massachusetts institute of technology — artificial intelligence laboratory

Teaching an Old Robot New Tricks: Learning Novel Tasks via Interaction with People and Things

Matthew J. Marjanovic

AI Technical Report 2003-013 June 2003

© 2003 massachusetts institute of technology, cambridge, ma 02139 usa — www.ai.mit.edu

Teaching an Old Robot New Tricks: Learning Novel Tasks via Interaction with People and Things

by

Matthew J. Marjanović

S.B. Physics (1993),
S.B. Mathematics (1993),
S.M. Electrical Engineering and Computer Science (1995),
Massachusetts Institute of Technology

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2003

© Massachusetts Institute of Technology 2003.
All rights reserved.

Certified by: Rodney Brooks
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by: Arthur C. Smith
Chairman, Department Committee on Graduate Students

Teaching an Old Robot New Tricks: Learning Novel Tasks via Interaction with People and Things

by

Matthew J. Marjanović

Submitted to the Department of Electrical Engineering and Computer Science on May 27, 2003, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

As AI has begun to reach out beyond its symbolic, objectivist roots into the embodied, experientialist realm, many projects are exploring different aspects of creating machines which interact with and respond to the world as humans do. Techniques for visual processing, object recognition, emotional response, gesture production and recognition, etc., are necessary components of a complete humanoid robot. However, most projects invariably concentrate on developing a few of these individual components, neglecting the issue of how all of these pieces would eventually fit together.

The focus of the work in this dissertation is on creating a framework into which such specific competencies can be embedded, in a way that they can interact with each other and build layers of new functionality. To be of any practical value, such a framework must satisfy the real-world constraints of functioning in real-time with noisy sensors and actuators. The humanoid robot Cog provides an unapologetically adequate platform from which to take on such a challenge.

This work makes three contributions to embodied AI. First, it offers a general-purpose architecture for developing behavior-based systems distributed over networks of PC's. Second, it provides a motor-control system that simulates several biological features which impact the development of motor behavior. Third, it develops a framework for a

system which enables a robot to learn new behaviors via interacting with itself and the outside world. A few basic functional modules are built into this framework, enough to demonstrate the robot learning some very simple behaviors taught by a human trainer.

A primary motivation for this project is the notion that it is practically impossible to build an "intelligent" machine unless it is designed partly to build itself. This work is a proof-of-concept of such an approach to integrating multiple perceptual and motor systems into a complete learning agent.

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

Acknowledgments

"Cog, you silent smirking bastard. Damn you!"

Phew... it feels good to get that out of my system. I should have known I was in trouble way back in June of 1993, when Rodney Brooks both handed me a copy of Lakoff's *Women, Fire, and Dangerous Things* and suggested that I start designing framegrabbers for "the new robot". That was the beginning of ten years of keeping my eyes on the sky above while slogging around in the mud below.

The truth is, it's been *great*: I've been working on intriguing problems while surrounded by smart, fun people and provided with all the tools I could ask for. But after spending a third of my life in graduate school, and half of my life at MIT, it is time to wrap up this episode and try something new.

Many people have helped over the last weeks and months and years to get me and this document out the door. I could not have completed this research without them.

Despite a regular lament that I never listen to him, my advisor, Rodney Brooks, has consistently told me to be true to myself in my research. That is uncommon advice in this world, and I appreciate it. Looking back, I wish we'd had more opportunities to argue, especially in the last few years; I learned something new every time we butted heads. Maybe we'll find some time for friendly sparring in the future.

Thanks to Leslie Kaelbling and Bruce Blumberg for reading this dissertation and gently pushing me to get the last little bits out. They gave me great pointers and hints all along the way. I should have bugged them more often over the last couple of years.

Jerry Pratt gave me many kinematic insights which contributed to the work on muscle models. Likewise, I'd like to thank Hugh Herr for the inspiring discussions of muscle physiology and real-estate in northern California.

Paul Fitzpatrick and Giorgio Metta kept me company while working on Cog over the last couple of years. I regret that I was so focused on getting my own thesis finished that we didn't have a chance to collaborate on anything more exciting than keeping the robot bolted together.

I'm awed by the care and patience displayed by Aaron Edsinger and Jeff Weber during their run-ins with Cog, often at my request a day before a deadline. Those guys are two top-notch mechanics; I admire their work.

Brian "Scaz" Scassellati and Matt "Matt" Williamson were my com-

rades in the earlier years working on Cog. Many ideas in this dissertation were born out of discussions and musings with them. The times we worked together were high-points of the project; I have missed them since they graduated.

My latest officemates, Bryan "Beep" Adams and Jessica "Hojo" Howe, have been exceptionally patient with my creeping piles of papers and books, especially over the last semester. Jessica was a sweetheart for leaving soda, PB&J's, and party-mix on my desk to make sure I maintained some modicum of caloric (if not nutritional) intake. Bryan is a cynic with a top-secret heart of gold. Nothing I can write could do justice to the good times I've had working with him, and I mean that with both the utmost sincerity and sarcasm.

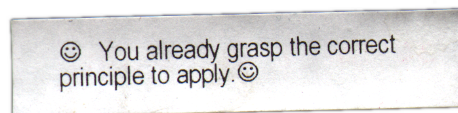
I've had some of the best late-night brainstorming rambles over the years with Charlie Kemp. Charlie also generously provided me with what turned out to be the last meal I ate before my thesis defense.

Tracy Hammond proofread this dissertation, completely voluntarily. That saved me from having to actually read this thing myself; I cannot thank her enough.

I would like to point out that the library copy of this dissertation is printed on a stock of archival bond paper given to me years ago by Maja Matarić. (Yes, Maja, I saved it and finally used it!) Maja was a great mentor to me during my early years at the lab. The best summer I ever had at MIT was the summer I worked for her as a UROP on the "Nerd Herd", back in 1992. That summer turned me into a roboticist. (And that summer would have never happened if Cynthia Breazeal hadn't hired me in the first place.)

Those of us on the 9th Floor who *build stuff* wouldn't be half as productive without the help of Ron Wiken. The man knows where to find the right tools and he knows how to use them — and, he is happy to teach you to do both. Jack Constanza, Leigh Heyman, and Mark Pearrow have earned my perpetual regard for keeping life running smoothly at the AI Lab, and for helping me with my goofy sanity-preserving projects.

A fortune cookie with this fortune



reassured me that I was on the right track with my research.

Carlin Vieri contributed to the completion of this work by pointing out, in 2001, that my MIT career had touched three decades. Carlin

made this same contribution numerous times over the next two years.

My grandmother, Karolina Hajek, my mother, Johanna, and my sister, Natasha, have been giving me encouragement and support for years, and years, and years. Three generations of strong women is not a pillar that can be toppled easily.

I have always thought that I wouldn't have gotten into MIT without the lessons I learned as a child from my grandfather, Josef Hajek. Those lessons were just as important for getting out of MIT, too. They are always important.

Finally, I could not have survived the ordeal of *finishing* without the unwavering love of Brooke Cowan. She has been patient and tough and caring, a steadfast companion in a little lifeboat which made slow progress on an ever-heaving sea. Brooke proofread this entire document from cover to cover, and she convinced me to wear pants to my defense. For months, she has made sure that I slept, that I smiled, and that I brushed my teeth. Most importantly, every day, Brooke gave me something to look forward to after this was all over.

And now it is! Thank you!

Contents

1 Looking Forward **19**
1.1 The Platform: Cog . . . 21
1.2 A Philosophy of Interaction . . . 23
1.3 Related Work . . . 27
1.3.1 Drescher's Schema Mechanism . . . 27
1.3.2 Billard's DRAMA . . . 30
1.3.3 Pierce's Map Learning . . . 31
1.3.4 Metta's Babybot . . . 33
1.3.5 Terence the Terrier . . . 34
1.3.6 Previous Work on Cog (and Cousins) . . . 35

2 sok **37**
2.1 Design Goals . . . 37
2.2 System Overview . . . 38
2.3 Life Cycle of a sok-process . . . 40
2.4 Anatomy of a sok-process . . . 41
2.5 Arbitrators and Inports . . . 43
2.6 Simple Type Compiler . . . 43
2.7 Dump and Restore . . . 45

3 meso **47**
3.1 Biomechanical Basis for Control . . . 47
3.2 Low-level Motor Control . . . 53
3.3 Skeletal Model . . . 55
3.3.1 Complex Coupling . . . 56
3.3.2 Simple Coupling . . . 59
3.4 Muscular Model . . . 61
3.5 Performance Feedback Mechanisms . . . 63
3.5.1 Joint Pain . . . 63
3.5.2 Muscle Fatigue . . . 63

4 Touch and Vision **69**
4.1 The Hand and Touch 69
4.2 The Head and Vision 72
4.2.1 Motor System 74
4.2.2 Image Processing 75
4.2.3 Vergence 76
4.2.4 Saliency and Attention 76
4.2.5 Tracking 79
4.2.6 Vision System as a Black Box 79

5 pamet **81**
5.1 A Toy Example: The Finger Robot 82
5.1.1 Learning to Move 82
5.1.2 Learning What to Do 84
5.1.3 Learning When to Do It 85
5.2 Names and Data Types 87
5.3 A Menagerie of Modules and Models 88
5.3.1 Movers 89
5.3.2 Controllers 90
5.3.3 Actors 91
5.3.4 Triggers 94
5.3.5 Transformers 96
5.4 Other Modules and Facilities 98
5.4.1 Age 98
5.4.2 Emotion 99

6 Models and Modellers **103**
6.1 Mover Models 103
6.2 Action Models 104
6.3 Trigger Models 109
6.4 Transform Models 118

7 Learning Simple Behaviors **123**
7.1 Moving via Movers 123
7.2 The Toy Finger Robot, Realized 129
7.3 “It’s a red ball! It’s a green tube!” 138
7.4 Reaching Out 147
7.5 The Final Picture 151

8 Looking Back, and Forward Again **153**
8.1 Creating Structure: What pamet Can and Cannot Do . 154
8.2 Unsatisfying Structure: Hacks 156
8.3 New Structure: Future Work 159
8.4 Unintended Structure 163
8.5 Round, Flat Structure: Flapjacks 163

A Details of the Complex Coupling Model **167**

B Two Transform Models **173**
B.1 Non-parametric: Memory-based Model 173
B.2 Semi-Parametric: Loose Coordinate Transform Model . 174

List of Figures

1.1 Grand overview of the components of this thesis work. . 20
1.2 Cog, a humanoid robot. 22
1.3 Two of Johnson's *image schemata*. 25
1.4 A schema, from Drescher's schema mechanism. 28

2.1 A network of *sok-processes*. 39
2.2 Typical life-cycle of a sok-process. 40
2.3 Typical code flow for a sok-process, using the sok C library. 42

3.1 Overview of meso. 48
3.2 Virtual muscles act like antagonistic pairs of real muscles. 49
3.3 Waste of energy due to "lengthening contractions" of single-joint muscles. 51
3.4 Torque feedback loop controlling the torso motors. . . . 54
3.5 Example of the "complex coupling" skeletal model. . . . 57
3.6 Description of a vector in two different linked coordinate frames. 58
3.7 Failure of the "complex coupling" skeletal model. 60
3.8 Example of the "simple coupling" skeletal model. 60
3.9 Classic Hill model of biological muscle tissue. 62
3.10 Joint pain response through the full range of a joint's motion. 64
3.11 Effects of virtual fatigue on a virtual muscle. 67

4.1 Cog's right hand, mounted on the arm. 70
4.2 The four primary gestures of Cog's hand. 71
4.3 Response curves of the tactile sensors. 72
4.4 Detail image of the FSR sensors installed on the hand. . 73
4.5 Outline of the vision system. 73
4.6 Cog's head, viewed from three angles. 74

4.7 Saliency and attention processing while looking at a walking person. 77
4.8 Saliency and attention processing while looking at the moving robot arm. 78

5.1 A one degree-of-freedom toy robot. 82
5.2 Two mover modules for the toy robot. 83
5.3 Acquiring an action model and actor for the toy robot. . . 85
5.4 Acquiring a trigger model and trigger for the toy robot. 86
5.5 General form of a *controller* module. 91
5.6 Two types of *actor* modules. 93
5.7 Two types of *trigger* modules. 95
5.8 General form of a *transformer* module. 97
5.9 The basic emotional system employed by pamet. 100

6.1 Two types of action modellers. 105
6.2 An example of position-constant-action data analysis. . 106
6.3 Comparison of CDF's to discover active axes. 107
6.4 General form of position-trigger modeller. 109
6.5 A sample of a position-trigger training data set. 111
6.6 Action and reward windows for the example position-trigger dataset. 113
6.7 Comparison of CDF's for the position-trigger example data. 114
6.8 Possible partitions of a 2-D parameter space by a Gaussian binary classifier. 117

7.1 Schematic of the initial state of the system. 124
7.2 Joint angles θ and mover activation A recorded by the elbow mover modeller. 126
7.3 Joint velocities $\dot{\theta}$ and mover activation A for the elbow mover modeller. 127
7.4 Linear fit of $\dot{\theta}$ versus A by the elbow mover modeller for elbow and thumb. 128
7.5 Recorded raw data while training robot to point its finger. 130
7.6 Comparison of rewarded/unrewarded CDF's while training robot to point. 131
7.7 Prototype positions of the pointing and grasping action models. 132
7.8 Raw data for training the robot to point its finger in response to touch. 133

7.9 Reward and action windows for the pointing-trigger training session. 134
7.10 CDF comparison for two components of the tactile sense vector. 135
7.11 The stimulus model learned for trigger the pointing action. 136
7.12 State of the system after learning to move and point. . . 137
7.13 Prototype postures of three position-constant actions for the arm. 139
7.14 Data acquired in training robot to reach outward in response to a red ball. 140
7.15 Reward, action windows for training robot to reach in response to a red ball. 141
7.16 Comparison of stimulus vs. background CDF's for red, green, and blue. 142
7.17 Stimulus decision boundary of the red-ball trigger. . . . 143
7.18 Stimulus decision boundary of the green-tube trigger. . 144
7.19 Data acquired by a trigger modeller while training a different action/trigger. 145
7.20 Learned arm postures (and any other actions) can be modified over time. 146
7.21 Training performance degradation of transform models, RMS-error. 149
7.22 Ability of transform models to lock on to signal in the face of noisy data. 150
7.23 Schematic of the final state of the system. 152

A.1 Description of a vector in two different linked coordinate frames. 168
A.2 General parameterization of linked coordinate frames. . 169

List of Tables

1.1 Johnson's twenty-seven "most important" image schemata. 26

2.1 Syntax of the sok type description language. 44

5.1 Classes of modules implemented in pamet. 89

7.1 The complete set of mover modules which connect pamet to meso. 125

Chapter 1

Looking Forward

This thesis work began with some grand goals in mind. I'm sure this is typical of the beginnings of many thesis projects, but it is all the more unremarkable considering that this work is part of the Cog Project. The original visions behind the Cog Project were to build a "robot baby", which could interact with people and objects, imitate the motions of its teachers, and even communicate with hand gestures and winks and nods. Cog was to slowly develop more and more advanced faculties over time, via both learning and the steady addition of more complex code and hardware.

My own pet goal was to end up with a robot which I could successfully teach to fry me a batch of pancakes. I was happy to settle for starting with a box of *Just Add Water!* mix and a rectangular electric griddle. (And a spatula bolted to the end of one of Cog's paddles.) There is no sarcasm intended here. This is a task which is quite easily performed by children. One could also quite easily design a machine specialized to perform precisely that task, using technology from even fifty years ago. However, to build a machine which *learns* to perform that task, using tools made for humans, is — still — no easy feat.

I would venture to say that none of the grandest goals of the Cog Project came to fruition, mine included. Cog is not yet able to learn to do a real, humanly-simple task. However, I was able to achieve some of my more humble and specific goals:

- Create a learning system in which actions and states are learned or learnable entities, not hard-coded primitives.
- Use a real robot, physically interacting with real people.
- Teach the robot to do *something*.

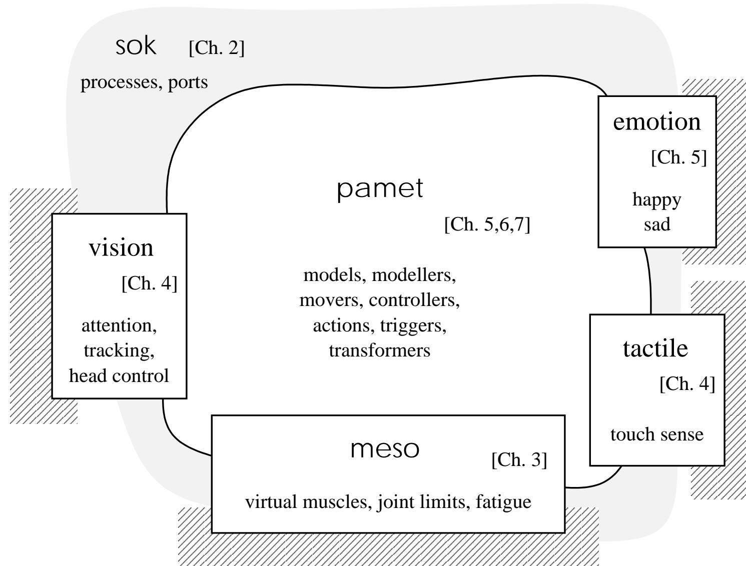


Figure 1.1: Grand overview of the components of this thesis work. sok (Chapter 2) is the process control and message-passing glue with which everything else is written. meso (Chapter 3) is a biologically-motivated motor control layer implementing virtual muscles. meso is grounded in the physical hardware, as are the vision system, tactile sense, and rudimentary emotional system (Chapters 4 & 5). pamet (Chapters 5, 6, & 7) is the "smarts" of the system, creating models of the interactions of the other subsystems in order to learn simple behaviors.

- Design a system which is capable of long-term, continuous operation, both tended and untended.

This dissertation describes these goals and constraints, and the resulting system implemented on Cog.

A grand overview of the system is given in Figure 1.1. The work comprises three significant components. The first is sok (Chapter 2), an interprocess communication (IPC) and process control architecture. sok is specifically adapted to the QNX real-time operating system, but could be ported to other POSIX-based OS's. It is a general-purpose tool useful for anyone building a behavior-based system distributed over a network of processors. The second piece is meso (Chapter 3), a biologically-inspired motor control system which incorporates the notion of "virtual muscles". Although tuned for Cog, it provides a quite

general system for controlling a robot composed of torque-controlled actuators. The third and final piece, built upon the first two, is pamet (Chapters 5–7). pamet is a framework for designing an intelligent robot which can learn by self-exploration and by interacting with human teachers. It is by no means a complete system; it is missing several important elements, but the structure necessary for adding those elements is in place. As it stands, it is capable of being taught a simple class of actions by a human teacher and learning to perform those actions in response to a simple class of stimuli. Learning systems always live at the mercy of the sensory and motor systems they are built upon, and the sensory systems used here are quite basic. Chapter 7 discusses what Cog and pamet can currently do and explores what elements they would require to do more.

1.1 The Platform: Cog

As alluded to already, the robot platform used in this project is Cog (Figure 1.2). This is an anthropomorphic robot, with a design which captures significant features of the human body from the waist up. Cog's hips are a large, two degree-of-freedom (dof) gimble joint; the torso has a third dof in the rotation of the shoulder yoke. Cog has two arms, each with six degrees of freedom — three in the shoulder, one in the elbow, and two in the wrist. (The human wrist has three.) The left arm ends in a simple paddle, but the right arm is outfitted with a 2-dof hand with tactile sensors. Atop the shoulder yoke is a 7-dof head, which includes two eyes consisting of two video cameras apiece. The actuators used in the torso and arms are described in Chapter 3, and the hand and head are described in greater detail in Chapter 4. At various times, microphones have been mounted around the head to provide auditory input, but none are used in this project. The entire robot is bolted to a sturdy steel base so that it stands at average human eye-level. Cog is not a particularly "mobile" robot; the base stays where it is.

All the processing for Cog is performed off-board, by racks of 28 x86 architecture processors running the QNX4 operating system. These *nodes* are networked together via 100baseT 100Mb/s ethernet. All nodes are connected to one another via a locally-switched backbone hub. Many nodes with large data throughput requirements (such as those handling video data) are also connected to each other directly via full-duplex point-to-point 100baseT connections. The processor speeds range from 200 to 800 MHz, and each node hosts 128 to 512

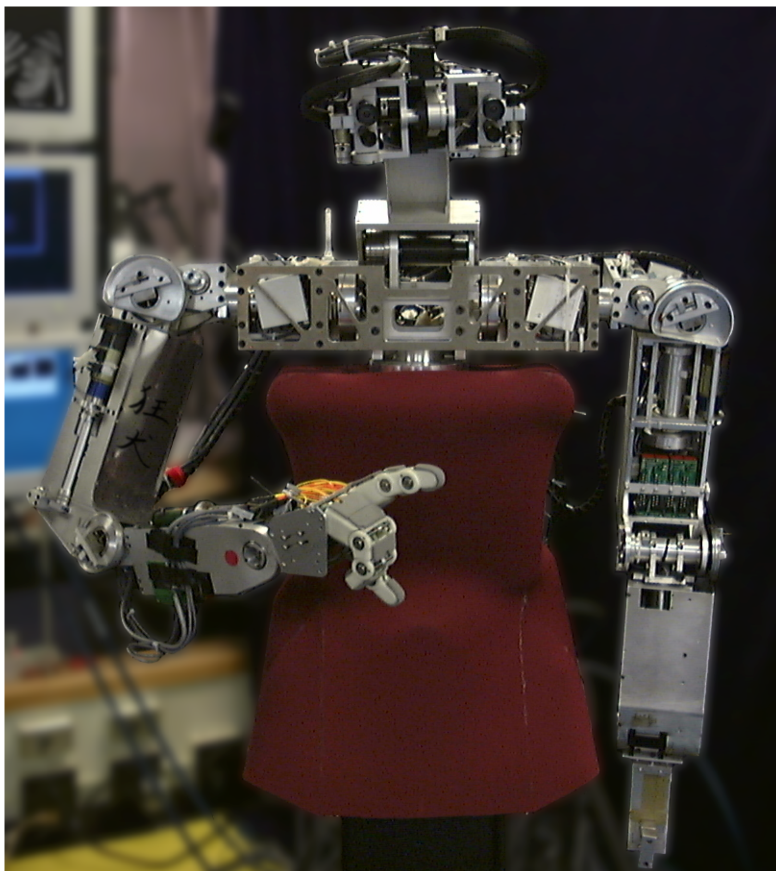


Figure 1.2: Cog is a humanoid robotics platform. It has a total of 24 degrees of freedom: 3 in the torso, 6 in each arm, 2 in the hand, and 7 in the head. The torso and arms feature torque-controlled actuators; the head/eyes are under conventional position control. Four cameras provide stereoscopic vision at two different resolutions and fields-of-view. The hand is equipped with tactile sensors. All processing and control is done off-board, on an expandable network of twenty-eight off-the-shelf x86 processors.

MB of RAM. About half of the nodes are dedicated to specific sensory or motor I/O tasks.

1.2 A Philosophy of Interaction

This project is motivated by the idea that *perception is meaningless without action.* The semantic content of a sensory experience is grounded in an organism's ability to affect its environment and in its need to decide what effect to produce. The meaning of what we see and hear and feel comes from what we can do about it.

A disembodied perceptual system cannot assign much intrinsic value to the information it processes. When a face detection algorithm draws bounding boxes around faces in a scene displayed on a computer screen, the *meaning* of those boxes typically arises from their observation by a human, to whom a "face" has meaning because it is attached to a large repertoire of social cues, interactions, desires, and memories.

The layers upon layers of interwoven concepts constituting intelligence are rooted in primitives that correspond to simple, direct interactions between motor systems and sensory systems, coupled either internally or through the environment. It is the interactions between these systems, and the patterns discovered among these interactions, which are the basis of thought.

Experience and Metaphor

In *Metaphors We Live By* [30], George Lakoff and Mark Johnson explore a philosophy of meaning and understanding which revolves around the pervasive use of metaphor in everyday life. Metaphors, they claim, are not simply poetic linguistic constructs:

> The most important claim we have made so far is that metaphor is not just a matter of language, that is, of mere words. We shall argue that, on the contrary, human *thought processes* are largely metaphorical. [p. 6]

Linguistic metaphors are expressions which describe one entity or concept as being another, drawing on structural parallels between the two: e.g. "Time is Money". Time and money are not literally one and the same, however we treat the abstract *Time* in many of the same ways we treat the more physical *Money*; they are subject to similar processes in our culture. We quantify *Time*, treating it as a commodity, which is valuable and often scarce. *Time* can be spent, wasted, given,

received — even *invested*. In a literal sense, we can do none of these things with ephemeral *Time*. Nonetheless, this metaphor permeates our daily interaction with *Time*; we treat *Time* as an entity capable of such interactions.

Lakoff and Johnson reject the objectivist philosophy that meaning is wholly reducible to propositional forms which are independent of the agent doing the understanding. Thought, at all levels, is structured by the interrelations of a great number of metaphors (if enumerable at all) derived from cultural, physical, and emotional experience.

The "Time is Money" metaphor is cultural and largely tied to post-Industrial Revolution western culture. In the later *The Body in the Mind* [27], Johnson focuses on metaphorical structures which are the result of the basic human physical form — and are thus (more or less) universally experienced by all humans. These simple structures, which he terms *image schemata*, become the basic elements out of which the more abstract metaphors eventually form. As he describes them:

> *A schema is a recurrent pattern, shape, and regularity in, or of, these ongoing ordering activities.* These patterns emerge as meaningful structures for us chiefly at the level of our bodily movements through space, our manipulation of objects, and our perceptual interactions.... [p. 29]
>
> ...[schemata] are not just templates for conceptualizing past experiences; some schemata are *plans* of a sort for interacting with objects and persons.... [p. 20]

Examples are the CONTAINER and related IN-OUT schemata (Figure 1.3). These schemata are manifest in many physical acts, such as "Lou got out of the car" or "Kate squeezed out some toothpaste." However, they also apply to *non*-physical acts, such as "Donald left out some important facts" or "June got out of doing the dishes."

Johnson goes so far as to explain formal logic itself in terms of the CONTAINER schema [27, p. 38]. The requirement that a proposition P be either true or false is the analog of the requirement that an object is either inside a container or outside the container. Transitivity is explained in the same way as a marble and a sack: if a marble is contained in a red sack, and the red sack is contained in a blue sack, then the marble is also contained in the blue sack. Negation, too: just as P is related to the objects contained in some box, $\neg P$ is equivalent to the objects outside of the box. In Johnson's view, abstract logical reasoning does not exist in some absolute objective sense; rather, it is derived from physical experience with containment:

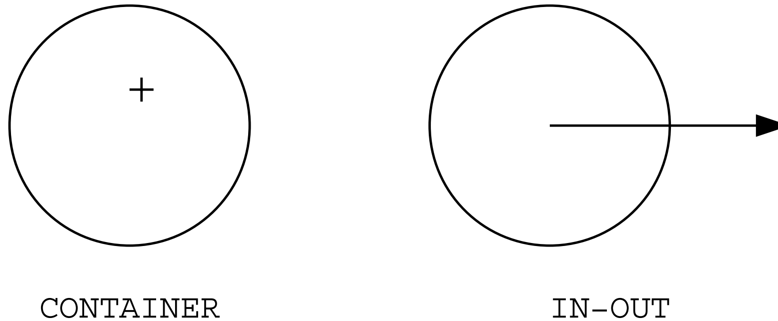


Figure 1.3: Two of Johnson's *image schemata*. The CONTAINER schema captures the various notions of containment, of something held with something else, of something comprising a part of something else. The IN-OUT schema captures the action of something leaving or entering a container. These schemata apply to physical events ("George put his toys in the box.") as well as abstract events ("Emil went out of his mind.").

> Since we are animals, it is only natural that our inferential patterns would emerge from our activities at the embodied level. [p.40]

Johnson produces a "highly-selective" list of 27 schemata (Table 1.1). Some (NEAR-FAR) are topological in nature, describing static relationships. Many (BLOCKAGE, COUNTERFORCE) are *force gestalts*, describing dynamic interactions. While not an exhaustive list, these schemata are pervasive in everyday understanding of the world. These schemata are not just tied to physical interactions, either; they also cross-correlated with recurrent emotional patterns and physiological patterns.

Philosophers and Auto Mechanics

The Cog Project was born out of these ideas [8, 12]. If even our most abstract thoughts are a product of metaphors and schemata which are themselves grounded in our bodies' physical interaction with the world, then human intelligence is inseparable from the human condition. Therefore, if we want to construct a machine with a human-like mind, that machine must also have a human-like body, so that it too can participate in human-like experiences.

The entire philosophical debate between objectivism, cognitivism, phenomenology, experientialism, etc., is just that, debatable. Maybe

CONTAINER	BALANCE	COMPULSION
BLOCKAGE	COUNTERFORCE	RESTRAINT REMOVAL
ENABLEMENT	ATTRACTION	MASS-COUNT
PATH	LINK	CENTER-PERIPHERY
CYCLE	NEAR-FAR	SCALE
PART-WHOLE	MERGING	SPLITTING
FULL-EMPTY	MATCHING	SUPERIMPOSITION
ITERATION	CONTACT	PROCESS
SURFACE	OBJECT	COLLECTION

Table 1.1: The twenty-seven "most important" image schemata listed by Johnson [27, p. 126].

reality can be reduced to a set of symbolic manipulations, maybe not. As roboticists however, we must eventually get our feet back on the ground and go and actually build something. The notion of embodied intelligence suggests an approach to the task worthy of exploration. We should build robots capable of physically interacting with the world (including people) in basic human-like ways. We should try to design these robots such that they can learn simple image-schema-like relationships via such interactions. We should work on mechanisms for connecting such relationships together, for creating more abstract layers woven from the same patterns. Perhaps we will then end up with not only a machine capable of some abstract thought, but a machine which shares enough experience with its creators that its thoughts are compatible with ours and communicable to us.

Even if philosophers eventually conclude that a complete shared experience is not a *formal* requirement for a human-like thinking machine, the approach has practical merit. For example, eyes are certainly no prerequisite for human thought — a congenitally blind person can be just as brilliant as a person with 20/20 vision. But, perhaps mechanisms which co-evolved with our sense of sight contribute to the greater mental process; if we force ourselves to solve problems in implementing human visual behavior, we might happen to discover those mechanisms as well.

This brings up a host of other questions: Have our brains evolved to accommodate *any* metaphors, or a particular limited set? What types of models underlie such metaphors, and which should we build? How much are the metaphors we develop an artifact of whatever brain/body combination we happen to have? (Visually, with our coordinated stereoscopic eyes and foveated retinas, we only *focus on one thing at a time*.

What if we were wired-up like chameleons, with eyes which could be controlled completely independently? Would we have expressions like "I can only focus on two things at a time, you know!")

1.3 Related Work

Many projects have taken these philosophies to heart to some degree. The entire subfield of "embodied AI", in contrast to the symbol-crunching "Good Old Fashioned AI", is driven onward by replacing cognitivism with experientialism. This section describes a representative sample of projects which explore some aspect of *knowledge as interaction*. Each of these projects shaped my own work in some way because they contained ideas which either appealed to me or unnerved me and thus provided vectors along which to push my research.

1.3.1 Drescher's Schema Mechanism

Drescher [16] presents a learning system in which a simulated "robot" learns progressively more abstract relations by exploring and interacting with objects in its grid-world. This system is presented as an implementation of the earliest stages of Piaget's model of human cognitive development [40], namely the sensorimotor stage, in which an agent discovers the basic elements of how it interacts with the world.

This *schema mechanism* comprises three basic entities: items, actions, and schemas (Figure 1.4). Items are binary state variables, which can be *on* or *off* as well as *unknown*. Actions correspond to a monolithic operation. Schemas are predictive or descriptive rules which specify the resulting state of a set of items after executing a particular action, given a particular context (specified by the states of another set of items). The system is created with a number of *primitive* items and actions, which are derived from the basic sensory and motor facilities built into the simulation. The goal of the system is to develop schemas which describe the relations between the items and actions and to develop a hierarchy of new items and actions based on the schemas.

Every action is automatically assigned a blank schema, with no items in its context or its result. In Drescher's notation, such a schema for action Q is written "$-/Q/-$" . Whenever the action is executed, the system updates statistics on the before and after states. Eventually, if the action seems to affect certain state items, a new schema will be "spun-off" which includes those items in the result slot, e.g. "$-/Q/a{\sim}b$". This schema predicts that executing Q always leads to a state in which

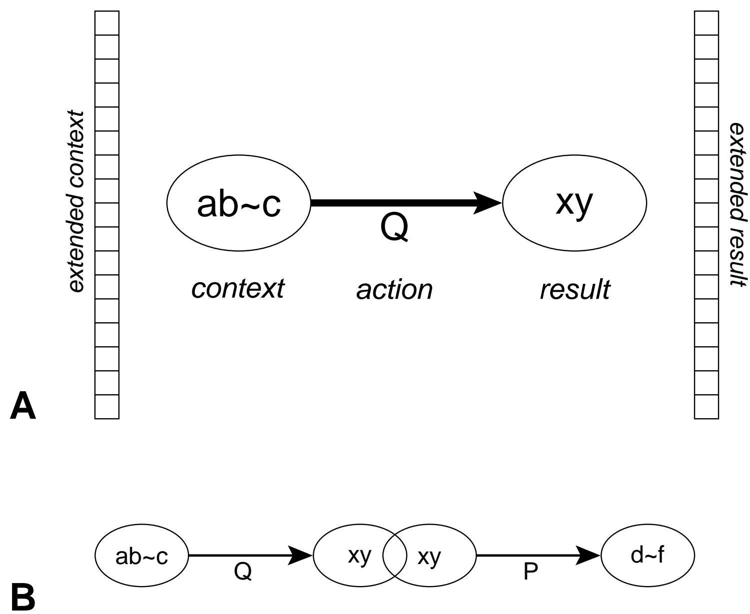


Figure 1.4: Drescher's schema mechanism [16]: A schema (A) is a rule which specifies the resulting state of some binary items ("x" and "y") if an action ("Q") is performed while the system's initial state satisfies some context ("a", "b", and "not c"). A schema maintains statistics on all other ("extended") context and result states as well, which are used to decide to "spin-off" new schemata with more specific context or results. Composite actions are instantiated as chains of schemas (B) with compatible result and context clauses.

a is on and b is off. Such a schema will be further refined if a particular context makes it more *reliable*. This would yield, for example, "$de{\sim}f/Q/a{\sim}b$", a schema which predicts that, when d and e are on and f is off, executing Q leads to a being on and b being off.

Composite actions can be created, which refer to chains of schemas in which the result of the first satisfies the context of the next, and so on. Executing a composite action amounts to executing each subaction in sequence. *Synthetic items* can also be created. Instead of being tied to some state in the world simulation, each synthetic item is tied to a base schema. The item is a statistical construct which represents the conditions that make its basic schema reliable.

These two methods for abstraction give the schema system a way to represent concepts beyond raw sensor and motor activity. Drescher gives the example that a schema that says "moving to (X, Y) results in a touch sensation" effectively defines the item of state "tactile object at position (X, Y)". For that schema to be reliable, that bit of knowledge must be true — so in the schema system, that schema *is* that knowledge.

Limitations The concepts espoused in Drescher's work resonate strongly with the founding goals of the Cog Project. The schema mechanism is, unfortunately, of little practical value in the context of a real-world robot. However, my own work was greatly influenced by the desire to address its unrealistic assumptions.

The schema system is essentially a symbolic AI engine. It operates in a toy grid-world with a small number of binary features. The "robot" has 10 possible primitive actions: moving its "hand" in 4 directions, shifting its "glance" in 4 directions, and opening or closing the hand. The primitive state items correspond to bits for each possible hand location (in a 3x3 region), each possible glance position, contact of "objects" with the "body", each visual location occupied by an object, etc. — a total of 141 bits. Sensing and actuation are perfect; there is no noise in any of those bits. Actions are completely serialized, carried out one-at-a-time and never overlapping or simultaneous. Furthermore, except for occasional random movement of the two objects in this world, the world-state is completely deterministic, governed by a small set of logical rules. The lack of a realistic notion of time and the lack of any material physics in the grid-world reduces the system to an almost purely symbolic exercise.

This grid world is very unlike the world inhabited by you or me or Cog. Cog's sensors and actuators are closer to continuous than discrete; they are certainly not binary. They are also (exceptionally) noisy. Cog

has mass and inertia and the dynamics that accompany them. And Cog interacts with very unpredictable people.

In my work, I have made a concerted effort to avoid any grid-world-like assumptions. The lowest-level primitive actions (roughly, the *movers* described in Section 5.3.1) are velocity-based and controlled by a continuous parameter (well, a `float`). Sensors provide time-series of vectors of real numbers, not binary states. Via external reward, the system distills discrete contexts representing regions of the parameter spaces of the sensors; the states of these contexts are represented as probabilities. Real-time is ever present in the system both explicitly and implicitly.

1.3.2 Billard's DRAMA

Billard's DRAMA (Dynamical Recurrent Associative Memory Architecture) also bills itself as a complete bottom-up learning system. It [3, p.35]

> tries to develop a single control architecture which enables a robot to learn and act independently of a specific task, environment or robot used for the implementation.

The core of the system is a recurrent neural network which learns relations between sensor states and motor activity. These relations can include time delays. Thus, the system can learn a bit more about the dynamics of the world than Drescher's schema mechanism.

DRAMA was implemented on mobile robots, both in simulation and the real world. (Further experiments were also conducted with a "doll robot" [2].) Two types of experiments were performed. In the first, a hard-wired "teacher" robot would trundle about a world populated with colored boxes (and, in simulation, sloped hills). As it encountered different landmarks, it would emit a preprogrammed radio signal describing the landmark. A "learner" robot would follow the teacher, and learn the radio names for landmarks (as it experienced them via its own sensors). In the second set of experiments, the learner would follow the teacher through a constrained twisting corridor and learn the time-series of sensory and motor perceptions as it navigated the maze.

The results of experiments were evaluated by inspecting the connections learned by the DRAMA network and verifying that the expected associations were made and that the knowledge was "in there". It is not clear, however, how that knowledge could later be put to use by

the robot. (Perhaps, once the trained learner robot were let loose, it would emit the right radio signals at the right landmarks?)

The fact that this system was implemented on real robots is significant, because it demonstrates that DRAMA could function with noisy sensing and actuation. The sensor and motor encodings are still overly simple, however. The robots have two motors each, controlled by a total of six bits (three per motor, corresponding to on/off, direction, and full/half speed settings). Each robot has five or six sensors each, totalling 26 bits of state. However, the encodings are unary. For single-bit sensors, like bump detectors, they are simply on/off. For multi-bit sensors, such as the 8-bit compass, each possible state is represented by a different bit. (The compass can register one of eight directions; only one bit is active at any given moment.) Overall, this situation is not very different from the discrete on/off *items* of Drescher's schema mechanism.

Although DRAMA can learn the time delays between sensor and motor bit-flips, it has no mechanism for abstraction. DRAMA cannot condense patterns of activation into new bits of state. The structure of the network is fixed from start to end.

1.3.3 Pierce's Map Learning

Pierce [41] created a system in which a simulated mobile robot learns the physical relationship of its sensors and then learns control laws which relate the sensors to its actuators. The simulated robot is simply a two-dimensional point with an orientation, which moves around in a variety of walled environments with immovable obstacles (e.g. more walls). The agent is equipped with a ring of 24 distance sensors, a 4-bit/direction compass, and a measurement of "battery voltage". It moves via two velocity-controlled actuators in a differential-drive "tank-style" configuration.

Pierce's system discovers its abilities in four stages:

1. Model the sensory apparatus.
2. Model the motor apparatus.
3. Generate a set of "local state variables".
4. Derive control laws using those variables.

In the first stage, the robot moves around its environment randomly. The sensors are exercised as the robot approaches and leaves the vicinity of walls. The sensors' receptive fields overlap, and thus the data

sampled by neighboring sensors is highly correlated. The system uses that correlation to group sensors together and derive their physical layout. In the second stage, the robot continues to move around randomly. The distance sensors are constantly measuring the distances to any obstacle in the line of sight — and thus they measure the robot's relative velocity with respect to the obstacles. Since the distance sensors are in a known configuration, these values give rise to velocity fields, and applying principle-components analysis to these fields yields a concise description of the principle ways in which the robot can move. The third stage amounts to applying a variety of filters to the sensor values to find combinations which result in constraints on the motion of the robot. The filtered values are used as new state variables and, finally, the constraints they impose are turned into control laws for the robot.

This system is intriguing because it uses regularities in the robot's interaction with the environment to distill the simple physics of the robot from a complex array of sensors. And, unlike the previous two projects, it broaches the confines of binary state and action, using real-valued sensors and actuators. Furthermore, it does incorporate a notion of abstraction, in the derivation of the "local state variables". However, it depends heavily on many assumptions which are not valid for a humanoid robot.

Pierce makes the claim that his system transcends its implementation [41, p. 3]:

> The learning methods are domain independent in that they are not based on a particular set of sensors or effectors and do not make assumptions about the structure or even the dimensionality of the robot's environment.

Actually, the methods are completely dependent on the linearity constraints imposed by the simulation. His system would not fare so well discovering the kinematics of a 6-dof arm, in which the relation between joint space and cartesian space is not translation invariant. The methods also depend on locality and continuity constraints applied to the sensors. They work with 24 distance sensors which exhibit a lot of redundancy and correlation; the methods would not work so well if there were only four sensors. Furthermore, the sensors and actuators in Pierce's simulation are completely free of noise. It is not clear how robust the system is in the face of imperfect information.

1.3.4 Metta's Babybot

Metta's graduate work [36] revolves around "Babybot", a humanoid robot consisting of a 5-dof stereoscopic head and a 6-dof torque-controlled arm. The robot follows a developmental progression tied extensively to results in developmental psychology and cognitive science:

1. The robot begins with no motor coordination at all, making a mixture of random eye movements and arm motions.

2. Using visual feedback, it learns to saccade (a one-shot eye movement to focus on a visual stimulus) progressively more accurately as it practices. The head moves very rarely.

3. As saccade performance improves, the head moves more frequently, and the robot learns to coordinate head and eye movement. The head is moved to keep the eyes centered "within their sockets".

4. As visual target tracking improves, now that the head can be controlled, the robot learns to coordinate movement of its arm, as a visual target.

5. Finally, the robot will look at moving targets and reach out its arm to touch them.

Each stage in the sensorimotor pipeline in this system depends on the stage before, so a succeeding stage cannot begin learning until the preceding stage has gained some competence. However, the noisier, lower-resolution data provided by the preceding stage early in its own development actually helps the succeeding stage in learning.

Metta's project culminates in essentially the same demo goal as my work: to have the robot reach out and touch objects. We have very different approaches, though. Babybot follows a preset, preprogrammed developmental progression of learning motor control tasks. Its brain is prewired with all the functions and look-up tables it will ever need, only they are missing the correct parameters. These parameters are acquired by hard-coded learning algorithms which are waiting to learn particular models as soon as the training data is good enough. In my work, on the other hand, I have tried to avoid as many such assumptions about what needs to happen as possible. The goal of my system is to try to discover where certain models can be learned, and which models are worth learning.

Both approaches have their places. The *tabula rasa* makes for a cruel classroom; no learning is successful without being bootstrapped

by some initial structure. On the other hand, in a dynamic, complex world, there is only so much scaffolding that one can build — at some point a learning agent must be provided with pipes and planks and allowed to continue the construction on its own.

1.3.5 Terence the Terrier

Blumberg et al [5] have created an animated dog, Terence (third in a distinguished pedigree, following Duncan and Sydney [52]). This creature, living in a computer graphics world, can be trained to perform tricks by a human trainer who interacts with it using two rendered hands (controlled via joystick) and vocal commands (via microphone). The trainer can reward the dog with a CG treat. Using a *clicker training* technique, the trainer clicks (makes a sharp sound with a mechanical clicker) and rewards the dog when it (randomly) performs the desired action. The click tells the dog when the action is complete, and the dog soon associates the action with receiving reward. The dog starts performing the action more frequently, and then the trainer rewards the dog only when the action is performed in conjunction with a verbal utterance. The dog then learns to perform the action on cue. This process is, in a reinforcement-learning-like fashion, a matter of linking states to actions. However, this dog's states and actions are not necessarily discrete and not completely enumerated at the outset.

Terence's states take the form of binary *percepts,* composed of individual model-based recognizers organized in a hierarchical fashion. As new raw sensory data arrives, it is passed down the *percept tree* to more and more specific recognizers, each dealing with a more specific subset of the data. These models are added to the tree dynamically, in response to input patterns that are reliably coincident with reward. Thus, only the regions of the sensory state space which are conducive to receiving reward are noted.

Terence's initial action space consists of a collection of short, hand-picked animation sequences which constitute its behavioral and motor primitives. These actions are represented as labelled trajectories through the *pose space* of the dog, which is itself a set of snapshots of the motor state (joint angles and velocities). The poses are organized in a directed graph which indicates preferential paths for transitioning from one pose to another. Some of the primitive actions are parameterized (the "shake-paw" amplitude is mentioned). It is further possible to create new actions (trajectories through the pose space). The percept tree includes recordings of short sequences of motion; if such a sequence is reliably rewarded, it is added to the action list. Note, however, that

all actions, even novel ones, are paths through the nodes of the same static pose graph.

1.3.6 Previous Work on Cog (and Cousins)

Over the years, Cog has spawned a lot of work on many elements of motor control, social interaction, and cognitive systems. Matt Williamson investigated control of the arms with kinematically-coupled non-linear oscillators [48]. Brian Scassellati explored a theory of body and mind, resulting in a system which could distinguish animate from inanimate objects and which could imitate simple gestures [45]. Cynthia Breazeal, working on Cog's close relation Kismet, developed a robot with a wide range of convincing facial and auditory gestures and responses [7]. Although it was only a head, Kismet was quite successful at "engaging" and shaping the attention of people around it. Bryan Adams developed a biochemical model for Cog's motor system [1]. I worked on using motor knowledge to enhance sensory performance [33, 32].

Until Paul Fitzpatrick's contemporaneous work on understanding objects by poking them [18], these projects all sorely lacked a significant feature: learning of any long-term behaviors. These projects all had adaptive components, where parameters were adjusted or calibrated as the robots ran, but the maps or functions learned there were hard-wired into the system. Scassellati's imitation system could observe, encode, and mimic the trajectory of an object, but that knowledge was transient. The last trajectory would be thrown away as soon as a new one was observed. Furthermore, that was the system's sole behavior, to imitate gestures; there were no mechanisms for deciding to do something else. Kismet could hold and direct a person's attention, could express delight and frustration in response to the moment — but all of its behavior was a transient dance of hard-coded primitives, responding to that moment. It attended to people and objects but didn't learn anything about them.

That's where this work comes in: creating a framework which enables Cog to actually learn to do new things, to retain that knowledge, and to manipulate that knowledge. Unfortunately, this work suffers from a converse problem: the components built for it so far, and thus the knowledge it can acquire, are few and simple. In a perfect world (i.e., if the robot were not quickly sliding into obsolescence) I would revisit all the previous projects and try to adapt their systems to this new framework. In doing so, the framework would certainly evolve. The dream is to reach a point where enough of the right common representations and interfaces are developed that it becomes trivial to drop

in new models which shuffle around and find their place and function among the old ones.

That, however, is for later. Now it is time to discuss what has actually been done.

Chapter 2

sok

sok is an API for designing behavior-based control systems, and it is the foundation upon which the software in this thesis is built. sok shares many of the same goals as Brooks' original *Behavior Language* (BL) [11], and it is the evolutionary successor to InterProcess Socks (IPS) [10] and MARS [9], which had been used in earlier work on Cog. Unlike its Lisp-based ancestors, sok is implemented as a C library and API, and it allows computation to be distributed throughout a multiprocessor network running the QNX operating system (i.e. Cog's current, third, and final computing environment).[1] sok provides a real-time, dynamic environment for data-driven programming, which is essential to realizing the goals of this project.

This chapter describes the essential features and structure of programming with sok. A complete description can be found in the *sok User Manual* [31].

2.1 Design Goals

sok was designed with a number of specific goals in mind. First and foremost, the purpose of sok is to enable coding which distributes computation over many processors. Much of the computation on Cog is I/O bound (i.e. simple computations applied to large continuous flows of data), so sok has to be lightweight and to make efficient use of the network. QNX provides an optimized network-transparent message-passing system, and sok uses this as its communication medium.

[1] sok is built on top of the message-passing features of the QNX operating system. However, it could probably be ported to another OS given the appropriate communication layers.

sok supports dynamic networks of processes. Processes, and connections between them, can be added to and removed from the running system. This is in contrast to Behavior Language (or the C40 network in Cog's second brain): processes and their connections were specified statically at compile time, and there was no runtime process control. sok builds on top of QNX's POSIX process control, so processes can be started, suspended, resumed, and killed from the QNX shell.

The network of processes maintained by sok is tangible. A program can traverse the network and explore how processes are connected together. This allows for code which programmatically spawns new processes and attaches them to appropriate places in the network. sok includes a simple typing system which allows programs to identify what type of data is being passed via various ports.

sok's process network is also saveable and restoreable. Cog's software has many adaptive and learning modules; the goal of the research is to create a system which grows and develops as it runs. It is crucial that sok be able to save and restore the complete state of the system, so that the system can continue to develop between power-cycles, and to aid off-line analysis. Since processes can be created and hooked into the network on the fly, this state consists of the connections between processes as well as their individual runtime states.

Lastly, sok is robust in the face of temporary failures in the system. sok allows dead or hung modules to be restarted without losing connection state. If a processing node fails, the processes which ran on it are lost (until restarted), but the rest of the system marches onward without deadlocking.

2.2 System Overview

The sok world consists of three parts: individual processes compiled with the sok C library, a locator daemon, and some shell utilities.

The fundamental unit in the sok paradigm is a *sok-process* (Figure 2.1), which can be considered a typical POSIX-like process executing in its own memory space, augmented with some built-in communication and control features. A sok-process exchanges data with peer sok-processes via *inports* and *outports*. Typically, a sok-process responds to received data on its inports, performs some calculation, and then sends messages with the results via its outports. The intent is that each sok-process encapsulate some behavioral primitive such as "visual motion detection" or "arm motor interface". Such primitives are coded independently of each other. Yet, by connecting ports, they are

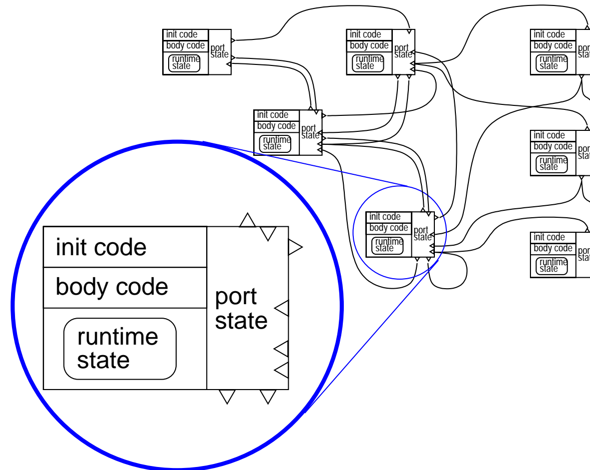


Figure 2.1: A network of *sok-processes,* connected via *inports* and *outports*. The inset highlights the structure of a sok-process with a number of ports. Multiple incoming and outgoing connections are allowed. The body code runs asynchronous to and independent of the message passing (both in time and process space).

glued together to form a complete control system.

The connections of a sok-process are independent of the process execution; messages are sent and received asynchronously. Furthermore, a sok-process can be suspended, or even killed and restarted, without affecting the state of its connections. This is useful for graceful crash recovery, preserving system state, and testing by "lesioning" the system.

Each sok-process has a unique name in a hierarchical namespace managed by the locator daemon, `soklocate`. This program runs on one node in the network and maintains a record of the names, process id's, and port lists of all registered sok-processes. The locator is consulted when a new sok-process is created, or when process and port names are referenced to create connections. Once sok-processes are running and connected, however, they will continue to run even if the locator goes down. When the locator is restarted, sok-processes will automatically reconnect to it, allowing it to re-establish the registry of sok space.

External to all of this is the `sok` utility program, which can be used in the shell (and in shell scripts) to connect and disconnect ports, start and stop processes, examine process status, etc. `sok` provides a command-line interface to the more useful public parts of the sok

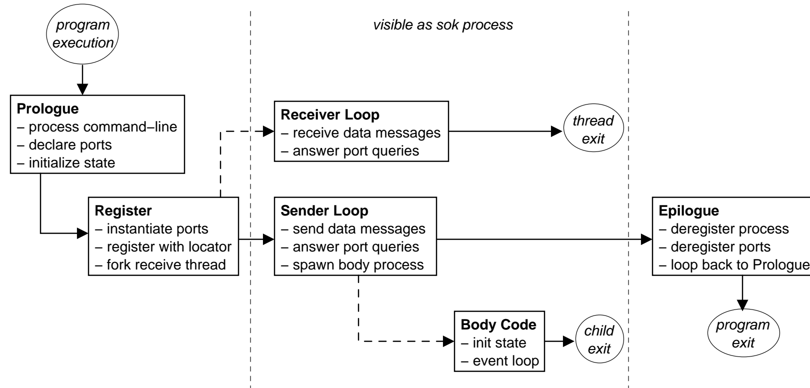


Figure 2.2: The typical life-cycle of a sok-process: it begins as a regular process, and does not become visible in *sok space* until it registers with the sok locator daemon. At that point, the original process forks into two threads to handle sending and receiving port messages. When the body code is spawned, the sender thread forks an independent child process. When the sok-process removes itself from sok space, it notifies the locator and then kills any extra threads and children.

messaging library. The final utility provided by sok is the simple type compiler, sokstc, which turns port type descriptions into code for creating typed ports.

2.3 Life Cycle of a sok-process

Figure 2.2 depicts the life cycle of a typical sok-process. It begins, like any other POSIX process, with the execution of a program. The program processes command-line arguments, and perhaps reads a configuration file. It is not actually a sok-process, however, until it registers with the locator daemon.

Upon registering with the locator daemon, the process declares its unique sok name and the names and types of all of its ports. It also forks into two threads: one for sending messages, and one for receiving them.[2] Once registration is complete, the process is fully visible in sok space. Its ports can be connected to ports on other processes, and it is ready to receive and send messages.

[2]This is necessary to avoid a deadlock condition in QNX which may occur when a cycle of processes attempt to send messages to each other.

The newborn sok-process will not actually do anything with messages, though, until the body code is "spawned". This causes the original process to fork again and run the user's code in a separate process, which protects the message handling code from segmentation faults and other damage. This new body process is the actual "meat" of the sok-process and performs the user's computation, acting on data received from inports (or hardware) and sending data via outports. The body process can be killed and respawned; this does not affect the ports or their connections, or the status of the process in `sok` space.

At this point, the sok-process is happily doing its job. It can then be told to "exit", which causes it to kill the receiver and body threads, disconnect all ports, deregister and disappear from `sok` space, and then, usually, exit. But, the process could be written to reconfigure itself, reregister with the locator, and begin the cycle anew.

2.4 Anatomy of a sok-process

The anatomy of a typical sok-process (Figure 2.3) reflects its life cycle. The first part, the prologue, is where the `sok` configuration is set up. `SokParseOptions()` is used to parse standard `sok`-related command-line options. All input and output ports are declared with `SokRegisterInport()` and `SokRegisterOutport()`. (The ports are not actually created until the sok-process is registered.) `SokParamAllocate()` can be used to create a block of memory which is preserved between invocations of the body code.

The process becomes a true sok-process once `SokInit()` is called. This function allocates memory for port data buffers, forks off the handler threads, and registers the process with the locator daemon.

`SokInit()` never actually returns to the original process until the sok-process is told to exit. However, whenever the sok-process is told to spawn the body code, `SokInit()` will fork and return `SOK_OK` to the child process. Thus, the code following a "successful" invocation of `SokInit()` is considered the *body block*.

The body block is usually an event-driven loop. At the beginning of a cycle, it waits for activity on a `sok` port (new data, new connection) or the expiration of a timer. Then, it may lock inports and read data, perform calculations, and finally send data via outports. If the body block ever exits, the body process dies, but may be respawned again.

Any code that executes after `SokInit()` returns a non-`SOK_OK` condition is considered part of the epilogue. Most processes will simply exit at this point. However, it is possible for a process to reconfigure it-

```
int main(int argc, char **argv)
{
  /*** Prologue ***/
  sok_args_t sargs;
  sok_inport_t *in;
  sok_outport_t *out;

  SokParseOptions(argc, argv, &sargs);
  in = SokRegisterInport("color",
                         SokTYPE(uint8), 0, NULL);
  out = SokRegisterOutport("shape",
                           SokTYPE(uint8), 0, NULL);

  /*** Registration ***/
  if (SokInit(&sargs) == SOK_OK) {
    /**** Body Code Block ****/
    /* ...setup body */
    .
    /* ...event loop */
    while (1) {
      SokEventWait(...);
      .
      .
    }
  }
  /*** Epilogue (SokInit() failed or returned) ***/
  .
  .
  exit(0);
}
```

Figure 2.3: Outline of typical code flow for a sok-process, created via the C library. The call to `SokInit()` instantiates all the ports and registers the process in sok space. The original process does not return from this call until the sok-process is deregistered. A child process is forked and returns from this call in order to spawn the body code block.

self — by declaring new ports, for example — and then call `SokInit()` again to return to `sok` space, reborn as a new sok-process.

2.5 Arbitrators and Inports

By default, an inport acts like a pigeonhole for incoming data from connected outports. When a new message is received, it overwrites any old message and a "new data" flag is set for the inport. This default behavior can be changed by defining an *arbitrator* for the inport.

An arbitrator allows one to implement more complex data handling, including processing which is connection-specific. It is essentially a stateful filter. Possibilities include accumulating inputs (the port delivers a running sum of all received messages), per-connection noise filtering, subsumption-type connections (where incoming messages on one connection inhibit other connections for a fixed period of time), and neural-net-like weighting of connections.

Arbitrators are implemented as sets of callback functions which are run at a number of points in an inport's life-cycle: creation/destruction, connection/disconnection, data reception, and dump/restore. These functions are called in the process space of the handler code — not the body code — so they must be written carefully. In particular, the data-received callback must be lightweight since it is called for every incoming message.

Arbitrators can also request a shared memory block so that they can communicate parameters with the body process, such as weights or timeout values for incoming connections.

2.6 Simple Type Compiler

One of the main goals of `sok` is to enable processes to automatically connect themselves to each other at runtime. To provide some clue as to when such connections are appropriate, `sok` ports are typed. Each port carries a type signature — the typeid — which identifies the type in terms of primitive integer and floating-point elements. Ports with mismatched typeid's are not allowed to connect to each other. Typeids are also catalogued by the locator daemon, so it is possible to query the locator for a lists of compatible ports on other sok-processes.

The `sok` type system is similar to the IDL of CORBA [38]. `sok` types are defined in a description file which is processed by `sokstc`, the `sok` type compiler, to produce appropriate C code and header files.

primitive types:	`float`, `double`, `int8`, `int16`, `int32`, `uint8`, `uint16`, `uint32`
compound types:	`(array` *subtype N*`)` `(struct (`*type-spec name*`) ... )`
constant definition:	`(defconst` *NAME value*`)`
type definition:	`(deftype` *name type-spec*`)`

Table 2.1: Syntax of the `sok` type description language. A *name* must be a valid C identifier, since type definitions are literally converted into C code. A *type-spec* can be any single primitive or compound type. The primitive types correspond to the standard floating point and integer C data types, and the compound types are equivalent to C arrays and structs.

`sokstc` is also embedded in the `sok` C library. This allows programs to dynamically parse type descriptions at run-time.

`sokstc` is implemented using SIOD [14], a small embeddable Scheme interpreter which compiles very easily on the QNX platform. `sokstc` description files are actually Scheme programs with support for rudimentary macro operations. The description language syntax is outlined in Table 2.1. `defconstant` is used to define symbolic constants, which appear as `#define`'d constants in the header files generated by `sokstc`. `deftype` defines a typeid in terms of predefined primitive or compound types. The ten primitive types correspond to the common floating-point and signed/unsigned integer types in C. The two compound types are arrays and structures. Arrays are fixed-length vectors of a single subtype; structures are fixed, named collections of any other defined types. Variable length or recursive type definitions are not allowed.

The standalone `sokstc` reads a description (`.stc`) file and produces a pair of C header (`.h`) and code (`.c`) files. These contain definitions for the type signatures as well as matching C type declarations (`typedef`) which can be used in user code. The signatures are used to register ports, and the declarations are used to access the port buffers. The embedded compiler can be accessed by calling `SokstcParseFile()`. This will open a file by name, parse it, and return an array of typeids.

The `sok` C library includes a number of other functions for working with typeids, such as walking through a typeid or typed buffer, locating elements in a typed buffer, and generating new structure or array typeids from other typeids.

2.7 Dump and Restore

sok supports the ability to dump and restore the state of sok space. This is important because the collection of running sok-processes and the connections between them can change as the robot runs. The results of learning and adaptation by the robot accumulate in the process network as well as the individual processes themselves.

sok-processes respond to a "dump" request by saving the following information to a disk file:

- process info: name, time stamp, command-line used to invoke the process;
- connection info: lists of connections (by name) for each port;
- runtime info: contents of a specially-allocated block of *parameter memory.*

The `sok` utility can be used to tell any or all processes to save their state to time-stamped files in a given directory.

When sok-processes receive a "restore" request, they read the designated state file and reverse the procedure, loading any parameter memory and establishing the specified connections. The `sok` utility can be used to recreate sok space from a collection of process state files. It will invoke each process in turn, as directed by the state file, and tell each new sok-process to restore itself.

If a process has any runtime state which should be saved, such as a neural net which has been learned at runtime, that data needs to be kept in a *parameter memory* buffer. This is a block of shared memory which is accessible to the handler code and which is preserved between invocations of the body code.

Chapter 3

meso

meso is the motor control architecture developed for this project. meso provides a uniform interface for controlling the arms, torso, and head of Cog via a collection of *virtual muscles*. It simulates a number of key features of the human musculoskeletal system, features which are important in a machine which will learn to move in a human-like way and which should experience human-like interaction with the world.

meso breaks down into roughly three layers: low-level (hardware) control, a skeletal model, and a muscular model. These are implemented by several sok-processes distributed over several processors (Figure 3.1). This chapter first describes the rationale behind meso, followed by details of each layer of the system. The last section describes feedback mechanisms provided by meso and how they affect the operation of the robot.

3.1 Biomechanical Basis for Control

Cog is an anthropomorphic robot: one fundamental principle of its mechanical design is that it should have enough degrees of freedom and articulation to enable it to recognizably emulate human motor behavior. This mandates a minimum hardware requirement: e.g. Cog needs two human-like arms, because it can't pretend to move an arm it doesn't have. There is no such fundamental requirement for the control system, though. A very elaborate animatronic motor controller can produce very life-like canned motion, although the controller itself bears little resemblance to a biological motor system.

Cog was not built to produce canned animations; the goal of the project is to explore mechanisms for generating and learning social

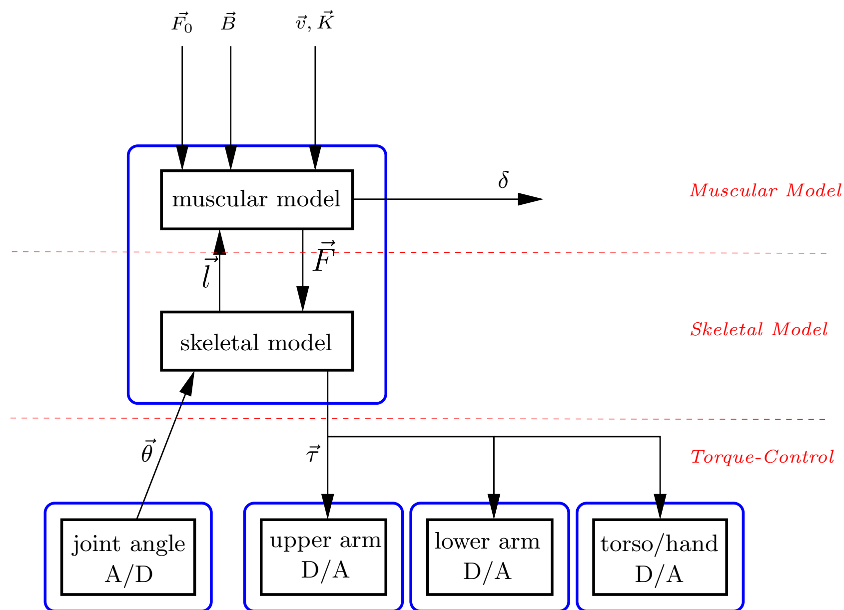


Figure 3.1: Overview of meso. Black boxes indicate sok-processes; blue boxes indicate separate processors. Due to hardware I/O peculiarities, each motor controller (D/A) board runs on its own processing node. Originally implemented as separate processes, the skeletal and muscular models eventually merged into the same process to reduce communication latencies and improve performance.

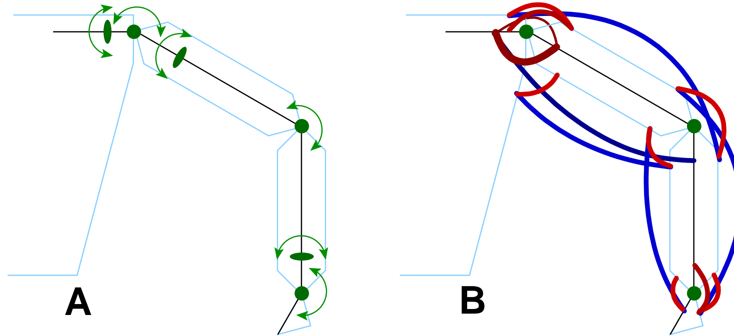


Figure 3.2: Cog's arms have six single-axis actuators (A); however, they are controlled as if they were actuated by antagonistic pairs of real muscles (B). These virtual muscles can span multiple joints, coupling their movements together.

behavior. Perhaps, if Cog's emulation of the human form includes key features of the motor system, the learning mechanisms will have a very natural form. With that in mind, Cog's actuators should incorporate human-like control as well as mechanics.

It would be impossible to give muscles to Cog without entirely rebuilding the robot from scratch. But, it is possible to implement an abstraction layer which provides an effective *simulation* of human musculature. meso is this abstraction layer. Cog's raw actuators are torque-controlled motors [49]. meso builds on top of them, simulating "virtual muscles" which mimic the operation of muscles in a human body (Figure 3.2).

meso incorporates three essential features of the human musculoskeletal system: reflex stiffness, polyarticulate coupling, and a fatigue model. This allows production of human-like movement which a higher-level control system can tune and optimize via biologically relevant feedback cues.

Reflex Stiffness

Human muscle tissue is mechanically very different from electric motors, even motors under force-control. Much work has been done to model how muscle tissues produce forces and react to loads [51]. However, an accurate simulation of muscle tissue itself is not necessary. The brain's motor cortex does not directly activate muscles; the cortex connects to spinal motor neurons which control the muscles in conjunction

with spinal reflex loops. Using input from stress and strain sensors in the muscles and tendons, these reflexes make antagonistic pairs of muscles act like simple damped, linear springs over a wide range of motion [29].

meso incorporates this subcortical machinery into its simulation in the form of a spring law to calculate muscle force. Each virtual muscle plays the role of a pair of biological muscles along with their associated spinal feedback loops.

Polyarticulate Coupling

Many muscles in the human body span more than one joint. For example, the biceps and triceps each span both the elbow and shoulder joints. Such muscles are *kinematically* redundant, because identical arm configurations can be produced using muscles which span only one joint. However, polyarticulate muscles have at least two important effects on the *dynamics* of the limbs.

First, a multi-joint arm actuated by single-joint linear springs will not have isotropic stiffness in endpoint coordinates [24]. In other words, the hand will react with varying stiffness when pushed in different directions, and the stiffness profile will be a function of limb configuration. Polyarticulate muscles add a tunable interjoint coupling, which allows for control of the endpoint stiffness over a wide range of the workspace, independent of the limb configuration. The endpoint stiffness can be made not only isotropic, but can be tuned to match the task at hand. For example, accurately placing a puzzle piece on a table requires high XY stiffness but low Z stiffness, to get precise position control in the plane of the table yet avoid bouncing when eventually making contact with the table.

A second dynamic effect is that polyarticulate muscles can make the musculoskeletal system more efficient [21, pp.298–303]. Applying a force in certain configurations of a multi-joint limb results in some muscles undergoing a "lengthening contraction" (Figure 3.3). That is, the muscle applies force while being stretched, thus doing negative work. Although this quantity of energy is wasted as heat, other muscles must provide that work, which is never seen at the output of the limb. In these cases, a stiff biarticulate muscle can act as a mechanical linkage which lets the limb produce the same force, but without wasting the work.

Cog has no real polyarticulate actuators (each joint is driven by a single motor), and a simulation of such won't make the robot any more efficient in terms of real physical energy. However, if energy consump-

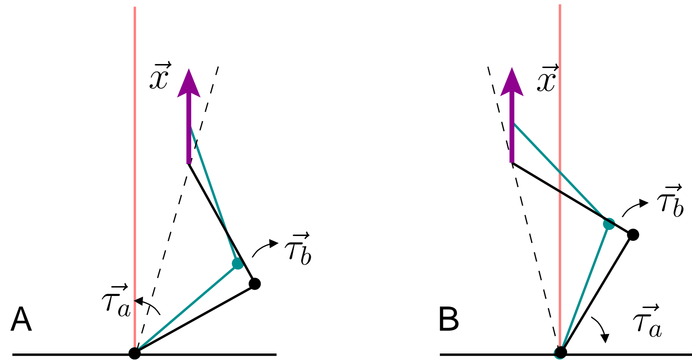


Figure 3.3: The illustrations depict a simple two-segment arm with monoarticulate muscles at each joint. The initial position of the arm is shown in black. (A) To do work along $\vec{x}$ — that is, to apply a force along that vector — the arm must move into the blue configuration. The torques required to produce such a force are given by $\vec{r} \times \vec{F}$, where moment arm $\vec{r}$ is the vector from the joint to the endpoint. The two joints apply torques $\vec{\tau}_a$ and $\vec{\tau}_b$ in the same directions as they are displaced, thus both contributing to the output work. (B) In a different configuration which does the same overall work, the lower joint must apply a torque in *opposition* to its displacement. (This is counterintuitive, but readily apparent when you consider that the dotted line is the moment arm.) This joint is *absorbing* energy, and that energy must ultimately be provided by the other joint.

tion is part of the simulation, then movements which utilize polyarticulate virtual muscles in this way will appear more optimal than those which don't. This bias will further favor the development of human-like movement.

Fatigue

Cog's motors have completely alien fatigue characteristics compared to human muscle. Given an unquenched power supply from the national power grid, the motors can apply a wide range of forces indefinitely, as long as they don't overheat. Human muscles get tired much more quickly, and this affects not only how much they are used, but also *how* they are used.

Constraining the motors to operate under a muscular fatigue model should encourage the development of both human-like movement and behavior. The model used by meso reflects the basic metabolic processes in muscle [35, ch. 6], including a reservoir of energy for penalty-free short-term activity. The fatigue level of a virtual muscle implicitly affects motor performance and is also accessible as direct feedback to the higher-level control system. The model is tunable, making it possible to simulate different stages of growth and ability, as well as different types of muscle tissue.

What meso Doesn't Do

meso does not implement motor control using *postural primitives* [50]. The notion of a postural primitive came out of work by Bizzi and Mussa-Ivaldi on motor activity in spinalized frogs [4, 37, 22]. They discovered that the endpoint forces in a frog's leg produced by activating neurons in certain regions of its spinal cord took the form of force fields modulated by the activation. Each neuron produced a specific force field (forces varying with the position of the endpoint), and the overall magnitude of the field was scaled by the activation. Activation of multiple neurons is additive; the fields of each individual neuron seem to be simply summed. Most fields were convergent, meaning there was some endpoint position to which the limb was pushed from any direction — in other words, an equilibrium point. On a simple level, this is all consistent with treating the muscle groups as spring-like when acting in conjunction with spinal feedback.

In a number of robotics projects ([36, 34, 32]), these equilibrium points are abstracted into a collection *postural primitives.* Each primitive corresponds to one vector of set-points for the set of spring-like controllers driving each joint. Movement is produced by interpolating

between primitives, i.e. moving the set-points of the springs from one position to another. The joint angle set-points can be moved around within the convex hull of the chosen primitives, although the joint angles themselves will stray because the joints are springy.

This has always been a profoundly unsatisfying control scheme to me. The way it is typically implemented amounts to little more than a sloppy (low-stiffness) position control loop. The stiffness of each spring is made constant, and all the interesting dynamic effects, such as changes in compliance over the path of a trajectory and adjusting for contact forces, are ignored.

In meso, virtual muscles are controlled by supplying the stiffness and the set-point velocity. The idea of controlling the velocity rather than the position was inspired by Pierce [41]. Since a zero velocity is equivalent to "no change", the learning modules of pamet can safely probe the operation of the virtual muscles by sending progressively larger velocity commands. Furthermore, the motion is intrinsically smoother: a hiccup in a stream of velocity commands has far less potential for damage than a $5.0 \rightarrow -1.2 \rightarrow 4.9$ glitch in a stream of position commands.

3.2 Low-level Motor Control

Cog's arms and torso are actuated by torque-controlled electric motors. Most robotic actuators, particularly in manufacturing automation, use position control: the primary control parameter is the actuator's position, and the motor is driven so as to track a target position as accurately as possible. Under torque control, the primary control parameter is the output torque of the actuator, regardless of position.

Position-controlled devices are typically very stiff. The ideal position-controlled actuator would lock on to its set-point and produce a potentially infinite force in response to a disturbance. Such a mechanism is perfect for, say, accurately placing parts on a circuit board in a carefully controlled assembly line. Cog, however, needs to interact with and explore an uncertain world full of breakable people and things. We want its actuators to be squishy and compliant: Cog should reach out to shake a visitor's hand without breaking her arm.

A low-gain feedback loop in position control will yield a softer, more compliant actuator but at the expense of sloppy positioning and still no real control of the forces it produces. Electric motors, with high ratio gearboxes, are not very backdriveable. The output shafts do not turn freely due to friction in the motor and gearbox. A position controller, whether sloppy or stiff, will also do nothing to reduce the drag and

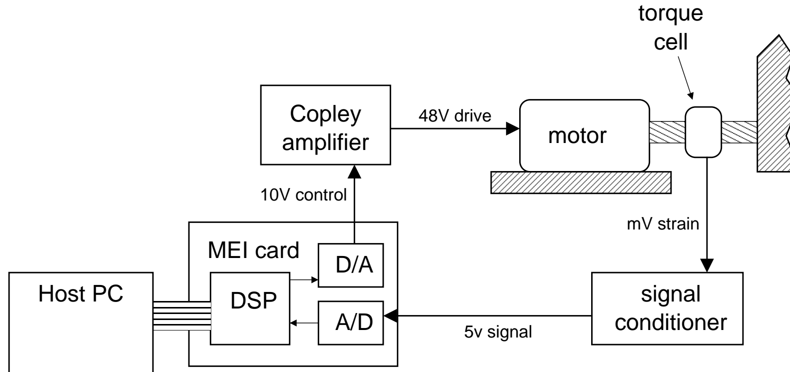


Figure 3.4: Torque feedback loop controlling the torso motors. A torsional load cell in series with each motor's output shaft measures the torque being applied to each joint. Control of the arm and hand motors is similar, except that the torque cell is replaced by a "series-elastic element", which is intentionally compliant.

resistance in an actuator. Under torque control, though, when zero torque is commanded, the motor will actually be actively driven to counteract the frictional forces. A torque-controlled actuator can be smoothly tuned down to a completely floppy stiffness of zero.

The complete feedback loop of a torso joint on Cog is illustrated in Figure 3.4. Torque is measured by a torsional load cell bolted in series with the motor. The controller is a Motion Engineering Inc. multi-axis motor control card ["MEI card"] with an embedded DSP. The torque signal is measured by the the card, which produces a motor control voltage according to a general PID (proportional-integral-derivative) control law. The control voltage is turned into a high-power drive current by a Copley amplifier which feeds the motor. The torso motors are also equipped with optical encoders and limit switches so that their positions can be accurately measured.

The arms are driven with a slightly more elaborate mechanism called *series elastic actuators*, described in detail by Williamson [49]. The torque measuring element is actually a torsion spring equipped with strain gauges. The spring acts as a mechanical low-pass filter which absorbs shock loads — impulses generated when the arm knocks into objects (including the robot itself). Shock loads can strip teeth in a motor's gearbox; the elastic element makes an actuator much more robust if it is expected to knock into things a lot. It also makes the

actuator more stable in response to contact forces, i.e. touching hard immovable objects.

The springs in the arms are quite stiff, but flexible enough that optical encoders in the motors cannot accurately measure joint position. Instead, arm position is measured by ring potentiometers installed at the spring output of each joint.

The MEI card is programmed to run the torque control loop at 1600 Hz. The effective bandwidth of the series elastic actuators, however, measures about 25-30 Hz, due to compliance of the spring and the cables in the drive mechanism. Four MEI cards are used in total, to control Cog's 22 actuators. Each card interfaces to the upper layers of `meso` via a `sok` program aptly titled `mei-glue`. This program provides an inport for torque commands and, for the torso motors, an outport for encoder position. The arm position potentiometers are actually scanned by a separate United Electronics Inc. A/D (analog-to-digital) card, interfaced to the system via `uei-glue`.

The head and eye actuators are, unfortunately, not equipped with torque sensors and are instead driven with position control. Since the head and eyes do not make physical contact with the environment in everyday use, this is not such a drawback. However, the head and eyes do not have variable compliance, and they cannot be coupled to the torso or arms via virtual muscles. In the human body, the eye muscles are independent of the rest of the musculature and are under a more position-like mode of control. The neck is, however, strongly coupled to the torso. Cog is unable to display such motor effects as compensatory stiffening of the neck when the torso leans forward.

3.3 Skeletal Model

Layered on top of the low-level torque control is a *skeletal model* which simulates the kinematics of the skeleton and virtual muscles. The skeletal model essentially computes two functions: the muscle lengths $\vec{l}(\vec{\theta})$ as a function of joint angles, and the joint torques $\vec{\tau}(\vec{F}, \vec{\theta})$ as a function of muscle forces and joint angles.

Two skeletal models were developed in the course of this research. The first is fairly sophisticated — incorporating a kinematic model of the robot — yet fatally flawed. The second model is almost trivial in comparison, but quite usable. Both models support virtual muscles which can span multiple joints.

In either case, the skeletal model was originally implemented as a sok-process wrapped around functions for the lengths $\vec{l}$ and torques $\vec{\tau}$.

New joint angle $\vec{\theta}$ messages from motor glue processes would cause $\vec{l}$ to be recalculated and sent up to the muscular model. Incoming force $\vec{F}$ messages from the muscular model would cause $\vec{\tau}$ to be recalculated and sent back to the motor glue processes. Eventually these two functions were merged into the same process as the muscular model to avoid communication latencies. The complete control loop of the merged system runs at 500 Hz.

3.3.1 Complex Coupling

In the complex coupling model, a virtual muscle is a mechanical element anchored at two points on different body segments of the skeleton (Figure 3.5). The muscle exerts a force directed along the line joining the two anchor points. As the robot moves (or is moved), the effective torques applied by the virtual muscles change according to the configuration of the skeleton. If a real force were being exerted between the anchor points, it would produce a torque at each joint spanned by the muscle. In the model, these torques are calculated and the resulting values are used to command the motors at those joints. Since the same torques appear in the body of the robot, the mechanical effect of the virtual muscle is equivalent to that of the real muscle. This method of simulating mechanical elements is developed much more elaborately in Pratt's *virtual model control* [42], in which the multi-joint legs of a bipedal walking robot are made to act like a simple spring between the body and the ground.

The expression for the torque on joint j due to muscle m is, by definition,

$$\vec{\tau}_{jm} = \vec{F}_m \times \vec{r}_j$$

where $\vec{F}$ is the force vector between two anchor points $\vec{p}_A$ and $\vec{p}_B$, and $\vec{r}_j$ is the vector joining the pivot $\vec{q}_j$ of joint j to either anchor (e.g. $\vec{r} = \vec{p}_A - \vec{q}_j$). Likewise, the length of the muscle is just the distance between the two anchor points:

$$l_m = ||\vec{p}_A - \vec{p}_B||$$

Once $\vec{\tau}_{jm}$ are calculated for all muscles, the total torque for joint j is

$$\vec{\tau}_j = \sum_m \vec{\tau}_{jm}$$

The complexity comes in calculating the actual vectors. This is explained in full detail in Appendix A and summarized below.

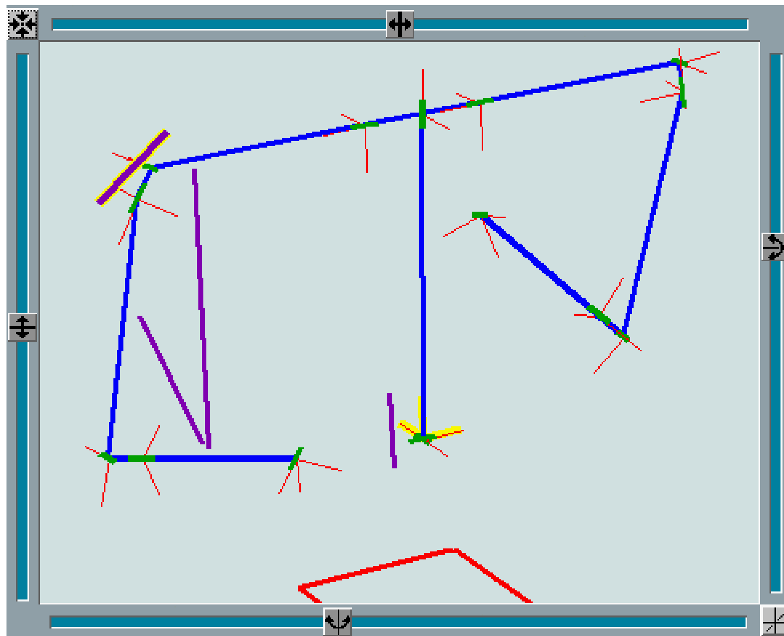


Figure 3.5: Example of complex coupling. Virtual muscles (purple) are lines of force acting between points anchored relative to different links (blue) in the skeleton. Thin red lines indicate the coordinate frames of each link. The figure itself is a screen capture of `mesokinescope`, the program created to compose skeletal models and monitor the skeleton in real-time.

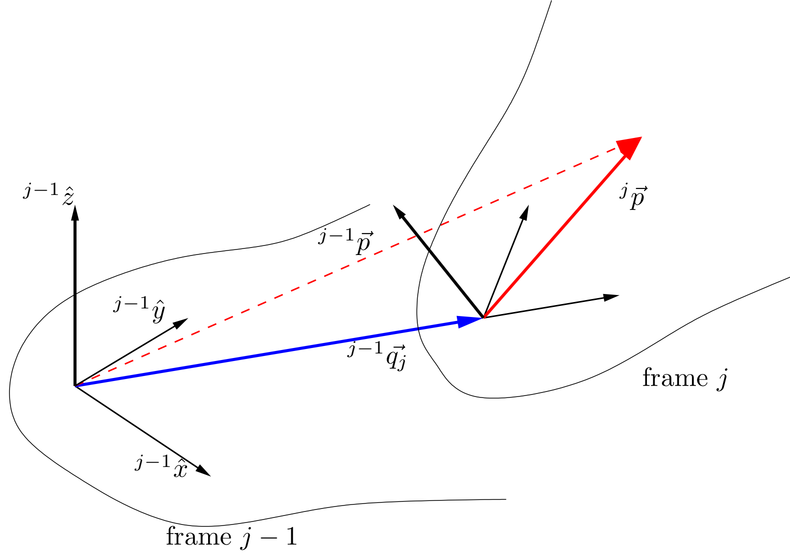


Figure 3.6: Coordinate frame $(j-1)$ is the parent of frame j. Vectors $^{j}\vec{p}$ and $^{j-1}\vec{p}$ describe the same point, but relative to the respective frames. $^{j-1}\vec{q}_j$ defines the origin of frame j.

Each jointed segment of the robot is a link in a kinematic chain.[1] Each link defines a local coordinate frame, with its origin at the axis of its joint (Figure 3.6). That joint is anchored in the frame of the link's predecessor; that is, the location of that joint is determined by a vector in the frame of the previous link in the chain. (The base of the robot, anchored to the ground, provides the base reference frame for the chain.) Each link is thus parameterized by a 3-d position vector ($\vec{q}_j$) and an additional 3 parameters (α, β, θ) which specify the orientation of the joint axis. Each muscle is defined by two anchor points anchored to different links, i and k, and specified by vectors, $^{i}\vec{p}_A$ and $^{k}\vec{p}_B$, in the corresponding frames.

Consecutive frames are related to each other by an affine transform, $^{j}_{j-1}T$, determined by the six parameters which locate one link within the other. These transforms can be cascaded to yield $^{k}_{i}T$ for any two frames along the chain. Using the appropriate $^{k}_{i}T$, one can transform all $\vec{p}_A$, $\vec{p}_B$, and $\vec{q}$ vectors into the same coordinate frame and then evaluate the length and cross-product using common vector operations.

[1] Actually, a kinematic tree in the case of Cog.

Only $2N - 1$ transforms need to be computed for a muscle which spans N joints. If the kinematic parameters are known in advance, the necessary transforms can be precomputed up to factors of the sine and cosine of the joint angles. This is precisely how the complex coupling model is implemented: a *muscle compiler*, `mesoc`, reads a description of the robot's skeleton and virtual muscles and produces C code for the functions $\vec{l}(\vec{\theta})$ and $\vec{\tau}(\vec{F}, \vec{\theta})$.

In Cog's case, this complex coupling was ultimately unusable because it isn't complex enough. In the human body, muscles are anchored to bones and they act via tendons which are constrained to slide over the joints. The moment arms of such action are determined by the routing of the tendons over knobby bones and other connective tissue. In Cog's virtualized skeleton, muscles apply a force directly between two points, and the moment arms are determined by how far the muscle travels away from the joint axis. As shown in Figure 3.7, in a straight configuration a muscle can only apply a torque if it is anchored away from the skeleton. However, in an angled configuration, the anchor points may touch or cross, or the line of force may cross to the other side of the joint, reversing the torque! This restricts the useful range of most muscles to the point where they are just not useful.

Fixing this problem requires modelling tendons and defining channels through which they are constrained to move. For the purposes of `meso` and this thesis, this seemed to be more trouble than it was worth. There exists at least one commercial package [25] which does create dynamic models of tendons, sliding joints, knobby bones, and muscles with multiple anchor points. It is not clear, though, that it can compute this fast enough to be used in a real-time controller.

3.3.2 Simple Coupling

The flawed complex coupling model was discarded in favor of a much simpler model which retained the key feature of polyarticulate coupling. In the simple coupling model, a muscle acts like a cable attached to a pulley driving the joint. The torque exerted by muscle m on joint j is $\tau_{jm} = F_m r_{jm}$, where r_{jm} is the "radius" of the pulley, and the total torque on joint j is thus $\tau_j = \sum_m F_m r_{jm}$. The length or displacement of muscle m is given by $l_m = \sum_j r_{jm}(\theta_j - \theta_{0j})$. The action of this model is easy to visualize with the cable analogy in the one- or two-joint cases (Figure 3.8); a muscle which couples three or more joints is more analogous to a hydraulic system.

The physical geometry of the robot is not important to this model, and the kinematic description of the skeleton is no longer needed. Yet,

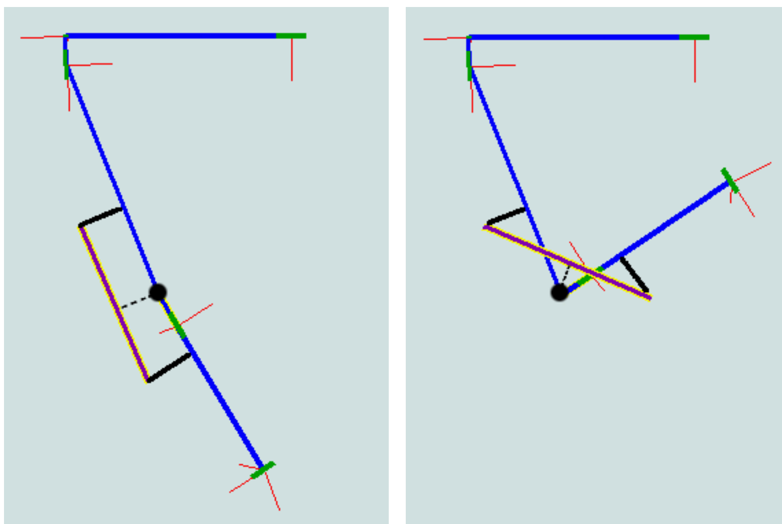


Figure 3.7: Failure of the complex coupling model: A single virtual muscle is shown highlighted in yellow. Its moment arm is the perpendicular (dotted line) from the joint to the muscle. When the joint bends, the point-to-point line of force may cross the axis of the joint. This effectively reverses the torque. A force which was expected to extend the arm may, depending on the joint angle, cause it to contract.

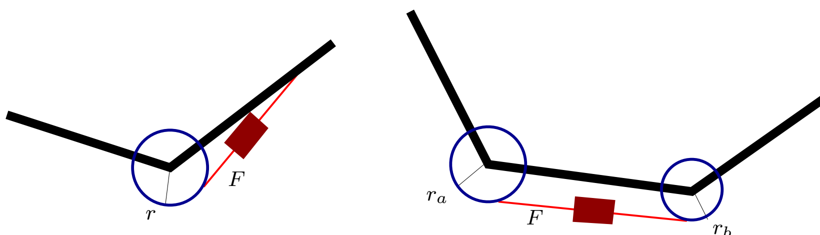


Figure 3.8: In the simple coupling model, virtual muscles act like a cable drawn around pulleys affixed to joints. Each pulley is described by a radius r_{jm} (the moment arm). The torque exerted by all muscles on a joint is $\tau_j = \sum_m F_m r_{jm}$. The length of a muscle is $l_m = \sum_j r_{jm}(\theta_j - \theta_{0j})$. Since the muscle's force law is linear, the absolute length (set by the offset angles θ_{0j}) is not important.

for a wide range of a joint's workspace, this model is actually a better approximation of the action of a tendon which wraps around the joint. Furthermore, the only parameters needed by this model are the pulley radii r_{jm}. No muscle compiler is necessary to achieve real-time performance, so the model can be modified at runtime. This makes development and testing much easier, too.

3.4 Muscular Model

The highest layer of meso is the *muscular model*, which determines the dynamics of the virtual muscles. The job of the muscular model is to compute the output forces $\vec{F}$ of the muscles as a function of their lengths $\vec{l}$. The basic model is a simple damped spring:

$$F = -K(l - l_0) - Bl' + F_0,$$

where l_0 is the equilibrium point (set-point), K is the stiffness, B is the damping constant, and F_0 is a length-independent bias.

This differs significantly from the behavior of an isolated biological muscle. Real muscle tissue is typically modelled as a combination of non-linear contractile and elastic elements (Figure 3.9). Pulse trains from afferent motor neurons cause muscle fibers to twitch, and over the bulk of the material a contraction is produced. The force is sensed by nerves terminating in the tendons (Golgi tendon organs), and muscle elongation is measured by nerves (spindle fibers) in the muscle itself [13]. Overall, real muscle acts as a source of non-linear contractile force, not a spring.

However, feedback loops which connect the force and elongation sensors, spinal ganglia, and muscle fibers do cause real muscles to exhibit a spring-like response to perturbations. Motor centers in the brain drive the muscles via the spinal ganglia by modulating the parameters of those feedback loops. The model I am using is a compromise between biological accuracy and convenience of control. It realizes the cumulative effects of muscle tissue, spinal feedback loops, and the action of antagonistic combinations of muscles.[2]

The spring law is essentially the equation for a proportional-derivative (PD) position controller. Unlike a typical PD controller, however, the stiffness K is not tuned to an optimal value and fixed in place. K is a

[2]Biological muscles can apply contractile forces only; one could easily make the virtual muscles do this by adding a constraint that $F \leq 0$. This would consequently require twice as many virtual muscles, to make sure that each had an antagonist pulling against it.

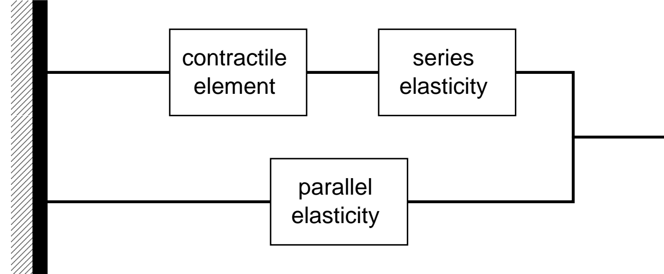


Figure 3.9: The classic Hill model of biological muscle tissue [51, 23]. All three elements are non-linear. The contractile element accounts for the force produced due to neuronal activation, but for a given activation this force is dependent on the velocity of contraction. The stiffness of the series elastic element is not constant, but is proportional to the exerted muscle force.

variable control parameter and is always set relatively low so that the joints remain "soft" in interactions with people and objects. When a muscle is inactive (not being driven by a higher-level process outside of meso), K drops to zero and the muscle is effectively limp. K also plays a role in a fatigue model which further modulates the magnitude of F (discussed in the next section). The damping constant B is the only fixed parameter; it must be set appropriately for each muscle to keep the controller stable.

The muscular model is implemented within a sok-process called `motor/msprings`, which computes $\vec{F}$ from $\vec{l}$ for all muscles at a frequency of 500 Hz. It has control inports for $\vec{B}$, $\vec{F}_0$, and a masked pair of $\vec{v}$ and $\vec{K}$ vectors. The equilibrium point l_0 of a muscle is not set directly; rather, it is controlled by commanding its velocity. This naturally constrains the robot's movements to be relatively smooth, no matter what random values other processes may decide to send to the muscles. The $\vec{v}$ and $\vec{K}$ vectors are accompanied by a bitmask which determines which muscles are affected by the input. A sok arbitrator is used so that multiple processes commanding different muscles can send their messages simultaneously to the same single control inport. The vectors are merged and processed as a single velocity/stiffness command at every timestep. If the stream of commands to a muscle stops (i.e. the message frequency falls below a limit of 5 Hz), then the muscle becomes inactive and its stiffness is ramped down to zero. When the stream begins again, the stiffness ramps back up to the commanded value.

3.5 Performance Feedback Mechanisms

Two important components of meso are the mechanisms with which it provides performance feedback to higher control centers. *Joint pain* is a "discomfort" signal produced when the joints are twisted close to their mechanical limits. *Muscle fatigue* is expressed as both a discomfort signal and a physical weakening produced when a muscle is overused. These signals provide negative feedback to the learning processes described in Chapter 5.

3.5.1 Joint Pain

Joint pain provides feedback to keep the robot from driving itself against its physical limits, which is as unhealthy for robots as it is for humans. Joint pain is produced by the `motor/jlimits` module, which observes the joint angles of the robot and generates a non-zero output δ per joint when the angle is within roughly 15% of its positive or negative limit. This output accelerates as the limit is reached:

$$\delta = \left(\frac{\theta - \theta_{L0}}{\theta_{L1} - \theta_{L0}} \right)^2 \text{ for } \theta_{L0} < \theta < \theta_{L1}$$

where θ is the current position, θ_{L1} is the physical limit, and θ_{L0} is threshold of the pain-inducing region (Figure 3.10). The joint limits are measured by simply keeping track of the min/max observed joint angles; these are initially discovered by "exercising" the robot (manually moving each joint to its limits), but can later be initialized from values stored in a file. The limits decay (shrink) very slowly over time so that the `jlimits` module can adapt to drift in the position sensors. This means that Cog, like its human operators, benefits from a good stretch every now and then to exercise the full ranges of its joints.

3.5.2 Muscle Fatigue

Just as joint pain provides the robot with feedback on the use of its skeleton, muscle fatigue provides feedback on the use of its muscles. A sense of fatigue gives Cog the means to optimize its movements to reduce its overall effort. Although Cog itself has no need to conserve energy (it doesn't foot the bill on its 60-amp AC circuit), it is hoped that energy-efficient motion will also be more elegant motion, i.e. more like a human and less like a (classic) robot. Furthermore, smooth, efficient motor activity does reduce the wear and tear on the machine.

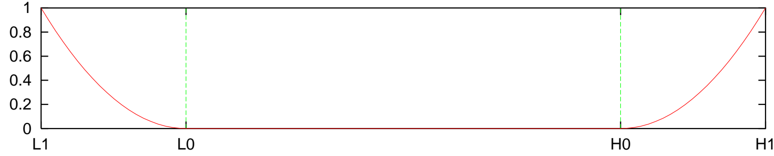


Figure 3.10: Joint pain response through the full range of a joint's motion. The hard limits, L_1 and H_1, are the mechanical limits set by the stops in each joint. The boundaries of the pain regions, L_0 and H_0, can be independently set for each joint.

Fatigue in biological muscles is still not very well understood. It is believed to arise from a combination of chemical and metabolic factors in the muscle tissue, as well as changes in the central nervous system (CNS). In the muscle, depletion of energy stores and blood oxygen, and accumulation of lactic acid, reduces the strength of muscle fiber contractions and slows them down. In the CNS, the motor centers themselves appear to habituate to motor commands and tire of firing the motor neurons.

Adams [1] created a model of human energy metabolism and muscle fatigue for use in Cog. This model simulates a circulatory system, several major organs, and the levels of six blood chemicals (three hormones and three fuels). It allows the robot to experience a wide range of physical conditions, such as exhaustion, fear-induced stress, or carbo-loading. However, none of those extremes are yet needed in this project, and the attention to detail tends to obfuscate the workings of the dynamics involved.

I have created a simple fatigue model, coded directly into the muscle model, which provides gross dynamic properties similar to Adams' work. As opposed to supplies of creatine phosphate, glycogen, glucose, and fat, virtual muscles have only two abstract energy stores: a short-term store S_S and a long-term store S_L. S_S is used with no fatigue effects until it is fully depleted. Subsequently, S_L takes over, with consequences to strength and comfort as it is exhausted.

The fatigue model works as follows. At each time step in the muscular model, after calculation of the desired output force $F(l)$ for a muscle, the required power P is computed according to

$$P = \alpha|Fv| + \beta|F| + \gamma K$$

where v and K are the velocity and stiffness. The first term accounts for the actual mechanical power involved (a muscle is penalized for pro-

ducing or absorbing work). The second term is a penalty for generating a static force (it takes energy just to maintain tension). The third term is a penalty for stiffness (it takes energy to hold antagonistic pairs in mutual tension). The three parameters α, β, and γ are tunable so that different types of muscle can be simulated: some muscles are better at quick exertions, others at providing large forces with little contraction. So far, however, all the virtual muscles in Cog use the same parameters, adjusted to favor static forces over stiffness.

The effect of P on the energy stores is computed via the following algorithm, where P_{avail} is available influx of power obtained from extramuscular metabolic sources, and Δt is the length of one time-step:

- Calculate the required energy $S_{req} = P\Delta t$ and the available metabolic energy $S_{avail} = P_{avail}\Delta t$.
- *Deplete* available source S_{avail} by the required amount:

$$\begin{aligned} S_{req} &= S_{req} - \min(S_{req}, S_{avail}) \\ S_{avail} &= S_{avail} - \min(S_{req}, S_{avail}) \end{aligned}$$

- If $S_{req} > 0$, then *deplete* S_S by remaining required amount.
- If $S_{req} > 0$ still, then *deplete* S_L by the remaining required amount.
- If $S_{avail} > 0$, then *replenish* S_S by the leftover available energy:

$$\begin{aligned} S_S &= S_S + \min(S_{avail}, (S_{S0} - S_S)) \\ S_{avail} &= S_{avail} - \min(S_{avail}, (S_{S0} - S_S)) \end{aligned}$$

- If $S_{avail} > 0$ still, then *replenish* S_L by the leftover available energy.

The energy stores never drop below zero, nor do they exceed their maximum capacities S_{S0} and S_{L0}. The stores only get replenished if the current required power P is less than the available extramuscular influx P_{avail}, and the short-term store is replenished before the long-term store.

The resulting S_L level, relative to its maximum S_{L0}, determines the discomfort signal δ and the efficiency level ϕ for the muscle:

$$\delta = 1 - \left(\frac{S_L}{S_{L0}}\right) \tag{3.1}$$

$$\phi = \left(\frac{S_L}{S_{L0}}\right)^{1/2} \tag{3.2}$$

The discomfort is signalled directly to other processes via a sok out-port, but the fatigue manifests itself solely by its effect on the muscle. ϕ modulates the force, so that the actual force value produced by the muscle is $F_{\text{out}} = \phi F$. As S_L decreases and fatigue increases, the effective muscle force and stiffness drop. Once $S_L = 0$, the muscle is completely exhausted and incapable of producing any force at all.

Figure 3.11 illustrates four stages of muscle exertion. When a muscle is lightly used and $P < P_{avail}$ (a), the muscle is completely unaffected. Once $P > P_{avail}$ (b), the muscle begins to draw energy from the short-term store S_S, which acts as a buffer for short-term exertion. At this point, muscle performance is still unaffected. After $S_S = 0$ (c), the long-term S_L begins to be depleted, and discomfort δ begins to rise and muscle efficiency ϕ begins to fall. Eventually, S_L drops to zero and the muscle abruptly exhausts itself. The muscle will no longer produce any force until P drops below P_{avail} (d) and it is allowed to recuperate.

The relative recovery times for S_S and S_L are proportional to their maximum capacities. S_{L0} is set to 100 times S_{S0}, and P_{avail} set so that full recovery takes approximately 15 minutes. S_{L0} is set roughly so that if the elbow is extended to 90 degrees via a single muscle, that muscle will exhaust itself after two minutes. This limit was chosen because, beyond that, the elbow motors tend to overheat and fail.

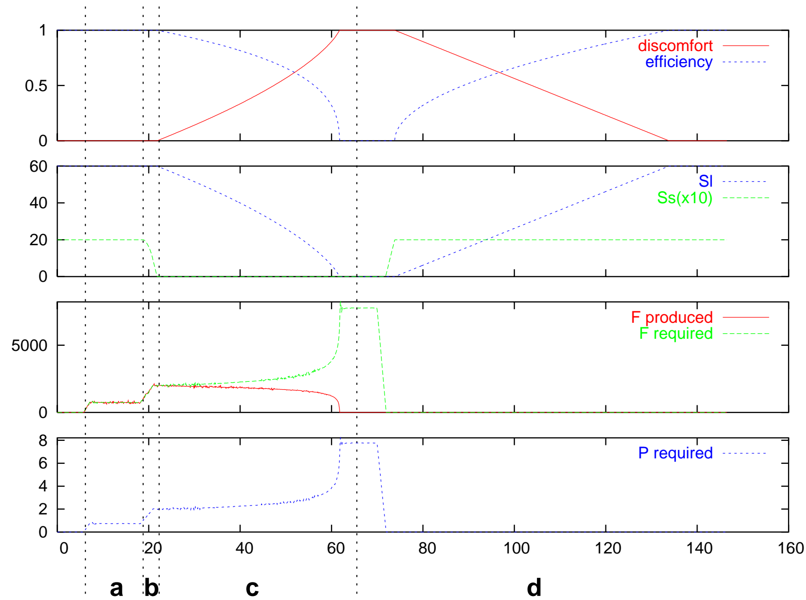


Figure 3.11: An example of the effects of virtual fatigue on a virtual muscle. The graphs show 160 seconds of discomfort δ and efficiency ϕ, short-term S_S and long-term S_L stores, the required force F and the actual force produced, and the required power P. (a) Under light exertion, the required power P is less than the modelled influx P_{avail} from metabolic processes, and the muscle is unaffected. (b) When $P > P_{avail}$, the short-term store S_S begins to be depleted, still with no effect on muscle performance. (c) Once S_S is used up, then the long-term store S_L begins to be depleted, resulting in a decrease in muscle efficiency and an increase in discomfort. At this point, the force produced by the muscle begins to diverge from the force required of it. Eventually, S_L is exhausted, and the muscle can no longer produce any force. (d) Once the muscle is allowed to relax and $P < P_{avail}$ again, it begins to recuperate. The short-term store is replenished first, followed by the long-term store.

Chapter 4

Touch and Vision

In addition to the torso and two arms, Cog has a hand (on its right arm) and a head. These two body parts provide new senses for the robot. The hand is outfitted with tactile sensors that give a coarse sense of touch. The head has eyes formed of two cameras each, the basis of a primitive vision system (by human standards). I am deeply indebted to my colleagues Giorgio Metta and Paul Fitzpatrick, who developed the head controller and much of the vision system.

This chapter describes the hand and the vision system as it is used in this project.

4.1 The Hand and Touch

Cog has a single, right hand[1] (Figure 4.1); its mechanism was sketched out by committee, but was fully designed by Aaron Edsinger, a fellow graduate student, and fabricated by Aaron and myself. The hand has three digits — thumb, finger, and "paddle" — actuated by only two motors. The thumb and finger are linked mechanically and move together to produce a pinching action.

Weight was a critical factor in the hand design, since the hand is mounted at the very end of the arm. The best way to reduce weight was to reduce the number of motors. This three-digit design was chosen as the minimal arrangement which could produce a grasping motion as well as a distinctly identifiable pointing gesture. Pointing is a key communicative gesture, and was desired for a parallel project [45].

[1] The left hand exists as a box of parts; it has never been put together.

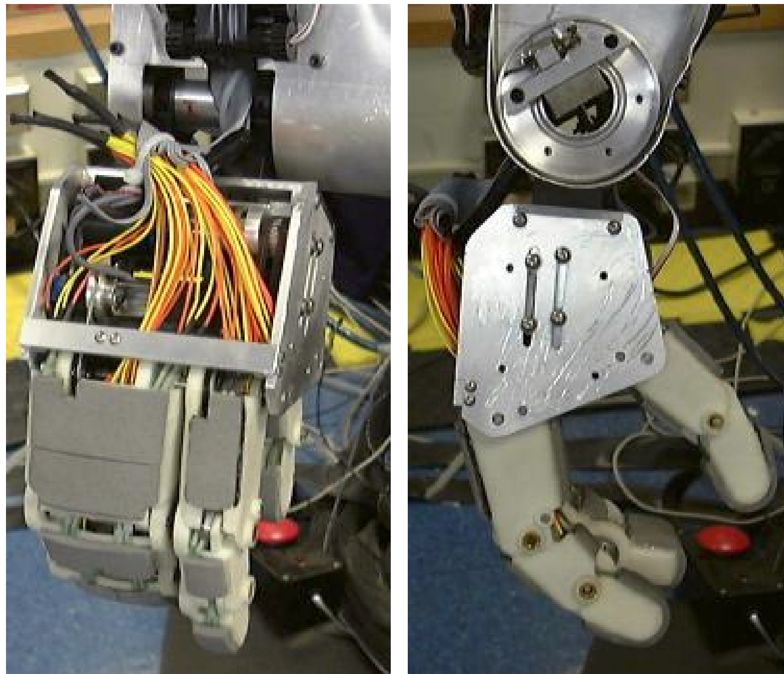


Figure 4.1: Cog's right hand, shown mounted on the right arm. It comprises three digits — finger, thumb, and paddle — but is actuated by only two torque-controlled motors. The thumb and finger are driven simultaneously by a single motor.

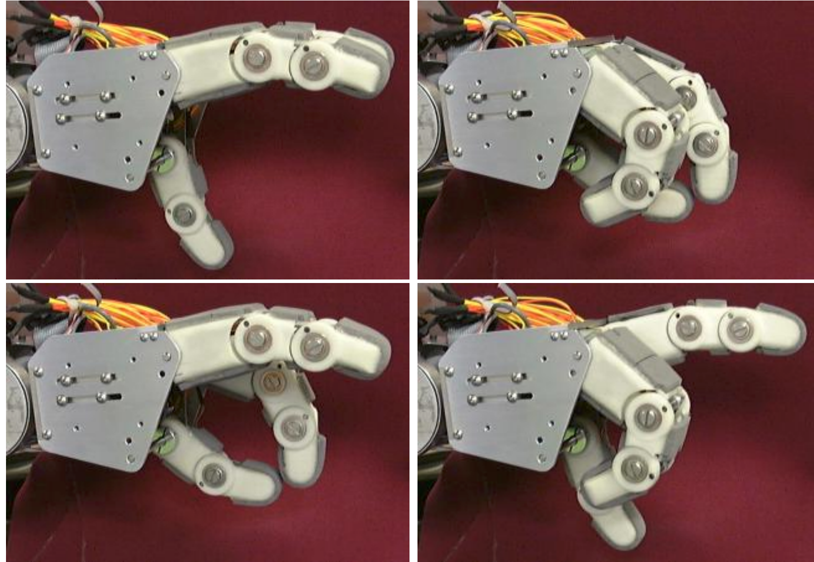


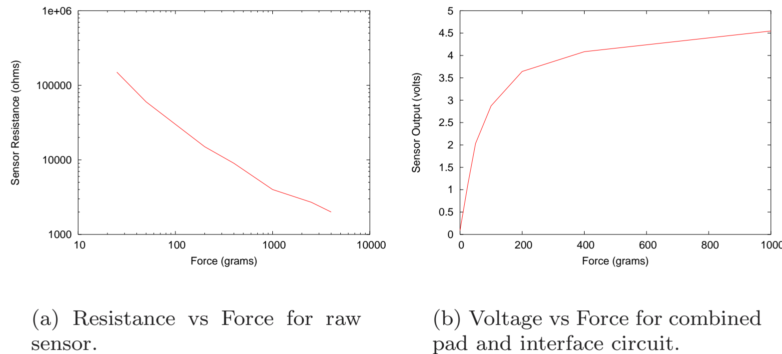
Figure 4.2: The four primary gestures of the hand: a) reaching, b) grasping, c) pinching, and d) pointing.

Like the arms and torso, the hand is driven by series elastic actuators and is torque-controlled. It also has absolute position feedback via small potentiometers installed at the actuator output. Driving the actuators to combinations of their position limits yields four primary gestures: pointing, grasping, pinching, and reaching (Figure 4.2).

Tactile Sense

The hand is equipped with tactile sensors to provide the robot with a sense of touch. The sensors are small force-sensitive resistor (FSR) pads which, as the name suggests, respond to a change in pressure by a change in resistance (Figure 4.3). The pads are covered with a thin layer of high-density foam, which both protects them and makes their mechanical response more uniform over a wider physical area. The foam also helps the hand grip objects.

Twenty-two pads are installed altogether (Figure 4.4). They are wired, however, to yield six tactile signals, one for each inner and outer surface of each digit. Six tactile signals are hardly enough to perform any dextrous manipulation, but they are plenty for the robot to tell if it is bumping against an object or holding something large in its hand.



(a) Resistance vs Force for raw sensor.

(b) Voltage vs Force for combined pad and interface circuit.

Figure 4.3: Response curves of the tactile sensors. The interface circuit is tuned so that the sensors produce the sharpest response in the region of low forces which are experienced when the robot touches objects and people.

In this project, the tactile sense is primarily used as a reinforcement signal for learning behaviors.

The analog signals from the tactile sensors are digitized by the same A/D hardware used for the joint angle sensors and made available to the system via the `glue/uei` module (Section 3.2). The signals are sampled at 50 Hz. The final form of the signal is a value ranging from 0 to 1, normalized and clipped between adaptive minimum and maximum values. The maximum value is simply taken as the maximum observed filtered signal. The minimum value tracks the filtered signal value with a one-second time constant on rising transitions and no delay on falling. This allows the filter to compensate for drift in the zero-offset of the sensors. Furthermore, it makes the tactile sense habituate to stimuli over a short time period.

4.2 The Head and Vision

The vision software on Cog, which controls both the head and cameras, was designed by Giorgio Metta and Paul Fitzpatrick [19] and based on earlier work by Brian Scasselatti [44]. This section gives a brief explanation of the significant features of that system and describes how it is put to use in my own work. Figure 4.5 summarizes the entire system; each subsection is described below.

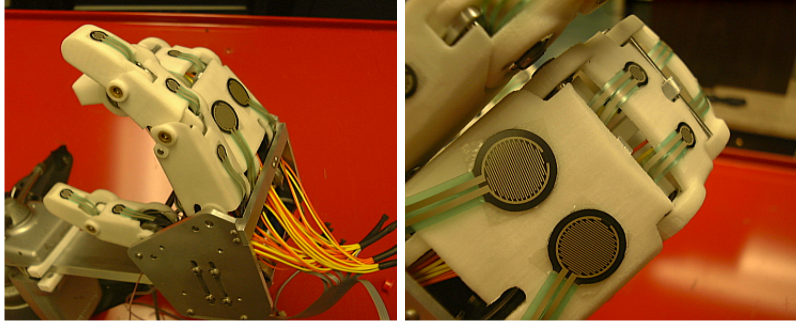


Figure 4.4: Detail of the FSR sensors installed on the hand, shown before the layer of protective foam was applied. The commercially-produced sensors consist of conductive thin-film electrodes affixed to a conductive rubber substrate. Compressing the substrate increases the density of conductive particles and thus lowers the resistivity.

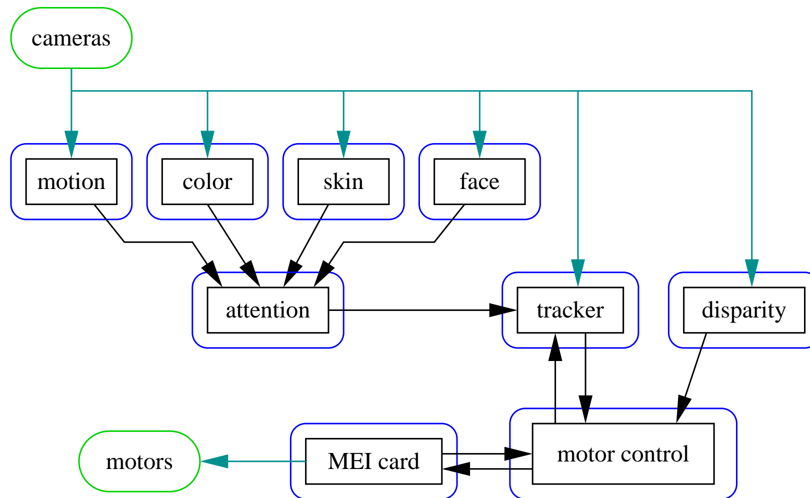


Figure 4.5: Outline of the vision system. Black boxes indicate processes; blue boxes indicate separate processor nodes. Analog video is fed to a framegrabber on each processor which needs it. Most processes only need the right wide-angle view. The disparity (stereopsis) module uses both left and right cameras.

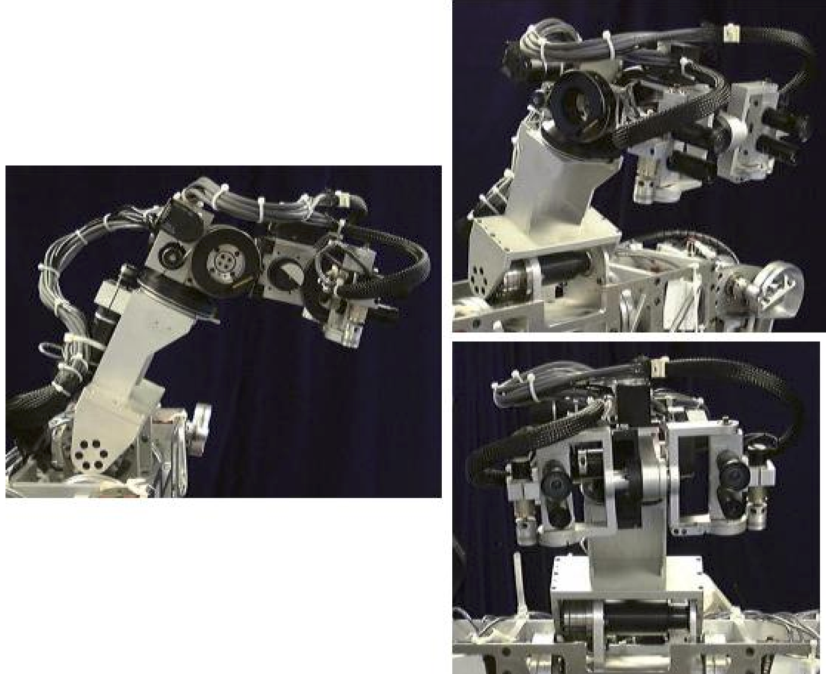


Figure 4.6: Cog's head, viewed from three angles. The eyes have three degrees of freedom: shared tilt and individual pan. The head/neck has four degrees of freedom: pan, tilt, roll, and a "lean". The head motors are position-controlled, using optical encoders to measure the position of each joint. An electronic gyroscope, measuring inclination and angular velocity, is mounted on the head, between and just behind the eyes.

4.2.1 Motor System

Cog's head has a total of seven degrees-of-freedom. Three belong to the eyes, which have independent pan control and a single tilt actuator. The remaining four belong to the head itself: pan, tilt, roll, and a "forward lean" axis (Figure 4.6). The axes are actuated by position-controlled motors, using optical encoders for accurate position feedback. Some axes (e.g. head roll and tilt) are differentially driven; they are controlled by the combined action of two motors. The head is also equipped with an InterSense electronic gyroscope which measures inclination (tilt with respect to gravity) and angular velocity.

Like the arm and torso, the head motors are driven by Copley amplifiers and an MEI motion control card; however, the MEI card operates in a position-feedback mode. The lowest level of motor control is via command of velocities for each of the seven joints. The actuators have no torque sensors, so torque feedback is impossible, and control of the head cannot be fully integrated into meso. In other words, there can be no virtual muscles which couple the neck with the torso or arms. However, since the arms and torso are also controlled via velocities (of virtual muscles), similar high-level controllers could be used to drive all three. The head has, of course, no tunable stiffness parameters ($\vec{K}$); it is under accurate position-control and is always very stiff.

The head has two default motor reflexes: fixating both eyes on the same target (vergence, discussed in the next section), and keeping the eyes centered within their range of motion. When the gaze (with respect to a central point-of-view) is not centered with respect to the head, the head turns while simultaneously counter-rotating the eyes. If the eyes are focused on some target in the robot's periphery, the head will, after a short delay, begin turning in the same direction. The eyes rotate in the opposite direction, so that gaze remains fixed on the target. Once the head is pointing at the target, and the eyes are centered with respect to the head, movement stops.

The gaze direction is stabilized by a combination of feed-forward control (using the pre-computed kinematics of the head) and gyroscopic feedback. Thus, the gaze angle is maintained even if the head is moved externally, by motion of the torso. This is essentially an implementation of the human vestibular ocular reflex (VOR) [39].

Although the head motor system can be directly controlled by commanding velocities for all seven joints, typically only the eye velocities are specified. The centering reflex then moves the rest of the head in a smooth natural-looking manner as it responds to the movement of the eyes.

4.2.2 Image Processing

Cog has two eyes, and each eye has two color NTSC cameras. One camera has a wide-angle (120°) lens to provide full peripheral vision. The other has a narrow-angle (15°) lens to provide a higher resolution in the center of the field of view, much like the human eye's fovea. The four cameras are genlocked together (synchronized to a common timebase).

The video streams are digitized by Imagenation PXC-1000 PCI framegrabbers. Multiple processors in separate computing nodes work

on the same stream simultaneously (performing different operations). Each such node has its own framegrabber(s); the analog camera signals are multiplexed via distribution amplifiers to each node as required. This is far more efficient (vis-à-vis network bandwidth) than digitizing each camera stream once and piping the video around in digital form. Those nodes which do need to exchange processed streams have direct point-to-point full duplex 100base-T ethernet links.

Video streams are captured in 8-bit per channel $R'G'B'$ encoding, typically sampled at a resolution of 128×128 non-square pixels. Streams are non-interlaced; one field is simply discarded. Much of the image processing is also performed using a log-polar representation [17]. This is a lossy transform which reduces bandwidth by reducing the image resolution at the periphery of the frame while maintaining resolution at the center. A 16 kB rectilinear frame (128×128) requires only 8 kB in log-polar form, a factor of two reduction in framesize (and thus, processor cycles) for an image with little loss of utility. With these optimizations, most of the image processing on Cog is able to run at the full frame rate of 30 frames per second.

4.2.3 Vergence

The vergence system mentioned earlier computes the visual disparity (measured in pixels) between the left and right wide camera images. Using the known kinematics of the eyes and head, the pixel disparity is transformed into a corrective velocity for the left eye and sent to the motor control system. Cog is thus a right-eye dominant robot; the left eye attempts to follow what the right eye is focused on. The disparity measure and the differential pan angle of the two eyes together provide the robot with a measure of the depth (distance) of the target.

The vergence control loop runs below full framerate, at 20 frames per second.

4.2.4 Saliency and Attention

The vision system focuses on one target at a time, and this target is chosen by an attentional mechanism. Several filters run in parallel over the video stream of the right wide-angle camera, appraising each frame for certain *salient* features. The attention system weighs the opinions of these filters and decides on the image coordinates of the region yielding the greatest total saliency. These coordinates are updated and sent to the tracking system at frame rate. When the tracker decides to switch to a new target, it uses the last coordinates it received.

Figure 4.7: Saliency and attention processing while looking at a walking person. One frame each from the three saliency filters (vividness, skin tone, and motion) and the attention system is shown. The vividness filter picks out parts of the blue floor. The skin tone filter picks out the person's arm and hair, a door, and the couch. The motion detector highlights picks out two targets on the person's body. The regions chosen by the filters are weighted together by the attention mechanism, which favors the motion detector.

The attention mechanism currently uses three filters. Two come from the original Fitzpatrick/Metta vision system: for every video frame, each filter generates a list of up to five bounding boxes which describe the image regions it considers most salient. The third filter, a motion detector, was designed by me; instead of bounding boxes, it outputs a list of up to five bounding circles (center coordinates and radius). Figures 4.7 and 4.8 illustrate the output of the filters and the attention mechanism in two different visual scenarios.

The first filter distinguishes vivid colors; it describes regions with high chroma content. This makes Cog sensitive to the brightly colored toys which we researchers often wave in its face. The second filter is a skin tone detector. Regardless of race, skin tones generally fit the constraint (for R', G', and B' values ranging from 0 to 255):

$$\begin{array}{ccccc} 1.05G' & < & R' & < & 2.0G' \\ 0.9B' & < & R' & < & 2.0B' \\ 20 & < & R' & < & 250 \end{array}$$

The "skin-tonedness" of pixels satisfying that constraint is estimated via the formula [6]:

$$\begin{aligned} S \;=\; & 2.5(0.000968R'^2 + 0.217R' - 0.000501G'^2 - 0.364G' \\ & -0.00287B'^2 + 0.906B' - 50.1), \end{aligned}$$

which is clipped to the range $[0, 255]$. (Pixels which do not fit the $R'G'B'$ constraint are assigned zero.) A region-growing algorithm is

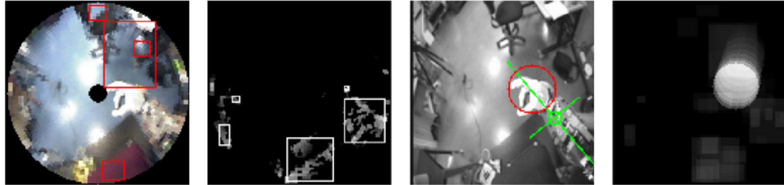


Figure 4.8: Saliency and attention processing while looking at the moving arm. One frame each from the three saliency filters (vividness, skin tone, and motion) and the attention system is shown. The vividness filter picks out a portion of the blue floor and the red fabric on the robot. The skin tone filter also picks out the fabric, and part of the robot arm. The motion detector highlights a circular region containing the robot's hand. The regions chosen by the filters are weighted together by the attention mechanism, which favors the motion detector.

run over the image of S pixels to yield bounding boxes around skin-toned portions of the frame. This filter makes Cog's eyes sensitive to people, particularly their heads and hands. Unfortunately it also makes Cog sensitive to wooden doors and furniture and cream-colored walls.

The third filter is a simple motion detector based on inter-frame image differencing. Each new frame is subtracted from the previous one, and then the difference image is thresholded, dilated, and tagged using an 8-connected region-growing algorithm. The orientation of each region is calculated, along with the extent along the major and minor axes; this provides a description of each region by a rotated bounding box. A salient disc for each region is chosen by picking the center point halfway between the centroid and the maximum extent of the major axis, in the direction closest to the upper left-hand corner of the screen, with radius equal to the minor axis. This choice of position tends to pick out both the heads or upper bodies of people moving around the scene as well as the hand of the right arm when it moves into the field of view. Note that the motion detection algorithm is useless when the eyes are moving. Fortunately, the tracking system is implemented in such a way that the eyes and head are stationary for a few moments before a new target is acquired. The motion detector is gated so that if it registers too much motion (e.g. a significant fraction of the scene), it decides that the camera has moved and suppresses its output.

The attention system maintains a 128×128 saliency map; each value in this map represents the saliency of the corresponding pixel in the video stream. With each new frame, the system multiplies the current

map by a decay factor δ. It then adds the weighted bounding boxes or discs from each filter to the map. That is to say, if location (x, y) is contained within a bounding box returned by a filter, and the weight assigned to that filter is w, then $w(1-\delta)$ is added to the value at location (x, y) in the saliency map. δ determines the persistence of saliency over time. After the saliency map is updated, the location with the highest saliency is determined, and these coordinates are output as the new center of attention.

4.2.5 Tracking

A separate tracking module is used to consistently follow a visual target as it moves around in the field of vision. The tracker receives the initial target coordinates from the attention system. It then records a small 6x6 image patch centered around those coordinates, and this becomes the target. Within each succeeding video frame, the tracker searches for the target patch and outputs its new location.

The tracker anticipates the new location based on the last location and knowledge of the motion of the head and eyes. Starting at that initial position, it searches around in a 10×10 region using straightforward correlation until it finds the best match. The tracker then records the matching region as the new target patch. If the tracker cannot find a good enough match, it decides that the target has been lost, and it then grabs a new target as directed by the attention system. The robot never becomes hopelessly fixated on one target because the tracker is not that good; in an active, moving world, a new target will usually "get its attention" every few seconds or so.

The tracker sends the retinotopic coordinates of the target to the eye motor control system, which then moves the eyes to try to center the gaze on the target. Thus, the eyes will follow the target, and since the head control attempts to keep the eyes centered in their range of motion, the head will follow as well.

4.2.6 Vision System as a Black Box

Much of the vision system is built with an interprocess communications protocol called YARP,[2] which is not particularly compatible with the ports and connections created by sok. So, the vision system is (for this project) interfaced to the rest of pamet via bridge modules which speak both YARP and sok. This creates an opaque interface which hides all the vision modules and their interconnections.

[2] YARP was designed by Paul Fitzpatrick.

The vision system internally uses several coordinate frames: joint angle, retinotopic, and gaze angle in the "world" coordinate frame. The `sok` interface uses the world coordinates exclusively. Everything visual is expressed in terms of 2-d gaze angle relative to the coordinate frame of the shoulders, i.e. the mounting point of the base of the head.

A `sok` module named `eye/tracker` provides an interface to the tracker. Its single outport streams the position of the target, expressed as gaze angle and distance. The gaze angle is computed in world coordinates from the eye/head joint angles and the target's retinotopic position. The retinotopic-to-gaze transform is increasingly inaccurate as the target moves off-center, but since the tracking system is always trying to re-center the target, this is not a problem in practice. The target distance is estimated from the differential angle of the eyes, which is actively controlled by the disparity feedback loop.

A module named `eye/saliency` provides an interface to the visual filters of the attention system. It has one outport for each filter and simply forwards the bounding box lists which each produces, after converting the retinotopic coordinates into gaze angles. As mentioned above, this transform is not completely accurate, but it is good enough to give a rough estimate of salient areas — which is all the filters provide anyway.

Finally, a module named `eye/features` provides a feature vector describing the target region being tracked by the attention tracker. This vector has six parameters: red, green, blue, vividness, skin-tone, and motion — each ranging from 0 to 1. The values are computed using similar algorithms as the saliency filters but are averaged over a 16×16 region of pixels centered over the tracker target.

Chapter 5

pamet

pamet is the system in Cog which does the "thinking" (as much as you can call it that). It is a collection of modules (sok-processes) which look for patterns in data flow, generate movements, respond to reward and punishment, discover useful actions, and recognize and respond to sensory stimuli. The modules are roughly divided into those which *do something*, those which *make models* of what is being done, and those which *generate instances* of the other two. The classes of modules are distinguished primarily by the types of inputs and outputs they have.

pamet is a dynamic system: modules are created and destroyed as time goes on. In general, the creation of a module indicates a hypothesis that there is something to observe or to do which is potentially useful. The same module may later be destroyed (reclaimed by the Great Bit Bucket) due to disuse; this indicates that the hypothesis has proven false.

pamet is also a distributed system: all modules run as separate processes on a network of processors. This system could have been implemented as a giant monolithic time-stepped process — in Matlab (or, preferably, Octave) no less. In many ways, that would have been substantially simpler to build and debug. It would probably even work for simulation. However, two of the challenges of this project were to create a system which runs a real, physical robot, and to create a system which can scale up to many, many more modules. In this distributed implementation, the learning and analysis modules can use cycles on isolated processors without interfering with the real-time behavior of the action modules. As the system develops and more modules are created and more processing cycles and memory are required, additional processors can be added to the robot *as it is running* — the processing

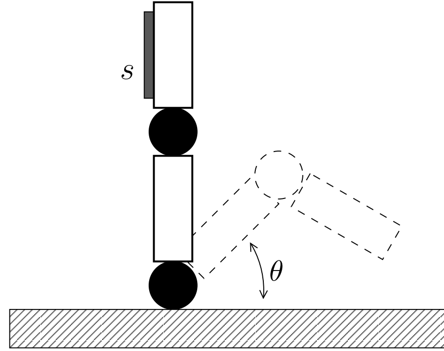


Figure 5.1: A toy robot. It is just a finger with a single actuator and a single tactile sensor. Its position is fully determined by the single value θ; the sensor yields a single value s.

hardware and operating system were chosen such that this is possible. The system is far from perfect, though, and a number of snags and bottlenecks exist which would need to be resolved before "infinite scalability" is possible. However, adhering to these goals has made the resulting system much more real than it would have been otherwise.

5.1 A Toy Example: The Finger Robot

To get a feel for how pamet is organized, let's look at a toy implementation. Suppose that Cog is a very simple robot with only one working actuator: a finger, which has a full one-dimensional repertoire of expression, from curled-up to pointing (Figure 5.1). This finger has tactile sensors on it, and the robot's primitive emotional system is hard-coded to signal pleasure in response to any tactile stimulus. Movement of the finger is controlled under meso by a single velocity-controlled virtual muscle, and the configuration of the finger is defined by a single angular position (also available from meso).

5.1.1 Learning to Move

The lowest level of pamet's control of the robot will be two *mover* modules, hard-coded by us, the roboticists. A mover module is very simple; it does nothing until it receives a positive scalar *activation* signal from some other process. While it is thus activated, it sends a fixed stiffness vector and a scaled velocity vector to meso. Thus, a mover

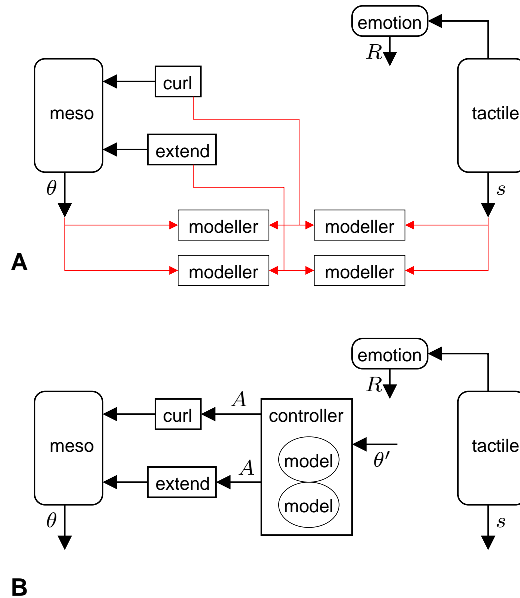


Figure 5.2: The robot has two movers, one for curling and one for extension. (A) Each is observed by mover modellers, to see what effect, if any, they have on θ and s. (B) Two models relating the movers to θ are learned, and a controller for the velocity θ' is created.

module makes the robot move by driving a subset of the muscles with some profile of velocities. Our toy finger robot will have two movers: one for curling the finger, and one for extending it (Figure 5.2(A)).

The movers are not actually completely subservient to external activation; they also have internal *random activation*. At random times, a mover will activate itself for a random duration and with a random activation level. The rate of random activation is initialized to some maximum value when the mover is created. As the mover is activated more and more by external sources, this rate of internal activation decreases.

The robot's first order of business is to discover what these two movers do. The robot has two *state parameters*, the tactile sensor value and the finger position value, which are assumed to be independent. pamet's *proto-mover-modeller* will spawn four *mover-modeller* modules, one for each combination of state parameter and mover.

Each mover-modeller analyzes the interaction of its assigned state parameter and mover module. It observes and records data from both

and then tries to fit a linear model of the state parameter's velocity as a function of the mover's activation. Essentially, it tries to determine if the mover has a direct causal effect on the state parameter. In our toy robot, we would expect such an effect to exist between the movers and the finger position, but not between the movers and the tactile signal.

If no model can be fit after some length of time, the mover-modeller simply gives up and exits gracefully. If the mover-modeller does discover a model, it records it in pamet's *model registry*.

Another module, the *proto-controller*, monitors the registry for mover models. It scans the registry and groups together all mover models which affect the same state parameter. It then spawns a *controller* module, which loads the models and connects itself to the relevant movers (Figure 5.2(B)).

The controller module is an open-loop velocity controller for the given state parameter. When it receives a velocity command on its inport, it activates the mover which will most closely generate such a velocity in the state parameter. The existence of a controller module for a state parameter means that it has become a *controllable parameter*.

Once we turn on our toy robot, it will begin wiggling its finger, due to the random activation of its movers. The mover-modellers will observe this and create models relating the finger position to the mover activation. Finally, a single controller will be spawned, which can drive finger velocity by activating movers. The finger position has become a controllable parameter, and the robot has effectively learned how to move its finger.

5.1.2 Learning What to Do

At this point, the robot is just occasionally wiggling its finger back and forth. We want it to learn a trick: "point the finger straight out when touched". The first thing the robot needs to learn is "point the finger straight out".

When the finger controller module is created, pamet's *proto-action-modeller* will spawn an *action-modeller* which will observe the controllable parameter (finger position) and the robot's emotional state. It will try to create an *action model* which correlates the finger position with positive changes in emotional state (Figure 5.3(A)).

Squeezing the finger will cause a tactile sensation which is hard-coded as a pleasurable response. So, we sit with the robot, and whenever it randomly happens to point its finger straight, we reward it by squeezing the finger. The action-modeller will observe this and create a model which essentially says that reward is expected when the finger

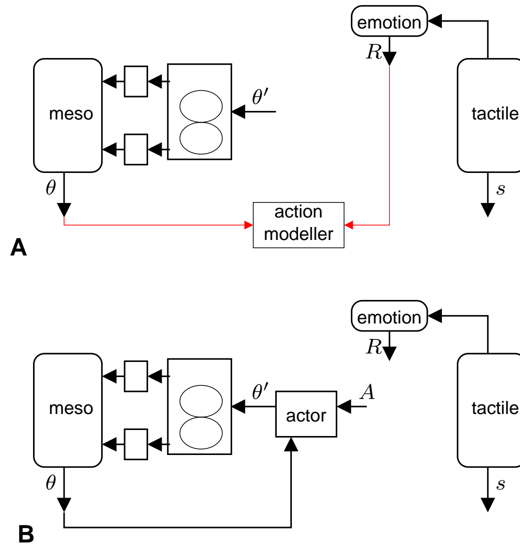


Figure 5.3: (A) An action modeller is generated to find rewarding configurations of θ. (B) After training, an actor module is created. When activated, it drives θ to a prototypical value θ_0 via the controller.

is pointing straight.

The action-modeller will register this model and then spawn an *actor* module (Figure 5.3(B)). The actor has an activation inport, an inport to monitor its controlled parameter, and a velocity outport connected to the controller. Like a mover, an actor is silent until it receives an activation signal. Once activated, it drives the controllable parameter to its most rewarding value (as encoded in the action model) and tries to hold it there as long as the activation persists. An actor also has an internal random activation rate, which decays over the life of the actor.

So, after a session of rewarding our robot when it randomly straightens and points its finger, it will create an actor module which, at random, *explicitly* points its finger. The robot has learned that pointing is a good thing to do.

5.1.3 Learning When to Do It

The robot now knows that pointing its finger can be a good thing to do, but it doesn't know *when* to do it. The next thing the robot needs

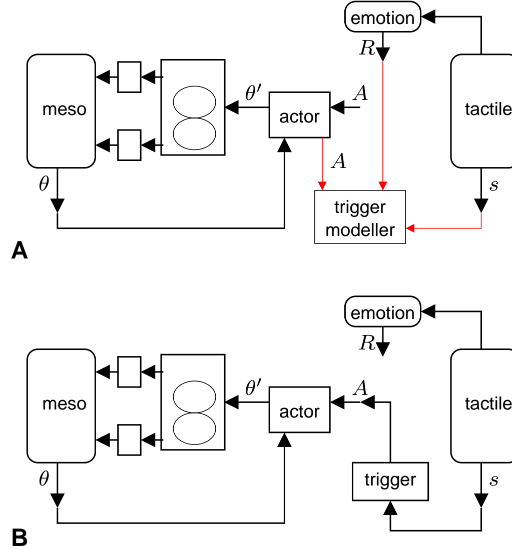


Figure 5.4: (A) A trigger modeller is created to discover rewarding conditions (values of s) in which to activate the action. (B) After training, a trigger module is created, which activates the action when the sensor value s is close enough to a prototype s_0.

to learn is an appropriate *trigger* for which pointing is the appropriate response.

When the pointing actor is created, pamet's *proto-trigger-modeller* will spawn one or more *trigger-modellers* (Figure 5.4(A)). Each of these will observe different state parameters and will try to correlate them with both the robot's emotional state and the activation of the pointing actor. The different state parameters can be sensor values, motor values, or even the activation levels of other modules in the system.

Suppose we want to train the robot to point when it gets a tap on the finger. When the robot starts pointing, we tap its finger, and then we give it a big squeeze as a reward. The trigger-modeller assigned to monitor the tactile sensor will develop a *trigger model* which shows that a tap to the finger, in conjunction with activation of the pointing actor, will reliably precede a positive reward. The modeller will register this model and then spawn a *trigger* module which uses it (Figure 5.4(B)).

The model essentially says that when a certain state context is satisfied (e.g. tactile sensor registers a tap) and an action is performed, then a reward is imminent. The job of the trigger is to put this model to

work. The trigger monitors the tactile sensor, and when a tap occurs, it will activate the pointing actor. Thus, we have successfully trained our toy robot to point its finger when we tap it.

5.2 Names and Data Types

All of the modules which make up pamet are implemented as sok-processes; data is passed between these modules via messages sent along connections between inports and outports. Data streams typically fall into one of three categories: state parameters, activation values, and velocity drive values.

State parameters ($\vec{s}$) are continuous streams of fixed-size vectors, such as the 17-dof vector of joint angles on the robot. Such data are samples of the state of some continuous process on the robot. Data is sent at a fixed frequency which is a compromise between temporal resolution and data rates. Most velocity-controlled motor data flows at 50 Hz; the motor control of meso operates at 500Hz; vision data (e.g. visual feature vector) is frame-rate limited to 30 Hz.

Activation values (A) are intermittent streams of scalar samples which activate a receiving process. When "off", no data is transmitted. When "on", messages are sent at a fixed rate (typically 50 Hz). The receiving process remains active as long as a continuous stream of data comes in. Although the values of the data itself are generally ignored by the receiver, it usually reflects the probability behind the decision to produce the activation.

Some state parameters (e.g. joint angles) can be actively controlled. Each such parameter will have an associated inport somewhere in the system for a drive value consisting of a velocity vector ($\vec{v}$). The state parameter will be driven with the commanded velocity as long as a continuous stream of commands is received on the port.

pamet is a dynamic system; ideally, all of these various parameters are discovered, connected, and modelled automatically. This is accomplished primarily by conventions for naming the ports. Regardless of the sok-process names, all outports which export a state parameter have names of the form `STATE.abc`, where the "abc" is an optional label for any human observers. Likewise, activation inports are labeled `ACT.abc`. Modules which have an activation input usually have an activation output as well, which broadcasts the activation state of the module to observers; these are named `AOUT.abc`. Velocity drive inports are named `VEL.abc`.

The various modules themselves also have canonical names. Dy-

namically-generated modules have names of the form _ABC/0xNNNNNNNN, where _ABC is a prefix determined by the type of module and the suffix is the hexadecimal hash value of an automatically-generated string which describes what the module does. These hash values are not particularly human-readable, but they are at least consistent. Modules created in successive runs or experiments, but which serve the same function, will keep the same name.[1]

Models and other metadata are stored in flat-file databases on the filesystem. Such data is only necessary for the development of the system (creating new models and modules and connecting them), not for real-time functioning at any particular stage of development. Both the filesystem and various support daemons (such as the sok locator) can go down, but Cog will continue to function. It won't be able to get any smarter, but it won't get any dumber, either.

5.3 A Menagerie of Modules and Models

pamet as a framework has a cyclic nature. In general, for every module that generates activity, there is a model of that activity, and in turn a module that creates and updates that model. And for every model, there is another module which uses it to create a new activity. Each model acts as an abstraction barrier, taking a nebulous set of data and statistics and packaging it into a discrete, tangible entity.

As illustrated in Table 5.1, there are 5 basic classes of *activity* modules: movers, controllers, actors, triggers, and transformers. Each one has a notion of "activation"; these modules sit quietly until input from another module (or random input from within) causes them to wake up and perform a stereotyped function. Associated with these are a variety of models: mover, position-constant action, position-parameter action, velocity-constant action, position trigger, delay trigger, etc. Each type of model has a *modeller* module which creates and manages instances of the model.

This set of classes has developed as the minimal set necessary for the robot to learn the complete task of "pointing at some visible target". Originally, I began this project envisioning a complete, self-contained, "universal" set of modules and models which could be endlessly combined and connected to enable the robot to learn anything that a human could. In retrospect, that universal set might be extremely large. The

[1] After a while, you look at the list of running processes and just *know* that _PCMod/0x2F788AD2 is the position-constant action modeller observing the right arm joint angle vector. It's a bit like "reading the Matrix".

Class	*Inputs*	*Outputs*	*Instance*	*Modeller*
mover	A		`_MV/`	`_MvMOD/`
controller	$\vec{v}$	A	`_CTRL/`	
actor				
position-constant	A, $\vec{s}$	$\vec{v}$	`_PCA/`	`_PCMod/`
position-parameter	A, $\vec{s}_1$, $\vec{s}_2$	"	`_PPA/`	`_PPMod/`
velocity-constant	A, $\vec{s}$	"	`_VCA/`	`_VCMod/`
trigger				
position	$\vec{s}$	A	`_PTG/`	`_PTMod/`
activation-delay	A_1, A_2	A	`_ATG/`	`_ATMod/`
transformer	$\vec{s}_1, \vec{v}_2$	$\vec{s}_2, \vec{v}_1, A$	`_XFM/`	`_XFMMod/`

Table 5.1: Classes of modules implemented in pamet; most classes include a modeller module for creating models, and an instance module for using the models. Each class is distinguished by the type of data it consumes and produces. A is an activation signal; $\vec{s}$ is a state parameter vector; $\vec{v}$ is the velocity of a state parameter.

module/model learning cycle works because each class encompasses a very focused and specific type of knowledge.

The rest of this chapter describes the various instance/activity modules. Most modules are associated with some type of model; these models are discussed in Chapter 6.

5.3.1 Movers

A *mover* module is, in the simplest terms, a module that causes something to move when it is activated. It has a single scalar input which determines the rate or intensity of the movement. In particular, *meso movers* are the basic interface between pamet and the virtual muscles provided by meso. When inactive (not receiving an activation input), meso movers do nothing. When active, they send a hard-coded vector of velocities and stiffness values to meso to activate a certain combination of virtual muscles. For example, the `_MV/hand/right-grasp` mover will make the fingers of the right hand contract (without affecting any other actuators).

Movers have internal *random activation*, which causes them to turn on spontaneously. This is roughly a Poisson process, set by a baseline average activation rate. This base rate will decrease over time as more and more explicit external activation is received. The idea is that the

mover starts out firing spontaneously to facilitate an exploratory phase of activity. As the behavior and effects of the mover are modelled and other modules begin to explicitly use the mover to produce actions, then the spontaneous firing is inhibited.

The meso mover modules are essentially a well-justified hack. They were created as a way to bootstrap the learning process, to get the robot to simply *move* before it has learned any reasons to move. The complete set is hand-crafted to reflect a repertoire of basic two- or three-actuator motion combinations. They are further described in Section 7.1.

When the robot is first turned on (i.e. early on in a run of Cog's "development cycle"), *mover modellers* are spawned by a *proto-mover-modeller* module. Each mover modeller attempts to find a reliable relationship between a particular mover's activation level and some state vector, e.g. the joint angles of the robot. For each such relationship found, a *mover model* is created. These models are used to create *controllers*, described next.

5.3.2 Controllers

A *controller* is a module which multiplexes a group of movers together to control the velocity of a state parameter. That parameter then becomes a *controllable parameter.* Controllers are created dynamically depending on the available mover models. Recall that each mover model characterizes a mover as influencing a particular state parameter (e.g. a subset of the robot's joint angles). The set of all mover models is partitioned into groups such that models within the same group have overlapping influence (intersecting subsets), and models in separate groups are independent. For each group, a controller is created, to which those models are assigned. Each controller registers itself as capable of controlling a state parameter which is the union of all parameters influenced by its group of movers.

The basic structure of a controller is shown in Figure 5.5. It has single *velocity command* input $\vec{v}$, and one drive output A_i for each mover under its command. A controller is an activatable module; it sits quietly until it receives a stream of velocity commands. It then activates the single mover m which best matches $\vec{v}$ according to the criterion

$$m = \arg\max(\frac{\vec{v} \cdot \vec{v}_m}{\|\vec{v}_m\|})$$

where $\vec{v}_m$ is the state velocity produced by mover m (given by the

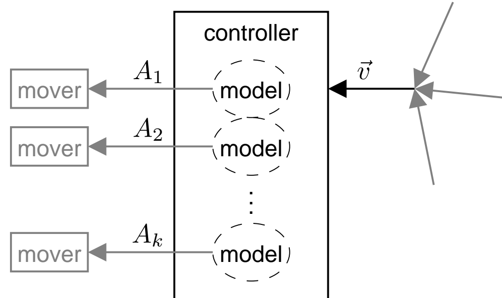


Figure 5.5: A *controller* module, which receives a velocity command $\vec{v}$ at its inport. Using the mover models it has loaded, the controller decides which of the connected mover modules to activate, via drive outports A_i.

mover model). The activation sent to m is simply:

$$A_m = \frac{\|\vec{v}_m\|}{\vec{v} \cdot \vec{v}_m}.$$

The other drive outports remain silent. (Although, when an outport is going to become silent, the last value sent is a 0.0; this is just to aid in diagnosing the system.)

Note that many modules can connect to a controller's $\vec{v}$ command inport. To resolve the possible contention due to multiple modules trying to command different velocities simultaneously, this inport is equipped with a *GrabArbitrator*. When the $\vec{v}$ port receives a message and becomes active, the source process of that message receives control. All other message sources are ignored until the controlling source stops transmitting (i..e. fails to send another message within a timeout period of ˜60 ms).

5.3.3 Actors

Each instantiated controller module gives pamet a knob to twist. *Actors* are the modules which do the twisting, and *action models* describe what twisting they do. An action model is essentially a description of a discrete, primitive action. Two types of action models have been implemented for pamet:

- *position-constant action:* Drive a controllable parameter to a particular constant value.

- *position-parameter action:* Drive a controllable parameter to match a varying input parameter.

Other varieties are envisioned (e.g. "velocity-constant" and "velocity-parameter"); however these are the only two implemented so far.

The impetus for having action models is to simplify learning by distinguishing "what to do" from "when to do it". An action model defines a very simple small behavior and makes it tangible. Action models can be used as building blocks for more complex actions. Action models can be compared with one another.

Action models are created by *action modeller* modules, which are themselves generated by a *proto-action-modeller*. In general, for each type of action, there is one action modeller assigned to each controllable parameter in the system. When new controllable parameters arise (via the birth of a new controller), then new action modellers are spawned as well. An action modeller analyzes the controllable parameter, looking for a correlation with a reward signal. If it discovers a reliable correlation over some short time period (minutes), it creates a model, and spawns an *actor* module for that model. Thereafter, it continues to monitor the model and may refine it. One action modeller may create and manage multiple models, since there may be multiple actions involving a single controllable parameter.

An actor module (Figure 5.6) instantiates its action model: it carries out the action described by the model. Actors are quiet until they are activated by input to the activation inport A. As long as they are kept activated, they will continue trying to achieve whatever action they perform, sending a drive signal out to some controller. Actor modules also have internal *random activation*. When first created, an actor will activate itself spontaneously. This allows the robot to manifest whatever behavior it has just learned so that the behavior may be linked to a stimulus (via a trigger, Section 5.3.4).

The rate of the random activation drops off over time, and eventually an actor will only activate in response to an explicit external signal. If, after some length of time, an actor has not received *any* external activation, then the actor exits and its action model is discarded. One can consider an action model to be a hypothesis of sorts, claiming that a particular controller activity leads to utile, rewarding behavior. If it turns out that nothing ever triggers that action, then pamet decides that the hypothesis is false and the action is "forgotten".

Action models can also evolve over time. When a new action model is first devised by a modeller, it is compared to all the pre-existing models which the modeller is managing. If the new model is similar

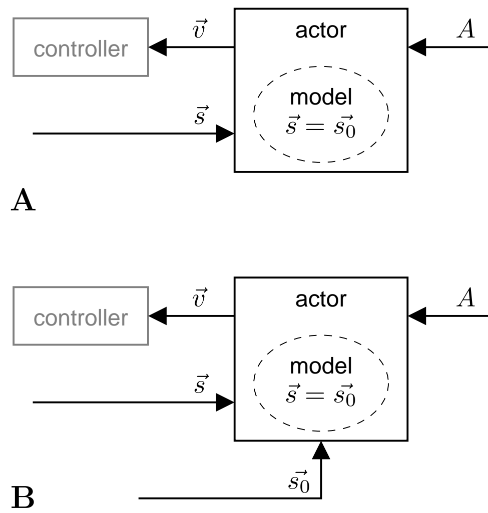


Figure 5.6: Two types of *actor* modules. (A) The position-constant actor has a model which specifies a prototype goal position $\vec{s}_0$ for a state parameter $\vec{s}$. When activated via messages to inport A, the actor sends velocity commands $\vec{v}$ to a controller, to drive $\vec{s}$ to the goal. (B) The position-parameter actor has similar behavior, except that the goal position $\vec{s}_0$ is read from another state parameter instead of being a constant.

enough to an old model, then the old model is refined by blending in the new model. No new actor is spawned (and the new model is never registered), but the actor corresponding to the old model is notified of the update. This mechanism allows actions to be continually shaped by a trainer over time.

The action models themselves are explored in Section 6.2.

5.3.4 Triggers

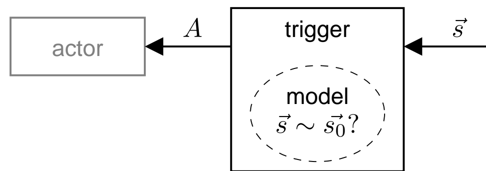
An actor is able to make the robot do some simple thing, but it does not know when to do it. By itself, an actor acts at random. A *trigger* module bridges the gap between cause and effect by explicitly activating an actor in response to a stimulus. For each trigger, the stimulus is described by a *trigger model*, and these models are themselves created by *trigger modellers.*

Trigger models link state to action: they describe a context, and an action (literally an actor module) which should be activated when the context is satisfied. Classes of models are distinguished by what type of context they describe. One type of model has been implemented in pamet, the *position trigger*, in which the triggering context is tied to the value of a state parameter (Figure 5.7): the trigger is activated when the parameter is close to some prototypical value. Another useful model envisioned is an *activation-delay trigger*, which triggers an action after a delay in response to another activation signal in the system. This would allow actions to be chained together into sequences.

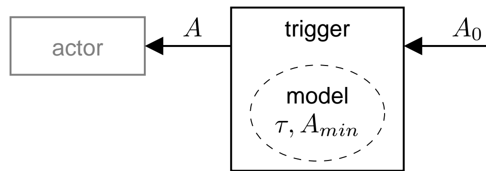
Trigger modellers are created by *proto-trigger-modeller* modules. Whenever a new actor appears in sok space, the prototype module creates a new modeller for every possible stimulus for that actor. Each modeller has inports for monitoring a state vector $\vec{s}$ (the potential stimulus), the activation level A of the actor, and a reward signal R. The basic operation of the modeller is to look for a correlation between the action, the state vector, and the reward; the modeller is trying to determine if the action is consistently rewarded when associated with some stimulus. How this happens is explained in Section 6.3.

Note that the trigger modeller has no idea what the actor is actually *doing*. The modeller only sees whether or not the actor is activated. The internal random activation of the actor insures that it will be occasionally active, at least at the outset.

If the modeller does discover a reliable correlation between action, stimulus, and reward, then it will register a trigger model and spawn a trigger module. The trigger connects itself to the stimulus' state vector and the actor's activation inport. The trigger will activate the actor



A



B

Figure 5.7: Two *trigger* modules. (A) The *position trigger* outputs an activation signal A when the input state parameter $\vec{s}$ is close enough to a prototype $\vec{s}_0$. (B) The proposed *activation-delay trigger* is activated at some time delay τ after an input activation $A > A_{min}$.

whenever (and for as long as) the trigger's context is satisfied. The modeller continues to monitor the action, stimulus, and reward, and may later refine the model, modifying the context.

The activation output of the trigger does more than just activate an actor. The value of that output is the probability that the its context is satisfied by the current state vector $\vec{s}(t)$. This probability is a useful parameter in its own right: it is an evaluation of some state which is deemed relevant to the robot. A trigger's activation output is thus considered a state parameter which may be used as input to other triggers. (The actor itself does not pay attention to the value, just to whether or not any message is being sent at all.)

Like an actor, a trigger is best considered to be a *hypothesis* that a particular stimulus should trigger an action. However, there is currently no mechanism for deciding if the hypothesis is no longer true and retiring a trigger. This issue will be further discussed in Section 6.3.

5.3.5 Transformers

Suppose that two state parameters, $\vec{x}$ and $\vec{y}$, describe the same process, but with different coordinate systems. They are two different views of the same phenomenon. Then, there should be a *coordinate transformation* $\vec{f}$ which maps $\vec{x}$ onto $\vec{y}$. The basis of a *transform model* is to learn such a transformation between two state variables, if it exists. A *transformer* module can then use that model to provide new state variables, classification, and control.

The motivating example behind the transform model is the task of looking at the hand. With respect to Cog's shoulders, the position of its hand is determined by the state parameter $\vec{\theta}$, the vector of the six joint angles in the arm. Recall from Chapter 4 that the target of the vision system is given as $\vec{\gamma} = (\theta, \phi, d)$, the pan, tilt, and distance in a shoulder-centered global-coordinate frame. If the eyes are focused on the hand, then $\vec{\gamma}$ also tells us the position of the hand, simply in a different coordinate system.

Of course, the hand position can also be described by $\vec{r} = (x, y, z)$, the 3-d cartesian coordinates. This could be considered the most fundamental description, since it specifies where the hand is in "real space", and the $\vec{\gamma}$ and $\vec{\theta}$ descriptions can be derived from $\vec{r}$ and knowledge of the head and arm kinematics. However, real space $\vec{r}$ isn't of any immediate consequence to the robot, whereas $\vec{\gamma}$ and $\vec{\theta}$ are. $\vec{\theta}$ is required for any motor activity which involves moving the hand around, and $\vec{\gamma}$ is needed for any sensory activity which involves looking at something.

Whenever Cog is looking at its own hand, $\vec{\theta}$ and $\vec{\gamma}$ relay the same

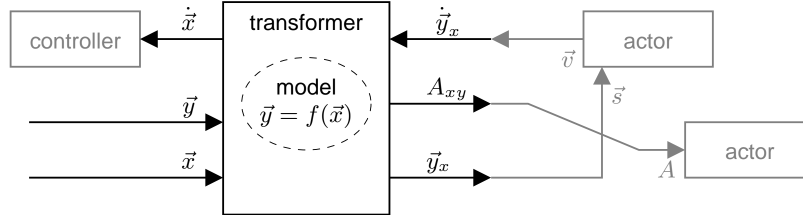


Figure 5.8: General form of a *transformer* module. The module contains a model which describes how the view of a process in one parameter space ($\hat{x}$ space) maps into another ($\hat{y}$ space). The transform of the input $\vec{x}$ is continuously estimated and output as $\vec{y}_x = f(\vec{x})$. The input $\vec{y}$ is used with $\vec{x}$ to continuously assess the probability A_{xy} that the two inputs are indeed tracking the same process. Since the model can perform the inverse transform on $\vec{y}$ velocities, the module can also act as a controller. When it receives a velocity drive signal at input $\dot{\vec{y}}_x$, it transforms that into a drive signal $\dot{\vec{x}}$ which it forwards to the controller for $\vec{x}$.

information: the position of the hand. The two vectors are related by a function $f : \vec{\theta} \mapsto \vec{\gamma}$ which maps from one coordinate system to the other. This function is the key to using the hand in conjunction with the eyes together.

This f turns out to be a very useful function: it can tell Cog three different things. First, given some random arm configuration $\vec{\theta}$, $f(\vec{\theta})$ will tell Cog where to direct its gaze in order to look at the hand (if, for instance, Cog forgets what it is holding). Second, if Cog is intently looking at some object with gaze angle $\vec{\gamma}$, then $f^{-1}(\vec{\gamma})$ will tell Cog where to move its arm in order to reach out and touch the object. Finally, if Cog's current arm position and gaze angle satisfy the relationship $\vec{\gamma} = f(\vec{\theta})$, then Cog knows that the target it is looking at *is* its hand.

Figure 5.8 illustrates a generic transformer module. It has inports for the current values of two state parameters $\vec{x}$ and $\vec{y}$. To act as a transform, it has an outport for the computed/predicted state vector $\vec{y}_x = f(\vec{x})$. As a classifier, it has an outport A_{xy} for the probability that $\vec{x}$ and $\vec{y}$ are observing the same process (e.g. the position of the hand). Finally, as a controller, it has a velocity inport $\dot{\vec{y}}_x$ and outport $\dot{\vec{x}}$. If $\vec{x}$ is a controllable parameter, then the latter port is connected to the $\dot{\vec{x}}$ controller. $\dot{\vec{y}}_x$ will then act as a controller for $\vec{x}$, in terms of its transform $\vec{y}_x$.

The details of how transform models are trained and used are discussed in Section 6.4.

5.4 Other Modules and Facilities

The dynamic modules discussed in the previous sections operate by connecting to parameters provided by a static foundation of hard-wired modules. meso and the visual and tactile senses comprise the bulk of that base layer. Two other static subsystems which have been alluded to but not yet described are the age mechanism and the emotional system.

5.4.1 Age

Most processes in pamet are time-dependent, but the dependency operates at two different scales. Control processes operate at the millisecond level. Developmental processes operate over minutes and hours, perhaps even days. Two different timebases are used in pamet to accommodate this.

The short-term timebase is the CPU clock, which is utilized for all real-time tasks. This includes sending activation signals and command streams, and sampling data — events which occur at frequent regular intervals. A precise timebase is necessary, for example, for accurately calculating velocities from state data: any jitter in the timebase adds noise to the measurement. All such code is implemented using the real-time timer facilities built into the QNX operating system, which provide microsecond resolution (as long as a processor is not overloaded). There is no synchronization between processes at these timescales, even on the same processor; each process essentially has its own local clock.

The long-term timebase is provided by a module called age. This is a sok-process with an outport which, once per second, broadcasts the "age" of the robot in days. This provides the system with a common global timebase for coordinating developmental events. These events themselves fall into two categories: absolute and relative.

Absolute events are keyed to the absolute age. Many prototype modules, such as the proto-controller, stay in hibernation until a certain developmental age is reached. These modules cannot function successfully until other structures in the system, such as the mover models, have stabilized, so this allows them to be hard-coded to wait for an appropriate length of time. Relative events are just keyed to a change in the age, usually relative to whenever a module was created.

The decay of random activation and the expiration of unused actors fall into this category. The "clock starts ticking" whenever an actor is created; the actor is given a deadline in robot minutes to prove itself useful, after which it is destroyed.

The `age` module has a few controls attached to it: time intervals can be added or subtracted from the age at the click of a button, and the aging rate (robot seconds vs. real-time seconds) can be varied from the default of 1.0. This freedom eases debugging of and experimentation with developmental events, and was the primary reason for decoupling the robot age from the local CPU clocks.

5.4.2 Emotion

The reward signals involved in training *actors* and *triggers* are provided by an extremely simple emotional system. The job of this system is merely to act as a clearinghouse for reinforcement in the robot. The emotional system is composed of two modules, `emo/happy` and `emo/sad`, which process positive and negative signals, respectively. Both modules have the same basic structure (Figure 5.9), and differ only in how their inports and outports are routed. Each module acts like a leaky integrator. The output R is a running sum of each input message s which is received, but R also decays by a factor of λ at each time step. The output value reflects the total *rate* of the input, averaged over a period of roughly the length of the time constant. The modules run at 40 Hz; λ is typically set to yield a time constant of two seconds, which equates to a fairly short "attention span" for reward.

The sources of raw reward are the *primal motivators* for the system, the hard-wired measures of what is good and bad to the robot. The sole source of positive reward is the tactile sense: when the robot's hand touches something or receives a squeeze, that increases its happy state. Negative reward is derived from `meso`'s joint-limit pain and muscle fatigue discomfort. Cog's sad state will increase if someone literally twists its arm behind its back.

Although it is satisfactory for the tasks implemented so far, this emotional model acts as little more than a global variable. `pamet` will eventually need a mechanism for expecting and accounting for reward (i.e. credit assignment). As more and more modules such as actors and triggers inhabit the system, each will expect reward for its own niche of state-space, and those niches will begin to overlap. Without a careful accounting for the reward, it may become impossible to tell if the reward is meant for an existing model or if it is marking a new experience. This system also needs more primal motivators — such

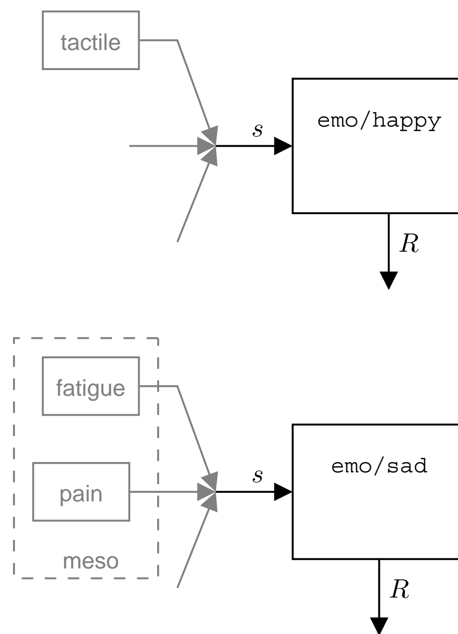


Figure 5.9: The basic emotional system employed by pamet consists of two modules, one for positive rewards and one for negative. Each acts as a leaky integrator which sums its input messages and produces an output which reflects the rate of reward input. The only source of positive reward is the tactile sense (any touch is good touch). Negative reward is derived from joint limit pain and muscle fatigue discomfort.

as simple facial expression or vocal prosody recognition, or even more abstract sources, such as senses of satisfaction and frustration (positive and negative reward sources) for models whose predictions are fulfilled or unfulfilled. An emotional system with states tied to gestural reflexes, such as the one found in Kismet [7], would give much needed natural feedback to people who interact with the system.

Chapter 6

Models and Modellers

This chapter details the implementation of the models and modellers introduced in the previous chapter. Recall that every module in pamet which produces some activity does so in accordance with some kind of model, generated dynamically by a modeller module. A modeller may update one of its models over time. It may also discard a model and destroy any other modules which were using it.

Most of the models are, ultimately, binary classifiers. Given a particular set of input signals, they attempt to sort samples into one of two classes — e.g. "rewarded" vs. "unrewarded", or "correlated" vs. "noise". The models differ in how the distributions of each class are modeled and which features of the distributions are put to use. In these terms, the primary job of a modeller is to decide if a reliable classifier exists for the data it sees, i.e. if there *are* two classes.

This chapter discusses the models and modellers in fairly generic terms. Many modellers have tunable parameters. The tuning of the parameters, and the successes and failures of the models, are explored along with the behaviors that this system produced, in Chapter 7.

6.1 Mover Models

A mover module is expected to produce a velocity in a state parameter $\vec{s}$ in proportion to the module's activation input A. By definition, then, a mover model is a linear model of the velocity $\dot{\vec{s}}$ as function of A which takes the simple form

$$\dot{\vec{s}} = A\vec{m}.$$

A mover may not affect all of the components of a state vector, so the model also includes a mask which specifies which axes of $\vec{s}$ are relevant.

Mover models are learned by mover modellers. For any given pairing of state $\vec{s}$ and mover activation A, a modeller needs to determine which axes of $\vec{s}$ are affected, if any, and what constants are involved. It does this by linear regression.

The modeller collects $(\vec{s}_i, A_i)$ sample pairs over a number of episodes of mover activity. A mover only transmits activation data when it is active, so there is no $(\vec{s}, A)$ data when the mover is inactive. The $\vec{s}$ samples are preprocessed episode-by-episode: each time-series is low-pass filtered, and sample-to-sample differencing is used to determine the velocity $\vec{v}$. A block of samples at the beginning and end of each episode — the tails of the filtering step — are discarded. Finally, the $(\vec{v}_i, A_i)$ pairs from all episodes are batched together and subjected to linear regression.

For each axis k, m_k is estimated by

$$m_k = \frac{\sum_i v_{ki} A_i}{\sum_i A_i^2}.$$

The correlation coefficients R_k, and their maximum, are also computed:

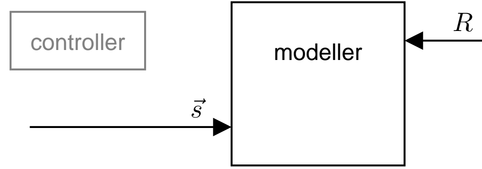
$$R_k = \frac{(\sum_i v_{ki} a_i)^2}{(\sum_i v_{ki}^2)(\sum_i a_i^2)}, \;\; R_{max} = \max(R_k).$$

A model is only created if R_{max} is greater than a threshold R_{thr}. If not, then the modeller decides that no component of the state vector is reliably linearly affected by the mover. If the threshold test does pass, then the axes for which $R_k > \lambda R_{max}$ are chosen as the relevant axes (and the appropriate bits in the mask are set).

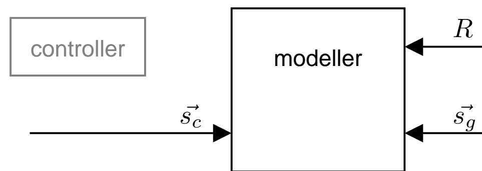
The result of this analysis is a model consisting of a vector $\vec{m}$ and a bit mask over the components of $\vec{m}$. $\vec{m}$ is an estimate of the velocity of $\vec{s}$ induced by the mover when it receives a unit activation $A = 1.0$.

6.2 Action Models

As discussed in Section 5.3.3, the purpose of an *actor* is to twist a knob. An actor relies on an *action model* to tell it which knob to twist and how to twist it. Two types of action models have been implemented: position-constant and position-parameter.



A



B

Figure 6.1: Two types of action modellers. (A) The position-constant modeller tries to determine which prototypical values of a controllable state parameter $\vec{s}$ are associated with reward R. (B) The position-parameter modeller tries to determine if reward occurs when controllable state parameter $\vec{s_c}$ is equal to or tracking a goal state parameter $\vec{s_g}$.

Position-Constant Action

The gist of a position-constant action is "drive the controllable parameter to a specific constant value". The job of the action modeller is to discover constant positions that correspond to useful goals. The position-constant action model is essentially a binary classifier, which decides when a state parameter is *rewarded* (useful) or *non-rewarded*.

Figure 6.1(A) illustrates a position-constant modeller. It has two inputs: the (controllable) state parameter $\vec{s}$ and a reward signal R. The modeller tries to correlate the reward signal with the state vector data stream to discover values that are consistently rewarded. It records pairs of samples $(\vec{s}, R)$ and analyzes the data in short batches (corresponding to around two minutes of real time). The first step of this analysis is to condition the reward levels by detrending, smoothing, differentiating, and then thresholding. This produces a series of pulses, where the onset of each pulse corresponds to a "reward event" — a moment when the robot was rewarded. Each state vector sample

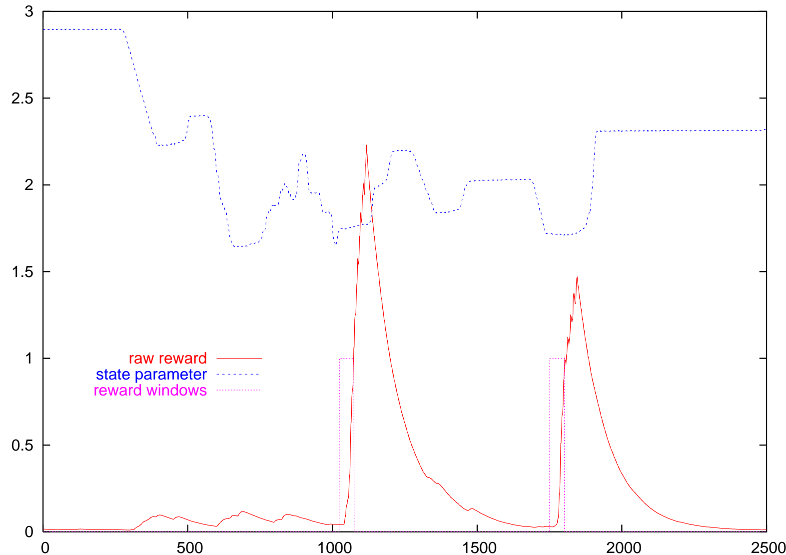


Figure 6.2: An example of position-constant-action data analysis. The raw reward is smoothed and thresholded to decide on a precise moment when reward occurred so that *reward windows* preceding the reward can be constructed. The distribution of state parameter samples occurring during reward windows ("rewarded") will be compared to the distribution of all the rest ("unrewarded").

$\vec{s}$ is then classified as "T" (rewarded) or "F" (not rewarded) according to the criterion of whether or not it falls into a fixed window of time preceding each reward event.

A basic assumption made here is that the reward immediately follows the desired action. The state vector values immediately preceding the reward are the ones which elicit it. Figure 6.2 illustrates part of a dataset from learning a simple "point-the-finger" action.

The next step is to decide if there is actually anything to be modelled. It may be the case that the reward R which the modeller sees is meant for some action involving some other state variable. If the distribution of T samples is significantly different from the distribution of F samples, then we conclude that the T samples are indeed being rewarded. If the T sample and F sample distributions are essentially the same, then we conclude that the reward is not, in fact, correlated with our state vector (not during this training session, at least), and

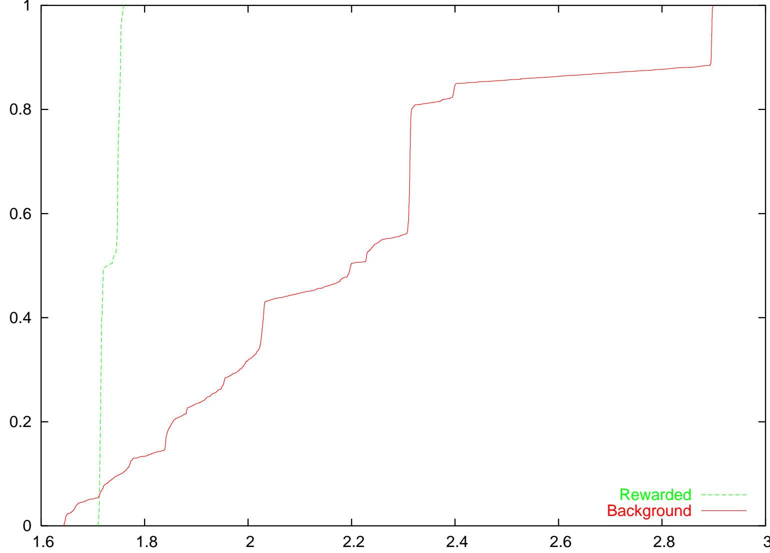


Figure 6.3: Comparison of CDF's to discover active axes. The rewarded and unrewarded sample distributions from the sample dataset are significantly different, both to the eye and by a measure of absolute area.

we toss out the data and start over again.

The distributions are compared by measuring the absolute area α between their cumulative distribution functions (CDF) $P(s_i|T)$ and $P(s_i|F)$, where

$$\alpha = \int |P(s_i|T) - P(s_i|F)|\, ds_i$$

normalized over the range of s_i. If $\alpha > \lambda = 0.1$, the T and F distributions are determined to be significantly different (Figure 6.3). This test is similar to the Kolmogorov-Smirnov (K-S) test [20], which measures

$$D = \max |P(s_i|T) - P(s_i|F)|,$$

but the K-S test is less sensitive to differences in variance.

Both tests are only applicable to 1-D distributions, thus this test is applied to each component of the state vector independently. This is necessary anyway, since not every component of $\vec{s}$ is necessarily germane to the action. For example, if the state vector is the complete vector of joint angles of the robot, perhaps only the position of the elbow is being

rewarded. The results of each test are combined to form the active mask for the state vector; each axis which demonstrates a difference between T and F distributions has its mask bit set. If no bits are set in the mask, then none of the state vector is correlated with the reward data and no model is created.

If at least one axis can claim rewarded samples, then a position-constant action model is created, consisting itself of models of the T and F distributions. These models are simply multivariate Gaussians. Each model is just the mean and variance of the set of active (masked) components of the class's state vector samples. Assuming the distributions are actually near-Gaussian, the T mean is the state vector with the maximum-likelihood for receiving reward.

Whenever a new model is created, it is compared to all existing models (for the same $\vec{s}$). If the new model is similar to an existing model, then the same action is probably being rewarded and reinforced again, and the old model should be *refined* instead of duplicated.

Models are compared by their $p(\vec{s}|T)$ distributions. If the means are within 1.5 standard deviations (taken from the variances), then they are considered similar. In that case the models are merged by averaging their parameters (the means and variances), and the old model is updated with the new values. If the new model is not similar to any existing ones, then it is registered and a new actor is spawned.[1]

Position-Parameter Action

The gist of a *position-parameter action* is "drive the controllable parameter to match another state value". The modeller for such an action (Figure 6.1(B)) has two state inputs instead of one: the controllable parameter $\vec{s}_c$ and the target parameter $\vec{s}_g$.

The entire discussion of position-constant models applies equally to position-parameter models, except that the controllable state $\vec{s}$ is replaced by the difference $\vec{e} = \vec{s}_c - \vec{s}_g$. Whereas the position-constant model contains the reward-relative distributions $p(\vec{s}|T)$ and $p(\vec{s}|F)$, the position-parameter model consists of $p(\vec{e}|T)$ and $p(\vec{e}|F)$. The modeller records batches of samples $(\vec{s}_c, \vec{s}_g, R)$, which are processed into streams of $(\vec{e}, R)$. The same distribution comparison algorithms are applied to determine if there is a significant difference between rewarded and non-rewarded $\vec{e}$ values.

The model is sensitive to the condition that $\vec{s}_c$ tracks $\vec{s}_g$ with a constant offset. If that offset is zero, then $\vec{s}_c$ is exactly following $\vec{s}_g$, but

[1] If the new model is similar to *two* old ones, only the closest one is currently merged. There is no mechanism for condensing existing models.

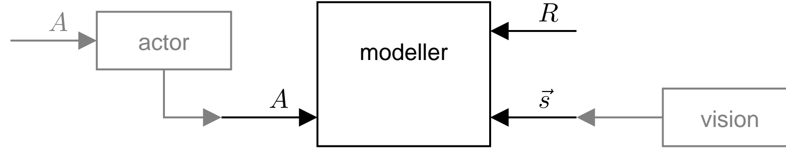


Figure 6.4: A position-trigger modeller tries to find an association between a state parameter $\vec{s}$, an actor's activation A, and reward R. The goal is to determine a region of the space of $\vec{s}$ (a context) which leads to the actor being activated (triggering the execution of an action).

that is not a critical condition for the basic behavior. When the model is instantiated by an activated actor, the actor continuously drives $\vec{s}_c$ to the goal position ($\vec{s}_g + \hat{\vec{e}}$), where $\hat{\vec{e}}$ is the mean encoded in $p(\vec{e}|T)$.

6.3 Trigger Models

A trigger module activates an action in response to some stimulus, and that stimulus is defined by a *trigger model*. Only one type of model (and thus one type of stimulus) has been implemented so far, and that is the *position-trigger* model.

Trigger modellers are created by *proto-trigger-modeller* modules. Whenever a new actor appears in sok space, the prototype module creates a new modeller for every possible stimulus for that actor. Each modeller has inports for monitoring a state vector $\vec{s}$ (the potential stimulus), the activation level A of the actor, and a reward signal R. Figure 6.4 illustrates an example of a trigger modeller observing an actor and a vector of visual features. The basic operation of the modeller is to look for a correlation between the action, the state vector, and the reward; the modeller is trying to determine if the action, when associated with some stimulus, is consistently rewarded.

Position-Trigger Model

Position-trigger models are sensitive to state parameters, which can be either sensory or motor parameters (or anything in between). They can define a context related to a sensory stimulus, such as "looking at a red object" (i.e. high 'redness' reported by the visual features module). Or, the context can refer to a particular motor configuration, such as "an outstretched arm".

The position-trigger model can be viewed as a binary classifier where

the two classes are *stimulus* and *non-stimulus*. As with the action models, it's convenient to label the classes "T" and "F" respectively. It is convenient to think of the non-stimulus distribution as the "background" distribution.

Each class is modelled by a distribution $p(\vec{s}|C)$. The context represented by the model is considered satisfied when the input state $\vec{s}$ is classified as stimulus. The criterion for this is:

$$log\left(\frac{P(T|\vec{s})}{P(F|\vec{s})}\right) = log\left(\frac{p(\vec{s}|T)P(T)}{p(\vec{s}|F)P(F)}\right) > \lambda$$

where λ is a threshold typically set to zero.

Within this basic framework, the distributions $p(\vec{s}|C)$ could be modelled in any number of ways. Following the frugal example set by action models, here they are modelled as unimodal, multivariate Gaussian distributions. This is sufficient to capture the concept of a stimulus consisting of a single prototypical parameter value.

Position-Trigger Modeller

The modeller operates by collecting and analyzing batches of sample triplets $(\vec{s}, A, R)$, where $\vec{s}$ is a state parameter vector, A is the activation status of an actor, and R is a reward signal. The basic operation loop is:

1. Collect and preprocess a batch of data.
2. Try to extract a position-trigger model from the data.
3. If no useful model is discovered, goto 1 and try again.
4. Compare the new model to existing models. If the new model is similar to an existing one, refine the existing one; goto 1.
5. Register the new model; spawn a trigger; goto 1.

The data is first preprocessed by low-pass filtering $\vec{s}$ and filtering R to produce a series of pulses signifying reward events. Figure 6.5 gives an example of raw and preprocessed data.

The next step in creating a model is to classify the $\vec{s}$ samples as T (stimulus) or F (background). The basic assumption is that a trainer is going to reward the robot when it performs an action in response to a stimulus. Thus, the stimulus should be represented by those samples which occurred in conjunction with both the action and the reward. However, it is possible that the recorded rewards pertain to something

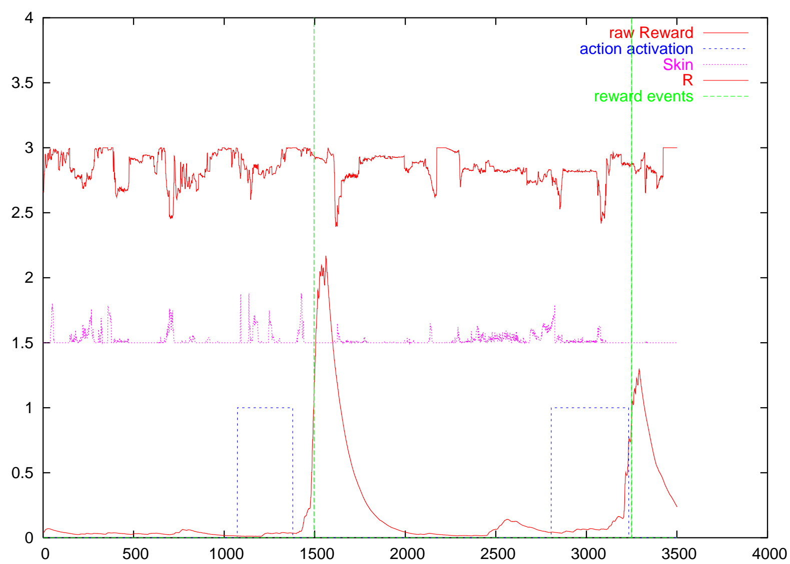


Figure 6.5: A sample of a position-trigger training data set. The two components of state parameter $\vec{s}$ are target features from the vision system (skin tone and redness). A is the activation of a "reaching" actor; R is the reward signal (derived from tactile reinforcement). The R signal is processed to yield discrete reward events.

else; perhaps the action which is being rewarded is not the action which the modeller is observing. Likewise, it is possible that the actual stimulus involves only a few of the components of the state parameter, or a different state parameter altogether. The modeller will need to be able to distinguish these cases.

What does it mean for a particular sample $\vec{s}$ to occur "in conjunction" with an action and reward? The onset or endpoint of the action will not necessarily coincide with the reward, and certainly the single state sample that is recorded simultaneously with either of these events is of little statistical value. What we really want is the *window* of samples which encompasses the stimulus. This will give us a sample set over which we can estimate a distribution. A couple of different ways of determining such windows were implemented.

Action-Onset Basis

The first technique makes the following assumptions about the training process: One, the stimulus will be presented near the beginning of the action, and the action will probably take some length of time to complete (e.g. to move the arm to some position). Two, the reward will be delivered near the end of the action. The stimulus windows determined under these assumptions are *action-onset based*.

We find likely windows in two steps. The intervals during which the action is activated are the primary timebase. Let us sequentially label each such active interval as $a_0, a_1, ..a_i, ..$, and the onset and ending times of those intervals by t_{i0} and t_{i1}, respectively.

First, over the entire dataset, we find the *action windows*, the samples which are to be associated with initiating the action. This is the set S_a of samples which fall in a window of length τ_a which begins at offset τ_{0a} before an action *onset*. (The length and offset are tunable parameters.) In other words:

$$S_a = \{\vec{s}(t) \,|\, (t_{i0} + \tau_{a0}) < t < (t_{i0} + \tau_{a0} + \tau_a), \text{ for some } a_i\}$$

The action windows for our example data set are illustrated in Figure 6.6.

Next, we find the *reward windows* S_r, the samples to be associated with receiving reward. This is a bit more complicated. First, we need to decide whether or not each reward is associated with an action. For a reward to be associated with an action, it should closely follow the action. So, we use the criterion that a reward event occurring at time t is associated with action a_i if

$$(t_{i1} + \tau_{r0}) < t < (t_{i1} + \tau_{r0} + \tau_r)$$

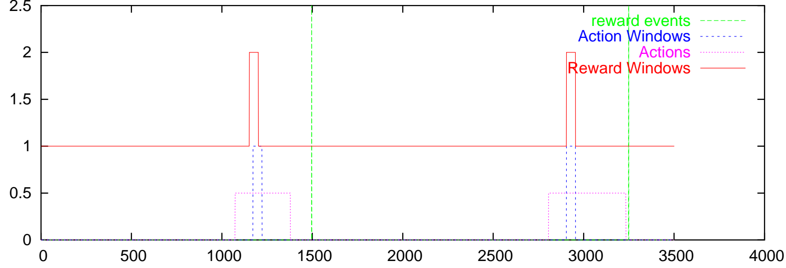


Figure 6.6: Action and reward windows for the example dataset. The action windows are 1s in duration and follow the onset by 2s. The reward windows apply only to those actions which have a reward which occurs within 2s of completion.

where τ_{r0} and τ_r define the offset and length of the interval of association. A reward event is associated with the closest action which satisfies that criterion, and if no action satisfies the criterion, the reward event is considered a spurious event. If a reward event is associated with an action, then we say that action has been rewarded. We also calculate t_{avg}, the mean delay from action onset to reward.

The reward windows S_r are going to be all the $\vec{s}$ samples which fit the action window criterion, whether there was an action or not. Concisely,

$$S_r = \begin{cases} \vec{s}(t) \,|\, (t_{i0} + \tau_{a0}) < t < (t_{i0} + \tau_{a0} + \tau_a), \text{ for some rewarded } a_i \\ \vec{s}(t) \,|\, (t_r - t_{avg} + \tau_{a0}) < t < (t_r - t_{avg} + \tau_{a0} + \tau_a), \\ \qquad \text{for spurious events at } t_r \end{cases}$$

The first term is all reward-worthy samples, for actions which received reward. The second term is an estimate of reward-worthy samples for rewards which had no corresponding action. The intersection $S_a \cap S_r$ is the set of all samples $\vec{s}$ which occurred in conjunction with both an action *and* a reward.

If $S_r \cap S_a$ is non-empty, then at least one invocation of the action appears to be coincident with a reward. It is possible, however, that the reward was intended for different action or a different state parameter, or both. If the distribution of S_r is the same as that of $\neg S_r$, then the context for the trigger does *not* involve $\vec{s}$ because the rewarded samples look the same as the unrewarded samples. Likewise, if the distribution of $S_r \cap S_a$ is the same as $\neg(S_r \cap S_a)$, then the trigger does not pertain to the given action.

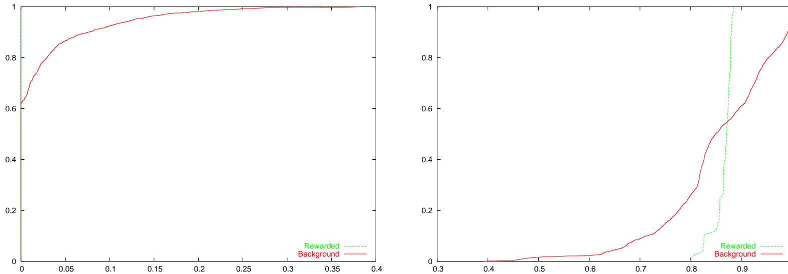


Figure 6.7: Cumulative distribution functions (CDF's) of the stimulus samples $(S_r \cap S_a)$ and background samples $\neg(S_r \cap S_a)$, as classified by the reward windows in Figure 6.6. CDF's are computed independently for each component of the sample vector, and the absolute area is measured. The difference is significant for the second component but not the first. Hence, only the second component would be used in the stimulus.

The two sets of distributions are compared using the same CDF measure used for the action models (Section 6.2). As before, the comparison of distributions is performed independently on each component of $\vec{s}$ in order to decide which ones, if any, are actually part of the stimulus. Figure 6.7 illustrates the CDF's for the sample data set.

Sliding Windows

Note that this algorithm depends on a few important parameters: the lengths and lags of the windows used for actions and for reward association. These numbers must reflect the interaction between the trainer and the robot. The reward association window (τ_{r0}, τ_r) is fairly forgiving. A typical working value is $(0, 2)$. That is, any reward occurring within the two seconds immediately following an action is considered a reward for that action. The action window (τ_{a0}, τ_a) is more critical. The offset describes when the stimulus is expected to happen with respect to the action and will vary depending on the nature of each. The ideal offset also depends on the reaction time of the trainer: the trainer will be applying the stimulus when the robot spontaneously performs the action, to make the robot associate the two. The length of the window is crucial as well. Ideally, it would match the length of the stimulus so as to yield the most accurate statistics. If too long, non-stimulus background samples will get mixed into the model. If too short, the window may provide poor statistics or even completely miss

the stimulus, depending on how precisely the offset is known.

In practice, a fixed-length action window τ_a of one second seems to work. However, the lag τ_{a0} needs to be tuned. The solution is to compare the CDF's of S_r and S_a computed using several different lag values, effectively sliding the action windows around until the optimal value is discovered. (The optimum is the the value which produces the largest above-threshold difference measure, summed over all axes.) The lag is varied in increments of one-half the window length, over a range of typically one to three seconds.

Distributions

After all this busywork to calculate an appropriate $S_T = (S_R \cap S_A)$ and $S_F = \neg(S_R \cap S_A)$, the rest of the model is simple: just calculate the means and variances of each set to yield unimodal Gaussian models $p(\vec{s}|T)$ and $p(\vec{s}|F)$. The *a priori* probabilities of each class are

$$P(T) = \frac{|S_T|}{N},\ P(F) = \frac{|S_F|}{N}$$

where N is the total number of samples in the data set. Note that the preceding discussion is independent of how the distributions of S_T and its complement are modelled. The only assumption is that the distributions are separable, and this arises when the distributions are compared.

The Gaussian model is used because it is very, very simple and gets the job done. It expresses the concept of a single prototype parameter value (the mean) which is the ideal stimulus. This makes the instantiation of the model by a trigger module straightforward.

Again, as with action models, once a new model is created from a batch of samples, the trigger modeller compares it with any other models which it has already created. It may be the case that the robot is still (or again) being trained to do the same thing as in an earlier session. If so, instead of registering a duplicate new model, the older model should be refined.

Two models are compared by examining the means of their stimulus distributions. If the means are within 1.5 standard deviations of each other (derived from the mean of the variances), then the models are considered to represent the same stimulus. The new model is merged into the old model by averaging the means and variances together. Then the trigger assigned to the old model is notified that its model has been updated.

Position Trigger

The *position-trigger* module introduced in Section 5.3.4 instantiates the model. A trigger has an inport for the stimulus state parameter and an outport connected to an actor's activation input. When the stimulus input matches the stimulus in its model (and as long as it does), the trigger activates the actor, which in turn causes the robot to perform some action.

As mentioned earlier, the stimulus context is satisfied for an input $\vec{s}$ when

$$\rho = log\left(\frac{P(T|\vec{s})}{P(F|\vec{s})}\right) = log\left(\frac{p(\vec{s}|T)P(T)}{p(\vec{s}|F)P(F)}\right) > \lambda$$

The activation level output by the trigger is $P(T|\vec{s})$; the message is only sent when $\rho > \lambda$. This value does not directly affect the actor — as long as the actor receives a message, it activates — however, it may also be used as state parameter, to trigger other actions.

Note that the trigger criterion depends on a comparison of the distribution $p(\vec{s}|T)$ for the desired stimulus signal with the distribution $p(\vec{s}|F)$ of the non-stimulus, or "background", signal. How these two relate changes the region of the parameter space which constitutes the stimulus (Figure 6.8). Since the two distributions are modelled as Gaussians, each class will be a single connected region in the space, and the decision boundary will be a (hyper)conic section.

The trigger module updates the background distribution over time. The $p(\vec{s}|F)$ distribution created by the modeller reflects the statistics of the state parameter when the model was trained. However, this is not necessarily a stationary process. As the robot moves about, and as the robot begins doing new or different things, the background distribution will probably change. (Imagine the visual impact of different tour groups coming to visit the robot, with each group wearing a different color t-shirt so that its members don't get lost.) The trigger uses $p(\vec{s}|F)$ supplied by the trigger-modeller as a seed when it starts up, but modifies it over time as it takes in new state parameter samples.

The trigger updates the background distribution by averaging in each new sample, weighted to yield a half-life of 300 seconds. Thus, the background distribution eventually reflects the mean and variance of the entire signal (over the last 10 minutes or so). This seems statistically inappropriate, since it conflates the stimulus and non-stimulus samples, but it works just fine. The rationale is that stimulus events should generally be rare ($P(T) \ll P(F)$), so including the stimulus samples in the calculation of the background should not significantly affect it. If for some reason the stimulus becomes so very frequent that the background

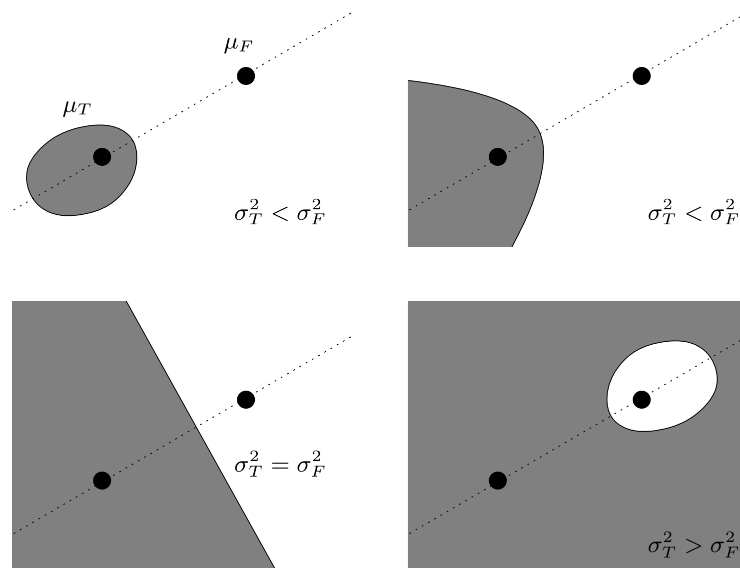


Figure 6.8: Possible partitions of a 2-D parameter space by a Gaussian binary classifier. The shaded region corresponds to the “stimulus”. The shape of the decision boundary is a conic section determined by the variances σ_T^s and σ_F^2 of each class. The “stimulus” may be confined to a closed region, or it may be everything *but* a closed region.

distribution begins to look like the stimulus, then the trigger criterion will become more difficult to satisfy. In effect, the trigger will habituate to the stimulus, and this is actually a desirable behavior.

6.4 Transform Models

A transform model describes a situation in which two state parameters are different views of the same underlying process. As discussed in the previous chapter, the motivating example is the case when the eyes are tracking the hand. Since the gaze angle of the vision system is locked onto the position of the hand, the gaze angles and the arm's joint angles become causally related, and in particular that relation is a coordinate transformation. Of course, the eyes are not always fixated on the hand. The transform model must not only learn the transformation function; it must also be able to discern when it is applicable.

Generic Transform Model

A *transform model* in pamet is another type of binary classifier. It is defined on a pair of state parameters $\vec{x}$ and $\vec{y}$ by a pair of distributions over the two classes, $p(\vec{x}, \vec{y}|T)$ and $p(\vec{x}, \vec{y}|F)$. The first distribution is the *tracking* distribution; it describes the statistics of $(\vec{x}, \vec{y})$ when the two parameters describe the same phenomenon — e.g. when the eyes are targeting the hand. The second distribution is the *background* distribution, which describes the other case — e.g. when the eyes are looking at anything *but* the hand.

For the non-tracking, background case, it is assumed that $\vec{x}$ and $\vec{y}$ are independent. Then,

$$p(\vec{x}, \vec{y}|F) = p(\vec{x}|F)p(\vec{y}|F)$$

and those two component distributions are simply modelled as multivariate Gaussians.

For the tracking case, $\vec{x}$ and $\vec{y}$ are decidedly dependent (that is the whole point of this exercise), and that dependency is described by a transform function f. Let's assume a Gaussian noise source in the measurement of $\vec{y}$; then when $\vec{y}$ is tracking $\vec{x}$,

$$\vec{y} = f(\vec{x}) + \varepsilon$$

and

$$p(\vec{y}|\vec{x}, T) = g(f(\vec{x}), \sigma_\varepsilon^2),$$

where $g(\mu, \sigma^2)$ is a Gaussian distribution with mean μ and variance σ^2, and σ_ε^2 is the variance of the noise source. This gives us

$$p(\vec{x}, \vec{y}|T) = g(f(\vec{x}), \sigma_\varepsilon^2)p(\vec{x}|T)$$

as the tracking distribution.

The transform function f needs to be learned. This is a problem of function approximation, and there are no good general answers. Two techniques were used in this thesis. The first is a non-parametric memory-based technique with a smoothness constraint. The second is a semi-parametric model for a true coordinate transformation. The former is quicker (trivial) to train than the latter, but the latter is significantly faster than the former for producing estimates. These models are discussed further in Appendix B.

Transform Model as Predictor, Classifier, and Controller

A transform model can serve three different purposes. Most trivially, it can serve as a predictor of the "y-space position" of $\vec{x}$, by evaluating the transform function, $\vec{y}_x = f(\vec{x})$. In the eye-arm case, this function predicts "where the eyes need to look, in order to look at the hand", given the current joint angles of the arm. The model could also do the converse, predicting "where the arm needs to move, so that the eyes focus on it", given the current gaze angle. This would also be useful, but not so trivial, since it requires solving $\vec{y} = f(\vec{x})$ for $\vec{x}$, when f is not necessarily invertible. Fortunately, the structure of pamet obviates a direct need for this.

The second use of a transform model is as a classifier, to estimate the probability γ that process $\vec{y}$ is tracking process $\vec{x}$. In practical terms, it can answer the question "Are the eyes looking at the arm?" This estimate is a direct result of the distributions in the model:

$$\gamma = P(T|\vec{x}, \vec{y}) = \frac{p(\vec{x}, \vec{y}|T)P(T)}{p(\vec{x}, \vec{y}|T)P(T) + p(\vec{x}, \vec{y}|F)P(F)}$$

The third use is as a controller. Recall that in pamet, a controller (Section 5.3.2) is a module that can drive a particular state parameter according to an input velocity. If $\vec{x}$ is a controllable state parameter, then a transform model can specify how to drive $\vec{x}$, given a velocity $\dot{\vec{y}}_x$ expressed in the "y-space". For example, this would allow the arms to be moved around in the gaze space of the robot, i.e. the robot could move the hand to whatever it happened to be looking at. This is possible because a controller only needs to convert a velocity from one space to the another, and that amounts to solving the linear equation

$$\dot{\vec{y}}_x = F(\vec{x}_0) \cdot \dot{\vec{x}}$$

for $\dot{\vec{x}}$ where $F(\vec{x}_0)$ is the Jacobian of f evaluated at the current position of the system $\vec{x}_0$. Even if f is not invertible (and thus F is not an invertible matrix), this can be solved using singular value decomposition (SVD) [43, 47]. The particular solution provided by SVD minimizes the norm $\|\dot{\vec{x}}\|$. In general, that is precisely the solution we want: it is the solution which achieves the goal velocity with minimal extraneous movement.

This controller method is essentially the same process as using the Newton-Raphson method to iteratively solve $\vec{y} = f(\vec{x})$ for $\vec{x}$, by step-by-step moving $\vec{x}$ along the gradient which minimizes $(\vec{y} - f(\vec{x}))$. Instead of just repeating the calculation until it converges to a solution for $\vec{x}$, however, the $\vec{x}$ state is physically changed at every step (e.g. the robot arm actually moves a bit). This method will fail as a controller in the same ways it can fail as a root solver. The fact that $\vec{x}$ is physically manifested can be helpful, though. For example, slew limits and the natural wobbliness of the arm help keep it from zooming off to infinity or getting caught in a cycle.

Transform Modeller

Transform models are created by a *transform modeller* module, which, as one might expect by now, is itself spawned by a *proto-transform-modeller*. One transform modeller is created for every pair of state parameters. Like all other modeller modules, the basic order of operations of the transform modeller is the following:

1. Collect a set of $(\vec{x}, \vec{y})$ samples.
2. If a model already exists, update the model using the new data.
3. If no model already exists, analyze the data to try to create a valid transform model.
4. If a new model is created, register it and spawn a *transformer* to put it to work.

If each $(\vec{x}, \vec{y})$ sample were already tagged as "tracking" or "background", the analysis would be trivial. We could just calculate the mean and variance of the background samples to produce the background distribution $p(\vec{x}, \vec{y}|F)$. Then we could train an approximator $\hat{f}$ on the tracking samples (straightforward supervised learning), and use its performance to

estimate σ_ε^2, yielding $p(\vec{x}, \vec{y}|T)$. Unfortunately, the samples are not conveniently tagged. The transform model, acting as a classifier, could do this tagging for us, but we don't have a transform model yet!

A way to wriggle out of this conundrum is to use the Expectation-Maximization (EM) algorithm, applied to classifying the samples as tagged or background. The basic idea is to iterate between these two steps:

1. For each sample $s_i = (\vec{x}_i, \vec{y}_i)$, use the current model to estimate the probability γ_i that s_i is a tracking sample (versus a background sample). γ_i is also known as the *responsibility coefficient*.

2. Update the model. Recompute the tracking and background distributions, weighting each sample's contribution to each distribution by γ_i and $(1 - \gamma_i)$ respectively.

EM is guaranteed to converge to something, but it's not guaranteed to converge to the answer we're looking for. The main question here is how well the tracking samples can be resolved from the background noise of all the rest. The answer to that question rests in the nature of the approximator used to learn $\hat{f}$.

Unfortunately, of the two approximators which were implemented, neither worked well enough in the sample task (learning the transform between arm joint angles and hand visual position) to yield an accurate "tracking" versus "background" classifier. With the available data sources, this algorithm could not be made to converge properly. However, the semi-parametric approximator was far more robust in the face of training data which contained many junk, non-tracking samples. Since its form is more constrained than the non-parametric model (whose only constraint is smoothness), it did a better job of locking on to samples that actually had a coordinate-transform relationship. These efforts are further described in Section 7.4.

Refinement

The preceding discussion was mostly concerned with bootstrapping: discovering a brand new model within a set of data. If a decent model already exists, refining it via the addition of new data is a much simpler process. We simply run one iteration of the EM algorithm:

1. Using the current model, compute the responsibility coefficients γ_i for each sample. (If we expect a useful temporal continuity constraint, $\{\gamma_i\}$ can be filtered over time.)

2. Iteratively update the transform function f and the background distribution, weighting each new sample with by its γ_i.

If the transform function uses an online update rule, this sequence can be applied online as well, as each sample is recorded. After a modeller has created its transform model, it continues to collect data and to refine the model.

Chapter 7

Learning Simple Behaviors

This chapter documents how the system learns a progression of simple behaviors through a combination of robotic self-exploration and human training. The first behavior is just controlled movement, developing controllers for the fingers and arm. This allows the robot to be taught specific hand and arm gestures. The gestures can then be conditioned to follow tactile or visual stimuli. The trainer can refine a gesture over time, keeping it linked to the same stimulus. Once the arm is moving around in the visual workspace, the robot will develop a controller for actively moving in that workspace. The robot can then be taught to reach to where it is looking. Finally, that reaching can be triggered by a particular visual cue.

Figure 7.1 shows a schematic diagram of the system when it is first started up. Only pamet modules are enumerated. Since the motor, vision, and tactile subsystems are static, they are shown as monolithic boxes. The contents of those boxes have already been described in Figures 3.1 and 4.5.

7.1 Moving via Movers

The motor primitives, which link pamet to the virtual muscles, are the mover modules (Section 5.3.1). Each mover is hard-coded to command a subset of virtual muscles with a particular velocity vector, which is scaled by the activation level of the mover. The system has two independent sets of movers, one for the right arm and one for the hand,

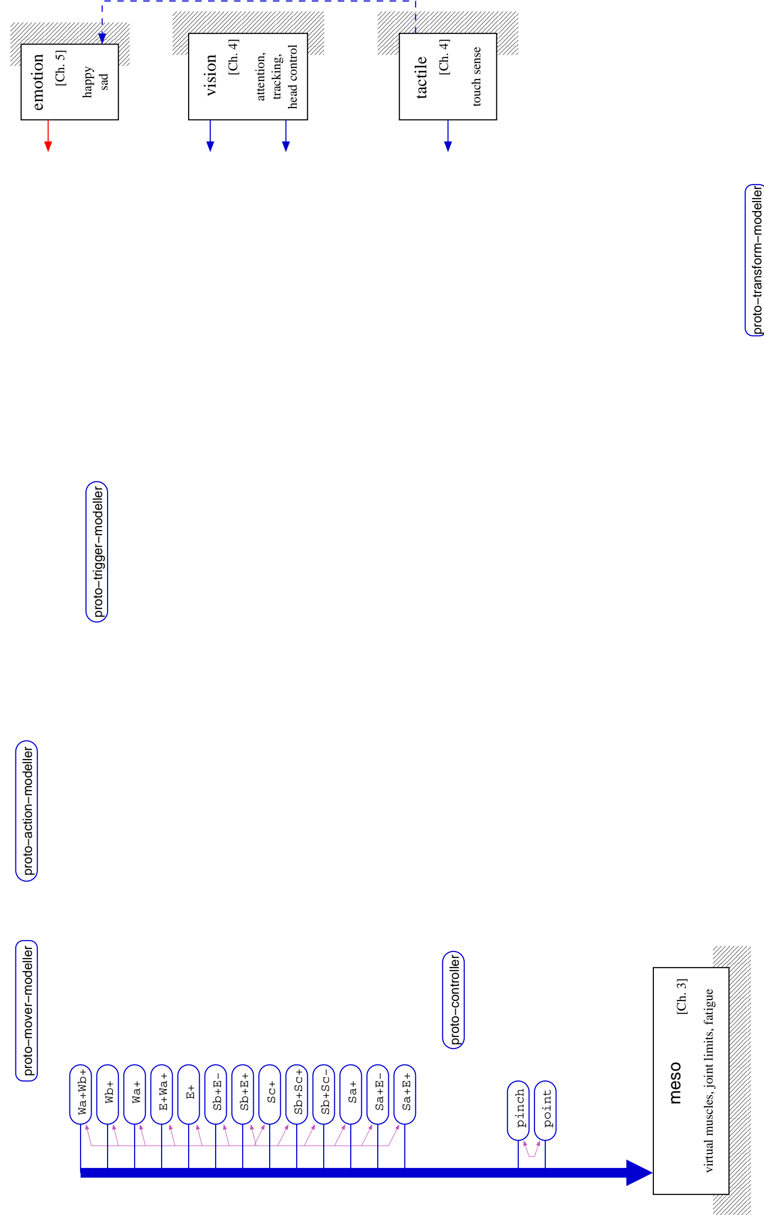


Figure 7.1: Schematic of the initial state of the system. Aside from the static structures of the motor and sensory systems, sok space is populated by only mover modules and the various proto-modellers.

Right Arm Movers

name	ShA	ShB	ShC	El	WrA	WrB
`Sb+Sc-`	0.0	2.0	-2.0	0.0	0.0	0.0
`Sb+Sc+`	0.0	2.0	2.0	0.0	0.0	0.0
`Sc+`	0.0	0.0	3.0	0.0	0.0	0.0
`Sa+E+`	3.0	0.0	0.0	2.0	0.0	0.0
`Sa+E-`	3.0	0.0	0.0	-2.0	0.0	0.0
`Sa+`	2.0	0.0	0.0	0.0	0.0	0.0
`Sb+E+`	0.0	2.0	0.0	3.0	0.0	0.0
`Sb+E-`	0.0	2.0	0.0	-3.0	0.0	0.0
`E+`	0.0	0.0	0.0	3.0	0.0	0.0
`E+Wa+`	0.0	0.0	0.0	3.0	2.0	0.0
`Wa+`	0.0	0.0	0.0	0.0	2.0	0.0
`Wb+`	0.0	0.0	0.0	0.0	0.0	2.0
`Wa+Wb+`	0.0	0.0	0.0	0.0	2.0	2.0

Right Hand Movers

name	Thumb	Paddle
`pinch`	-1.0	1.0
`point`	-1.0	-1.0

Table 7.1: The complete set of mover modules which connect pamet to meso. For a given mover (row), each coefficient specifies the velocity of the given muscle (column) when the mover has unit activation. Most movers drive two muscles.

listed in Table 7.1. The movers are hand-crafted to reflect a repertoire of basic two- or three-actuator motion combinations. They fully cover the joint velocity space, just as a complete collection of single-joint vectors would. However, when the robot is not executing learned actions (such as in the early stages of its development, when it has none), all its movement is derived from random activation of single movers. (Mutual inhibition connections allow only one mover to be active at a time.) Movers which encode multi-joint motor combinations produce more life-like motion than those which only move a single joint.

When the system starts up, random activation of the mover modules causes the robot's arm and hand to begin moving, exploring the workspace. Mover modellers are automatically created for each mover, to determine which state parameters the movers are driving. The movers directly control muscle velocity, and via the action of the muscles this affects the joint velocity. The system has no *a priori* knowledge

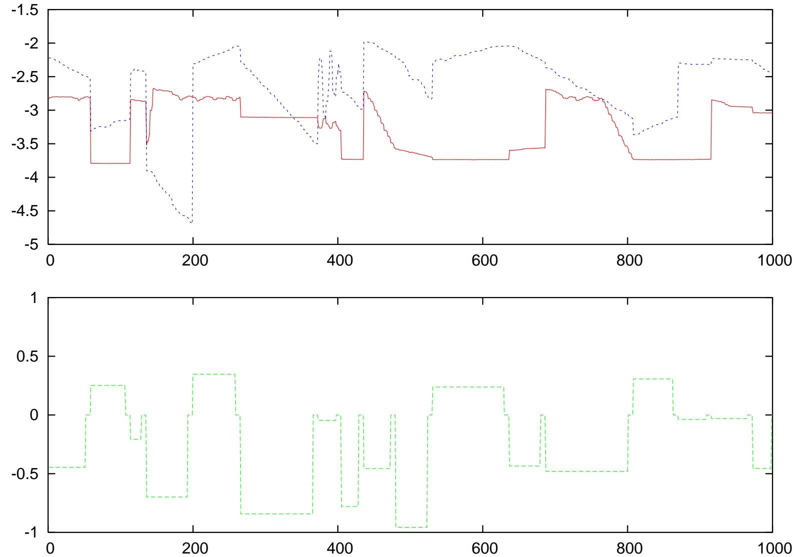


Figure 7.2: Elbow and thumb joint angles θ (top) and mover activation A (bottom) versus time, recorded by the modeller for the elbow mover (`E+`).

of this connection and must discover it.

Each mover modeller records and analyzes series of $(\vec{s}, A)$ samples, where $\vec{s}$ is a state parameter and A is the activation of the associated mover. Figure 7.2 shows a sample dataset of elbow and thumb joint angles and elbow mover activation over a period of 200 seconds of activity. The data was collected over a larger period of real-time; only the intervals in which the elbow mover was activated are recorded, because the mover has no activation A to record when it is inactive.

The modeller filters and differentiates each episode of mover activity to yield the joint velocity over time (Figure 7.3). It then tries to fit a linear model of $\vec{\theta}$ as a function of A (Figure 7.4). The fit is performed by linear regression independently on each component of $\vec{\theta}$. Not every component is necessarily affected by every mover. A mover controlling finger muscles, for example, shouldn't have any impact on the joint angles of the shoulder. The correlation coefficient R^2 of each fit is used to decide whether or not a component is being controlled by the mover. The velocity of a joint which is not controlled by the mover should show less correlation with the activation value than joints which are

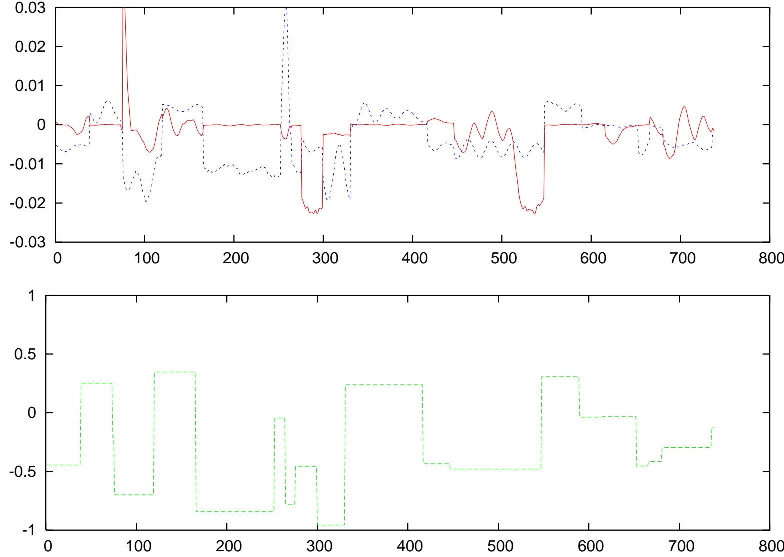


Figure 7.3: Elbow and thumb joint velocities $\dot{\theta}$ (top) and mover activation A (bottom) versus time for the elbow mover modeller. Velocities are estimated by smoothing and differencing the joint angle data.

controlled. The disparity is enhanced because typically other movers are randomly moving the uncontrolled joint.

For any joint to be considered affected by a mover, its R^2 must surpass 0.4. Furthermore, if the maximum coefficient over all the joints is R^2_{max}, then R^2_j must be greater than $(0.2)R^2_{max}$ for joint j to be considered. This local threshold allows the decision of choosing one joint over another to be relative to the quality of the given data.

Problems

The coefficient vector $\vec{m}$ learned by a mover model is an estimate of the joint velocities produced by unit activation of a mover. The mover itself commands a set of muscle velocities. For a mover which only commands single-joint muscles, we can predict what the ideal $\vec{m}_*$ should be, since the virtual muscle is specified by the single ratio between its length and the joint angle.

It turns out that the learned $\vec{m}$ always underestimates $\vec{m}_*$ (i.e. is biased towards zero). Three effects contribute to this error:

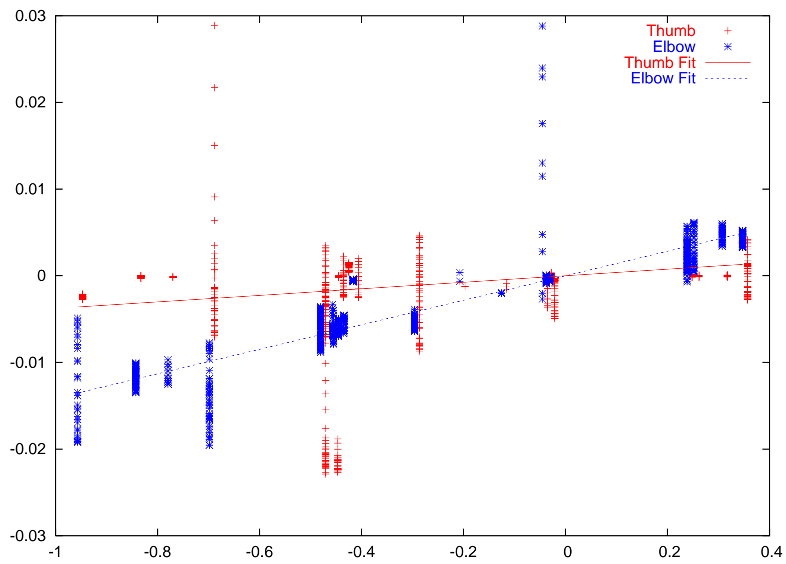


Figure 7.4: The linear fit of $\dot{\theta}$ versus A by the elbow mover modeller, shown for the elbow and thumb joints. The correlation coefficients for each joint are: elbow $R^2 = 0.817$, thumb $R^2 = 0.081$. The elbow is sufficiently correlated with the mover activation, but the thumb is not — just as one would hope for an "elbow mover".

1. Joint limits: When a joint reaches its mechanical limit, its velocity is clipped to zero, no matter what the mover is commanding.
2. Inertia: When the muscle starts moving and pulling on the joint, there is a lag before the joint begins to move. Likewise, there is a lag (and oscillation) when the muscle decelerates; but deceleration occurs after the mover has become inactivated. Thus, only the acceleration lag is recorded in training data.
3. Gravity: Virtual muscles are spring-like. When a joint is in a configuration which opposes gravity, the equilibrium point of the muscle is stretched out farther than the actual joint position. Large changes in the equilibrium point are required to produce small changes in the joint angle. Thus, a large muscle velocity is needed to achieve the same joint velocity.

The first problem could be mitigated by filtering out samples which are recorded when the joint is at its limits, since the limits are known to the motor system (and used, for example, in the calculation of the pain response). This would, however, complicate the issue of using mover modellers to model other linear phenomena in addition to virtual muscle activation — how would knowledge of such "extenuating circumstances" be discovered in general? The second problem could be partially resolved by eliminating more data from the beginning of each episode; the problem of discovering how much is necessary is still a problem. The third issue, the gravity effect, actually demonstrates a strength of the adaptive paradigm: the real system does not completely match the theoretical operation, and the learned model reflects that. However, the model is not sophisticated enough to account for the fact that this effect varies over the workspace.

7.2 The Toy Finger Robot, Realized

After the mover models have been created and registered, pamet's proto-controller module can group them together and spawn controller modules for each independent subset. With the given movers, two controllers are created: one for the hand and one for the right arm. The controllers know how to control joint velocities by activating the appropriate mover modules.

Once those controllers are created, the hand and arm joint velocities become *controllable* state parameters, and the proto-action-modeller spawns action-modellers to learn useful configurations for each. Those configurations become encapsulated within action models.

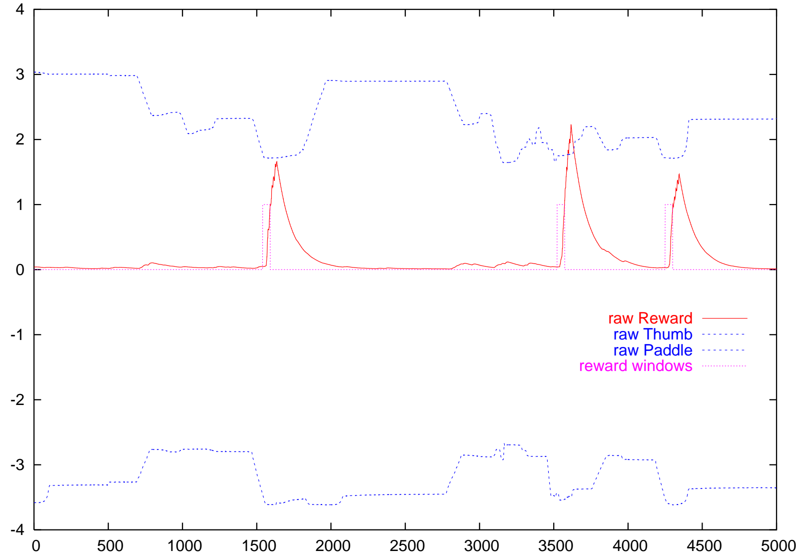


Figure 7.5: Recorded hand joint positions $\vec{\theta}$, raw reward R, and resulting reward windows, as a function of time (50 samples per second), while training the robot to point its finger.

In the spirit of the toy robot example of Section 5.1, we can now train the robot to point its finger. Due to random mover activation, the hand moves through its range of gestures, occasionally pointing. The robot receives positive feedback from repeated squeezes to the hand. Thus, by lovingly squeezing when the finger is pointing, we can tell the robot that "pointing is good".

Figure 7.5 shows data collected by an action-modeller while the robot was being trained. The modeller collects samples $(\vec{\theta}, R)$ of joint angle and reward and then analyzes them in two minute intervals. The one-second time windows preceding the rewards (illustrated as magenta pulses in the figure) are taken as the samples which correspond to the reward joint configuration. By comparing the distributions of rewarded and unrewarded samples, the modeller decides if the reward is actually correlated with the recorded joint angles. If not, then the robot was probably rewarded for activity in some other part of the system, so the training set is discarded. Figure 7.6 shows the cumulative distribution functions (CDF's) of rewarded vs. unrewarded samples for thumb and paddle joints in the example data. In this case, both axes are considered

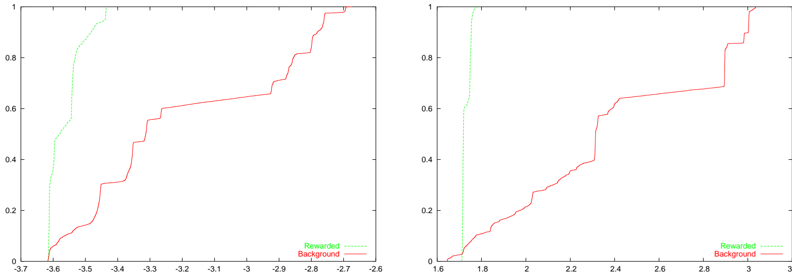


Figure 7.6: Comparison of CDF's for rewarded versus unrewarded samples, for the thumb and paddle joints, while training the robot to point its finger. The distribution of rewarded samples differs significantly from the distribution of unrewarded samples for both joints.

relevant.

If there are any relevant axes, then an action model is created. This model is a Gaussian model of the distribution of rewarded joint angles (Figure 7.7). The mean is taken as the prototype hand position — the goal position of the action. The variance is used in deciding if the action is equivalent to any new actions learned later on.

Once the action model is registered, an actor module is spawned. This changes the behavior of the hand. Previously, the fingers and paddle just moved around with random velocities. Now, in addition to that, the hand moves to a specific pointing configuration quite regularly, once every 10 seconds on average. This is due to the random activation of the actor module.

At this point, the training cycle repeats itself. The action modeller makes more observations, and tries to learn new actions; if successful, it spawns new actors. Figure 7.7 also includes the mean and variance of a second model corresponding to a "grasp" action. Once this model is learned, the robot's hand begins to alternate between pointing, grasping, and random motion in between.

Pointing On Cue

The new pointing and grasping actions occur regularly, yet still randomly. However, when the actor modules for these actions are spawned, the *proto-trigger-modeller* also spawns trigger-modeller modules for each, which try to learn appropriate contexts in which to activate the actors. In the toy-robot scenario, these trigger-modellers are only sensitive to one sensory source, the vector of tactile sensor readings.

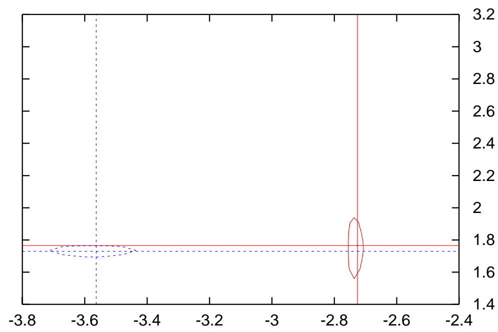


Figure 7.7: Pointing and grasping action models. The crosshairs indicate the prototype joint positions for each action (blue = pointing, red = grasping). The ellipses indicate the rewarded vs. unrewarded decision boundaries determined during training.

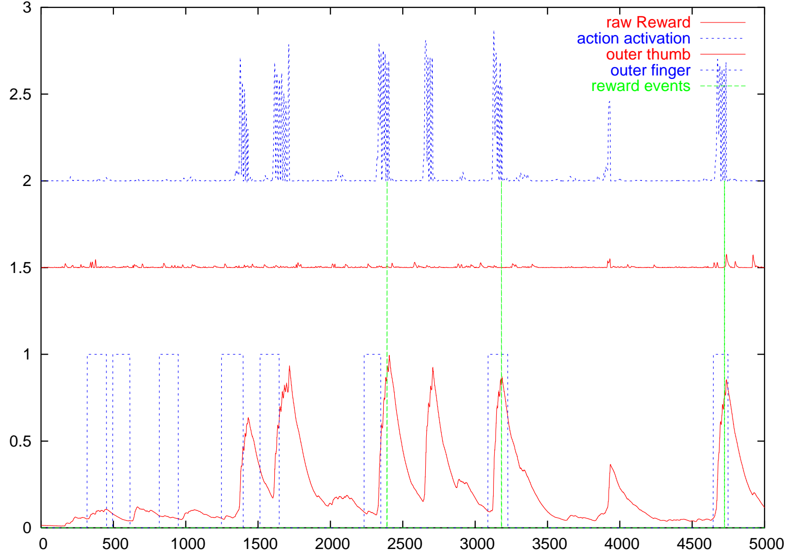


Figure 7.8: Data acquired while training the robot to point its finger in response to touch on the outside of the finger. Pointing actor activation A, raw reward R, and tactile sense for the outer-finger and outer-thumb sensors are shown. The vertical green lines mark the discrete reward events detected after filtering R.

The trigger-modellers capture time-series of samples of $(\vec{s}, A, R)$ triplets, where $\vec{s}$ is the tactile vector (the state of the potential stimulus), A is the activation level of an actor, and R is the reward signal. Figure 7.8 illustrates such a dataset from the trigger-modeller assigned to the pointing action. The goal of the modeller is to discover what values of the sensor vector, if any, are consistently associated with both the action and a reward.

The reward signal is filtered to determine discrete instants at which reward was received. Relative to the reward events, *reward windows* are constructed (Figure 7.9) which demarcate the set S_R of sensor samples which might be associated with the reward. Relative to the episodes of action activation, *action windows* are constructed which indicate the set S_A of sensor samples that might be associated with cues for the action. The intersection, $S_R \cap S_A$ is the set of samples which should be associated with both the action and the reward.

To decide if the sensor samples $S_R \cap S_A$ are actually relevant to

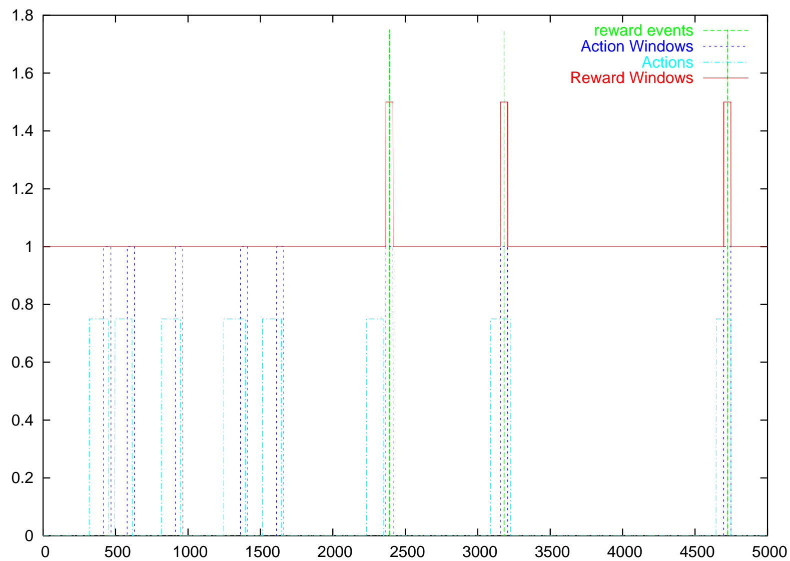


Figure 7.9: Reward and action windows for the pointing-trigger training session. Not every instance of the action was rewarded; however, every reward event appears to be associated with the action.

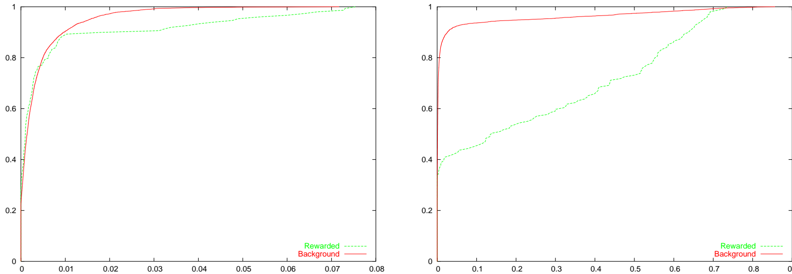


Figure 7.10: CDF comparisons for two components of the tactile sense vector. Each graph compares the "stimulus" samples (those occurring in conjunction with both reward and action) to the "background" samples. The left graph, for the outer thumb sensor, shows no significant difference in distribution. The right graph, for the outer finger sensor, does display a significant difference. Only the outer-finger sensor will be used in detecting the stimulus.

cueing the action and receiving reward, the distribution of those samples is compared component-by-component with the distribution of the rest of the samples (the "background"). Figure 7.10 shows the cumulative distribution functions (CDF) of stimulus vs. background samples for two of the tactile sensors, the outer-finger and the outer-thumb. The thumb sensor shows no significant difference between stimulus and background sets; it is dismissed as irrelevant. The finger sensor does show a significant difference and becomes the only component used in the stimulus model of the trigger. If none of the sensors had been relevant, the modeller would have decided that there was no interesting training data, thrown it away, and started over from scratch.

Now that it knows that the outer-finger sensor is indeed correlated with action and reward, the modeller models the stimulus $(S_R \cap S_A)$ and background $\neg(S_R \cap S_A)$ sets as Gaussian distributions. This model is saved in the registry and a *trigger* module is spawned. The trigger module continuously monitors the outer-finger tactile sensor and calculates the probability $p(T|\vec{s})$ that it corresponds to a previously-rewarded stimulus. When that probability breaches a threshold (0.5), then the trigger will activate the pointing action. Figure 7.11 illustrates the stimulus and background distributions learned for this task, and the resulting stimulus probability function. Pointing will be triggered whenever the outer-finger sensor values exceeds ~ 0.38. From the background distribution, we can see that that sensor is usually not

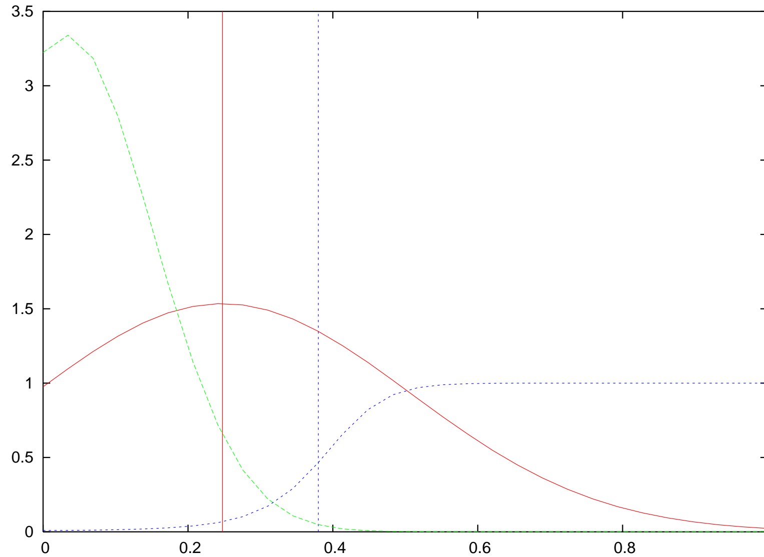


Figure 7.11: The stimulus model learned for triggering the pointing action. The green and red curves show the stimulus and background distributions, respectively, of the outer-finger tactile sensor. The red line indicates the stimulus mean or prototype. The blue curve is $p(T|\vec{s})$, the probability of a sensor value resulting in a stimulus, and the blue line is the resulting decision boundary. Sensor values greater than ~ 0.38 will trigger the pointing action; values beneath that will not.

squeezed and thus the background mean (and the variance) is low.

Figure 7.12 illustrates the state of sok space after all this learning has taken place. New to the scene are actors for pointing and grasping (connected to the hand controller), the pointing trigger, and a variety of modellers. As the robot ages (via its developmental age), the random activation of the actors will decrease, until eventually they will only be activated by triggers. The grasping actor does not yet have a trigger. If it does not acquire a trigger within a short time ($\sim$30 minutes), then it will be purged from the system, along with any trigger-modellers associated with it. Of course, in the future, the robot could always be taught a new grasping action.

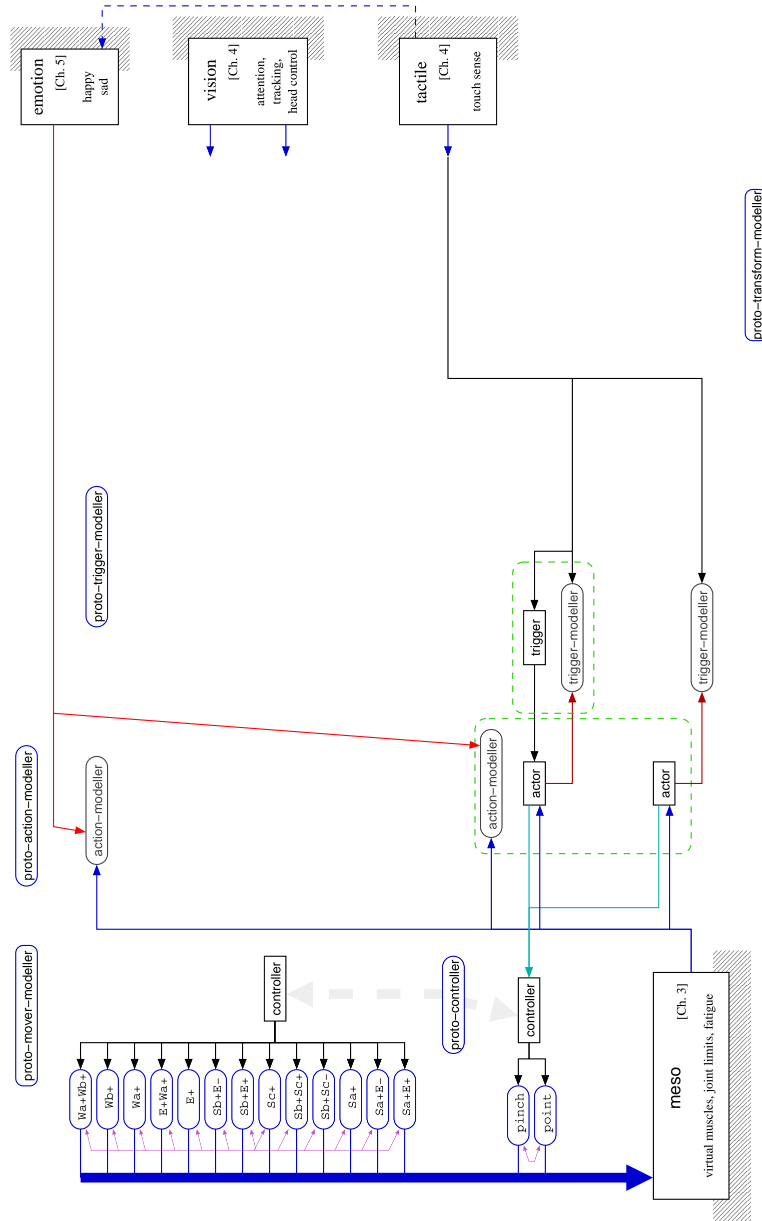


Figure 7.12: State of the system after learning how to move and being taught to point in response to touch.

7.3 "It's a red ball! It's a green tube!"

The same mechanisms which enable Cog to learn hand gestures also apply to learning static arm postures. When a controller for the arm joint angles appears in sok space, a position-constant action modeller is also spawned for the arm. By rewarding Cog (squeezing its hand) when the arm is in some configuration, it will learn that moving to that configuration is a worthwhile activity and will create an actor that randomly repeats that motion, again and again. This uses all the same algorithms and types of modules as with the finger-pointing in the previous section, only the connections are different and the dimensionality is higher.

In the finger-pointing, the hand initially moves completely randomly, due to random activation of the hand mover modules. This causes the hand to explore its state space, and gives the trainer an opportunity to reward it when it comes across useful configurations. It turns out that for learning arm postures, the random activation of the arm movers tends to make training more difficult. The state space of the arm is much larger than that of the hand, and the random movement explores it quite slowly. Instead of waiting hours for the arm to spontaneously "do the right thing", it is far easier for the trainer to just grab and move the arm to the desired position and then reward the robot. The action-modeller doesn't care how the arm got there. If the arm decides to move itself at the same time, though, it will compete with the trainer. Thus, it is beneficial to have the random activation rate of the arm movers *decrease* after the arm controller has been created.

Figure 7.13 shows three arm postures taught using this technique: outward, forward, and across. The figure illustrates the prototype goal positions of the respective action models. Once the robot has been taught a few postures, it begins moving among them (due to random activation of the actors). If some postures are located at the periphery of the useful workspace of the arm, then the actors will actually do a better job of exploring the workspace than the randomly-activated movers do.

Triggering

The next step is to train the robot to move in response to a stimulus. In this case, a visual stimulus is used. The vision system outputs a 6-element feature vector describing the target image of the tracking system. This vector is identified as a state parameter to pamet (by

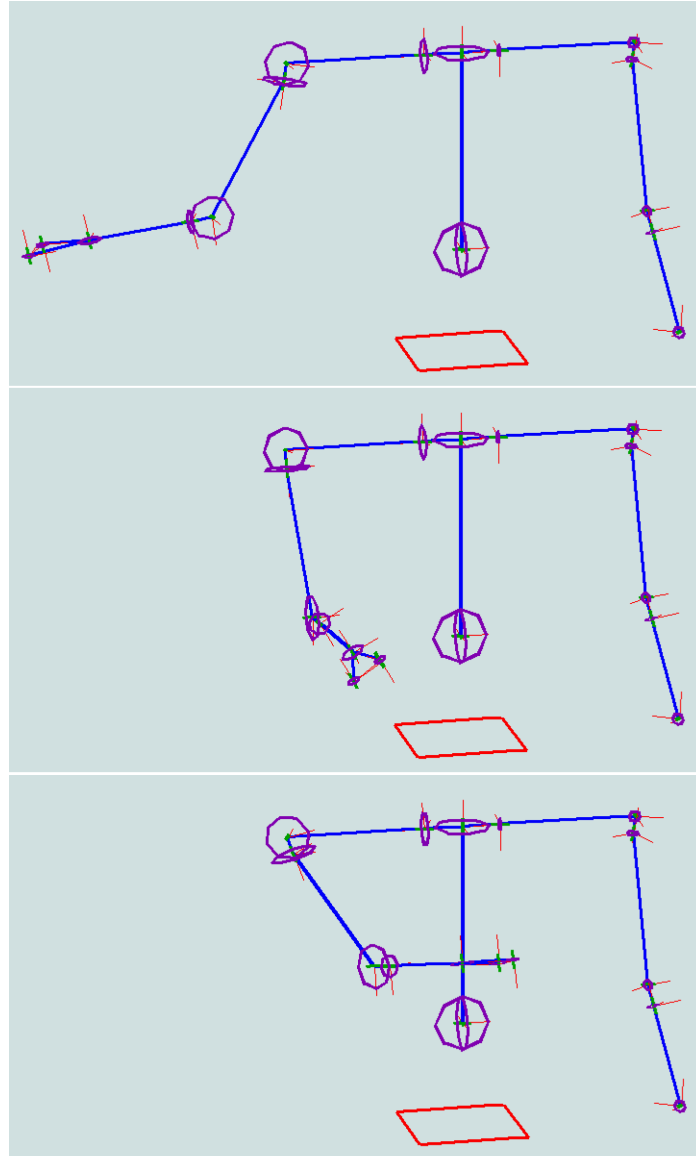


Figure 7.13: Prototype postures of three position-constant actions for the arm: “outward”, “forward”, and “across”. The robot was taught by repeatedly moving its arm to a desired position and rewarding it. Once one posture was acquired (and the robot began moving there spontaneously), the next one was taught.

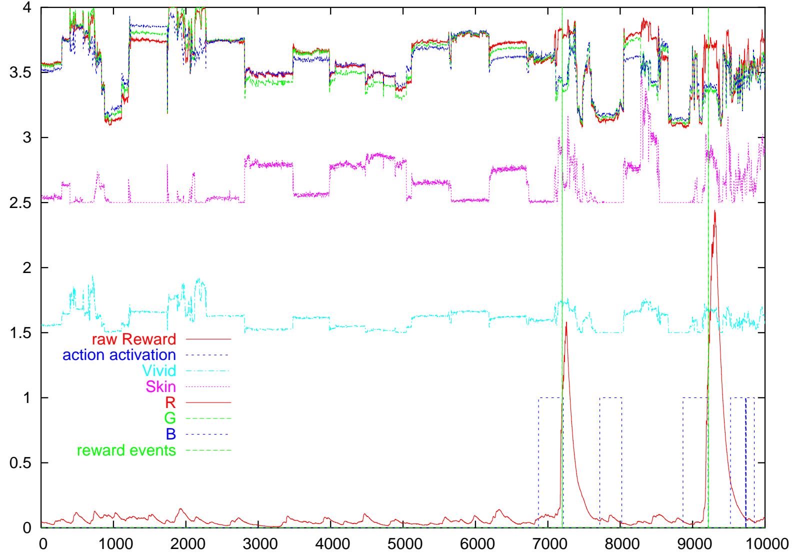


Figure 7.14: Data acquired while training the robot to reach outward in response to seeing a red ball. Actor activation A, raw reward R, and all components of the visual target feature vector $\vec{s}$ are shown. The vertical green lines mark the discrete reward events detected after filtering R.

its hard-coded name), and every actor is assigned a trigger-modeller that tries to find rewarding visual stimuli in terms of this vector. Figure 7.14 shows data acquired while training the robot to reach outward in response to seeing a red ball. The corresponding reward and action windows for that session are shown in Figure 7.15.

The training paradigm is to wait for the robot to execute the action, and then simultaneously present the stimulus and give it a reward (hand-squeeze). In other words, the windows for determining the stimulus are based on the instant of the reward. In the initial implementation, these windows were based on the onset of the action. This corresponds to a training paradigm of presenting the cue to the robot when it begins the action, and rewarding it when the action is complete.

The onset-based method seems like the most sensible and appropriate. However, a couple of features of the robot conspire to make it extremely difficult to use in practice. First of all, due to the wobbliness in the robot's arm movements, it is difficult at the onset of an

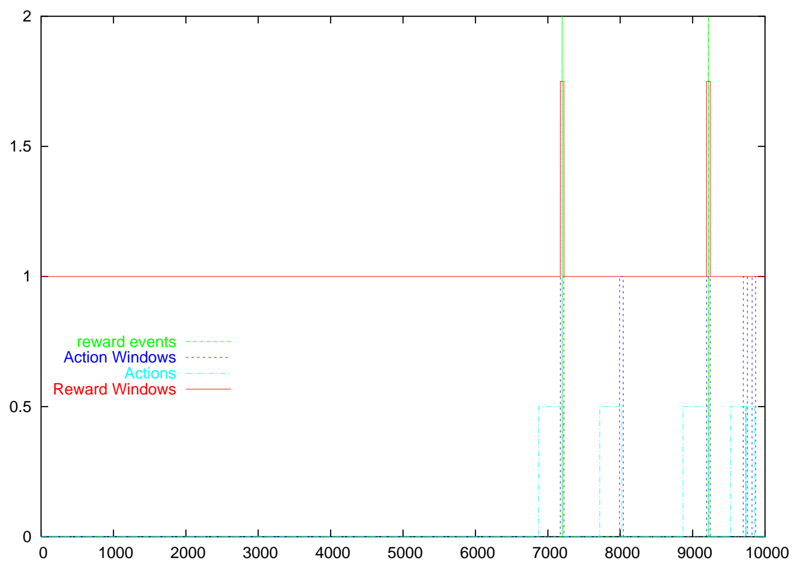


Figure 7.15: The reward and action windows for training the robot to reach outward in response to seeing a red ball. Note that only a couple of instances of the action were rewarded; presumably, the red ball was not presented to the robot during those other instances.

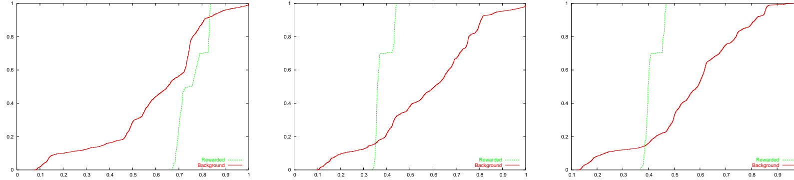


Figure 7.16: Comparison of stimulus and background CDF's for the red, green, and blue components of the visual feature vector. All three components (and skin-tone and vividness as well) are deemed relevant.

arm movement to predict where that arm movement is going to end up. Some movements are indistinguishable at the onset in any case: if the arm is in the "outward" posture, movements to the "forward" and "across" postures are identical until they stop. Second, the vision system is always tracking something, but it has no deeper concept of a target than a small image patch, and the feature vector is just the moment-to-moment description of that patch. Until a red-ball is stuck in Cog's face, the last thing it tends to focus on is the trainer's head, when the trainer moves close enough to reward the robot. If the trigger-modeller is searching in time for a consistent feature vector following the action onset, more often than not it will just decide that the best stimulus was due to the trainer's head, not the ball that appeared next.

All-in-all, it was just too cumbersome for the trainer to keep out of Cog's sight, figure out what action was starting, get the vision system to attend to the desired target, and then finally reach over to squeeze that hand after the action was complete. It was much easier to constrain the timing of the desired stimulus to accompany the reward, and have both occur in some window near the completion of the action.

For the red-ball training session described above, Figure 7.16 shows comparisons of the stimulus and background distributions for the red, green, and blue components of the visual feature samples. All three are deemed relevant to determining the stimulus (the sight of the red-ball). From the CDF's, one sees that the background distributions were fairly uniform (precisely uniform would translate into a straight diagonal line), whereas the stimulus components were much more sharply defined. This assessment is borne out in the decision-boundary of the red-ball trigger, shown in the $R'G'B'$ subspace only in Figure 7.17. Feature vectors falling within the boundary of the ellipsoid will trigger the "outward" arm action.

A second trigger was also trained in this experiment, this time to

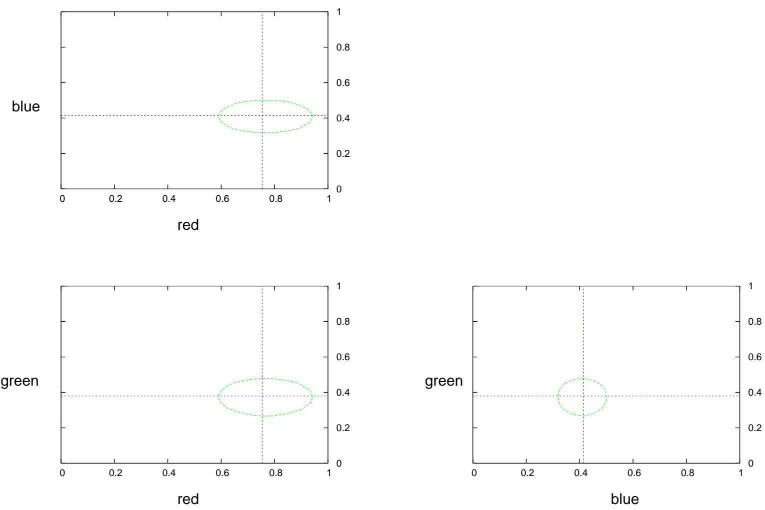


Figure 7.17: The stimulus decision boundary of the red-ball trigger, shown projected into the $R'G'B'$ subspace of the feature vector (and evaluated at the prototype mean of the skin-tone and vividness components). When the feature vector falls within the ellipsoid, the trigger will activate the outward-reach action. The crosshairs mark the prototype stimulus vector in the trigger model.

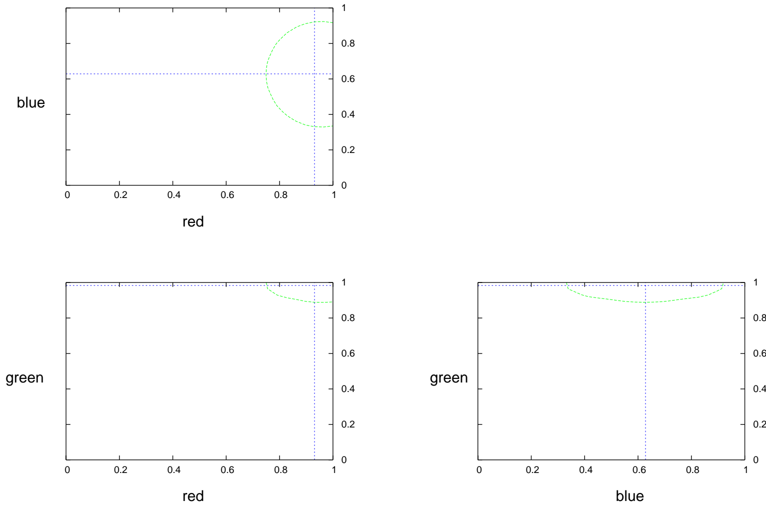


Figure 7.18: The stimulus decision boundary of the green-tube trigger, shown projected into the $R'G'B'$ subspace of the feature vector (and evaluated at the prototype mean of the skin-tone and vividness components). When the feature vector falls within the ellipsoid, the trigger will activate the outward-reach action. The crosshairs mark the prototype stimulus vector in the trigger model.

activate the "forward" action in response to seeing a bright green tube (another one of the toys in Cog's toybox). The $R'G'B'$ decision boundary for that trigger is shown in Figure 7.18, and as one would expect, the ellipsoid is shifted toward a high green value.

It is worthwhile to take a peek at what the "outward" trigger-modeller was doing in the meantime. Figure 7.19 shows the dataset captured by the "outward" modeller while the "forward" trigger-modeller was learning about the green tube. Several reward events were registered, but none of them was near enough to the endpoints of the active action intervals (of which there is only one) to be considered coincident with the action. Since the "outward" action was never rewarded in this dataset, the dataset was discarded.

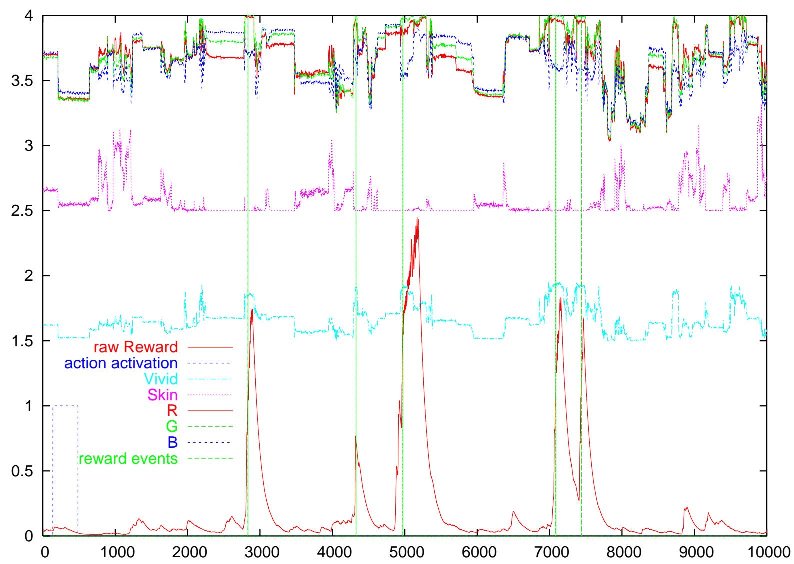


Figure 7.19: Data acquired by the outward-reach trigger modeller while another action/trigger combination was being trained. Several reward events occur, but none of them is close enough to the endpoint of the (single) activity period of the outward-reach actor. This dataset was discarded by the modeller.

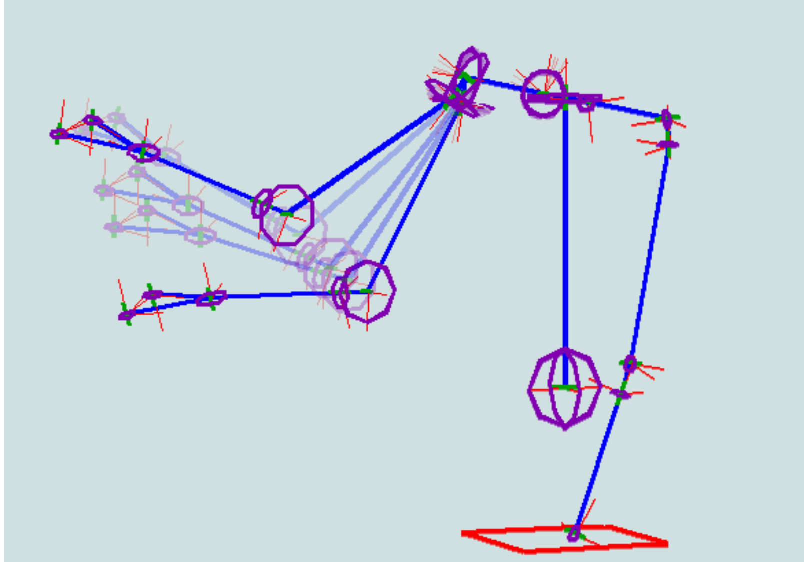


Figure 7.20: Learned arm postures (and any other actions) can be modified over time. The “forward” arm action is shaped into a “forward and up” posture by tugging the arm a bit higher and rewarding it every time it moves to that position. The associated action-modeller will adjust the prototype posture in the model (in several stages) instead of creating a brand new model.

Shaping

Recall that action-modellers continue to manage the actions they have created. Instead of spawning a new model in response to new training data, an action-modeller may choose to refine an existing action. This allows a trainer to shape and modify an action over time. Figure 7.20 shows the result of such a shaping activity. The lowest posture shown is the original “forward” posture discussed previously. Over a ten minute period, every time the robot moved to that posture, the trainer grabbed the arm and stretched it a little bit higher (i.e. “encouraged” it), and rewarded the robot. Since the new posture was similar enough to the original, the modeller associated with that action simply adjusted the action model. Over that time period, the prototype posture was modified four times, culminating in the highest posture shown in the figure.

Since only the action model was modified in this exercise, and the

actor and its connections were unchanged, the previously-learned trigger still applies. In response to seeing the green tube, the robot will now move its arm to the new, higher "forward" posture.

7.4 Reaching Out

A fourth type of modeller, the transform modeller, opens the door to a whole new class of behaviors. As discussed in Sections 5.3.5 and 6.4, if two state parameters have a functional relationship, a transform modeller can learn this relationship; the resulting transform model and transformer module can act as a predictor, classifier, and controller.

My original intent was to apply this notion to the specific case of a kinematic coordinate transformation function, in particular, the transformation between two representations of the position of the right hand: as joint angles $\vec{\theta}$ of the arm, and as the gaze angles $\vec{\Gamma}$ of the hand when it is a visual target. If such a transform model is acquired, then it could be used to perform two important operations. One, it could add an element of visual state, determining whether or not the eyes are looking at the head at any particular instant. Two, it could be used to control the position of the hand in space of gaze angles, i.e. head-centered world coordinate space. This would lead directly to a reaching behavior. If the eyes are focused on a target at some gaze angle $\vec{\Gamma}_0$, reaching to that target amounts to moving the hand so that its gaze angle $\vec{\Gamma}$ is equal to $\vec{\Gamma}_0$. Once the transform-moderated controller were in place, this action could be implemented by a position-parameter action model.

Much like the mover models, the appropriate transform model should be learned automatically by a modeller observing the data streams of arm joint angles and visual target positions. In practice, a number of factors prevented this. The kinematic model built into the vision system, used to calculate the gaze angle of a visual target from its retinotopic position and the head/eye joint angles, was not accurate enough and in particular did not account for parallax errors. Just locating the hand was a noisy operation. The motion detection filter was tuned so that the tracker would fixate on *some part* of the hand, but it could be anywhere on the hand, and sometimes on the wrist or arm, too. The visual size of the hand varies with its distance; when the hand is close, there is a lot of variance in the recorded hand position. Finally, the random arm movement produced by random mover activation tended to be out to the side of the robot (the bulk of the arm's workspace), not out in front. Thus, it was rare for the eyes to ever actually *see* the

hand!

In order to experiment with the learning techniques and make some assessment of how the transform models work, a set of data was recorded in a contrived scenario. Instead of allowing the head and eyes to track targets, the head and eyes were fixated, and only the raw retinotopic coordinates $\vec{x}$ of the target were considered. This eliminated any errors due to the vision system's kinematic model of the head. An eye-hand-arm transform modeller was set to recording $(\vec{\theta}, \vec{x})$ samples while the arm was manually moved around within the visual field. The saliency filters, attention module, and tracker otherwise operated normally, sometimes locking on to and following the hand or arm, other times focusing on other random visual stimuli. Only the four most significant joint angles (shoulder and elbow joints) were used. As the samples were recorded, a human observer manually labelled them as "tracking the hand" ($\gamma = 1$) or "background noise" ($\gamma = 0$). The result of this manual labor was ten datasets of 5000 $(\vec{\theta}, \vec{x}, \gamma)$ samples apiece, of which 65% were labelled as "tracking".

Comparing Two Models

Two different types of models were trained to estimate $\vec{x} = f(\vec{\theta})$ (Appendix B). The first is a generic non-parametric, memory-based model with a smoothness constraint — each estimate is essentially a locally-weighted average of the samples in the training set. The second model is a more constrained "semi-parametric" model consisting of terms which are the sines and cosines of the joint angles. This model is capable of exactly representing a kinematic coordinate transformation, however it has many more parameters than are necessary for a compact model of the kinematics. The performance of each model is assessed using 10-fold cross-validation; models are trained on combinations of nine out of the ten data sets, and tested on the tenth.

With optimal smoothing parameters and learning rates, and training exclusively on the good samples hand-labeled as "tracking", the RMS (root-mean-square) prediction error of the memory-based model is 21.3 ± 2.7 pixels; for the parametric model, it is 19.9 ± 2.5 pixels. Both models appear to do pretty much the same job, and it's not very good: the full range of the retinotopic data is $[0, 127]$ and the standard deviation of the training data is 36.3 pixels.

Figure 7.21 shows how the RMS-error performance of each model degrades as it is exposed to "background noise". In these trials, a percentage of the training $\vec{x}$ samples are replaced with spurious synthetic data, tailored to act like the acquisition of random visual targets. The

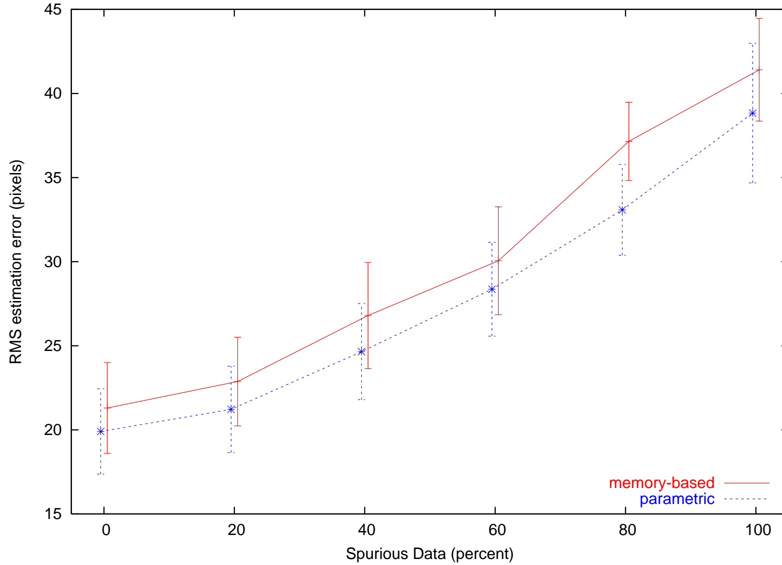


Figure 7.21: Training performance degradation of transform models in the face of noisy data. The graph shows the RMS-error of a memory-based and a parametric model of $\vec{x} = f(\vec{\theta})$. The $\vec{x}$ training data is corrupted with a percentage of synthetic spurious samples which are not correlated with $\vec{\theta}$. Both models perform similarly. As the percentage increases to 100%, their performance drops to no better than random, as expected.

models are still tested with unadulterated data. As one expects, the performance of both models degrades to essentially random when the training data is 100% noise.

In this test, the parametric model performs consistently better, but not significantly better. However, the two models are not equal. Figure 7.22 shows the *correlation* of the test set $\vec{x}$ with the models' estimates. Here it is apparent that the parametric model does a much better job of locking on to the $\vec{x}$ data which is actually functionally related to $\vec{\theta}$, even as it is flooded by more and more spurious data. In the presence of 80% junk data, the correlation of the parametric model's estimate with the measured signal only drops to 0.64 (from 0.82), whereas the memory-based model drops to 0.19 (from 0.79). This corresponds to the situation where the eyes are only tracking the hand 20% of the time, which is hardly unreasonable. If the hand-tracking data were it-

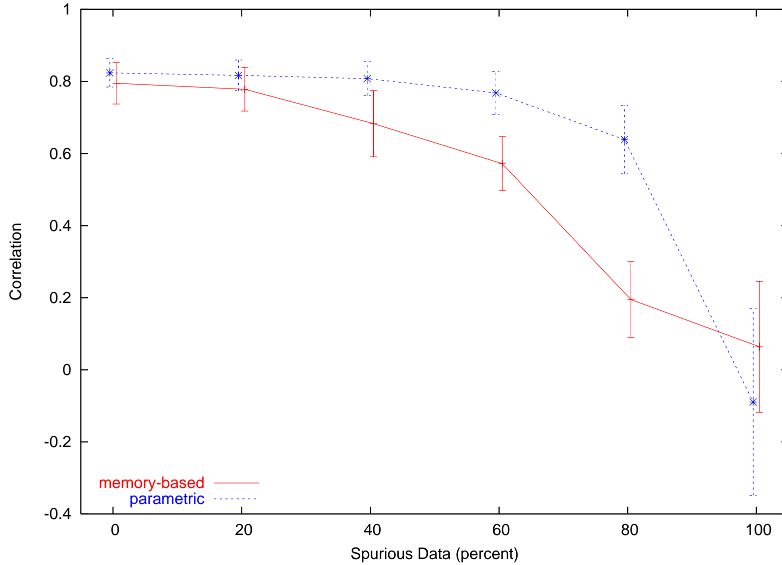


Figure 7.22: Ability of transform models to lock on to signal in the face of noisy data. The graph shows the performance of a memory-based and a parametric model of $\vec{x} = f(\vec{\theta})$, as measured by the correlation of the $\vec{x}$ estimate with a test set. The $\vec{x}$ training data is corrupted with a percentage of synthetic spurious samples which are not correlated with $\vec{\theta}$. The parametric model maintains better correlation than the memory-based model in the presence of much more junk data; it appears to "lock on" better to the underlying signal.

self better (more precise localization of the hand), then the RMS-error performance of the parametric model would certainly improve, and this ability to lock on to the kinematics might allow it to discover and model the eye-arm correlation completely automatically.

This is also an argument for populating pamet with very specific models tuned to specific classes of phenomena. A true parametric model of the arm kinematics, with no more free parameters than necessary, would presumably lock on to the good training data in the face of an even higher percentage of background noise.

7.5 The Final Picture

Figure 7.23 shows a schematic of the system after all this learning and training has taken place. In contrast to Figure 7.1, the system has acquired quite a bit of new, explicit structure linking the sensory systems to the motor systems via learned behaviors.

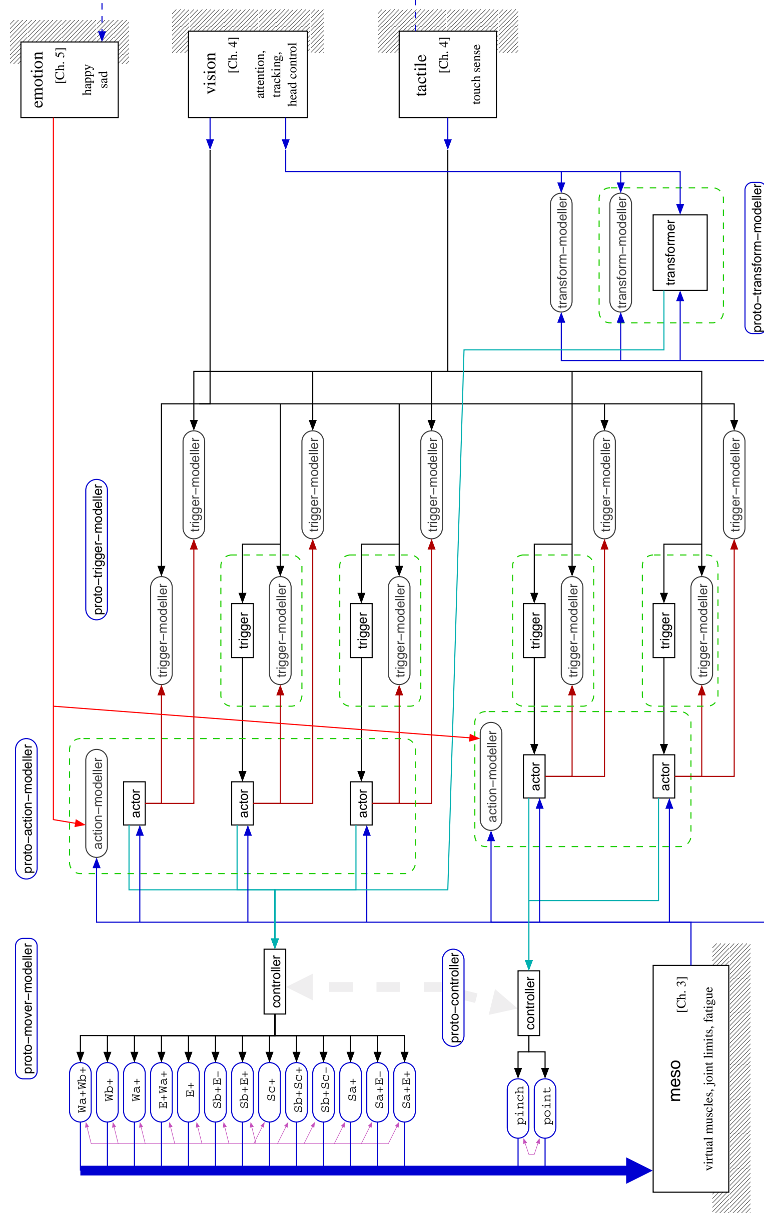


Figure 7.23: Schematic of the system after the robot has learned to point its finger, learned to move its arm to various postures in response to visual stimuli, and learned how to move its hand in the visual field.

Chapter 8

Looking Back, and Forward Again

This dissertation describes a system which enables a humanoid robot, Cog, to be taught simple behaviors by a human trainer. The system begins with little explicit structure beyond basic sensory and motor systems grounded in the robot's hardware. However, it does have rules for creating structure. By exploring and interacting with the world, it builds a small hierarchy of knowledge on top of the initial foundation.

In many ways, this work seeks to be a synthesis of two projects described in the introduction: Drescher's schema mechanism [16] and Metta's Babybot [36]. Starting with raw sensory and motor primitives, the schema mechanism's simulated robot could explore its grid-world and learn progressively more abstract relationships about that world. However, the binary grid-world is too simple and perfect; the mechanism developed for it cannot be directly applied to a robot which exists in the real world. Babybot, on the other hand, is a real robot, which learns to look at and reach out to real objects. However, Babybot is prewired to learn precisely this task and the component faculties which lead up to it. Without enough sensory stimulation, Babybot may fail to learn this task, but no amount of training will ever make it learn to do anything else. The *qualitative* structure of what Babybot can and cannot do is predetermined by the connections of the very specific models which are coded into it. My own work is an experiment in combining the dynamic properties of the schema mechanism with the real-world constraints of Babybot.

8.1 Creating Structure: What pamet Can and Cannot Do

With its pamet, Cog can learn some simple motor behaviors triggered by simple sensory cues. It can move its hand and arm to static postures, and it can almost move its arm to where it is looking. The triggers can be tactile — touch to a part of the hand — or visual — gazing at something of a particular color. Some learning is automatic, such as the acquisition of mover models to describe the motor system. Actions and triggers, on the other hand, are trained by a human teacher who rewards the robot for doing the right thing. Both the actions and triggers can be slowly refined over time, modifying the response or the stimulus while maintaining the causal connection between them.

Although the words "action", "state", and "reward" evoke reinforcement learning [28], in this system the words are used with a different emphasis. The classic reinforcement learning scenario involves sets of discrete states and actions; the problem is to decide on a policy for which actions to take in each state in order to maximize a reward (which might only be received once, at the end of a journey through the state space). In contrast, the learning in actors and triggers is really directed at a precursory problem: determining what the actions and states should be. An action model, instantiated by an actor, defines a discrete action in terms of a class of manipulations of a controllable system parameter. A position-trigger model defines a discrete state in terms of a fuzzy region of some portion of a continuous parameter space. In this context, the reward signal is not really used as the long-term optimization criterion. The positive reward signal is better thought of as a perceptual cue, with certain learning mechanisms hard-wired to correlate that specific cue with other perceptual data. Once an action or trigger is learned and well-established, the robot should not expect to continue to receive direct reward for further instances of that action or trigger. A child shouldn't expect a gumdrop every time she wipes her nose, but she should keep wiping her nose!

In pamet, triggering stimuli are modelled as unimodal Gaussians, but the system can acquire a multimodal stimulus as long as the training can be decomposed by presenting one mode at a time. The system will then learn a separate model for each component of the complete distribution. What the system can't do is acquire a stimulus which is a conjunction of two different sensory modalities (i.e. tactile *and* visual, two separate state parameters). This could be implemented, though, by outfitting each trigger-modeller with an arbitrary number of inports $\vec{s}_i$ and letting them be treated as one big stimulus vector $\vec{s}$.

When a trigger model is created, a decision is made as to which components of the stimulus vector are relevant. If the model is later refined, the distributions of those components are updated, but the choice of components is never changed. A new model with even one more or one less component than an old model is considered completely different and would never be used to refine the old model. This behavior could possibly be changed by keeping and comparing the relevance statistics (the difference between stimulus and background CDF's, Section 6.3); if a component of the old model was just under the relevance threshold, and is just over it in the new model, the two models could be considered similar. This same limitation (and possible solution) applies to action models as well.

Any one action or trigger modeller can only learn one thing at a time. For example, if an arm action modeller sees reward for two different postures during the same training session, it will either conflate the two (and learn the average posture), or ignore them (and discard the data). This also means that the robot cannot be taught independent arm and hand postures simultaneously. Two action modellers are involved (one for the hand, one for the arm), but both see the same reward signal, and if the reward is mixed between modellers, they will be confused. On the other hand, it *is* possible to simultaneously train two trigger stimuli, as long as the trigger modellers are assigned to different actors. The learning is tied to the execution of the actions; as long as the actors are activated at different times, they will remove the ambiguity in the reward signal.

The transform modellers will not be able to learn an eye-hand-arm transformation without a more elaborate vision system or, perhaps, a more constrained coordinate transform model. However, once working automatically, the same mechanism would apply to both right and left arms. If transform models connecting joint angles to eye-gaze angles were learned for both, then the positions of the right and left hands would share a common representation. This would immediately allow for coordination of their motion. A simple implementation of "hand-clapping" could be "adjust the joint angles of each arm such that the hands move to the same location in eye-gaze space". Such a behavior does not actually require any visual input. Once the common representation was established via interaction with the vision system, the eyes could just as well be disconnected.

8.2 Unsatisfying Structure: Hacks

In a project involving machine learning, the inevitable research compromise between "principles" and "finishing" means that structures creep in as crutches which help to get the job done. One hopes that these crutches will be discarded in further revisions of a project, so it helps to point out where they are.

Meso Movers

The *meso mover* modules were implemented as a simple way to parameterize motion and to get the robot moving when first powered up. However, they are *too* simple and are only usable if a number of features and complexities of the motor system are ignored.

First, the movers activate virtual muscles with fixed, hard-coded stiffness values. If these movers are the only link between pamet and meso, then there is no way to modulate muscle stiffness throughout movements. All the arguments for implementing controllable stiffness (task-specific tuning, increased efficiency) are moot if the controls are never adjusted!

Second, the action of a mover is implicitly linear. Mover models assume a linear relation between mover activation and an affected state parameter, i.e. joint angle. Even without the measurement problems described in Section 7.1, such a relationship will only hold for meso movers if all the virtual muscles involved are *single-joint* muscles, in which the muscle length set-point translates into an equilibrium *point* in joint angle. For *multi-joint* muscles, the length set-point defines an equilibrium *surface* in joint space, which cannot be represented by the simple mover model. Again, if multi-joint muscles are never going to be activated by movers, then all the work that went into implementing them is for naught.

Third, the split of movers into "arm" and "hand" groups is rather artificial. Recall that there are two hand movers and thirteen arm movers. Members of each group have mutual inhibition interconnections so that only one member of each group is ever active at a time. Most movers affect more than one muscle, but no muscle is driven by movers from different groups. If the torso and the left arm were brought on-line, presumably each would have its own group of movers as well. The result of such a division is that independent controller modules develop, one corresponding to each group of movers, and this partitions the joint angle space into more manageable chunks, e.g. hand joints, right arm joints, torso joints, etc.

What is needed is a representation which can capture the locality constraints of the muscles and joints without making hard divisions. The fingers and the wrist are close together, kinematically, and thus will often need to be used in synchrony. Imagine pushing a button: the whole arm needs to be positioned in roughly the right place, but then the finger and maybe the wrist handle the details of the action. Likewise, shoulder joints and torso joints affect each other: when lifting an object, changes in the torques produced by the shoulder also need to be accounted for by counterbalancing forces in Cog's hips (or, for a person, her back). However, torque changes in the wrist which are small enough to not affect the shoulder probably won't affect the torso either.

In terms of pamet, I imagine that such a representation would lead to action-modellers which observe and model overlapping subsets of the joint angles of the robot: one looking for utile postures of hand only, one for hand and wrist, one for wrist and upper arm, etc.

Finally: the meso movers don't even do a very good job of making the robot move randomly, especially with respect to the arm. Since the movers drive the muscles with random velocities, the resulting motion amounts to a random walk in joint space. At any one moment, at least one of the arm joints tends to be driven against its mechanical limits; it stays that way until all thirteen movers happen to become deactivated and the arm is allowed to relax. A secondary issue is that much of the actual workspace of the arm is out to the side of the robot, out of the range of the vision system. A better scheme for random movement would bias the motion to the front of the robot, the region where most interaction with objects and people will occur.

Actor Timing

Actors are little position-controllers: when activated, they produce a stream of appropriate velocity commands to drive a state parameter to a goal value. They are full of hard-coded constants which should eventually be removed or parameterized. For instance, the overall *speed* at which the parameter is moved is regulated by an arbitrary constant which has been tuned to result in reasonable hand and arm motions. This constant will not necessarily work for any other actuators or parameters. Also, this constant should not be constant — at some point, the robot should be able to decide how fast or slow a movement (or other parameter adjustment) needs to be.

When an actor is activated, it remains activated until the target parameter is within some threshold distance to the goal value, or un-

til a timeout occurs. The threshold distance is currently an arbitrary constant which could be reasonably parameterized based on the variance of the goal prototype in the action model. Likewise, the timeout is an arbitrary constant of a few seconds duration. Since an actor doggedly pursues its goal, even when initially activated for only an instant, the timeout is necessary to allow it to give up on unobtainable goals (e.g. if the arm is physically restrained, it will not be able to move to the "forward" posture). The timeout could be eliminated if goal *pursuit* were pushed back into the triggers or some intermediary module, which would then be in charge of keeping the actor activated as long as necessary.

Another set of time constants in actors control the random activation rates and expiration of the actor. They have been hand-tuned to make training the robot, in the scenarios presented in Chapter 7, a reasonably snappy experience. It is not clear how these constants should change as the system scales since the random activation rate, the number of competing actors, and the trade-off between exploration and exploitation are all tied together.

Vision Black Box

As it stands, the vision system (Section 4.2) is a black box sensory system, providing only outputs describing where it is looking and what it is looking at. It operates in a purely reflexive manner, semi-randomly picking targets to gaze at in accordance with the hard-coded saliency filters. This box needs to be opened up so that the gaze can be controlled by learned behaviors.

Although head-eye coordination has been made adaptive in previous work on Cog [32], in this vision system, head-eye coordination is computed using the measured kinematics of the head. It would be worthwhile to try reimplementing the tracking system using transformer modules which learn the transform between retinotopic and head motor coordinates.

The saliency filters and attention system should be exposed to pamet. This means that the weighting of the filters could be changed to suit the task at hand. Also, the output of the filters themselves would be available to the behavioral system, giving it the potential to determine better criteria for choosing what to look at.

Finally, the motion detection filter was tweaked specifically to make Cog's right hand a salient feature to the vision system. When something moves in the field of view, the motion detector reports saliency near the end of the moving region which is closest to the top-left corner of

the screen. If the moving object is the right arm, coming into view from the bottom-right, this targets the hand. Fortunately, this same heuristic also picks out the heads of people walking around the room, which is another appropriate target for Cog's vision system. In future work, these biases should at least be balanced so that, for example, the left hand is as salient as the right.

Hard-wired Tactile Reward

The tactile sense is hard-wired into the emotional system to produce positive reward when the hand is squeezed. In principle, this is a reasonable idea: pamet needs sources of innate primitive and negative reward which ground-out the behavioral decisions it makes. However, since this is currently the *only* source of positive reward, it makes the hand seem like a bit of a "magic training button".

This innate tactile reward percept needs to be refined, perhaps to more distinctly favor gentle, "soothing" hand squeezes over abrupt, sharp tactile sensations. Overall, the system needs more sources of innate reward, tied to other sensory modalities, such as detection of smiles and vocal encouragement ("*Good* robot, Cog!").

8.3 New Structure: Future Work

pamet is by no means finished. There are several features on the drawing board which should be implemented in the next cycle of development of this project.

Negative Reward, Inhibition, and Un-learning

Although meso provides two innate sources of negative reward (joint pain and muscle fatigue), and the emotional system's emo/sad module turns those into a global negative reward signal, that signal is not yet used anywhere by pamet. The positive reward signal is a cue to tell the robot when it is doing something it should be doing; likewise, the negative reward signal should be a cue that the robot is doing something that it should *not* be doing.

pamet needs modules for learning and instantiating models of *inhibition*. One manifestation could be a model which specifies regions of a parameter space to be avoided. The model could be used to send an inhibition signal to, say, a mover module which was driving the arm to its joint limits. As another example, a subsystem which gives the robot upright posture could be implemented as a cascade of two types

of inhibition. First, reaction to the joint pain signal causes the robot to avoid driving its hip joints to their limits, so the robot will not rest in a slumped over position. Second, maintaining any torso position besides fully upright and balanced requires a lot of force, and thus induces muscle fatigue. Reaction to the fatigue signal would favor movements which keep the torso upright.

A connection from an inhibitory model could also augment a trigger; the inhibition model would specify an overriding condition I under which a stimulus S should be ignored. This is different from just learning a model of a stimulus $S \cap \neg I$ in two ways. First, it may be easier to learn in two separate steps, first S and then I; the inhibiting context I might not even arise until much later in the robot's development. Second, the two contexts S and I may each be independent in their own right. The state represented by S might be valuable and used by other modules, even if one particular action should be inhibited. A common state represented by I might lead to inhibition of, say, three triggers, so it makes sense to learn it once in one place instead of modifying three other contexts simultaneously.

A further use of inhibition is *un-learning*: instead of repeatedly inhibiting a learned action or trigger, it may be better to just forget it. Already in pamet, an action model expires if the actor instantiating it is never explicitly activated; if no other module ever requires the action, then it is considered a mistake and is removed from the system. However, there is no mechanism for forgetting a trigger: once the robot learns to raise its arm in response to seeing the "green tube", it will always try to do that. If an action or trigger is consistently inhibited, however, it should probably just be removed from the system.

Feedback Channels in sok Connections

If two actors — connected to the same controller module — are activated simultaneously, only one will have any effect. Arbitration logic in the controller's drive inport will allow messages from the first active incoming connection to pass through; the drive signal from the second activated actor will be discarded (until the first actor is done). In the current implementation, these messages are discarded *silently*: the second actor will not be told that it is being ignored. The problem here is that this actor will continue to tell the rest of the system that it is active. A trigger-modeller observing the activity of the actor may then falsely associate some stimulus with it, even though the ignored actor is not having any effect on the behavior of the robot.

A solution to this problem is to implement a feedback channel in

sok connections, such that the sending process is informed when its messages are being rejected or when the receiver is under some type of inhibition. The snag in implementing such a solution is that a sok outport supports multiple outbound connections. If an outport is distributing a message stream to six different receivers, and only one of them rejects it, should the process be told that it is being ignored by the system or not?

Activation-Delay Triggers

In addition to the *position-triggers* already implemented in pamet, an *activation-delay-trigger* was suggested earlier. The stimulus for such a trigger is the activation signal from (or to) another module. The trigger fires after some delay following the stimulus activation. This delay is a learned parameter in the trigger model. Such a trigger would allow the system to learn sequences of actions. One trigger would fire, activating the first action and the second trigger. After a delay, the second trigger would fire, activating the second action, and so on.

Attention, for Learning

Learning in pamet is currently an anarchic process: each and every modeller is always observing its inputs, watching for correlations and trying to find something to learn. The modellers do not directly interfere with each other; however, it is possible for one modeller to latch onto some spurious correlation while the true target of a trainer's intentions is simultaneously being learned by another modeller. This leads to *behavioral noise* due to extra actions, etc., that get created and end up hanging around in the system. This is not much of a problem when the system is small, with few state parameters which are mostly orthogonal to each other. But as the system grows in size and complexity, there will be many more modellers and parameters and the problem will get much worse.

What the system needs is an attention mechanism which allows learning to be focused on a particular region of the behavioral space. If a trainer is teaching the robot to shake hands, it should be able to concentrate on the arm movement and the presentation of the trainer's hand, while ignoring the color of his hat. This could perhaps be implemented as a distributed inhibition — modellers would compete with each other for the privilege of learning, somehow based on how relevant their models are to the current activity in the system.

Online Learning

All of the modellers have been implemented using batched training: they record a large set of samples and then analyze the whole set at once to look for useful models. This is convenient from the researcher's point of view; because each batch of data is independent, it is easy to simulate and analyze the modelling off-line on recorded datasets. From the behavioral viewpoint, though, it is a nuisance. In trigger training, for example, there is a five minute window in which a trigger modeller is silently collecting data, followed by 30 or so seconds of analysis. If the trainer happens to engage the robot in the second half of a recording cycle, that cycle may not end up with enough significant data overall. Two minutes of training could be discarded just because the trainer was out of phase with the robot. If the learning used online algorithms without these arbitrary, invisible batch boundaries, training would be a more fluid and efficient process.

The action and trigger models cannot be trained with strictly online algorithms, since some historical analysis is necessary, e.g. the "reward windows" which precede action events. But, the modellers already record five minutes of samples at a time. This data could instead be kept in a rolling buffer. There would be a constant latency in model updates but no acquired samples would ever be wasted.

Revisiting Previous Projects

The best sources of brand-new directions for this work are the previous projects done on Cog. Much of the previous work explores particular faculties or subsystems which are important components in a humanoid robot. The trick would be to reimplement earlier work such that it stays true to the transparent, distributed design philosophy of pamet.

Work such as Scassellati's theory-of-body [45] would translate into new classes of models. Breazeal's work [7] could be transplanted as a much more significant emotional system, accompanied by innate social gestures. Fitzpatrick's project [18] would lead to models for "objects" and interactions with objects, as well as more robust visual primitives. Trying to up-end any one of these projects and integrate it into pamet's framework would require a lot of conceptual reorganization, probably worthy of another dissertation for another graduate student.

8.4 Unintended Structure

In this kind of project, we try to devise simple systems which can cope with a complex world. However, the interaction between the world and the system is sometimes more complex than we imagine. Even when something "works", it is not necessarily working the way we think it does. I present here one small cautionary tale.

As long as there is a bit of activity in the room, Cog's vision system keeps the robot looking around fairly randomly and uniformly. To the casual observer, the robot appears to be naturally looking at different things in the room. What is not obvious, however, even to the people who designed the system, is that this behavior requires that the floor is blue.

One night while working on the robot, the color-content saliency filter froze and stopped producing output. Since the code I was working on at the time only needed motion detection, I didn't bother to restart the color-content filter. Soon, however, the robot's gaze was fixated on the ceiling. It turns out that the color-content filter is necessary for the "natural around-the-room" gaze behavior of the robot, because it allows the blue floor in the room to exert a downward influence which counteracts an overall upward influence from the ceiling (exerted via the skin-tone filter). If Cog were pushed into a room without primary-color linoleum flooring, it would stare at the ceiling all the time.

The moral of this little story is that the real world sometimes has even more structure than one hopes for. Dependencies on such structure will inevitably creep into any design. The process of *developing* a system is often more organic than the system itself. It takes on a life of its own, and life has a way of exploiting all the structure it can find.

8.5 Round, Flat Structure: Flapjacks

I began this dissertation with the admission of my own pet dream for Cog: to create a robot which I could teach to make pancakes. Cooking pancakes is a relatively simple procedure by human standards, something I learned to do myself when I was only a few years old (with adult supervision, of course). However, learning this task from another person requires a lot of background knowledge and the ability to handle some fairly abstract concepts.

Let's break down the "make pancakes" task into bite-sized pieces:

1. Turn on the electric griddle; let it warm up.

2. Pour batter from pitcher onto the griddle.
3. Wait until first side is cooked.
4. Flip.
5. Wait until second side is cooked.
6. Serve.

Conceptually, the two simplest steps may be "4. Flip" and "6. Serve", although they are also the most mechanically complicated. The basic operation is "lift a discrete object and turn it over". To learn this by watching a human, the robot would need some ability to imitate third-person body movements. It would also need an understanding of object manipulation and the ability to distinguish the "pancake" from other objects, so that it could understand the actual *goals* of the arm movements and hone its performance accordingly.

Step 2, "Pour batter", is mechanically simpler than flipping a pancake. However, now the robot has to deal with a *fluid*, flowing freely under the influence of gravity. The speed of that flow is related to how the pitcher is being held and the amount of fluid remaining in the pitcher. The goal of this operation is to create a fluid disk of a certain size on the griddle. The robot will have to know how fluid flow relates to growth.

Steps 3 and 5 are both waiting, but for vague visual criteria rather than a concrete duration. Usually, in pancake parlance, the first side is done "when bubbles stop forming and rising to the top". To evaluate this, the robot needs a vision system capable of seeing and gauging the density of air bubbles in the batter, and the saliency of these bubbles has to be pointed out to the robot. Furthermore, the existence or even the density of the bubbles is not the important feature, but rather the *rate of change of the density.* It is almost impossible to imagine indicating such an abstract concept to the robot without some form of language, be it vocal or gestural. And for this, the robot will need to have already acquired that concept of *rate of change* via prior experience.

Finally, there is an implicit seventh step, "Repeat", in which the robot closes its training loop and makes a new pancake, tuning its technique and improving on the last one. How will the robot assess its performance? The human chef judges her skills by *eating the pancake*. The important qualities of pancake are its taste and texture, as experienced by the mouth. The visual appearance is secondary, important mostly as a predictive indicator of the taste. Other people eating

and judging the pancake will report on it in terms of taste and texture, possibly connecting back to cooking technique: "too crispy" or "undercooked". As the chef cooks more pancakes, she makes connections between changes in technique and changes in aspects of the taste, thus learning how to adjust her cooking style to achieve different results. The poor robot, however, does not eat its own pancakes! Even if the robot is given a differential performance report from human tasters — "better", "ok", "too dark" — and even if it tries to connect this to the visual appearance, the most important "state" of the pancake is completely hidden from its perception. This puts the robot at a great disadvantage in its training as pancake chef.

As this pancake task illustrates, the greatest difficulties in creating a human-level machine intelligence appear to arise from the *mundane*, the sheer volume of the interrelated everyday experiences which comprise our understanding of the world. Our ability to perform a single narrow task depends on an enormous breadth of "simple" knowledge. Human interaction and communication is based on shared experiences which are so very common that we rarely take notice of them. For a machine to seamlessly interact with and learn from humans, it needs to be able to participate in these pervasive but all-too-easily overlooked experiences.

Appendix A

Details of the Complex Coupling Model

The lower level of meso (Section 3.3) is a skeletal model, which handles the kinematics of the musculature simulation. The purpose of the skeletal model is to calculate two functions: $\vec{l}(\vec{\theta})$, the vector of lengths of the virtual muscles, and $\vec{\tau}(\vec{\theta}, \vec{F})$, the vector of joint torques. The inputs to the skeletal model are $\vec{\theta}$, the skeletal configuration expressed as a vector of joint angles, and $\vec{F}$, the vector of muscle forces provided by the muscular model (Section 3.4).

Of the two different skeletal models implemented on Cog, the "complex coupling" version treats virtual muscles as lines of force in space, connecting points anchored to different parts of the robot. To calculate the two functions, this model requires a kinematic description of the mechanical linkages in the real robot and a list of the anchor points of the virtual muscles.

Coordinate Frames and Transforms

The robot can be described as a tree of segments, or *links*, connected by revolute joints. Each link has an associated coordinate frame, and a kinematic description of the robot amounts to specifying how these coordinate frames relate to each other.

A frame is aligned such that the $\hat{z}$ axis matches the joint axis, and usually the $\hat{x}$ axis is parallel to the major axis of the associated limb segment (Figure A.1). A point in space which is described by a vector $\vec{p}$ in the jth frame is labeled ${}^{j}\vec{p}$. The origin of the jth frame is defined

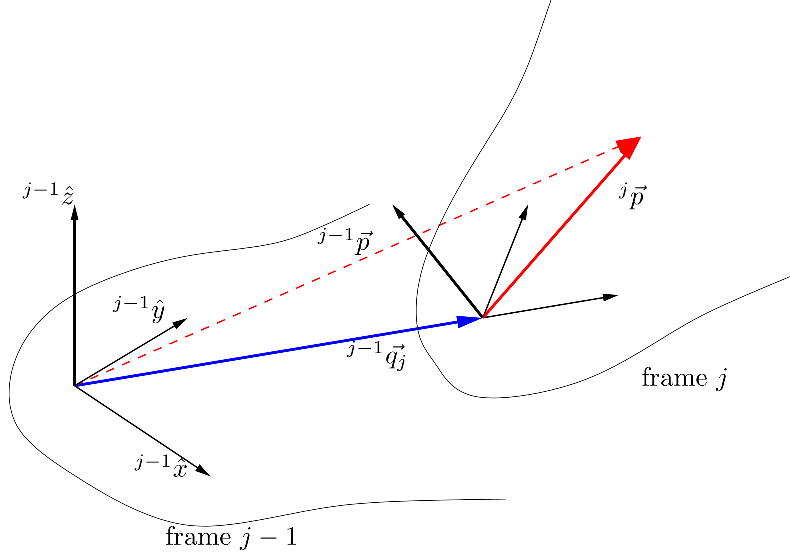


Figure A.1: Coordinate frame $(j-1)$ is the parent of frame j. Vectors $^{j}\vec{p}$ and $^{j-1}\vec{p}$ describe a point relative to the respective frames. $^{j-1}\vec{q_j}$ defines the origin of frame j.

by a vector $^{j-1}\vec{q_j}$ in its *parent* frame $j-1$. (Note that $^{j}\vec{q_j} = 0$. It's the origin, after all, for that frame.)

The transform $^{i}_{j}T$ changes the reference frame of a point from j to i. That is, $^{i}\vec{p} = {}^{i}_{j}T\,{}^{j}\vec{p}$. For any two frames $i < j$ connected by a kinematic chain,

$$^{i}_{j}T = {}^{i}_{i+1}T\,{}^{i+1}_{i+2}T \cdots {}^{j-1}_{j}T.$$

$^{i}_{j}T$ is actually a rotation and a translation,

$$^{i}_{j}T\,{}^{j}\vec{p} = {}^{i}_{j}R\,{}^{j}\vec{p} + {}^{i}\vec{q_j}, \tag{A.1}$$

where $^{i}\vec{q_j}$ is the position of the jth origin relative to the ith frame; $^{i}_{j}R$ is expressed as a 3×3 matrix, which depends on the parameterization used to specify the relative orientation.

Linkage and Muscle Description

The canonical Denavit-Hartenberg form allows each frame to be very compactly described by four parameters [15]. However, this form does not allow for the description of branching chains, i.e. a ground-rooted

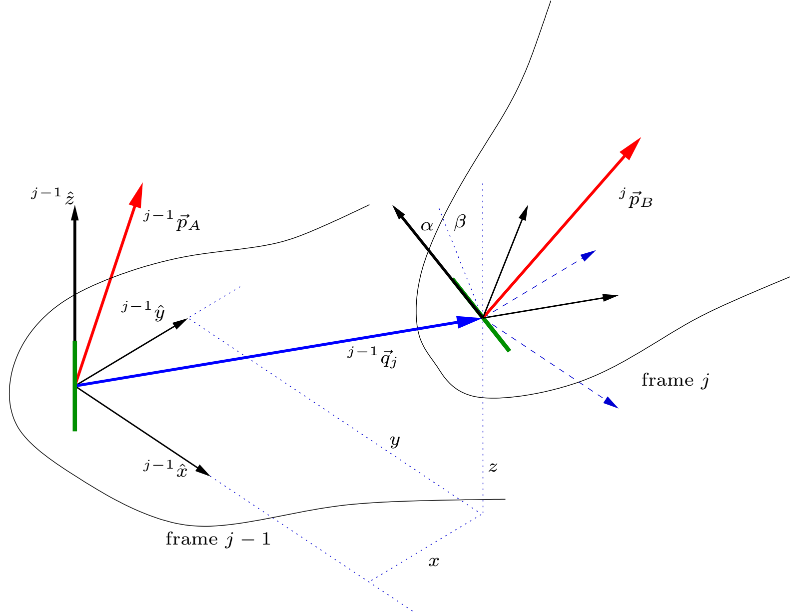


Figure A.2: The skeleton is described as a tree of connected links. The origin of a link is specified by (x, y, z) position relative to the parent link. The orientation is specified by relative rotations β (around $\hat{y}$) and α (around $\hat{x}'$).

torso with two arms connected to it (and potentially a head, as well). I have chosen a more general description with six parameters: x, y, z (all constant) to position a joint in its parent frame, α, β (both constant) to set the joint's orientation, and θ (variable) to indicate the joint angle (Figure A.2). An implicit seventh parameter in this description is the identity of the upstream parent frame in which a joint is rooted.

A parameterization is mostly a matter of convenience; all it needs to do is to provide the transformation ${}^{j-1}_{j}T$ which describes how a child frame relates to its parent. Under the specified parameterization, the transformation is given by:

$$ {}^{j-1}_{j}R = \begin{pmatrix} c\beta\, c\theta + s\beta\, s\alpha\, s\theta & -c\beta\, s\theta + s\beta\, s\alpha\, c\theta & -s\beta\, c\alpha \\ c\alpha\, s\theta & c\alpha\, c\theta & s\alpha \\ s\beta\, c\theta - c\beta\, s\alpha\, s\theta & -s\beta\, s\theta - c\beta\, s\alpha\, c\theta & c\beta\, c\alpha \end{pmatrix} \quad \text{(A.2)} $$

$$ {}^{j-1}\vec{q}_j = (x, y, z)^T \quad \text{(A.3)} $$

where $c\beta = \cos\beta$, $s\beta = \sin\beta$, etc. Note that since ${}^{j-1}_{j}R$ is a rotation

(an orthogonal matrix),

$$ {}^{j}_{j-1}R = ({}^{j-1}_{j}R)^{-1} = ({}^{j-1}_{j}R)^{T}. $$

Thus the "push-back" transformation (sending a vector into the parent frame) is trivially invertible into the "push-forward" transformation.

Once the skeleton is described as a collection of frames, virtual muscles have a simple description. They are just lines defined by their two endpoints, ${}^{j}\vec{p}_A$ and ${}^{k}\vec{p}_B$. Each endpoint (A or B) is specified by a position vector anchored in a particular link's coordinate frame (j or k), corresponding to the parts of the robot to which the muscle is attached.

Calculation of Muscle Length

Muscle length l_{AB} is simply the cartesian distance between a muscle's two anchor points, ${}^{j}\vec{p}_A$ and ${}^{k}\vec{p}_B$. To calculate the distance, however, we must first transform the endpoint vectors into the same reference frame:

$$ {}^{k}\vec{p}_A = {}^{k}_{j}T\,{}^{j}\vec{p}_A $$

Although the anchor vectors are constants (in their respective frames), the transformation is a function of the skeleton/joint configuration. Hence, the length becomes a function of the joint angles $\vec{\theta}$.

Calculation of Joint Torque

Given the magnitudes of the forces to be applied by each virtual muscle, the skeletal model must also calculate the resulting equivalent torque to be applied by the motor controllers.

A torque $\vec{\tau}$ can be calculated as the cross-product of a force vector $\vec{F}$ with a moment arm $\vec{r}$. For a virtual muscle anchored at ${}^{i}\vec{p}_A$ and ${}^{k}\vec{p}_B$ and spanning joint j (where $i < j \leq k$), the force vector lies on the line connecting the anchor points, and the moment arm is any vector from joint origin ${}^{j}\vec{q}_j$ to any point on that line. We can simply choose anchor ${}^{k}\vec{p}_B$ as that point.

Given the force magnitude F:

$$
\begin{aligned}
\vec{F} &= F\frac{\vec{p}_B - \vec{p}_A}{\|\vec{p}_b - \vec{p}_A\|} = \frac{F}{l_{AB}}(\vec{p}_B - \vec{p}_A) \\
\vec{r} &= \vec{p}_B - \vec{q}_j \\
\vec{\tau}_j &= \vec{F} \times \vec{r} = \frac{F}{l_{AB}}(\vec{p}_B - \vec{p}_A) \times (\vec{p}_B - \vec{q}_j)
\end{aligned}
$$

Again, all vectors must be transformed into a common reference frame. We choose the jth reference frame, which is what we need to get the torque experienced by the joint. Recall that ${}^j\vec{q}_j = 0$:

$$\begin{aligned} {}^j\vec{\tau}_j &= {}^j\vec{F} \times {}^j\vec{r} \\ &= \frac{F}{l_{AB}}({}^j\vec{p}_A \times {}^j\vec{p}_B). \end{aligned}$$

Only the $\hat{z}$ component is needed (the $\hat{x}$ and $\hat{y}$ components become strains in the joint bearings):

$$\tau_{jz} = \left(\frac{F}{l_{AB}}\right)({}^jp_{Ax}{}^jp_{By} - {}^jp_{Ay}{}^jp_{Bx}).$$

This calculation is performed for every joint j ($i < j \leq k$) spanned by the virtual muscle.

Each virtual muscle m contributes a vector of torques $\vec{\tau}_m$ to the robot (one component per joint, where most are zero) to yield $\vec{\tau}(\vec{\theta}, \vec{F})$, the skeletal torque of the virtual musculature.

Appendix B

Two Transform Models

Two function approximation techniques were applied to the *transform model* discussed in Section 6.4. The first is a non-parametric, memory-based approach; the second is a semi-parametric approach designed to capture a coordinate transformation. The memory-based approach is generic and trivial to train, but estimation is compute-intensive. The semi-parametric approach requires more tuning, but provides faster estimates and exact derivatives.

B.1 Non-parametric: Memory-based Model

Given a corpus of N training samples $(\vec{x}_i, \vec{y}_i)$, an estimate $\hat{\vec{y}}$ for $f(\vec{x})$ is calculated via

$$\hat{\vec{y}} = \frac{\sum_i \gamma_i \vec{y}_i}{\sum_i \gamma_i}, \quad \gamma_i = e^{-\left(\frac{\|\vec{x}-\vec{x}_i\|^2}{\sigma^2}\right)}$$

The estimate is a locally-weighted average of the training samples. The variance σ^2 sets the size of the averaging neighborhood. This imposes a smoothness constraint on the function $f()$.

This estimator only provides good estimates in regions of the input space with sufficient sample density. Thus, it requires a quantity of training data exponential in the number of input dimensions. Because the estimator must iterate through all the samples, this slows down the computation.

Many variations on the basic algorithm address these issues. The computational load can be lightened by taking advantage of the fact that relatively distant samples contribute almost nothing to an estimate. At the expense of accuracy, nearby samples can be identified

quickly using approximate-nearest-neighbor techniques [26]. The need for high sample density can be reduced by techniques such as *locally weighted regression* [46] which impose additional constraints on the data.

B.2 Semi-Parametric: Loose Coordinate Transform Model

Suppose the transform we wish to learn is the particular case of finding the cartesian coordinate position of the endpoint of a multi-joint limb. Let's call the cartesian position $\vec{x}$ and the vector of joint angles $\vec{\theta}$. We want to learn f such that $\vec{x} = f(\vec{\theta})$.

The mathematics involved has already been overviewed in Appendix A. In the language used there, we are trying to determine

$$\vec{x} = {}^0\vec{p} = {}^0_jT(\vec{\theta})\,{}^j\vec{p}$$

where ${}^j\vec{p}$ is the position of the endpoint of the limb in the coordinate frame of the last segment in the limb and ${}^0_jT$ is the composite transform which pushes back a vector from the jth frame to the base frame of the robot (or whatever frame we want the cartesian position in). Given that the limb segments themselves are rigid, ${}^j\vec{p}$ is a constant, but ${}^0_jT$ is a function of $\vec{\theta}$. In fact, ${}^0_jT(\vec{\theta})\,{}^j\vec{p}$ is just the $f(\vec{\theta})$ which we are looking for.

How shall we model this? ${}^0_jT$ is the composition of all the link-to-link transforms,

$${}^0_jT = {}^0_1T\,{}^1_2T \cdots {}^{j-1}_{\;\;j}T,$$

and each of those one-step transforms is given by Equations A.1, A.2 and A.3. To create a "tight" parametric model, we could multiply these matrices together to get the explicit form for ${}^0_jT(\vec{\theta})\,{}^j\vec{p}$. This model will have six parameters per joint — the $(\alpha, \beta, \theta_0)$ and ${}^{i-1}\vec{q}_i$ which define each link's coordinate frame — plus three parameters for ${}^j\vec{p}$. Thus, the number of parameters is linear in the number of joints. A four-joint arm will require 27 parameters.

Now, if we are feeling a bit lazy (and after eight chapters of thesis, who isn't?), we aren't going to want to do all that long-hand matrix multiplication. And we certainly aren't going to want to evaluate all the derivatives needed to come up with training rules for each parameter. But at the expense of an exponential number of parameters, we can take a shortcut.

Note that each ${}^{i-1}_{i}T$ depends only on θ_i, and that dependency shows up as additive terms containing either $\sin\theta_i$ or $\cos\theta_i$. If we had overcome our laziness, the final expression for each component of $\vec{x}$ would be a sum of terms of the form $\omega Z_0 Z_1 ... Z_j$. Each Z_i is either $\sin\theta_i$, $\cos\theta_i$, or 1 (in case θ_i does not appear in that term), and ω accounts for all other parameters (α's, β's, etc.). We can write this as

$$\vec{x} = \sum_{i=0}^{3^n - 1} \vec{\omega}_i Z_i(\vec{\theta}), \tag{B.1}$$

where $Z_i(\vec{\theta}) = z_{i(1)}(\theta_1) \cdots z_{i(n)}(\theta_n)$ and $i(j)$ is the jth digit of i expressed in base-3, and

$$\begin{aligned} z_0(\theta) &= 1 \\ z_1(\theta) &= \cos\theta \\ z_2(\theta) &= \sin\theta \end{aligned}$$

Equation B.1 gives us a "loose" parametric model for a coordinate transformation. It's loose because it has many more parameters than are absolutely necessary — for n joints, it requires 3^{n+1} parameters instead of $6n + 3$. There are a lot of interdependencies among the $\vec{\omega}_i$. All the same, this model is capable of exactly representing the target function $\vec{x} = f(\vec{\theta})$. Since the model is linear in $\vec{\omega}_i$, a gradient-descent update rule is trivial to compute:

$$\Delta\vec{\omega}_i = -\lambda(\Delta\vec{x}) Z_i(\vec{\theta})$$

Exact derivatives can also be computed (necessary for a transformer module to act as a controller) by replacing the appropriate $z_{i(j)}(\theta_j)$ quantities with $\dot{z}_{i(j)}(\theta_j)$.

Bibliography

[1] Bryan P. Adams. Meso: A virtual musculature for humanoid motor control. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, September 2000.

[2] A. Billard, K. Dautenhahn, and G. Hayes. Experiments on human-robot communication with robots: an imitative learning and communicating dollrobot. Technical Report CPM-98-38, Centre for Policy Modelling, 1998.

[3] Aude Billard and Gillian Hayes. DRAMA, a connectionist architecture for control and learning in autonomous robots. *Adaptive Behavior*, 7(1):35–63, 1999.

[4] Emilio Bizzi, Simon F. Gistzer, Eric Loeb, Ferdinando A. Mussa-Ivaldi, and Philippe Saltiel. Modular organization of motor behavior in the frog's spinal cord. *TINS*, 18(10):442–446, 1995.

[5] Bruce Blumberg, Marc Downie, Yuri Ivanov, et al. Integrated learning for interactive synthetic characters. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, volume 29, New York, NY, 2002. International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH-2002), ACM Press.

[6] C. Breazeal, A. Edsinger, P. Fitzpatrick, B. Scassellati, and P. Varchavskaia. Social constraints on animate vision. *IEEE Intelligent Systems*, 15(4):32–37, 2000.

[7] Cynthia L. Breazeal. *Social Machines: Expressive Social Exchange Between Humans and Robots*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2000.

[8] R. Brooks and L. A. Stein. Building brains for bodies. *Autonomous Robots*, 1:1:7–25, 1994.

[9] Rodney Brooks. *L*. IS Robotics Internal Technical Report, 1996.

[10] Rodney Brooks, Johanna Bryson, Matthew Marjanović, Lynn Stein, and Mike Wessler. Humanoid software manual. MIT AI Lab internal document, 1996.

[11] Rodney A. Brooks. The behavior language user's guide. A.I. Technical Report 1227, Massachusetts Institute of Technology AI Lab, Cambridge, MA, April 1990.

[12] Rodney A. Brooks, Cynthia Ferrell, Robert Irie, Charles C. Kemp, Matthew Marjanovic, Brian Scassellati, and Matthew Williamson. Alternative essences of intelligence. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*. AAAI Press, 1998.

[13] Thomas J. Carew. Spinal cord I: Muscles and muscle receptors. In E. R. Kandel and J. H. Schwartz, editors, *Principles of Neural Science*, chapter 25, pages 284–292. Edward Arnold, London, second edition, 1981.

[14] George J. Carrette. SIOD: Scheme in one defun. http://www.cs.indiana.edu/scheme-repository/imp/siod.html, July 1996.

[15] John J. Craig. *Introduction to Robotics*. Addison-Wesley, Reading, MA, 1986.

[16] Gary L. Drescher. *Made-Up Minds: A Constructivist Approach to Artificial Intelligence*. MIT Press, Cambridge, MA, 1991.

[17] F. Ferrari, J. Nielsen, P. Questa, and G. Sandini. Space variant imaging. *Sensor Review*, 15(2):18–20, 1995.

[18] Paul Fitzpatrick. *From First Contact to Close Encounters: A Developmentally Deep Perceptual System for a Humanoid Robot*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering Computer Science, 2003.

[19] Paul Fitzpatrick and Giorgio Metta. Towards manipulation-driven vision. In *IEEE/RSI International Conference on Intelligent Robots and Systems (IROS)*, Lausanne, Switzerland, 2002.

[20] Jean Dickinson Gibbons. *Nonparametric Statistical Inference*, chapter 7. Marcel Dekker, New York, NY, second edition, 1985.

[21] Stan Gielen. Muscle activation patterns and joint-angle coordination in multijoint movements. In Alain Berthoz, editor, *Multisensory Control of Movement*, chapter 19, pages 293–313. Oxford University Press, 1993.

[22] Simon F. Giszter, Ferdinando A. Mussa-Ivaldi, and Emilio Bizzi. Convergent force fields organized in the frog's spinal cord. *Journal of Neuroscience*, 13(2):467–491, 1993.

[23] A. V. Hill. The heat of shortening and the dynamic constants of muscle. *Proceedings of the Royal Society of London*, 126:136–195, 1938.

[24] Neville Hogan. The mechanics of multi-joint posture and movement control. *Biological Cybernetics*, 52:315–331, 1985.

[25] MusculoGraphics Inc. *SIMM Software Suite*. Chicago, IL. http://www.musculographics.com/.

[26] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998.

[27] Mark Johnson. *The Body in the Mind*. University of Chicago Press, 1987.

[28] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[29] J. A. Scott Kelso. Concepts and issues in human motor behavior. In J. A. Scott Kelso, editor, *Human Motor Behavior: An Introduction*, chapter 2. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1982.

[30] George Lakoff and Mark Johnson. *Metaphors We Live By*. The University of Chicago Press, Chicago, IL, 1980.

[31] Matthew Marjanović. sok user's manual. MIT Humanoid Robotics Group, Cambridge, MA, November 2002.

[32] Matthew Marjanović, Brian Scassellati, and Matthew Williamson. Self-taught visually-guided pointing for a humanoid robot. In Pattie Maes, Maja J Matarić, et al., editors, *From animals to animats 4*, pages 35–44, North Falmouth, MA, September 1996.

Fourth International Conference on Simulation of Adaptive Behavior (SAB96), MIT Press, Cambridge, MA.

[33] Matthew J. Marjanović. Learning functional maps between sensorimotor systems on a humanoid robot. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1995.

[34] Maja J. Matarić, Victor B. Zordan, and Matthew M. Williamson. Making complex articulated agents dance. *Autonomous Agents and Multi-Agent Systems*, 2(1), July 1999.

[35] Thomas A. McMahon. *Muscles, Reflexes, and Locomotion*. Princeton University Press, 1984.

[36] Giorgio Metta. *Babyrobot, A Study on Sensori-motor Development*. PhD thesis, University of Genoa, Italy, 1999.

[37] Ferdinando A. Mussa-Ivaldi, Simon F. Giszter, and Emilio Bizzi. Linear combinations of primitives in vertebrate motor control. *Proceedings of the National Academy of Sciences*, 91:7534–7538, August 1994.

[38] Object Management Group. *C Language Mapping*, formal/99-07-35 edition, June 1999. `http://www.omg.org/-technology/documents/formal/c_language_mapping.html`.

[39] F. Panerai and Giulio Sandini. Oculo-motor stabilization reflexes: Integration of inertial and visual information. *Neural Networks*, 11, 1998.

[40] Jean Piaget. *The Origins of Intelligence in Children*. Norton, New York, NY, 1952.

[41] David Pierce and Benjamin Kuipers. Map learning with uninterpreted sensors and effectors. *Artificial Intelligence Journal*, 92:169–229, 1997.

[42] J. Pratt. Virtual model control of a biped walking robot. M.Eng. thesis, department of electrical engineering and computer science, Massachusetts Institute of Technology, Cambridge, MA, 1995.

[43] William H. Press et al. *Numerical Recipies in C*, chapter 2. Cambridge University Press, Cambridge, 1992.

[44] Brian Scassellati. A binocular, foveated, active vision system. MIT AI Memo 1628, MIT Artificial Intelligence Lab, Cambridge, MA, March 1998.

[45] Brian Michael Scassellati. *Foundations for a Theory of Mind for a Humanoid Robot*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, May 2001.

[46] Stefan Schaal, Christopher G. Atkeson, and Sethu Vijayakumar. Real-time robot learning with locally weighted statistical learning. In *International Conference on Robotics and Automation*, San Francisco, April 2000.

[47] Gilbert Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, Wellesley, MA, second edition, 1998.

[48] Matthew Williamson. *Robot Arm Control Exploiting Natural Dynamics*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, 1999.

[49] Matthew M. Williamson. Series elastic actuators. A.I. Technical Report 1524, Massachusetts Institute of Technology AI Lab, Cambridge, MA, January 1995.

[50] Matthew M. Williamson. Postural primitives: interactive behavior for a humanoid robot arm. In *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (SAB-96)*. Society of Adaptive Behavior, 1996.

[51] Jack M. Winters and Lawrence Stark. Estimated mechanical properties of synergistic muscles involved in movements of a variety of human joints. *Journal of Biomechanics*, 21(12):1027–1041, 1988.

[52] Song-Yee Yoon, Robert C. Burke, Bruce Blumberg, and Gerald E. Schneider. Interactive training for synthetic characters. In *AAAI/IAAI*, pages 249–254, 2000.