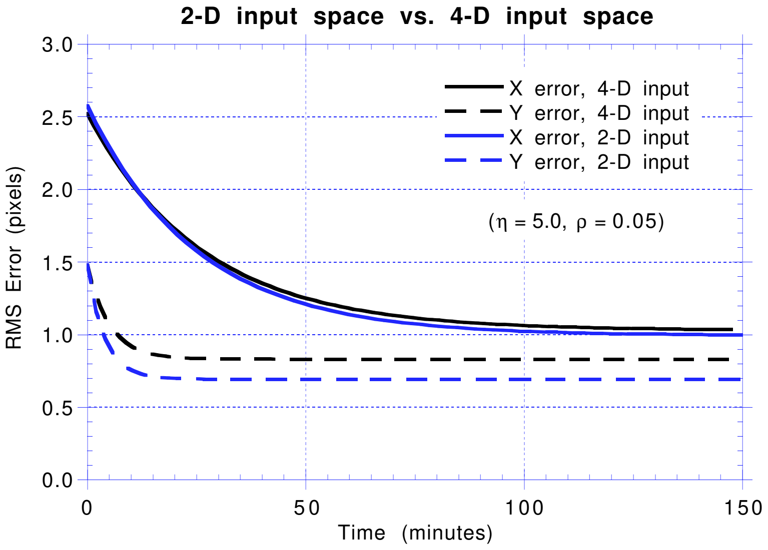


Figure 4-4: Comparison of fitted curves of motion detector error for learning over a 2-dimensional input space and a 4-dimensional input space.

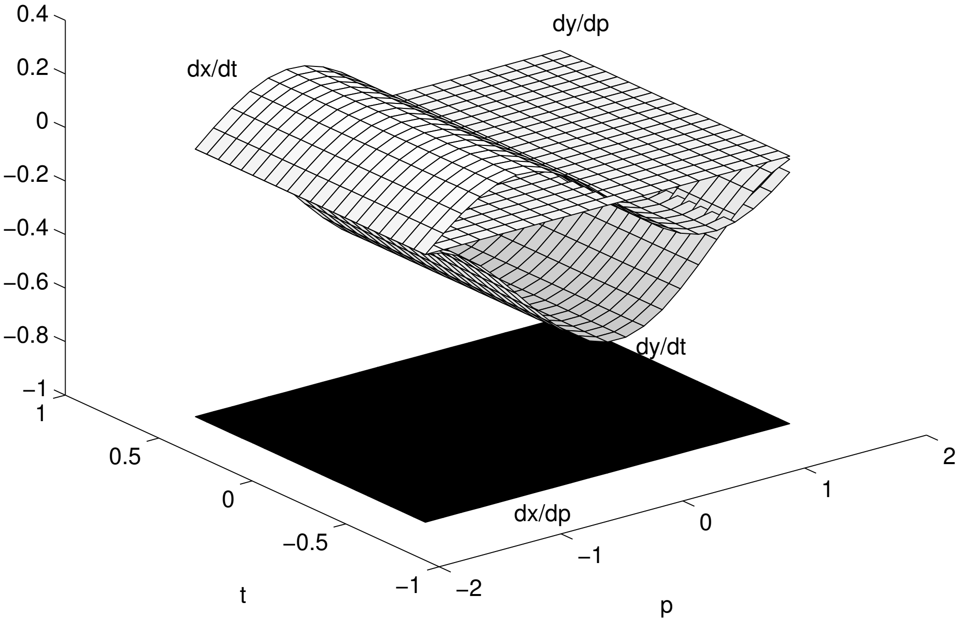


Figure 4-5: Illustrative plot of the analytical solution of $\mathbf{J}$, for $(x, y) = (0, 0)$. p and t correspond to θ_p and θ_t.

the end of Section 3.2.1. The most suspicious clue is the θ_t dependence of $\frac{\partial x}{\partial \theta_p}$, which should be θ_t independent. As Cog looks up or down, it sees the ceiling and floor, which are much closer than the far wall seen at center position. The closer an object is, the greater effect a change in θ_p will have, so the magnitude of the learned $\frac{\partial x}{\partial \theta_p}$ will be larger.

Another possibility is that the twenty-five experts are too sparsely distributed over the input space to accurately model $\mathbf{J}$. To test this hypothesis, I looked at the output from a network trained over two dimensions only. In such a case, the experts are distributed over the one plane of interest, with a sufficient density to capture all of the turning points in the analytical solution. The result is plotted in Figure 4-7. This is qualitatively the same as the output from the sparse 4-dimensional case.

Unfortunately, it seems that Cog is learning some twisted depth map of the room. The depth effects, combined with any other systematic effects which contribute to the base error, prevent the network from learning the expected clean-cut analytical solution.

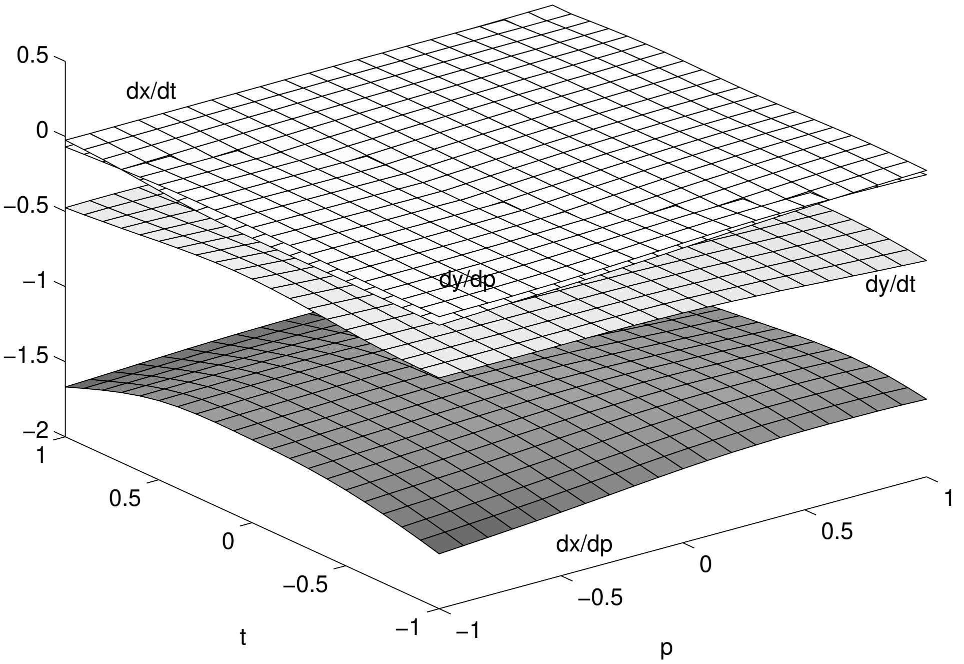


Figure 4-6: Plot of **J** as learned by the network, shown for $(x, y) = (0, 0)$. p and t correspond to θ_p and θ_t.

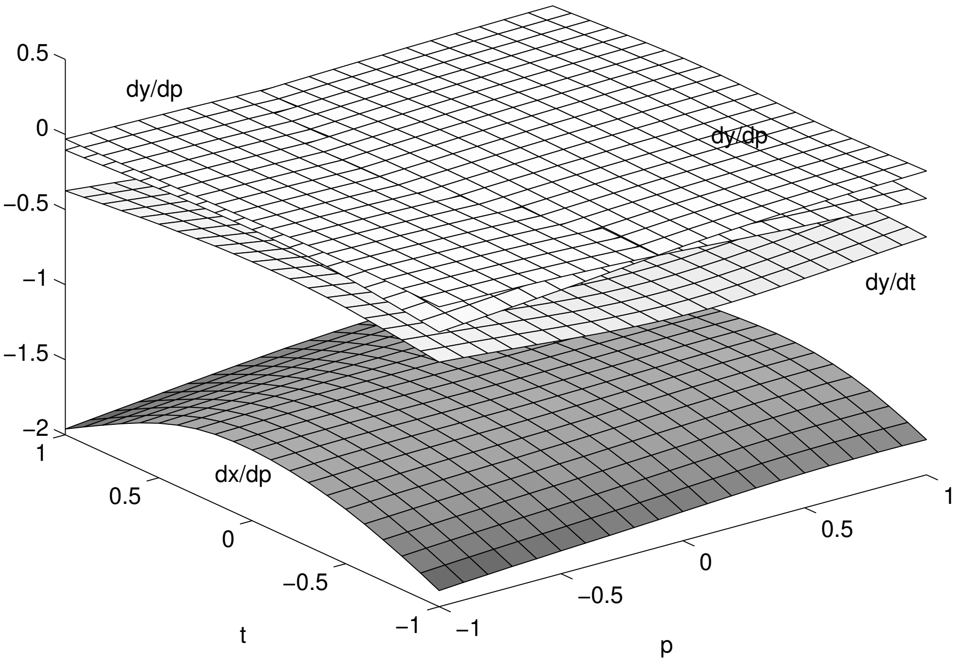


Figure 4-7: Plot of **J** as learned by the network, with learning restricted to $(x, y) = (0, 0)$. p and t correspond to θ_p and θ_t.

4.3 Expert Positions

The positions of the linear experts in the compensation network are not fixed; competitive learning causes them to move around and distribute themselves about the input space. As long as the learning rate ρ is left non-zero, this is a continuous process. The advantage of this is that the distribution of experts will adapt to track the distribution of input data. The disadvantage is that the structure of the network is always in flux, although this is not a problem if the experts learn fast enough to keep up with the motion.

Figure 4-8 shows the positions of linear experts at five timesteps in a network trained over two dimensions only. All the experts were contained in the plane of the page. The density is large enough that once the experts have dispersed, they tend to stay in about the same place. In this graph, each position is separated by 5000 network updates; since one expert is moved at each update, this amounts to about 200 updates per expert between data points.

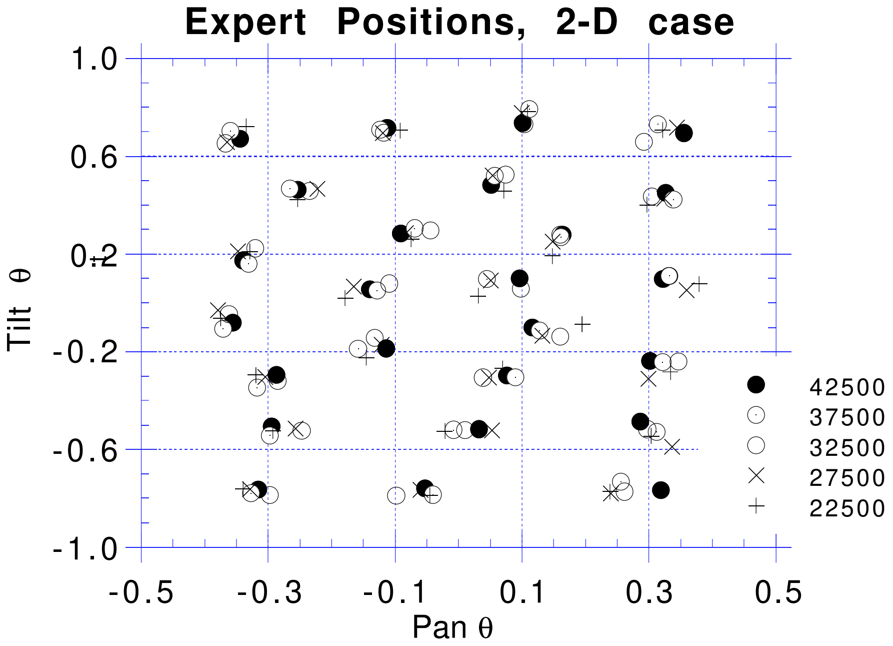


Figure 4-8: Positions of linear experts every 5000 updates of the network. Learning was constrained to the plane $(x, y) = (0, 0)$. $(\eta = 5.0, \rho = 0.05)$

Figure 4-9 shows the expert positions for a network trained over all four dimensions. The positions vary in x and y, but only the projection onto the θ_p-θ_t plane is shown. The density of experts in this case is much lower, and the experts seem to swim around without finding a stable configuration. Note that in this graph has only 100 updates per expert between data points. Despite this constant movement, the 4-D network did not perform any worse than the 2-D network.

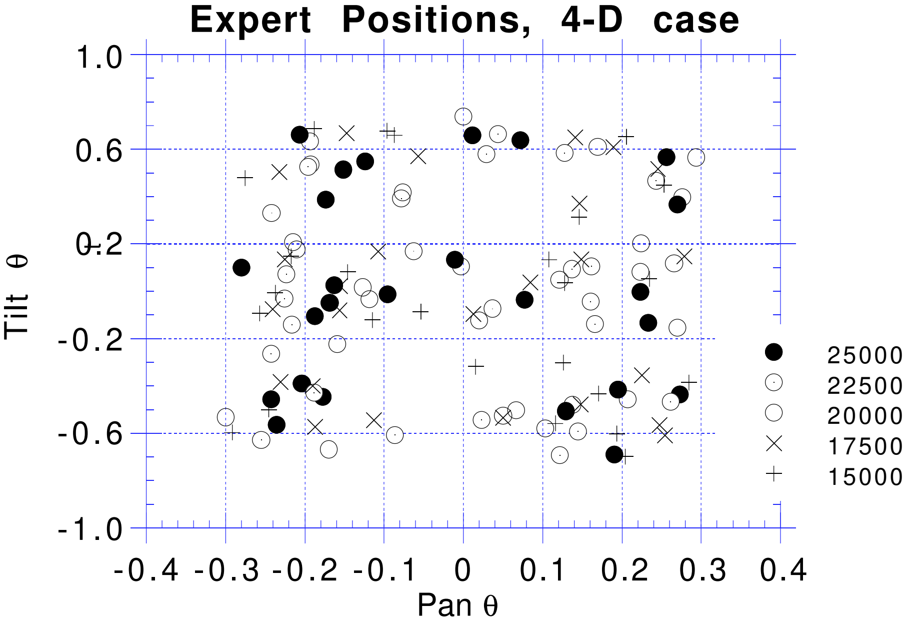


Figure 4-9: Positions of linear experts every 2500 updates of the network, projected onto a (θ_p, θ_t)-plane. The experts are actually distributed through a four dimensional space. ($\eta = 5.0, \rho = 0.05$)

Chapter 5

Future

To quote Karen Carpenter, "We've only just begun to live." The work presented here is the precursor to better solutions for harder problems. Future work will proceed on three levels:

- refinements to the current system
- extensions of the current system
- generalization to other systems

Much of the future work will require faster processing and distributed processing. The work will progress hand-in-hand with the development of other systems on Cog.

Refinements

Refinements to the current system include all the changes which improve function without really changing functionality. The causes of the baseline error in motion detection need to be pinned down. If this error is just one-bit noise in a three bit system, it could be reduced by improving the dynamic range, say by searching over 17 possible horizontal and vertical correlations to achieve a 4-bit result. Another possibility is grabbing a sequence of video frames and producing some kind of multi-scaled output that measures the movement over different time delays, using longer delays to resolve slower movements.

Some other refinements will require faster processors. The network computation would benefit from a hardware floating point unit. Before that becomes available, it may be worthwhile to recode the network to use a fixnum-based fixed-point calculation. Motion detection might also be more accurate if images were not reduced to black-and-white. However, any correlation more sophisticated than the current one bit per pixel `XOR` cannot be accomplished in real-time on a 68332.

I would also like to see multiple motion detector processes running on multiple cameras, so that Cog actually computes a motion vector field over the scene.

Extensions

The first and most simple extension to the current system is to account for all three degrees of freedom in the neck: pan, tilt, and the yet unused roll. Roll is not necessary to explore a scene, and analysis of the system was easier with only two degrees of freedom instead of three, so I left out roll in these experiments. Adding roll just means another adding another input parameter to the network, and running one more set of motor handler processes to control the extra motor. This will slow the system down even more.

The system also needs to account for depth. Adding depth to the network is easy, "just one more input parameter" (although this would bring our total to six, which is a very large input space). The tricky part is getting the depth information. This requires vergence control of the two eyes, a whole visuomotor subsystem unto itself. Furthermore, depth information will most likely pertain to the center of attention of the foveal view, not to odd patches in the periphery. The neck-compensated motion detection may be fated to always being a depth-free approximation. I am not sure how Cog will be able to learn this if it is stationary in the room.

I do want to extend the network to incorporate dynamics. Right now, it only models head kinematics, giving motion corrections based on neck joint position. The system might yield better results if it could also process neck motor commands, and anticipate neck motion. The network also needs to learn about delays in position feedback from the motor control boards and delays in video output from the frame grabbers. Control theories borrowed from the Smith predictor would help here.

Generalization

Sometime soon, the eyes have to start moving, instead of being fixed in place by cable-ties. Once the eyes are darting around with saccades and VOR reflexes, everything described here will cease to function. With eye movement, there will be at least three interacting sensorimotor systems — neck control, eye control, and motion detection — with mutual interactions and three maps to describe those interactions. How to learn and compose those maps, even in the absence of saccades, vergence, and VOR, is a wide-open question.

Appendix A

Budget Framegrabber v1.4.3

Just in case someone feels like building one from scratch, here is the parts list for Budget Framegrabber v1.4.3. The schematic is on the next page.

Quantity	Description	Package	DigiKey Part #’s
11	0.1μF capacitor	SMD	DK/PCC104B
1	0.01μF capacitor	SMD	DK/PCC1038
4	75Ω, 1% resistor	SMD	DK/P75.0FBK-ND
1	50Ω, 1% resistor	SMD	DK/P49.9FBK-ND
1	15Ω, 1% resistor	SMD	DK/P15.0FBK-ND
1	100Ω, 1% resistor	SMD	DK/P100FBK-ND
1	200Ω, 1% resistor	SMD	DK/P200FBK-ND
1	10kΩ, 5% bussed res. network	SIP	DK/R5103
1	200Ω 4mm trimmer pot.	SMD	DK/P1D201
1	100Ω 4mm trimmer pot.	SMD	DK/P1D101
1	1MΩ 4mm trimmer pot.	SMD	DK/P1D106
1	dual 1N4148 diode (BAV99)	SMD	DK/BAV99PH
1	green LED (Pana. LN1351C)	SMD	DK/P501CT-ND
1	hex-coded switch	DIP	DK/SW215-ND
6	44-pin/2-row female header	PC	
1	3-pin rt.ang. .156” header	PC	DK/WM4701-ND
1	3-pin .156” housing	—	DK/WM2101-ND
3	.156” crimp terminals	—	DK/WM2300-ND
2	BNC jack, right-angle mount	PC	DK/ARF1065-ND
1	Bt218 video ADC	PLCC	
1	Bt121 video DAC	PLCC	
1	LM1881 sync separator	SMD	
1	19.6608 MHz clock oscillator	DIP	DK/CTX090
1	16v8 PAL	PLCC	
1	22v10 PAL	PLCC	
2	74HC4520 dual 4-bit counter	SMD	DK/CD74HCT4520E
1	74HC4024 7-bit ripple counter	SMD	

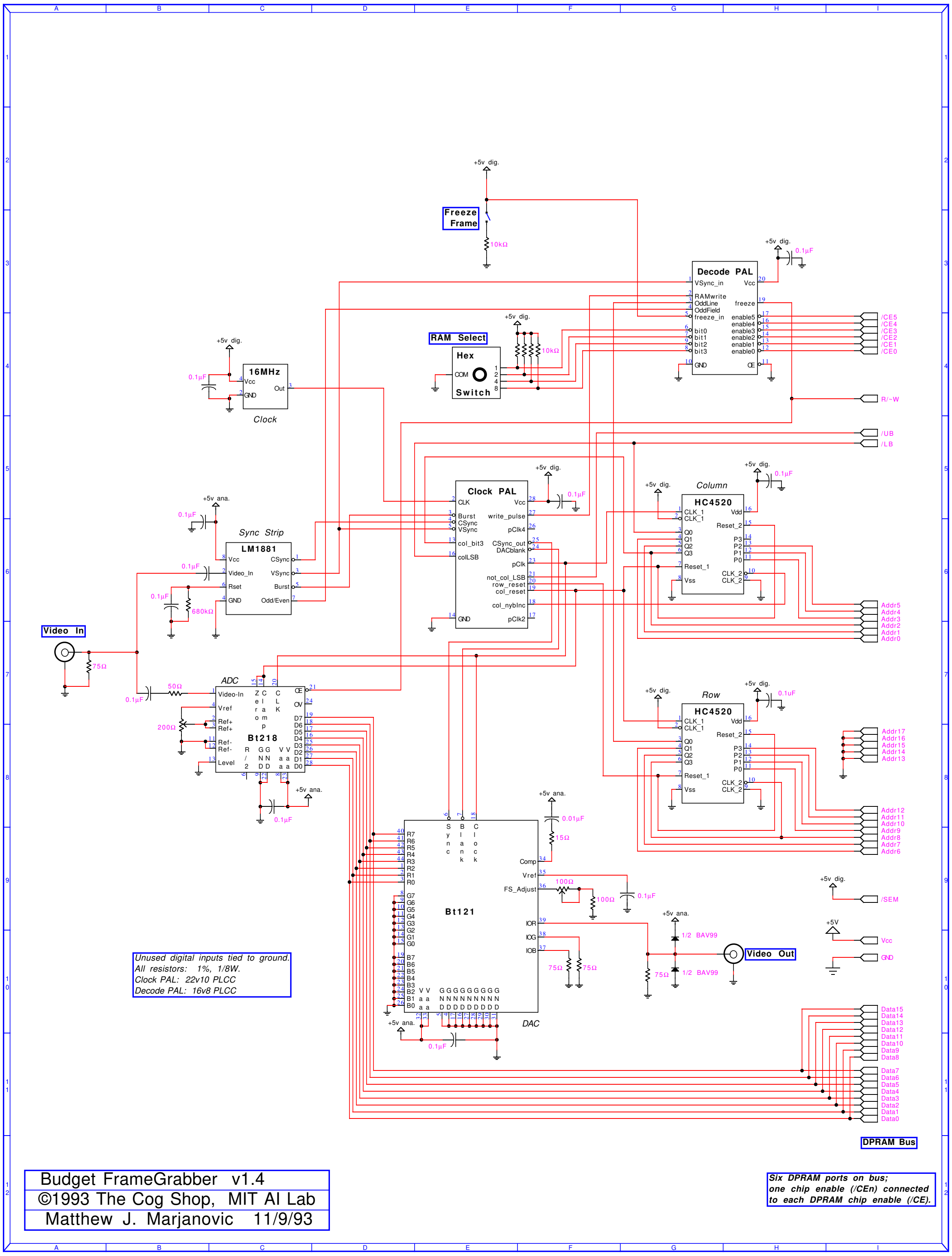


Figure A-1: Schematic diagram for the Budget Framegrabber v1.4.3.

Appendix B

InterProcess Socks v2.2

The following summary sheet lists the macros and functions defined by IPSv2 for manipulating svariables and sprocesses. This is an excerpt from the *Humanoid Software* reference manual (Brooks et al. 1994, Ch. 9).

IPSv2 Summary Sheet

Svariables

`def-svariable` *name* {*value-form*}	[*Macro*]
`make-svariable` *name* {*value*}	[*Function*]
`data` *variable-expression*	[*Macro*]
`flag` *variable-expression*	[*Macro*]
`variable` *variable-expression*	[*Macro*]
`notify` *variable-expression*	[*Macro*]

Sprocesses

`def-sprocess` *name* {*options*}* {*body*}	[*Macro*]
`make-sprocess` *name* {*options*}* {*body*}	[*Macro*]
`def-stemplate` *name* {*options*}* {*body*}	[*Macro*]
`make-stemplate` *name* {*options*}* {*body*}	[*Macro*]
`destroy-sprocess` *name*	[*Function*]
`spawn-sprocess` *name* {*arguments*}*	[*Function*]
`kill-sprocess` *name*	[*Function*]
`show-sprocess` *name*	[*Function*]

`spvref` *sproc name* [*Function*]

References port variable *name* in sprocess *sproc*. *name* should evaluate to a symbol (e.g. a quoted symbol).

Sprocess Options

- `:VARIABLES` signals a list of port variables. Each item in the rest of the list is a port variable descriptor, with these formats:
  - `name` — `name` will be, by default, a svariable with initial data `NIL`
  - `(name` {*value*}`)` — `name` will be a svariable with initial data *value*
  - `(name` *type-spec* {*type-arg*} {*value*}`)` — the port-variable `name` has a type specified by *type-spec* and *type-arg* (if required). Valid *type-spec*s are:
    - `:SHARED` — specifies a svariable, the default type. No *type-arg*.
    - `:S-ARRAY` — specifies an array of svariables. *type-arg* is a list of array dimensions suitable for passing to `make-array`.
    - `:GENERIC` — specifies that the port-variable is just a plain variable, as opposed to a svariable. No *type-arg*.

  In all of the above, the *value* form is optional and defaults to `NIL`. This form is evaluated only once, when the sprocess is created.

- `:ARGUMENTS` signals a list of passed-in arguments. In the sprocess body, these become variables whose value is initialized whenever the sprocess is spawned. Each argument specifier can take one of two forms:
  - `arg` — `arg` is a passed-in argument which defaults to `NIL`
  - `(arg` {*value*}`)` — `arg` is a passed-in argument which defaults to *value* (which is optional and defaults to `NIL`).

  Default values for passed-in arguments can be overridden in `spawn-sprocess` when the sprocess is spawned or respawned; if not overridden, a default value form is evaluated every time a sprocess is spawned.

- `:OPTIONS` signals a list of spawning options for the sprocess. This is a list of keywords and values, as you would find in an invocation of `spawn`. The following options are available:
  - `:NO-SPAWN` — specify `t` to *not* automatically spawn the sprocess after it is created
  - `:TICKS` — see `spawn` in the L manual
  - `:STACKSIZE` — see `spawn` in the L manual

  Any option available to `spawn` can be used here, except for `:PATTERN` and `:ARGLIST` which are used specially by IPS.

- `:TEMPLATE` — specify that the sprocess should be created from a pre-existing template instead of from scratch. `:TEMPLATE` should be followed by the name of[1] a stemplate. A new sprocess will be created with the procedure body, port variable, and passed-in argument structure specified by the template.

  If an `:OPTIONS` list is given, its spawn options will supersede those of the template. `:VARIABLES` and `:ARGUMENTS` lists can be used to specify new initialization forms, however they cannot specify new variables or arguments, nor can they specify different variable types.

  And, of course, you cannot specify a procedure body when using `:TEMPLATE`. If you did, you wouldn't need the template!

---

[1] or an expression that evaluates to...

Bibliography

Akshoomoff, N. A. & Courchesne, E. (1992), 'A new role for the cerebellum in cognitive operations', *Behavioral Neuroscience* **106**(5), 731–738.

Albus, J. S. (1971), 'A theory of cerebellar function', *Math. Biosci.* **10**, 25–61.

Atkeson, C. G. (1989), 'Learning arm kinematics and dynamics', *Annual Review of Neuroscience* **12**, 157–183.

Bizzi, E., Hogan, N. et al. (1992), 'Does the nervous system use equilibrium-point control to guide single and multiple joint movements?', *Behavioral and Brain Sciences* **15**, 603–613.

Bro (1993), *Graphics and Imaging Products Databook*, Brooktree Corporation, 9950 Barnes Canyon Rd., San Diego, CA 92121-2790. (databook).

Brooks, R. A. (1994), *L*, IS Robotics. (reference manual).

Brooks, R. A., Marjanović, M., Wessler, M. et al. (1994), *Humanoid Software*, MIT Artificial Intelligence Laboratory. (reference manual).

Daum, I. et al. (1993), 'The cerebellum and cognitive functions in humans', *Behavioral Neuroscience* **107**(3), 411–419.

Dichgans, J. & Diener, H. C. (1985), Clinical evidence for functional compartmentalization of the cerebellum, *in* J. R. Bloedel et al., eds, 'Cerebellar Function', Springer–Verlag, New York, pp. 127–147.

Eccles, J. C. et al. (1967), *The Cerebellum as a Neuronal Machine*, Springer–Verlag, Berlin.

Hertz, J., Krogh, A. & Palmer, R. G. (1991), *Introduction to the Theory of Neural Computation*, Addison–Wesley, Redwood City, CA.

Holmes, G. (1917), 'The symptoms of acute cerebellar injuries due to gunshot injuries', *Brain* **40**, 461–535.

Horn, B. K. P. (1986), *Robot Vision*, The MIT Press, Cambridge, Massachusetts.

Irie, R. (1995), Robust sound localization: an application of an auditory system for a humanoid robot, Master's thesis, Massachusetts Institute of Technology.

Ito, M. (1984), *The Cerebellum and Neural Control*, Raven Press, New York.

Kapogiannis, E. (1994), Design of a large scale MIMD computer, Master's thesis, Massachusetts Institute of Technology.

Kawato, M. et al. (1987), 'A hierarchical neural-network model for control and learning of voluntary movement', *Biological Cybernetics* **57**, 169–185.

Keeler, J. D. (1990), 'A dynamical system view of cerebellar function', *Physica D* **42**, 396–410.

Marr, D. (1969), 'A theory of cerebellar cortex', *J. Physiology* **202**, 437–470.

Matsuoka, Y. (1995), Embodiment and manipulation process for a humanoid hand, Master's thesis, Massachusetts Institute of Technology.

McIntyre, J. & Bizzi, E. (1993), 'Servo hypotheses for the biological control of movement', *Journal of Motor Behavior* **25**(3), 193–202.

Miall, R. C., Weir, D. J. et al. (1993), 'Is the cerebellum a Smith Predictor?', *Journal of Motor Behavior* **25**(3), 203–216.

Noback, C. R. (1981), *The Human Nervous System*, second edn, McGraw-Hill, New York.

Paulin, M. G. (1993*a*), 'A model of the role of the cerebellum in tracking and controlling movements', *Human Movement Science* **12**, 5–16.

Paulin, M. G. (1993*b*), 'The role of the cerebellum in motor control and perception', *Brain Behav Evol* **41**, 39–50.

Roberts, R. S. (1985), *Television engineering: broadcast, cable, and satellite*, Vol. 1, Pentech Press, London.

Stokbro, K. et al. (1990), 'Exploiting neurons with localized receptive fields to learn chaos', *Complex Systems* **4**, 603–622.

Williamson, M. (1995), Series elastic actuators, Master's thesis, Massachusetts Institute of Technology.