UNIVERSITY OF ALBERTA

The Use of Adaptive Logic Networks as Fast Predictors of Motion

by

Allen George Supynuk

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the Degree of Master of Science

in

Computing Science

Department of Computing Science

Edmonton, Alberta

Fall 1991

ISBN 0-315-69966-3

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled "The Use of Adaptive Logic Networks as Fast Predictors of Motion" submitted by Allen George Supynuk in partial fulfillment of the requirements for the degree of Master of Science.

[illegible]
Dr. W.W. Armstrong (Supervisor)

[illegible]
Dr. T. A. Marsland

[illegible]
Dr. H. Zhang

[illegible]
Dr. M. N. Oguztoreli (External)

Date: Oct 2/91

Dedication

I would like to dedicate this thesis to four groups of people:

To my supervisor, Dr. Bill Armstrong, for his enthusiasm, his unflagging support, and his wonderful adaptive logic networks; and to Andrew Dwelly, for his easy to understand and modify code implementing said networks.

To my best friend and loving wife: Wendy McDonald, for *everything*.

To my two buddies: Gregory and Brian Supynuk, for examples of how carbon-based content-addressable adaptive logic systems learn.

To two of my best friends: William and Marion Supynuk, for all their encouragement, support, love, and for constructing the carbon-based content-addressable adaptive logic system that wrote this thesis.

Abstract

The equations of motion are relatively straight-forward and simple, albeit computationally complex. While relatively efficient closed-form solutions exist, they still compute on the order of real time on current systems. Adaptive logic networks have the potential to approximate the quantities describing motion (after training on representative sample data) on current hardware in an efficient, naturally parallel way. They have the added benefit of executing in a few tens of propagation delays (which in state-of-the-art silicon is currently 10-9 seconds), doing the entire computation in about the time of an add or multiply on special purpose hardware. This thesis explores the application of ALNs to the problem of predicting the results of a closed form solution to the laws of motion. It also contains a detailed exploration of the problem of input/output coding for ALNs.

Results indicate that ALNs are a potentially viable method for predicting the results of a closed form solution to the laws of motion.

Two useful new methods of input/output coding for ALNs are presented, along with a fast method of decoding a useful class of coding schemes.

Table of Contents

1 Introduction

This thesis encompasses many disciplines. Besides being implemented in C, Yacc, Awk, C-shell scripts, IRIS Graphics, and a small custom designed language named LF [Dwel90]) on a Sun SPARCStation I and a Silicon Graphics Personal Iris (as well as maintaining the ability to run on a Myrias SPS-2 parallel computer), it touches on coding theory, statistics, animation, quaternion curves, content addressable parallel processing, programming languages, neural networks, adaptive logic networks, and the equations of motion.

The next chapter provides an introduction to Robot Control, including a historical perspective, leading up to adaptive logic networks. Subsequent chapters introduce: the equations of motion simulator (chapter 3); adaptive logic networks (chapter 4); and the coding of input and output (chapter 5). Chapter 5 also includes some original work on coding real intervals. With the background taken care of, Chapter 6 describes the setup of the adaptive logic network predictor, along with the results. Chapter 7 outlines the software modifications made, including some interesting and necessary performance enhancements. Chapter 8 contains some final conclusions and suggestions for further research.

2 Robot Control

2.1 History of Robot Programming Languages

For a historical overview of the state of the art in robot programming languages prior to 1985, see chapter eight of the edited volume by Lee, Gonzalez, and Fu [Loza83, Shim84,Tayl82, Taka81, Mujt82, Lieb77, Gesc83, Gruv84] as well as the work of Nakano et al [Naka85] and Mitsuishi et al [Mits85]. For a detailed description of one of the languages described therein (AL), see Goldman's book [Gold85].

The languages of that time range from robot oriented/robot level languages (where each motor and joint action are controlled) like AML [Tayl82], AUTOPASS [Lieb77], PAL [Taka81], TL-10 [Naka85], VAL-II [Shim84], and WAVE [Loza83] to object oriented/object level languages like AL [MujT82, Gold85], COL [Mits85], MINI [Loza83], and RSS [Gesc83]. All incorporate: some form of "guiding" - a human operator manually guides the robot through some motion, which the robot then repeats under program control; some form of concurrency - the ability to describe two or more actions that must happen at the same time; some way of integrating sensory information; and some way of describing the world in terms of Cartesian coordinates and Euler angles. In style they range from assembler (AUTOPASS, WAVE) through BASIC (TL-10), Algol (AL, AML, PAL, VAL-II), and Pascal (RSS, COL) to Lisp (MINI).

2.2 Imbedded Robot Programming Languages

An interesting approach, pioneered by MINI, is to imbed the robot programming language into an existing language. Blume et al [Blum87] describe a language imbedded into both Pascal and C and mentions that a further Ada imbedding has been implemented. This saves the language designer from having to re-implement features that are common to mainstream programming languages.

2.3 Tabular Learning

A more modern approach, described by Raibert [Raib86], uses tables of precomputed data to control a running robot. Memory efficiency is gained through the use of multivariate polynomial approximations to large tables at a cost of a 3 to 10 fold increase in processing time. The resulting controller was successfully used to maintain balance ana regulate forward running speed for a one-legged hopping robot.

2.4 A Modern Robot Programming Language

In his landmark text, Donner [Donn87] gives a list of five features he believes are necessary for a good robot programming language (summarized from page 64):

- some form of real-time guarantees on the time it takes for a (small) command to be performed
- the ability to control the order of events; some way of saying "wait until this event happens"
- lexically scoped concurrency
- true process abstraction; the ability to invoke a process without knowing how it is implemented
- the ability for one process to preempt another; some way for one process to say "stop what you are doing for now (or forever)" to another process

[Donn87] goes on to describe his language OWL which meets each of these needs to the extent necessary for him to control a six-legged, 1800 pound, 9 foot walking robot. The walking process was loosely based on studies of the gaits of cockroaches (which exhibit fairly independent control of each leg). It was able to walk 6 meters in 85 seconds (about $\frac{1}{4}$ km/h), and could walk over varied terrain and on five legs (at about half speed).

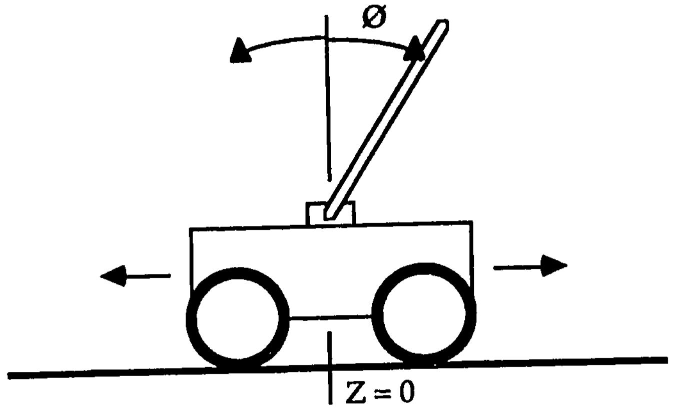
2.5 Expert systems

Another modern variant, based around an expert system, is described by Andersson [Ande88]. This controller takes in sensor data, makes an intelligent guess at a strategy for coping with the data, and modifies previous plans based both on the new data and on physical constraints. Plans are encapsulated and tuned over time. The system plays a moderately respectable game of ping-pong with a standard ping-pong ball on a 2m by 0.5m table.

A similar controller for a four-legged articulated expert system simulacrum (simulated robot) is described by Mohamed and Armstrong [Moha88]. It includes both a Learning Apprentice System (LAS) version, in which the simulacrum is trained by an external (usually human) trainer, and an Autonomous Intelligent System (AIS) version, which substitutes an evaluation subsystem for the external trainer. While no results are given (the article describes work in progress) the authors propose a "Turing Test" for an AIS, namely that it train as well its LAS version.

2.6 Trainable Adaptive Controllers

Guez and Selinsky [Guez88] describe a Trainable Adaptive Controller (TAC) for a two dimensional "broom balancer" using a feedforward linear threshold network trained using Back-Propagation (for a description of linear threshold networks see section 4.1.1 - *ALNs vs Linear Threshold Networks*). They used four input elements (one each for cart position, cart velocity, angular position of pole, angular velocity of pole), 16 elements in the first hidden layer, 4 elements in the second, and one output element feeding into a sigmoidal activation function.



Broom Balancer

Three learning strategies were tried. The first used a linear control law to determine which way to drive and at what speed to both balance the pole and to keep it centered around the origin (Z = 0). After 20,000 iterations of Back Propagation the average mean square error was less than 0.0005. The second used a non-linear control law to determine correct output and was successfully learned after 80,000 iterations, although the TAC still took longer to stabilize the pole than the teacher. The last strategy was based on data gathered from humans doing the balancing, and was learned in 40,000 iterations. Since the humans were unable to keep the cart both balanced and centered around the origin, neither was the resulting TAC.

An interesting further experiment used filtered human data - data with the human reaction time filtered out. The resulting TAC was much smoother in stabilizing the system.

2.7 Increasing Controller Accuracy

An innovative approach to using neural networks in a robot controller is described by Kozakiewicz et al [Koza90]. Here the neural network (and a set of least squares polynomials) was trained to correct the forward kinematic model of the arm; that is, it corrected differences between the theoretical model of the arm used by the controller and the real arm. They used a three-layer linear threshold network (with 6 input, 20 hidden, and 3 output elements) trained using back-propagation for 2500 cycles (for a description of linear threshold networks see section 4.1.1 - *ALNs vs Linear Threshold Networks*). The following table [Koza90, p 218] compares the original positioning error with the error after both least squares polynomial (LSQR) and neural network (NN) calibration:

quantity	before calibration (μm)	after LSQR calibration (μm)	after NN calibration (μm)
average error	434	34	71
maximum error	654	127	302
delta of error	115	31	50
range of error	89 to 779	-59 to 127	-79 to -221

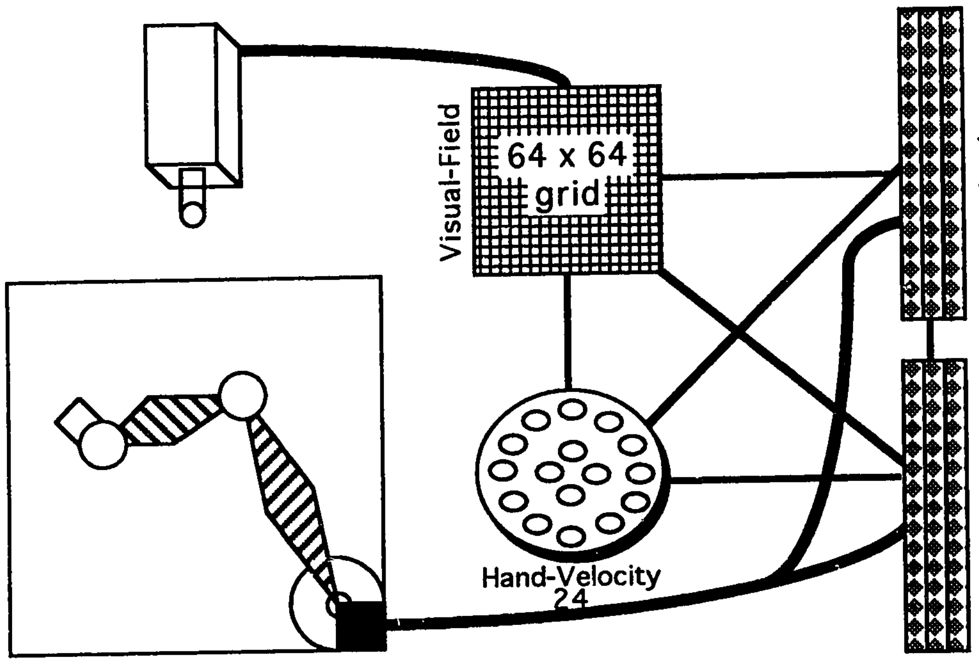
Positioning error of a SCARA robot due to static deflection

That is, the authors were able to achieve results within a rough factor of two of least squares polynomial correction using linear threshold networks, where days of careful programming are replaced by hours of training.

2.8 Connectionist Controller

In what appears to me to be a landmark work, Mel describes his biologically motivated controller for a robot arm [Mel90]. The arm is capable of solving "difficult, visually-guided reaching problems in the presence of obstacles."

The only teacher-based training is in the initial setup, wherein the arm is guided to a (small) representative sample of its possible configurations to learn the correspondence between the arm position and what the camera sees. Four Sigma-Pi (connectionist) networks were used during training, developing five million weights on the forward kinematics alone. A similar model was developed for the inverse differential kinematics (that is, the inverse kinematics were learned only for small movements backwards). The Sigma-Pi networks were then reduced to k-d trees. k-d trees are binary trees that decompose a multi-dimensional space into hyper-rectangular regions. An approximation to the mapped function (Sigma-Pi network in this case) over this region is stored at the leaf. In this case the approximating functions were all constants, making the approximation run in O(n log n). The reduction to k-d trees took 16 hours on a Sun 3-160, resulting in a speedup by a factor of 50 to 100.



Architecture of the Connectionist Controller
(Numbers indicate number of (conceptual) Sigma-Pi neurons)

2.9 Adaptive Logic Controller

Mohamed describes work-in-progress on an adaptive logic network for controlling the motion of a four-legged articulated simulacrum [Moha90]. The network is fed an indication of the desired direction (from an expert system) and the position and velocity of the leg joints. While some experiments have been performed, no results were given.

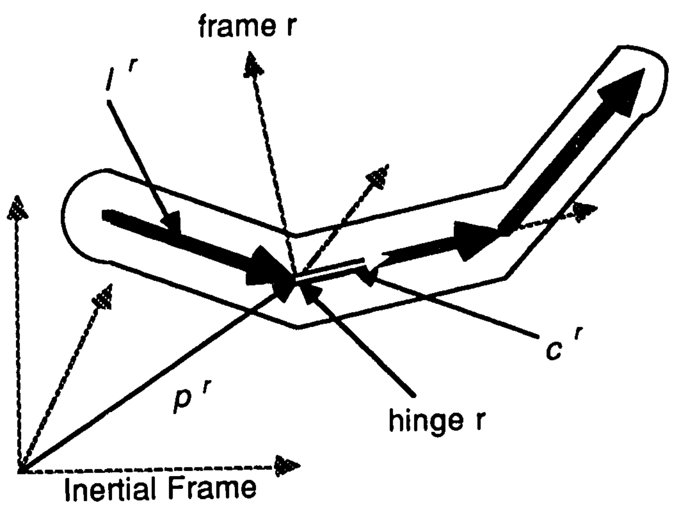
3 Equations of Motion Simulator

This chapter is a summary of a paper by Armstrong and Green [Arms85]. It describes the computational complexity of, and variables used by, the Equations of Motion simulator used to produce both the training and testing data for the adaptive logic networks used in this thesis. Besides the application described by Armstrong et al and used here, it has been successfully used in a simulation of a dancing human being [Lake90]. For an alternate model based on mass-spring systems see the paper by Miller [Mill88].

Note that as far as the adaptive logic networks are concerned, Dynatree (the equations of motion simulator) is a 'black box' that produces data for training and testing. Dynatree was given to me by Armstrong.

3.1 The Model

The model represents objects as a tree structure (that is, no loops) of flexibly linked objects. The point where an object joins its parent is called its hinge. Orthonormal moving three dimensional frames are attached to each object at its hinge. There is also a fixed, non-rotating inertial frame. The model calculates position, velocity, acceleration, mass, force, angular velocity, angular acceleration, moment of inertia, and torque based on external forces and torques.



The inertial frame and some selected variables

3.2 Quantities Used

The model uses the following quantities. Lower-case letters denote scalars and vectors; uppercase denote matrices. Superscripts denote the link number. Each link other than the root (link 1) has one proximal hinge connecting it to its parent. The middle column gives the variable name used in the C implementation:

Scalars

m^r	m	the mass of link r

Quantities in the inertial frame

a_G	aG	the acceleration of gravity
p^r	pH	the position vector of the hinge of link r which joins it to its parent (the proximal hinge of r)
v^r	vH	the velocity of the proximal hinge of link r
f^r_E	FE	an external force acting on link r at the proximal hinge
g^r_E	GE	an external torque acting on link r

Quantities in the frame of link r

a^r	aH	the acceleration of the proximal hinge of link r
ω^r	omega	the angular velocity of link r
$\dot{\omega}^r$	omegadot	the change in angular velocity of link r. (This would be the angular acceleration if it were represented in the inertial frame. When translated to the frame of link r it becomes a more complicated phenomena.)
c^r	c	the vector from the proximal hinge of link r to the center of mass of link r
f^r	fH	the force which link r exerts on its parent at the proximal hinge
g^r	gH	the torque which link r exerts on its parent at the proximal hinge
J^r	J	the moment of inertia matrix of link r about its proximal hinge

Quantities in the frame of the parent of link r

l^r	l	the vector from the proximal hinge of the parent of link r to the proximal hinge of link r (a constant vector in this frame)

Rotation Matrices

R^r	ROT	converts vector representations in the frame of link r to their representations in the frame of the parent link
R^{rT}	ROTT	the inverse (= transpose) of R^r
R_I^r	RI	converts vector representations in the frame of link r to their representations in the inertial frame
R_I^{rT}	RITT	the inverse (= transpose) of R_I^r

3.3 The Equations of Motion

In the following equations, S_r denotes the set of all links having link r as a parent.

Equation (1) relates the rate of change of angular momentum of link r to the applied torques:

$$J^r \dot{\omega}^r = g^r_\Sigma - \underbrace{m^r c^r \times a^r}_{\substack{\text{acceleration of center of mass}\\ \text{around the hinge of link r}}} + \underbrace{\sum_{s\in S_r} l^s \times R^s f^s}_{\substack{\text{forces from sons translated}\\ \text{to frame of link r}}} \tag{1}$$

where

$$g^r_\Sigma = -\underbrace{\omega^r \times \underbrace{(J^r \omega^r)}_{\substack{\text{angular}\\ \text{momentum}}}}_{\substack{\text{appears to rotate}\\ \text{in this frame}}} - \underbrace{g^r}_{\substack{\text{“equal and opposite”}\\ \text{torque to that exerted}\\ \text{on parent}}} + \underbrace{\sum_{s\in S_r} R^s g^s}_{\substack{\text{torques from sons}\\ \text{translated to link r}}} + \underbrace{R^{rT}_I g^r_E}_{\substack{\text{external torque translated}\\ \text{from Inertial frame}}} + \underbrace{m^r c^r \times R^{rT}_I a_G}_{\substack{\text{force of gravity on center of}\\ \text{mass translated to link r}}} \tag{2}$$

Equation (3) relates the force f^r acting on the parent of link r at the proximal hinge of link r to the applied forces:

$$f^r = f^r_\Sigma \underbrace{- m^r a^r}_{\substack{\text{from}\\ \text{acceleration}\\ \text{of frame}}} + \underbrace{m^r c^r \times \dot{\omega}^r}_{\substack{\text{from angular}\\ \text{acceleration}\\ \text{of frame}}} + \underbrace{\sum_{s\in S_r} R^s f^s}_{\substack{\text{forces from sons}\\ \text{translated to frame r}}} \tag{3}$$

where

$$f^r_\Sigma = \underbrace{- m^r \omega^r \times (\omega^r \times c^r)}_{\text{centripetal force}} + \underbrace{R^{rT}_I (f^r_E + m^r a_G)}_{\substack{\text{external force and gravity}\\ \text{translated to frame r}}} \tag{4}$$

Equation (5) relates the acceleration at the proximal hinge of a son link *s* of link *r* to the linear and angular accelerations at the proximal hinge of link *r*:

$$R^s \; a^s = a^s_c + \underbrace{a^r}_{\substack{\text{linear acceleration}\\ \text{of hinge}}} \underbrace{- \, l^s \times \dot{\omega}^r}_{\substack{\text{angular acceleration}\\ \text{of son}}} \tag{5}$$

where

$$a^s_c = \underbrace{\omega^r \times (\, \omega^r \times l^s \,)}_{\substack{\text{centripetal acceleration}\\ \text{of son}}} \tag{6}$$

The model goes on to express the motion (positions, velocities, accelerations, and orientations of the links) given the torques at the hinges and the external forces and torques in a computationally efficient manner.

For a description of which variables were selected for learning by the ALN predictor and why, see section 6.1 - *Variables Learned.*

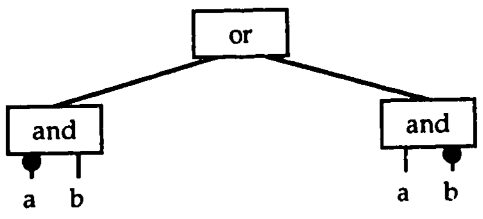
4 Adaptive Logic Networks (ALNs)

Adaptive logic networks are (conceptually) binary trees. Each interior node calculates one of four binary functions: **and, right, left,** and **or.** The input to the tree (a sequence of 0's and 1's) is fed in through the leaves. Negation (logical not of an input) can happen only at a leaf. Each tree calculates one bit of the result.

a	b	a **and** b	a **right** b	a **left** b	a **or** b
0	0	0	0	0	0
0	1	0	1	0	1
1	0	0	0	1	1
1	1	1	1	1	1

Table of functions at nodes of adaptive logic networks

For example, the following adaptive logic network calculates the function on the right:



a	b	r
0	0	0
0	1	1
1	0	1
1	1	0

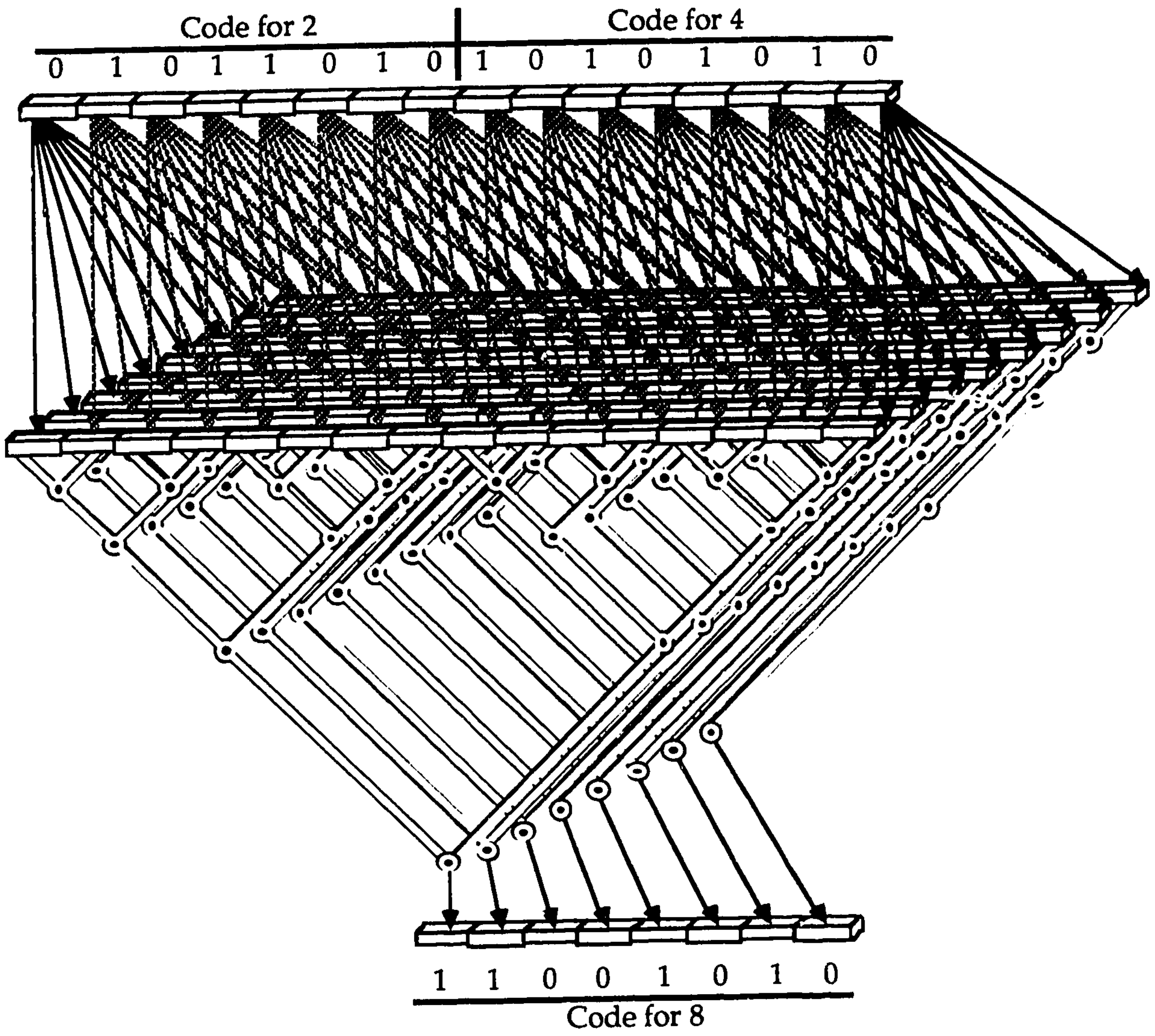
An adaptive logic network and the function it implements (• denotes negation)

To see how a "forest" of these trees can be used to calculate results that are more than one bit, consider training a neural network to learn the 9 x 9 times table:

Now, to multiply 2 times 4 to get 8, we input our code for 2 and our code for 4 and then train the network until it responds with the code for 8. We then train on other inputs (like 3 times 3 to get 9), while checking that 2 times 4 still gives us 8.

First we find a special way of representing 1 thru 9 with a group of (in this example) eight 0's and 1's:

1	01011001
2	01011010
3	10011010
4	10101010
5	10110010
6	10111110
7	10001110
8	11001010
9	11001001



A forest of ARLO trees calculating an multi-bit output

4.1 ALNs vs Neural Networks

This section explores the relationship between adaptive logic networks and the most popular mainstream neural networks: linear threshold networks. Since sigma-pi networks were mentioned in section 2.8 - *Connectionist Controller,* a relationship between them and adaptive logic networks is also drawn. For an introduction to the field of neurocomputing see Hecht-

Nielsen's excellent book [Hech90]. For a more informal yet comprehensive of the state of the art in neurocomputing see Shriver's video notes [Shri89] and the associated video tape.

4.1.1 ALNs vs Linear Threshold Networks

Each node of a traditional neural networks typically calculates a function of the form:

$$\sum_{i=0}^{n} w_i \cdot x_i \geq 0,$$

where w_i '$0 \leq i \leq n$) are weights and x_i ($1 \leq i \leq n$) are the inputs (possibly an output from another node), w_0 is a bias (constant after training), and x_0 is a constant 1. The output is a 1 if this relation is true (ie. if the sum is ≥ 0), otherwise the output is 0. (For more information see Hecht-Nielsen's book [Hech90].)

To get an adaptive logic network from this model set w_0 to -2, and restrict w_i and x_i ($1 \leq i \leq n$) to the set {1,2}. This gives us:

$$-2 \cdot 1 + w_1 \cdot x_1 + w_2 \cdot x_2 \geq 0 \Leftrightarrow w_1 \cdot x_1 + w_2 \cdot x_2 \geq 2$$

If we set w_1 and w_2 to 1 our output will be 1 if and only if both x_1 and x_2 are 1, which is equivalent to **and**. Setting w_1 and w_2 to 2 give us **or**, $w_1 = 1$ and $w_2 = 2$ gives us **right**, and $w_1 = 2$ and $w_2 = 1$ gives us **left**.

w_1	w_2	output on $x_1=0,x_2=0$	output on $x_1=0,x_2=1$	output on $x_1=1,x_2=0$	output on $x_1=1,x_2=1$	equivalent function
1	1	0	0	0	1	**and**
1	2	0	1	0	1	**right**
2	1	0	0	1	1	**left**
2	2	0	1	1	1	**or**

Table of outputs of the function $w_1 \cdot x_1 + w_2 \cdot x_2 \geq 2$

That is, adaptive logic networks are a special case of linear threshold networks.

4.1.2 ALNs vs Sigma-Pi Networks

The output y of a sigma-pi network is the sum of independent multiplicative clusters of input weights:

$$y = \sum_j v_j \cdot c_j > t$$

where

$$c_j = \prod_{i=1}^{n} w_i \cdot x_i$$

where w_i, $1 \leq i \leq n$ are weights on the input, x_i, $1 \leq i \leq n$ are the input, and the v_j are the weights on the cluster as a whole. (For more information see Mel's book [Mel90, pp 42-3].) A feature (selected by one or more of the c_j's) is recognized if y>t.

(Note that the sigma-pi model is suspect. Since the weights w_i can be factored out of the c_j's, there is no association of weights with inputs.)

To get an adaptive logic network from this model restrict v_j and x_i (and by implication, c_j) to the set {0,1}, set w_i=1 and use t=0 as the output threshold. The product (Pi) now becomes a multi-way **and**, and the sum (Sigma) is now equivalent to a multi-way **or**. The network in effect uses the **ands** to recognize single features and **ors** to say that one of the features was seen. This kind of behavior of sigma-pi networks was what was desired in at least one major application [Mel90, p 43] (which also set all the w_i to 1). That is, adaptive logic networks are a special case of sigma-pi networks.

4.2 Why and, right, left, or?

There are 16 possible binary functions of two variables. Of these, 12 functions have the property that changing one input has a 50% chance of changing the output; two of the functions (labelled **0** and **1**) ignore their inputs and never change output; two more, xor and its complement (labelled **a ≠ b** and **a = b**), always change output. However, the 12 functions can be handled by **and** (&), **right, left,** and **or** (|) if we allow the inputs to be negated. It is a simple exercise, left to the reader, to show that any tree made up of these 12 functions can be transformed to an equivalent tree having only the functions **and, or** , **right,** or **left** on the interior nodes and negations appearing only on the leaves; from here on, we will only talk about such trees, and will refer to them as ARLO (<u>a</u>nd, <u>r</u>ight, <u>l</u>eft, <u>o</u>r) trees. Alternatively, these four functions are precisely the set of all nonconstant increasing Boolean functions of two variables [Arms79]. (By "increasing" we mean that if a zero input is changed to a 1, the value output never changes from 1 to 0.)

			and		**left**		**right**	$a\neq b$	**or**		$a=b$						
a	b	0	a&b	$a\&\overline{b}$	a	$\overline{a}\&b$	b	a^b	$a \vert b$	$\overline{a \vert b}$	$\overline{a\hat{}b}$	$\overline{b}$	$a \vert \overline{b}$	$\overline{a}$	$\overline{a} \vert b$	$\overline{a \vert b}$	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

The 16 possible binary functions of two variables

This 50% rule is very important to the learning of the network. We want to have a network which, when presented with an input that is close to one it has seen before, calculates an answer that is close to the previous answer. With ARLO trees the chance that changing a single bit of input will affect the output are 1 in 2^d, where d is the number of nodes between the input and the root of the ARLO tree.

4.3 Parsimonious or Lazy Evaluation of ALNs

The 50% rule has another important advantage: we almost never have to calculate the entire tree! Once we know one input to a node we have a 50% chance of knowing what the output is. (For **and** and **or** this is called McCarthy or lazy evaluation in the programming languages literature. Since "lazy" has negative connotations, and since "parsimony" has already been used in neural network literature [Meis90], we use the term *Parsimony* or *Parsimonious Evaluation.*) Not evaluating the input to a single interior node can save we from calculating as much as half the nodes in the tree! (Consider the root node. If it is an **and** node and the left input is already known to be 0, then there is no need to evaluate any node in the right half of the tree.)

To calculate the relative savings over the whole tree, consider the function ni(d), the number of inputs a node that is d levels from being a leaf has to evaluate in order to know its result. Clearly,

$$\mathrm{ni}(d) = \begin{cases} 1, & d = 0 \\ \mathrm{ni}(d\text{-}1) + 0.5 * \mathrm{ni}(d\text{-}1), & d > 0 \end{cases}$$

That is, we need to calculate ni(d-1) nodes to know one input, and then have a 50% chance of needing to calculate another ni(d-1) nodes to know the other input. Expanding a few terms of this recurrence relation quickly leads to the closed form solution:

$$\mathrm{ni}(d) = \left(\frac{3}{2}\right)^d$$

A tree of height d has 2^{d+1}-1 nodes (including the leaves), so the average relative savings are:

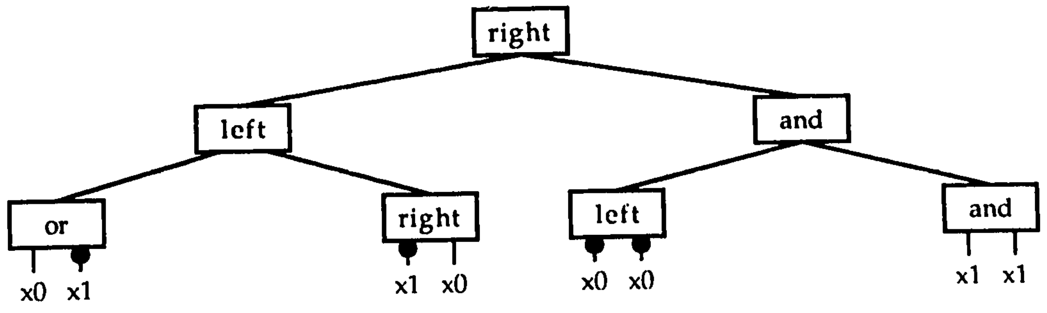
$$\frac{\mathrm{ni}(d)}{2^{d+1}-1} = \frac{\left(\frac{3}{2}\right)^d}{2^{d+1}-1} \cong \frac{3^d}{2^{2d+1}} = \frac{1}{2} \cdot \left(\frac{3}{4}\right)^d$$

This results in an exponential speedup in the evaluation of an ARLO tree; the larger the tree, the smaller the fraction of the tree that is likely to need to be evaluated.

4.4 How ALNs Learn

This section outlines the method which adaptive logic networks use to learn functions. For a more complete description see the papers by Bochman and Armstrong [Boch74], and Armstrong [Arms79, Arms90a, Arms90b].

Each ALN is a binary tree, initially set up with random functions at each of the interior nodes, and with the k binary inputs to the tree randomly distributed and randomly complemented onto the leaves of the tree. (Note that the number of leaves should be at least twice k so that each input and its complement can appear on a leaf, and preferably should be several times k so that useful pairings occur close together on the tree.) For example:



An ALN with 8 leaves, two inputs (x0 and x1), and one output

Now suppose that an input is presented to the tree for training, so we know what the expected output is. First we check to see if the answer is correct. If it is, the current behavior is reinforced. If not, the behavior is discouraged. To adjust behavior we recursively traverse the tree looking for nodes that might be responsible for the output. (In the example above, if we input X0=0, X1=1, and expect a 1 we instead get a 0. We then recurse through the tree looking for nodes which, if their output were changed, would cause the result of the tree to change.) Note that If we determine a node is not responsible, then none of its children are responsible either. Hence, training is parsimonious.

Since all four functions (**and**, **or**, **left**, **right**) produce the same value when their input is (0,0) or (1,1), the trick lies in adjusting responsible nodes whose inputs are either (0,1) or (1,0). For this purpose, two counters implemented in each node keep track of the number of times a (0,1) or a (1,0) was encountered by a node deemed responsible. If the desired output is 1 the relevant counter is incremented; if 0, decremented. The counters are bounded; trying to increment or decrement past their limit has no effect. In effect, each counter tries to keep track if it is expected to produce a 1 or a zero for the case it is tracking. If it is told more often to produce a 1 for this case than a 0, the counter will increase over time; if it is told it should mostly be producing 0's for this case, the counter will decrease. The following table itemizes the possible ranges of values for these counters, the corresponding response of the node to the four possible input pairs, and shows how the counters induce a function on the node:

(0,1) counter	(1,0) counter	output on (0,0)	output on (0,1)	output on (1,0)	output on (1,1)	induced function
<0	<0	0	0	0	1	**and**
<0	≥0	0	0	1	1	**left**
≥0	<0	0	1	0	1	**right**
≥0	≥0	0	1	1	1	**or**

The relation between relative values in counters and induced functions

The problem occurs, of course, in the assigning of responsibility to a node. If the wrong node is deemed responsible learning will be either hindered or blocked. Early work used what is now called *true responsibility;* a node was deemed responsible if changing its output would change the output of the tree:

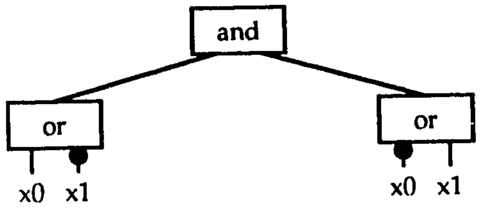
1) For an **and** node, if one child produces a 1, then the *other* child is *truly responsible*; if they both produce a 1, they are both *truly responsible*

2) For an **or** node, if one child produces a 0, then the *other* child is *truly responsible*; if they both produce a 0, they are both *truly responsible*

3) For a **left** node, the child on the left is *truly responsible*

4) For a **right** node, the child on the right is *truly responsible*

This resulted in very opportunistic learning that would fail to learn things like exclusive or (where two nodes have to cooperate in order for the function to be learned):



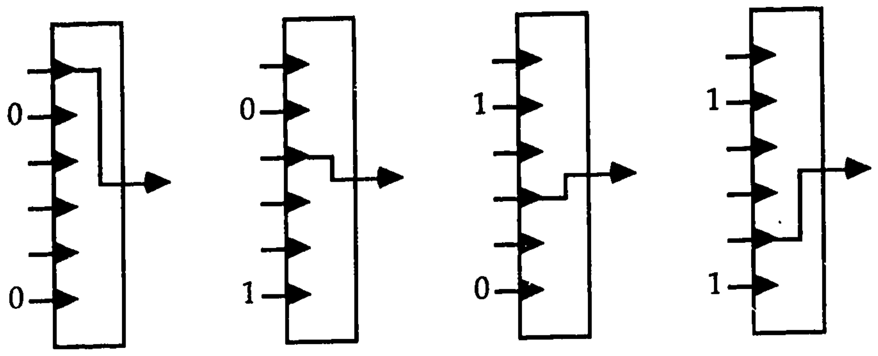
ARLO tree implementing exclusive or

Current work also uses another form of responsibility called *heuristic* or error *responsibility* . A simple form of heuristic responsibility is [Arms90b, p 5]:

1) the root is always *heuristically responsible*
2) if a node is *heuristically responsible* **and** one of its input signals is not equal to the desired network output **then** that input signal is called an *error*
3) the child on the <u>opposite</u> side to the *error* is *heuristically responsible*

The motivation behind heuristic responsibility is that an erroneous input signal (*error*) means that the tree on the other side needs to try harder to compensate. Remember, the error may be coming directly from one of the inputs to the tree, in which case it would be impossible to correct.

With a more sophisticated form of responsibility, incorporating both error and heuristic responsibility, Lin [Arms90a, pp 19-20] was able to train an adaptive logic network to learn a multiplexor with 8 control leads and 264 input leads to 99.9% accuracy, using 3 trees and a majority vote. Lin's algorithm implementing adaptive logic networks allowed the trees to grow in size when the algorithm deemed it necessary. The network was only given 6000 of the 2^{264} possible inputs to train on. The network was not told which 8 of the 264 input leads were the control leads. (The values on the control leads form a binary number in {0,...,255} that is used to select which of the other leads input is passed through as the output. This is extremely impressive.)



A 6-multiplexor. Depending on the values on the two control leads, one of the other four inputs is passed through.

5 Coding

5.1 Description of Problem

Each tree in an Adaptive Logic Network can learn one bit of a function. How can we use a collection or "forest" of trees to learn more complicated functions? How can we best code our input to these forests to maximize their ability to recognize features? This chapter explores several alternative methods of coding and discusses their advantages and disadvantages. Along the way we develop a set of heuristics for recognizing good and bad codes.

5.2 Radix-2 Codes

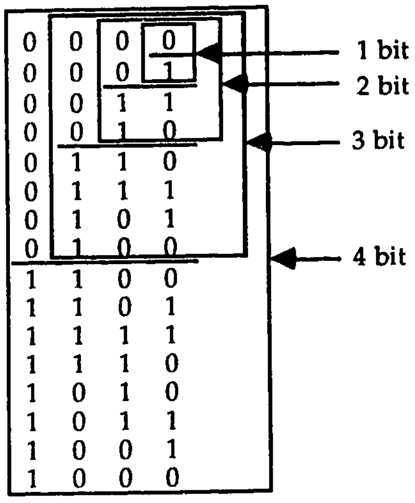
The traditional radix-2 binary codes used by nearly every computer today are highly unsuitable for adaptive logic networks. When moving from a number of the form 2^{k-1}-1 to 2^k, k+1 bits change (for example, moving from 7_{10} to 8_{10} is changing 0111_2 to 1000_2, ie. 4 bits change). So adjacent features can have codes that are arbitrarily far apart.

The number of bits that change between one number and another is known as the Hamming distance, which we write $|\ |_H$. This leads us to our first heuristic:

code rule 1: The Hamming distance between close features should be as small as possible.

5.3 Gray Codes

Gray codes are a cyclical reflected binary code. Cyclical binary codes have the property that only one bit changes when moving from one codeword to the next, so by code rule 1 Gray codes seem ideal. Gray codes are also reflected codes: the n-bit Gray code is formed from the n-1 bit Gray code and a reflection (with 1's and 0's reversed) of the n-1 bit Gray code. For example:



0	0	0	0
0	0	0	1
0	0	1	1
0	0	1	0
0	1	1	0
0	1	1	1
0	1	0	1
0	1	0	0
1	1	0	0
1	1	0	1
1	1	1	1
1	1	1	0
1	0	1	0
1	0	1	1
1	0	0	1
1	0	0	0

The 4 bit Gray code with mirrors

The first problem we notice is that the last code (1000) is only 1 bit away from the first code (0000). This leads us to our second rule:

code rule 2: The Hamming distance between distant features should be as far apart as possible.

As we will see later, this rule is probably the most problematical, since it has a strong effect on how many code words we can fit into a given number of bits.

Both radix-2 and Gray codes pose another problem - the amount of information coded in each bit is not equal. The least significant bit in both codes toggles between every code word, while the most significant bit only changes once. Misinterpreting the least significant bit even slightly means missing out on fine details; it is often the sole indicator that we have moved from one feature to another. Misinterpreting the most significant bit even slightly means losing gross details; it is the sole indicator of which half of the range of data the feature is in. This leads to the following rule:

code rule 3: Each bit in the codeword should contain about the same amount of information; that is, each bit in the codeword should change about the same number of times as you move from one codeword to the next.

In the Gray code above the least significant bit changed 15 times, whereas the most significant bit changed only once.

5.4 One Bit per Feature Codes

In this form of coding, each feature is assigned to a particular bit. n features are coded in n bits. To code the numbers 0 through 4 we would use the codes 00001, 00010, 00100, 01000, 10000.

While the amount of information per bit is uniform, satisfying code rule 3, the distance between the first two codewords is the same as the distance between any two codewords, violating code rule 2.

5.5 Thermometer Codes

Thermometer codes each feature is again linked with a particular bit, but the code words are formed by or-ing the new bit into the previous codeword. To code the numbers 0 through 4 we could use: 0000, 0001, 0011, 0111, 1111. Thermometer codes are unary (base-1) numbers.

Thermometer codes satisfy code rules 1 to 3; adjacent codewords are hamming distance 1 apart, inputs that are n apart have codewords that are n apart, and each bit flips the same number of times (once) as we go through the codewords.

The problem with thermometer codes lies in their inefficiency. In an industry where we are used to packing 2^n codes into n bits, it appears extremely inefficient to pack only n + 1 codes into n bits. This leads us to our next code rule:

code rule 4: *An efficient code should pack considerably more than n+1 features into n bits. Otherwise thermometer codes should be used.*

We now focus in on codes that have higher densities of packing than thermometer codes.

5.6 Combinatorial Walks

This is a moderately clever code based on combinations of k bits out of n. The first $\binom{n}{0}$ codewords (one codeword) is n 0's. The next $\binom{n}{1}$ codewords are all the combination of n-1 0's and one 1, on to the last codeword which is all 1's. Furthermore the codewords are ordered such that the Hamming distance between successive codewords is two, except when we are switching k (the number of 1's) in which case the Hamming distance is one. For example:

```
000
001
010
100
110
101
011
111
```

A 3-bit combinatorial walk

Pseudo code for combinatorial walk:

```
Start with 0...0 /* n 0's */
/* k is the number of 1's in the code to be generated */
for k from 1 to n
    /* Add a 1 to the inside edge of the right- or left-  */
    /* justified group of 1's from the previous iteration */
    if k is odd
        form k 1's by toggling the (n-k)th bit
        permute(n bits, k 1's, right_to_left)
    else
        form k 1's by toggling the (k-1)th bit
        permute(n bits, k 1's, left_to_right)
    endif
endfor
```

Function permute is a little more complicated, but is similar to the Towers of Hanoi problem. To move k 1's left-to-right while hitting every permutation:

1) recursively move (k-1) 1's left-to-right as far as they will go.
2) slide the k'th 1 right one bit using two flips (one to turn it off, the other to turn the new bit on). If done, stop.
3) recursively slide the (k-1) 1's right-to-left until they are immediately to the right of the k'th 1.
4) slide the whole group of (k-1) 1's one bit to the right using two flips (turn off the leftmost bit, then turn on the 0 immediately on the right).
5) go to step 1.

The basis step requires sliding a single bit either left or right, which can be done in two flips. Right-to-left moves are done in an analogous fashion. See the code listings for a program that generates this walk for any given n (**combinatorialWalk.c**).

Combinatorial walks fail due to code rule 2. Codes that are a long way away map onto codewords that are very close. (Note that codes 001 and 011 are at Hamming distance one, yet they are the second and second last codewords.) In particular, the first code for k=i and k=i+2 are hamming distance 1 apart, but are $\binom{n}{i} + \binom{n}{i+1}$ steps apart.

Note that no code that packs 2^n features into n bits is likely to do well on code rules 1 and 2. Every time a new step is taken, there are up to $\binom{n}{1}$ unchosen steps that are Hamming distance 1 away, up to $\binom{n}{2}$ unchosen steps that are Hamming distance 2 away, etc.

At this point we stop searching for completely dense codes and focus instead on codes that pack more than n+1 features into n bits, but less than 2^n.

5.7 Random walks

Random walks start from a given point on an n-dimensional hypercube (and flip a small number p of bits at random. They then pick another p bits at random and flip those. Each step is of Hamming distance p. Two points that are several steps apart should differ in at least half their bits, since this should be like choosing two points at random in the hypercube. So close codes should differ by small multiples of p, and farther codes should differ by around $\frac{n}{2}$.

Points are decoded by finding the closest (in Hamming distance) step on the walk and returning the step number. A faster method of decoding, perhaps using adaptive logic networks, would speed things up on sequential machines. It is relatively straightforward to build special purpose hardware using associative memory that would decode in a small fixed number of cycles. See section 5.15 - *Hardware for Decoding Walks,* later in this chapter. This does suggest, however, the following code rule:

code rule 5: A good code should have a computationally efficient method for decoding.

The naive random walk described here is not suitable. While distant steps *should* be Hamming distance $\frac{n}{2}$ apart, there is no guarantee. In fact, the path could even intersect itself, leading to ambiguous codes. This leads us to our last code rule:

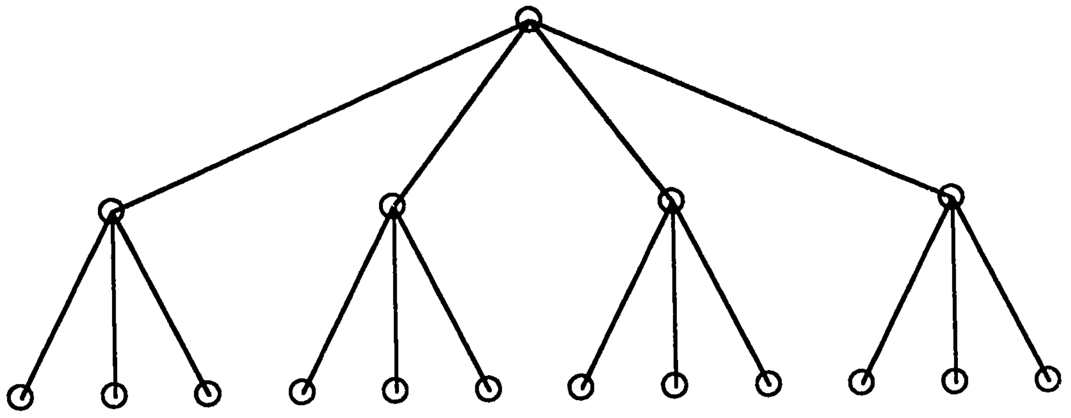
code rule 6: A good code should guarantee a minimum Hamming distance between distant features.

5.8 Constrained Random walks

Several refinements to Random walks immediately present themselves. Checking to make sure that the path does not intersect itself is an obvious improvement. Keeping track of recently flipped bits also helps to keep the path locally nice, by ensuring that it steers away from neighborhoods it has recently visited.

5.9 Sphere Packing

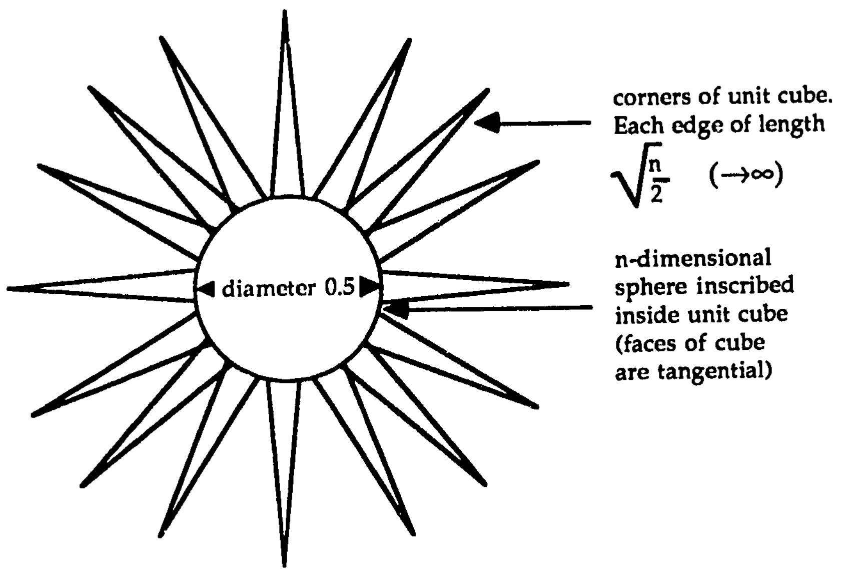
Sphere packing is used in the construction of error-correcting codes. Here each codeword (at the center of a sphere of radius r) is guaranteed to differ by at least $2 \cdot r+1$ bits from any other codeword; each codeword is at Hamming distance $2 \cdot r+1$ or greater from every other codeword. To get an idea of why sphere packing is difficult, consider what these "spheres" look like. A Hamming sphere of radius r imbedded in a hypercube of dimension n can be thought of as a tree whose root node has n- branches, and nodes at level i from the root have n-i branches. For example:



A Hamming sphere of radius 2 on a 4 dimensional hypercube

If this Hamming sphere was imbedded in a 5 dimensional hypercube then the root would have 5 subtrees each with 4 branches.

Note that higher dimensional hypercubes are also not smooth objects, but look more like spherical "porcupines" [Hech90, pp 42-3]. The following diagram is adapted from Hecht-Nielsen's book [Hech90, p 43]:


corners of unit cube.
Each edge of length
$\sqrt{\frac{n}{2}}$ $(\to\infty)$
diameter 0.5
n-dimensional sphere inscribed inside unit cube (faces of cube are tangential)

N-dimensional hypercube

Hamming codes are Hamming spheres of radius 1 that completely fill a hypercube of any dimension n. Codes that completely fill a hypercube with uniform size spheres are called perfect codes.

There is only one known perfect binary code with r>1: the Golay code. These will be discussed in the next section. See MacWilliams and Sloane [MacW77] and Berlekamp [Berl68] for a more complete description of error-correcting codes.

Why are we interested in error correcting codes and sphere packing? We would like to have codes that are forgiving; that is, if some small number of trees in our forest give the wrong answer, we would like the coding to correct the answer, or at least ensure that our answer is close.

Note that we do not need perfect codes. For our application, it is not necessary to have every possible codeword be valid. Also note that we are more interested in finding paths through a hypercube rather than an unordered collection of discrete points (which is what sphere packing gives us). We want to say "this codeword is close to the code for a 1; that codeword is close to the code for a 10; this codeword is a long way from any of our codes and is suspect (or alternatively, we may decide not to care where it maps)." For this reason, traditional sphere packing and error correcting codes provide us more with useful ideas and paradigms than with useful results.

5.10 Golay Codes

Golay codes [MACW77 pp 64-69] are the only known perfect codes with sphere radius other than 1. The only binary code is one with n=23 and sphere radius 3. Using Golay codes we can correct up to 3 bits of error in a 23 bit codeword. Because the Golay codes are perfect, every 23 bit codeword is within hamming distance 3 of exactly one sphere center. There are 4096 (2^{12}) sphere centers, totally covering the hypercube of dimension 23.

In other terms, Golay codes take boolean vectors of length 12 and code them as boolean vectors of length 23. Up to three elements of the output vector can be flipped and still have the vector correctly decoded back to the original boolean vector. If more than 3 elements are flipped then the result will decode incorrectly.

In the following section $\oplus$ is used to denote binary addition (addition modulo 2, or exclusive-or).

5.10.1 Some Properties of Golay Codes

A generating matrix G_{24} for the Golay-24 code (the 23 element code with parity) is [MacW77, p 516]:

$$\left[\begin{array}{cccccccccccc|cccccccccccc}
1&&&&&&&&&&&&1&1&1&1&1&1&1&1&1&1&1&\\
&1&&&&&&&&&&&1&&1&&&&1&1&1&&1&1\\
&&1&&&&&&&&&&1&1&&1&&&&1&1&1&&1\\
&&&1&&&&&&&&&&1&1&&1&&&&1&1&1&1\\
&&&&1&&&&&&&&1&&1&1&&1&&&&1&1&1\\
&&&&&1&&&&&&&1&1&&1&1&&1&&&&1&1\\
&&&&&&1&&&&&&1&1&1&&1&1&&1&&&&1\\
&&&&&&&1&&&&&&1&1&1&&1&1&&1&&&1\\
&&&&&&&&1&&&&&&1&1&1&&1&1&&1&&1\\
&&&&&&&&&1&&&&&&1&1&1&&1&1&&1&1\\
&&&&&&&&&&1&&1&&&&1&1&1&&1&1&&1\\
&&&&&&&&&&&1&&1&&&&1&1&1&&1&1&1
\end{array}\right]$$

To get a Golay-23 code we can delete any column in the above matrix [MacW77, 493].

Note that G_{24} is of the form [**I** | **A**]. G_{24} is self-dual, ie. $\mathbf{GG^T} = 0$. Hence, $\mathbf{I} \oplus \mathbf{AA^T} = 0$. Therefore $\mathbf{A^T} = \mathbf{A^{-1}}$. Also, if we define **H** as

$$\mathbf{H} = \begin{bmatrix} \mathbf{A} \\ \mathbf{I} \end{bmatrix}$$

Then $\mathbf{GH} = \mathbf{H^T G^T} = \mathbf{IA} \oplus \mathbf{AI} = \mathbf{A} \oplus \mathbf{A} = 0$.

5.10.2 Decoding Golay Codes

With this machinery in hand, suppose we encode an input boolean vector of length twelve, **u** by:

$$\mathbf{x} = \mathbf{uG}$$

now,

$$\mathbf{xH} = (\mathbf{uG})\mathbf{H} = \mathbf{u}(\mathbf{GH}) = 0$$

Then suppose **x** gets garbled while being transmitted, and is received as **y**. Define the error vector **e** as,

$$\mathbf{e} = \mathbf{x} \oplus \mathbf{y} \Leftrightarrow \mathbf{y} = \mathbf{x} \oplus \mathbf{e}$$

Now,

$$\mathbf{yH} = (\mathbf{x} \oplus \mathbf{e})\mathbf{H} = (\mathbf{uG} \oplus \mathbf{e})\mathbf{H} = \mathbf{uGH} \oplus \mathbf{eH} = \mathbf{eH}$$

yH is called the *syndrome* of **y**. Since **H** is a linear matrix, this means there is a one-to-one correspondence between syndromes and correctable errors. Since $\mathbf{G_{24}}$ has distance 3, we can correct up to 3 errors. Since $\mathbf{G_{24}}$ has parity, we can detect 4 bit errors (and, if we knew where the fourth error was, correct it too).

We use **G** and its inverse **H** to code and decode <u>23</u> bit numbers as follows

1) At the start of the program define an array SC of size 2^{12} (=4096), and set each element of SC to 0.

2) For all error vectors **e** such that $|\ \mathbf{e}\ |_H \leq 3$, set SC[**eH**] = **e**.

3) For all errors vectors **e** such that $|\ \mathbf{e}\ |_H = 4$ and $(\mathbf{e} \oplus 1) = 1$ (ie. one of the errors is in the parity bit), set SC[**eH**] = **e**.

   (We have just set $\binom{24}{1} + \binom{24}{2} + \binom{24}{3} + \binom{23}{3} = 4095$ unique elements in SC; SC[0] is the only element of SC that is still 0.)

4) To code **u** we return the leftmost 23 bits of **uG**.

5) When (the 23 bit quantity) **y** is received (with up to 3 errors), we shift it left by one bit (possibly introducing a fourth error in the rightmost (parity) bit) to form **y′**, calculate its syndrome (**y′H**), and look up its correction in SC. The leftmost 12 bits (information bits) of the corrected **y′** are returned. If we use the convention that **x** « n means shift **x** left n bits, and **x** » n means shift **x** right n bits we have:

   $((\mathbf{y} \ll 1) \oplus SC[(\mathbf{y} \ll 1)\mathbf{H}]) \gg 12$

5.11 Golay Walk

The Golay walk was a failed attempt to get a good random walk. The idea was to use the Hamming spheres imbedded in the 23 dimensional hypercube as steps on the path. The walk was constructed as follows:

```
S = set of "Golay spheres"
r = random selection from S
n = 0
walk[n] = r
remove r from S
While S is non-empty
     a = any member of S adjacent to r
     walk[++n] = a
     delete all members of S adjacent to r (including a)
     r = a
endWhile
```

This creates a "tube" of spheres, such that there is at least one sphere between any two non-adjacent spheres on the path. Several variations on selecting *a* were tried, with the most successful being the member of S adjacent to r with the most number of adjacent spheres still in S. This resulted in a path of 30 spheres, each at distance 7 or 8 from each other. The path started at 0 and ended at 0x7fffff.

When the tube was constructed, a thermometer code was used between steps on the sphere, giving a final path length of 214.

It was expected that this path would be faster to decode. With Golay codes it is possible to calculate the center of the sphere containing a given code (this is precisely how Golay codes are used for error correction). It was hoped that this could be used to quickly locate one or more spheres on the the path that would contain the closest point on the path. Unfortunately, points were found such that the closest point on the path was not in a sphere adjacent to the sphere containing the points. See section 5.14 - *Using Golay Codes to*

Optimize Decoding for more details. Also, it was eventually discovered that the path actually came within Hamming distance 7 of self-intersecting at several points, the closest being at Hamming distance 3 (see the program **mindist.c**, which calculates the closest non-local Hamming distance between points on the Golay walk). For these reasons, I no longer consider the Golay walk to be of interest, although the current software still uses the Golay walk, and consideration of them did lead to Helical Walks, which appear very useful.

5.12 Helical Walks

Helical walks were developed after discovering the shortcomings of the Golay walk. They are called "helical" walks because they maintain a minimum distance between non-local points on the walk as they wind their way, a bit at a time, through the hypercube. A count is kept of how many times each bit has been flipped during the course of the walk so far, and less frequently used bits are searched more often. The resulting walks are very balanced and have some nice provable features.

There are two parameters to the helical walks, n and m, where n is the dimension of the hypercube in which the walk takes place, and 2·m+1 is the minimum distance allowed between "non-local" points on the walk (points that are farther than 2·m+1 steps away).

Definition A *helical walk* H is an ordered sequence of points $<h_0, h_1, \ldots, h_n>$ such that

$$| h_i \oplus h_j |_H \begin{cases} = | i - j |, & | i - j | \leq 2 \cdot m \\ \geq 2 \cdot m + 1, & | i - j | > 2 \cdot m + 1 \end{cases} \quad \forall\, 0 \leq i,j \leq n$$

Like any other walks, a point p in the hypercube is decoded by finding a point h_i on the walk H such that $| p - h_i |_H \leq | p - h_j |_H \; \forall h_j \in H$. The point is decoded as i. Note that in general the point is not unique, as the following theorem attests.

Theorem A point p within m of a point h_i on the path will decode to a point h_j (i possibly equals j) such that $| h_j - h_i |_H \leq 2 \cdot m$. That is, it will decode to a point that is local to h_i.

Proof Suppose not. Then

$$| h_j - h_i |_H > 2 \cdot m$$

but

$$| p - h_i |_H \leq m \text{ (given)}$$

and

$$| p - h_j |_H \leq | p - h_i |_H \leq m$$

(h_j has to be the same distance or closer than h_i to p, else we would decode to h_i.) By the triangle inequality, this is a contradiction.

What happens if we flip more than 1 bit between steps on the walk? For helical walks as described above there is nothing that can be proven. However, for an improved form of helical walks we can prove something useful.

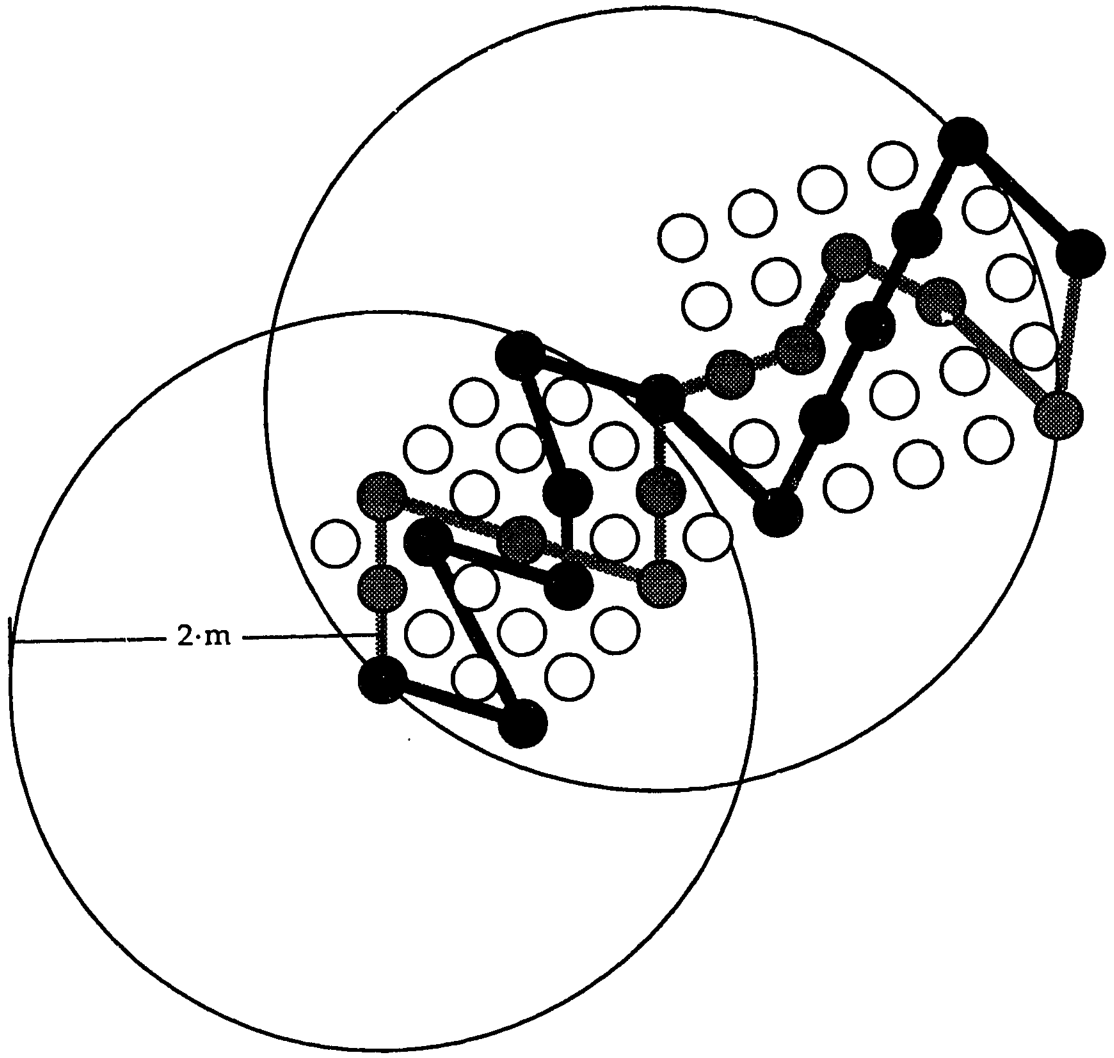
Definition A *helical parade* is an ordered sequence of points $<c_0, c_1, \ldots, c_n>$ such that

$$| c_i \oplus c_j |_H \begin{cases} = 0, & i = j \\ = 2\cdot m+1, & | i - j | = 1 \\ \geq 2\cdot m+1, & | i - j | \geq 2 \end{cases} \quad \forall\, 0 \leq i,j \leq n$$

So far this is not much different than a helical walk taking 2·m+1 steps at at time. However, we would like to be able to say that no point c_j comes within Hamming distance 2·m+1 of *any* path connecting two adjacent points, say c_i and c_{i+1}, that sticks to the bits in $c_i \oplus c_{i+1}$. So, for the case i < j-1, we add the further restriction:

$$| (c_i \oplus c_j) \text{ and } (\overline{c_i \oplus c_{i+1}}) |_H \geq 2\cdot m+1$$

A helical parade between two points can be visualized as:



Two steps on a helical parade. No other parade step is allowed to come within 2·m of any dot shown. Two possible helical paths are shown.

To derive a helical walk from a helical parade, we connect the centers by changing the 2·m+1 bits where they differ one bit at a time. A thermometer code on these bits would be just fine. (To connect 000 with 111 we could go 000,001,011,111 or 000,010,110,111.)

Theorem Let H be a helical walk derived from a helical parade, and for some k, $1 \le k \le 2 \cdot m+1$, let $R_k = \langle h_0, h_k, h_{2 \cdot k}, \rangle$. ($R_k$ is every k'th step in H.) Now let p be a point within m of a point $h_i \in$ H. Then p will decode to $h_j \in R_k$ such that $| h_i - h_j |_H \leq m$. (That is, p will decode to a point local to h_i.)

Proof

$| p - h_i |_H \leq m$ (given)

and

$\{h_{i-m}, \ldots, h_i, \ldots, h_{i+m}\}$ are all within m of h_i (since H is a helical walk)

now, one of {i-m, ..., i, ..., i+m} is evenly divisible by k, since $k \leq 2 \cdot m+1$ (call it j), and hence $h_j \in R_k$ and $| h_i - h_j |_H \leq m$. Therefore,

$| p - h_j |_H \leq 2 \cdot m$

However, by construction, all non-local points are > $2 \cdot m$ from h_j. Note that h_j may not be the point we decode to; we have just shown that at least one local point is within Hamming distance $2 \cdot m$ and no non-local point is within $2 \cdot m$.

Note that these strides through a helical walk constructed from a helical parade give us exactly the kind of error correction we want. If we stride at $k = 2 \cdot m+1$ steps, then any answer which comes within m of a point on the strided walk will decode to that point.

(*Note:* The helical walks were discovered too late into this thesis to be incorporated; the Golay walk was used instead.)

5.13 Combinatorial Hypercompression

Combinatorial hypercompression is a code developed by Hecht-Nielsen [Hech90, pp 210-214] where each feature is coded by setting k bits out of n to 1 (and the remaining n-k bits to 0). This allows the coding of $\binom{n}{k}$ features into n bits. For n = 1000 and k = 50 this allows 2^{282} features to be coded. A codeword x is decoded by having n linear threshold units of the form:

$$z_i = \sum_{j=1}^{n} \omega_{ij} \cdot \chi_j = \mathbf{w}_i \cdot \mathbf{x} \qquad (1 \leq i \leq n)$$

where $\mathbf{w}_i$ are unit length weight vectors. In the case where k=1 we would pick (by means of a competitive process) the unit i with z_i equal to the maximum of all the z_i. For k>1 the k largest units win the competition, and the feature is decoded accordingly.

The set of conditions for which the n values of $\mathbf{w}_i$ can be found is, in general, unknown. However, in the case where the **x** codewords are uniformly distributed over the entire n dimensional unit hypersphere sphere the $\mathbf{w}_i$ can be chosen to be the n orthonormal basis vectors (unit vectors on the axes of the hyperspace).

Note that combinatorial hypercompression is a tool for packing more features into n bits; it is not a tool for coding real intervals. It is included here as an interesting example of feature packing for linear threshold networks.

5.14 *Using Golay Codes to Optimize Decoding*

The 23 bit Golay codes uniformly partition the space of 23 dimensional bit vectors. We can use the Golay codes as sphere centers to decode efficiently all points within r of our walk:

1) (Offline) For each sphere center, record which points on the walk come within Hamming distance d=r+3, where r is the desired radius around the walk (the spheres induced by the Golay codes are of radius 3). Points outside this radius may not decode properly (see below).

2) To decode a point, calculate its sphere center then check only those points recorded in step 1.

Note that a sphere at distance r+3 from a point on the walk will have at least one interior point at distance r from the walk. Since the spheres are non-overlapping and exactly fill the space, every point in the space is in some sphere.

For the Golay walk described in section 5.11 - *Golay Walk* we get:

	r=2 *d=5*	*r=3* *d=6*	*r=4* *d=7*	*r=5* *d=8*
Maximum number of points recorded for any sphere	12	27	39	52
Minimum number of points recorded for any sphere	0	0	0	3
Average number of points recorded per sphere[†]	2	4	11	25
Number of spheres not within d of any point on walk	2593	589	10	0

That is, if we construct the tables above for r=3 (d=6), then at the cost of one Golay decode we only need to check an average of 4 points on the walk to find

† Not counting spheres with no points recorded.

the closest point on the walk. Our walk has 214 steps, so we are doing about 2% ($\frac{4}{214}$) of the work (on the average).

Note that we decode correctly only for points within r of the walk. To see why points farther than r from the walk decode incorrectly, consider:

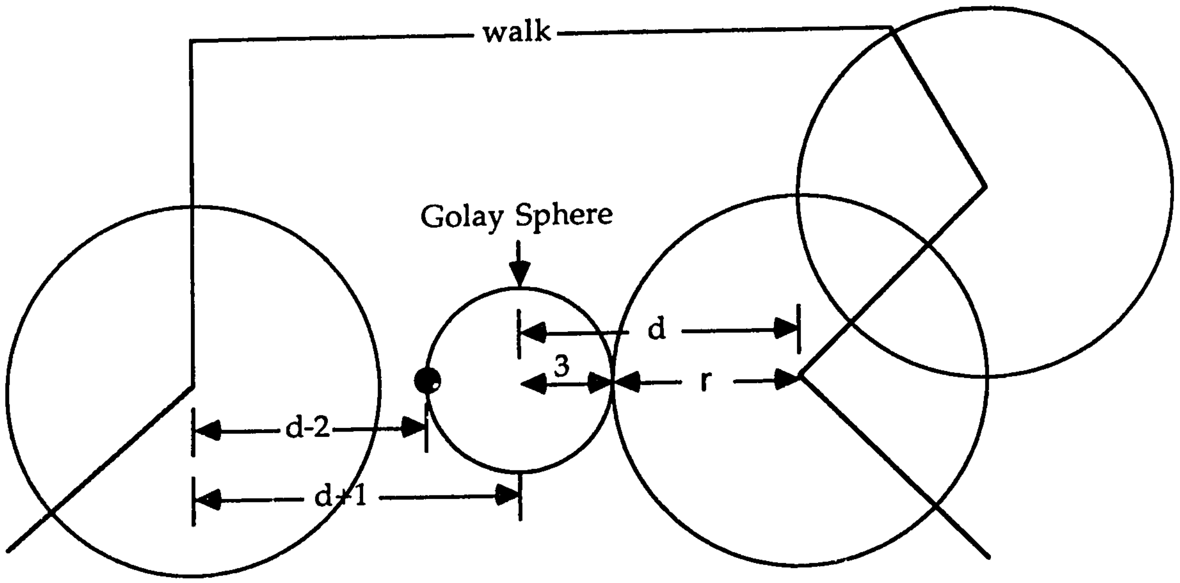


Illustration of a shortcoming in the optimized decoder. The point on the left is removed from consideration, even though it is the closest point.

Note that this method works for any walk over $\{0,1\}^{23}$ with a minimum distance greater than 2·r between non-local points on the walk. Since r=3 for the Golay walk used in this thesis, this method of fast decoding was not used. See the file **fastdecode.c** for programming details.

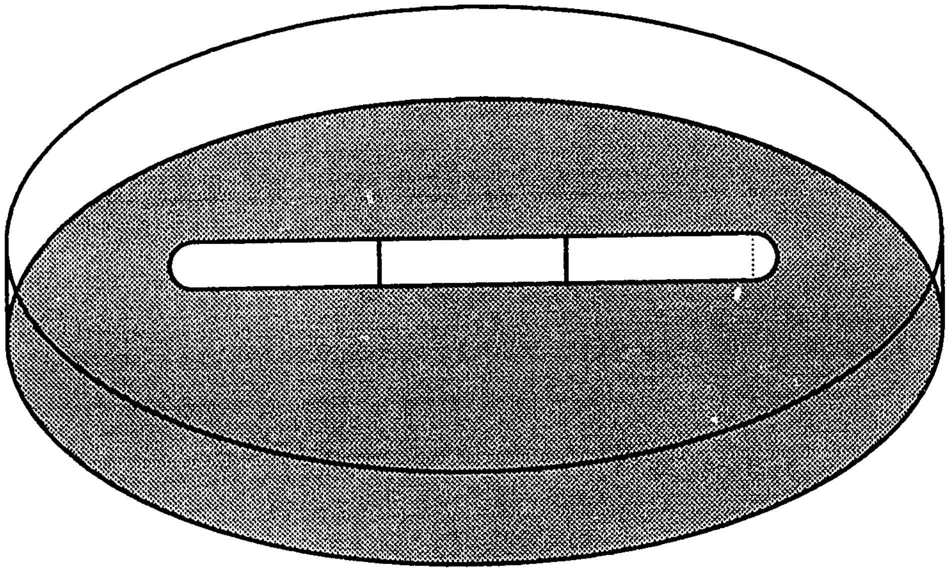
5.15 Hardware for Decoding Walks

Note that most of the walks used so far have been on the order of a few hundred steps long. Walks of 2^{10} (1024) steps will suffice for many applications for some time, easily fitting into the 2^{16} processing units available today.

Circuits to decode such small walks in time proportional to the width of the code independent of the number of steps should easily fit onto Programmable Gate arrays or similar technology. Foster describes a content-addressable parallel processor (CAPP) along with algorithms for finding the Hamming distance and minimum in time proportional to the width of the codewords, independent of the number of steps on the walk [Fost76,97,212].

6 The ALN predictor

The goal of this thesis was to use adaptive logic networks to predict the results of the model used in chapter 3 *Equations of Motion Simulator*. An existing program, *Dynatree*, was used that implemented this system for a three-segment "worm" rolling around in a circular walled arena [Arms85]. Since the environment was concave, the only points that need to be considered are the two end points and the two joints of the worm:



The worm in its lair

As Chapter 3 attests, the model has many variables. Which of these should be chosen for the predictor? The following section outlines the choices made and the motivation behind them.

6.1 Variables Learned

The brute force approach of learning all 18 variables, most of which are vectors and matrices, was quickly rejected. A biologically motivated decision was made not to include any variable that described the inherent properties of the worm itself. For example, the mass of each segment (m^r) was not included since you and I do not use the weight of our arm when deciding how to move it. Instead, we just move our arm around until we learn that "pushing that hard makes it go that fast." Similarly the acceleration of gravity (a_G), the position vector of the links (p^r), the position of the center of mass relative to the hinge (c^r), the forces (f^r) between the links, the relative position of the links (l^r), and the moment of inertia matrix (J^r)were not included. Nor were most of the rotation matrices (R^{rT}, R^r_I, and R^{rT}_I) used.

The variables chosen were the velocity (v^r), acceleration (a^r), angular velocity (ω^r), angular acceleration ($\dot{\omega}^r$) and torques (g^r) at each hinge (the constraining force (f^r) does not play a major role in the dynamics), the external forces (f^r_E) and torques (g^r_E), and the angle at each of the two joints in terms of roll and pitch (derived from (R^r)).† For consistency, and in keeping with an unsubstantiated expectation of the biological model, all variables were translated into the frame of the link.

Of these 8 variables, 6 are three dimensional vectors for each of the 3 links, the roll and pitch are 2 numbers for each of 2 links, and the torque is a three

† R. Lake argued persuasively that quaternions [Shoe85] should be used instead of roll and pitch, indeed that all the rotation matrices be recast as (the wonderful! mathematical objects) quaternions. While he convinced me that this is desirable, it is still on my list of "things to do later."

dimensional vector at each of the two interior hinges. That is, a total of 64 numbers that are both fed into and predicted by our ALN.

6.2 Quantizing the Variables

To determine the range of values each variable took on during the course of execution, the simulator was changed to determine the minimum, maximum, mean, and standard deviation. All available configuration files for *Dynatree* were then run, giving the following results:

Quantity	*Variable name*	*min*	*max*
a^r	aH	-500	500
f^r_E	FER†	-3000	4000
g^r_E	GEHR†	-3000	3000
g^r	gH	-150	150
(R^r)	roll&pitch	-1.57	1.57
ω^r	omega	-35	35
$\dot{\omega}^r$	omegadot	-1500	2000
v^r	vHR†	-18	11

Since the Golay walk was used, all variables were quantized to 214 levels using 23 bits.

6.3 Other ALN Parameters

Having 64 variables each quantized into 23 bits, gives $64 \cdot 23 = 1472$ input bits. Since both the input bit and its complement must be fed in, this requires a

† the R at the end of these variables names indicates they have been translated into the frame of the link.

tree with at least 1472·2=2944 leaves. Since it is highly desirable to have room for at least two copies of the input, and since tree sizes that are powers of two are marginally more efficient, the initial size of each tree was set to 8192 leaves (therefore having 8191 internal nodes).

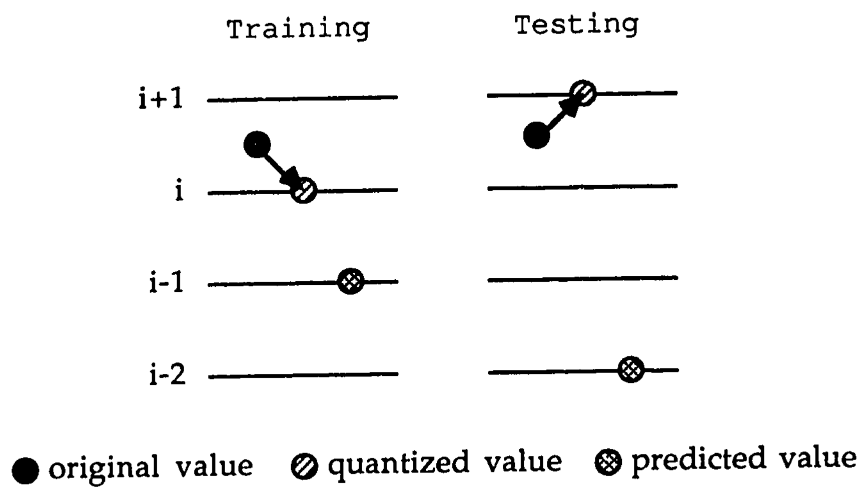
The trees were trained on 1000 input vectors for a maximum of 10 epochs (presentations of the training set). The training data was derived by instrumenting the original dynamics program and having it write out the values of all 64 variables after each time slice. A separate program (stride) was then run to pick out every nth line and its successor from this file, append the successor on the end of the nth line, and write it out. Hence the trees were trained with uniform samples of what the variables looked like before and after a time slice in the original program.

The trees were then tested on 9000 samples from the same data. One in nine tests was seen during training.

6.4 Analyzing the Results

6.4.1 What Can We Reasonably Expect?

Perfect prediction is an unrealistic goal for an ALN. The ALN sees and predicts *quantized* numbers. Suppose an output value v is quantized to level i in the test set, which causes the ALN to predict quantization level i in the output. Then during testing, a value $v'$is seen, where v' is just enough different from v to cause it to quantize to $i+1$. In this case, it is within reason for the ALN to predict any of quantization levels $i-1$, i, or $i+1$ as the result. So even "perfect" learning could cause the network to be off by one quantization level. If we allow the ALN to be off by one quantization level during training, a similar argument leads us to conclude that errors of 3 quantization levels are reasonable.



How "learn within one quantization level" turns into errors of three quantization levels. Note the original value on the right is slightly displaced from the one on the left

So, if the mean and the standard deviation suggest that 95% of the time we are within 3 quantization levels, we have an ALN that is very near optimal.

6.4.2 A View of the Raw Data

The complete ALN takes two to three days to train on a lightly loaded Sun SPARCStation I. The test results were the output of lf after piping them through **histogram**, The result is a histogram of errors that says, for each variable learned, "the ALN had m errors of magnitude n."

The results were characterized by having a small number of outliers - values a long way from the mean. While few in number, each has an enormous effect on the mean. For example, the results for the third variable (*a^r[1][z]*) look like this:

6300	at correct quantization level	11	out by 10 quantization levels
1854	out by 1 quantization level	8	out by 11 quantization levels
343	out by 2 quantization levels	7	out by 12 quantization levels
185	out by 3 quantization levels	4	out by 13 quantization levels
109	out by 4 quantization levels	3	out by 14 quantization levels
48	out by 5 quantization levels	6	out by 15 quantization levels
31	out by 6 quantization levels	2	out by 16 quantization levels
12	out by 7 quantization levels	3	out by 17 quantization levels
21	out by 8 quantization levels	8	out by 18 quantization levels
21	out by 9 quantization levels	6	out by 19 quantization levels

The remaining 18 values were spread between errors of quantization levels from 20 through 151. Removing the last 18 values cause the mean to drop from .6953 to .5795 and the standard deviation to drop from 3.4881 to 1.5434.

6.5 Statistical Methods Used

This section provides a brief description of each of the statistical methods used to analyze the test results from the ALN predictor. The descriptions are drawn from Harnett [Harn75] and Gellert et all [Gell75].

6.5.1 Mean

The mean of a collection of numbers is the average value, in our case the average number of quar'ization units of the error. Its formula is:

$$\mu = \frac{1}{n} \cdot \sum_{i=1}^{n} x_i$$

6.5.2 Standard Deviation

Standard deviation is a measure of the variability of the data. For data with a normal distribution (we have a Poisson distribution, since we do not record the sign of the errors; this does not in and of itself affect the standard deviation) the rule of thumb is 68% of all data is within one standard deviation, 95% within two. Its formula is:

$$\delta = \sqrt{\frac{1}{n-1} \cdot \sum_{i=1}^{n} (x_i - \mu)^2}$$

6.5.3 Standard Error of the Mean

This is a measure of the expected error when using the mean to make conclusions about the general population from the sample. Its formula is:

$$\delta_\mu = \frac{\delta}{\sqrt{n}}$$

6.5.4 5% Trim

This number is [illegible] of the data not including the most extreme 5% of the sample. [illegible] useful measure to us, since it tells us what would happen if the "outliers" (the errors that are a long way from the median) were removed.

6.5.5 Median

This is the value in the middle, $x_{\frac{n}{2}}$.

6.5.6 Interquartile Range (IQR)

The IQR is $x_{\frac{3 \cdot n}{4}} - x_{\frac{n}{4}}$. Since it contains the middle 50% of the values, and one standard deviation covers approximately 68% of the values, multiplying the IQR by $\frac{68}{50}$ gives us a rough approximation of the standard deviation. For our purposes, using the IQR to estimate the standard deviation gives us a good indication of what the standard deviation would look like without the outliers.

6.5.7 Maximum Error

This is the maximum error value recorded, x_n. (The minimum error in every case was 0).

6.6 Results

At the end of training, each variable was learned to the following accuracy:

Quantity	*Mean*	*Std Dev*	*Std Err*	*5% Trim*	*Median*	*IQR*	*Max*
a^r							
best	.6953	3.4881	.0368	.3332	.0000	1.0000	151.0000
worst	2.8976	9.3316	.0984	1.3563	1.0000	2.0000	168.0000
f^r_E							
best	.7422	2.8522	.0301	.4389	.0000	1.0000	161.0000
worst	4.0970	15.2223	.1605	1.4733	1.0000	2.0000	181.0000
g^r_E							
best	.7540	2.7485	.0290	.4857	.0000	1.0000	156.0000
worst	2.5918	9.2039	.0970	1.1065	1 0000	2.0000	150.0000
g^r							
best	1.1401	.9347	.0099	1.0737	1.0000	1.0000	10.0000
worst	1.5663	1.6206	.0171	1.3957	1.0000	2.0000	21.0000
R^r							
best	.5246	3.2197	.0339	.3900	.0000	1.0000	212.0000
worst	1.1103	8.1942	.0864	.6654	1.0000	1.0000	213.0000
ω^r							
best	1.0896	1.6230	.0171	.9591	1.0000	2.0000	95.0000
worst	2.5804	11.8330	.1247	.9584	1.0000	1.0000	190.0000
$\dot{\omega}^r$							
best	.9749	3.2439	.0342	.6127	.0000	1.0000	162.0000
worst	3.6872	10.5746	.1115	1.9826	1.0000	2.0000	190.0000
v^r							
best	.8927	1.5956	.0168	.7425	1.0000	1.0000	78.0000
worst	2.4616	9.6765	.1020	1.0393	1.0000	2.0000	166.0000

(*Notes:* f^r_E, g^r_E, and v^r were translated to the frame of the link before training, g^r is 0 for the first link (and hence was not included), and R^r was transformed into the roll and pitch (in radians) between links. For the complete table, see Appendix I - *Complete Training Results*. SPSSX was used to do the analysis.)

6.7 Conclusions

These results clearly show the viability of the ALN predictor, albeit with some caveats. The "best" mean and standard deviation for each variable are all reasonable approximations. The exciting numbers come from the 5% Trim and the IQR. These indicate in every case, that if the problem of the outliers can be solved, the ALN prediction would be very close to the best that can be expected.

How can these outliers be eliminated or reduced? Training three or five trees and doing a majority vote has improved performance of ALNs in similar circumstances [Arms90b], although the increased memory requirements (3 to 5 times) is a definite drawback. Early tests indicate no improvement with majority vote. A better walk for quantizing, like the helical walks of chapter 5, may help reduce the number of spurious errors. Finally, the simple expedient of not allowing any variable to change value by more than 10% should both limit the range of the errors seen, and reduce the time taken to dequantize (we only need to check 20% of the steps on the path).

7 Modifications to Software

This thesis involved a lot of programming. The following briefly outlines the major areas of development.

7.1 Dynamics

7.1.1 Statistics Gathering

In order to effectively quantize the variables used by the ALN predictor, the minimum, maximum, mean, and standard deviation of each of the variables needed to be accumulated. The file **stats.c** contains a moderately clever implementation of mean and standard deviation that allows incremental updates of the values to date. That is, it calculates the mean and standard deviation of n numbers from the mean and standard deviation of the first n-1 numbers and the nth number. For an excellent reference work on much of the field of mathematics including statistics, see Gellert et al [Gell75].

Note that just finding the minimum and maximum is not good enough. From the mean and standard deviation we can tell when we should be focussing more on the typical values than on the entire range. The software is set up so that values outside the minimum and maximum used for quantizing are mapped onto the closest endpoint.

7.1.2 Journaling of Values

After the effective range of values was discovered, the raw data needed to be written out for training. This was a straightforward insertion of print statements into **main.c**.

7.2 lf

7.2.1 Multiple Codomains

The biggest deficiency, from the perspective of this thesis, with the language **lf** [Dwel90], was that it only supported one codomain. So the YACC grammar for lf in synan.y was modified to add the following statements:

```
codomain dimension = {integer}
```

The associated code was also added to **lf.c**.

7.2.2 Save and Restore

The grammar for **lf** was modified to allow training of a previously created tree to be specified instead of specifying the function and the tree. The program definition was changed from:

```
     program : function_spec tree_spec
             | tree_spec function_spec
             ;
to
     program : function_spec tree_spec
             | tree_spec function_spec
             | train_spec
             ;
```

and the following definitions were added at the bottom:

```
train_spec : TRAIN train_statements
           { train_flag = TRUE; }

train_statements : train_statement
                 | train_statements train_statement
                 ;

train_statement : min_correct
                | max_epochs
                | train_table_size
                | train_table
                | test_table_size
                | test_table
                ;
```

All the referred symbols to were already defined. This also required changes to lf.c.

7.3 Atree

7.3.1 Fast-Trees

In section 6.3 - *Other ALN Parameters,* we mentioned that we have 64 · 23 trees, each initially with 8192 leaves (and 8191 interior nodes). Each internal node takes 8 bytes of storage; each leaf node takes 4 bytes. Since the number of internal nodes in a binary tree is 1 less than the number of leaves, the total memory requirements for the trees are: 64 · 23 · 8192· 12 = 144,703,488 ≅ 138M, which is clearly unmanageable.

However, this is before compression; each tree has a lot of **left** and **right** nodes which are not necessary after training. The unused subtree in each case may also be deleted. Typical compressions run between 5-30% of the original size, taking us down to the range 6.90 to 41.4M.

Further reduction of memory requirements is still possible, since we do not need to store the interior nodes of an **and-or** tree. To see this, consider a leaf of the tree: it must be evaluated if and only if all the **ands** on its path to the root have 1's preceeding, and all the **ors** have 0's preceding, otherwise the leaf's input will not affect the output of the tree.

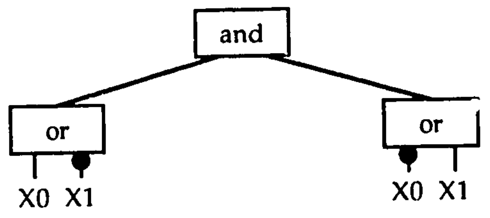
Conceptually, for each leaf we store:

bit_index	complement_flag	branch_if_0	branch_if_1

We look up the input bit using the index, complement it if complement_flag tells us to, then follow one branch or the other depending on the result.

Alternatively, a fast tree can be viewed as a binary decision tree based on the possible values of the input.

For example,



becomes:

Leaf No.	Bit_index	Complement	Branch_if_0	Branch_if_1
0:	0	0	1	2
1:	1	1	4	2
2:	0	1	4	3
3:	1	0	4	4

The output of the tree is the last bit value we looked at. Note that branches to leaf 4 stop the calculation.

To convert a tree to a fast_tree we can use a global array ft indexed from 0 to the number of leaves (nleaves) - 1, a global variable leafnum to keep track of our location in ft, and a post order recursive traversal function ftc:

```
ftc( node, next_if_0, next_if_1 )
atree *node;
int next_if_0, next_if_1;
{
        switch( node -> tag ) {
        leaf:
                ft[leafnum].branch_if_0 = next_if_0;
                ft[leafnum].branch_if_1 = next_if_1;
                ft[leafnum].bit_index = node -> bit_index;
                ft[leafnum].complement = node -> complement;
                leafnum--;
                break;
        and:
                ftc( node -> right, next_if_0, next_if_1 );
                ftc( node -> left,  next_if_0, leafnum+1 );
                break;
        or:
                ftc( node -> right, next_if_0, next_if_1 );
                ftc( node -> left,  leafnum+1, next_if_1 );
                break;
        }
}
```

To start things off:

```
leafnum = nleaves - 1;
ftc( root, nleaves, nleaves );
```

Note that one of the two branches will always be to the next leaf. (If this were not true, then one of the leaves could be cut from the tree with no effect.) So to get rid of the branch_if_0 field we set the complement flag so that we go to the next leaf if the (possibly complemented) input bit is a 0, and follow the branch_if_1 pointer if it is 1. A second flag field (complement_out) is used to keep track of whether or not the output of the tree needs to be complemented. This flag only needs to be checked if this is the last leaf processed. The case for a leaf in the above code becomes:

```
leaf:
     if( next_if_0 == leafnum + 1 ) {
          ft[leafnum].branch_if_1 = next_if_1;
          ft[leafnum].bit_index = node -> bit_index;
          ft[leafnum].comp_input = node -> complement;
          ft[leafnum].comp_output = 0;
     } else {
          assert( next_if_1 == leafnum + 1 );
          ft[leafnum].branch_if_1 = next_if_1;
          ft[leafnum].bit_index = node -> bit_index;
          ft[leafnum].comp_input = !(node -> complement);
          ft[leafnum].comp_output = 1;
     }
     leafnum--;
     break;
```

Since our trees have at most 2^{13} (8192) leaves, and our input vector is 64·23 bits long which is less than $2^{6}\cdot 2^{5}$, with our two complement bits we need 13+6+5+2 = 26 bits which easily fits into a four byte integer. The total tree size was 4.2M after compression and conversion to the above tabular form.

This tabular form is called fast_tree, because it allows us to evaluate the tree in about 30% of the time of the original form (according to R. Mandershied; I did not verify this). Code for converting regular trees (atrees) to and from fast_trees, and for saving and restoring fast_trees was written and added to **atree.c**.

7.3.2 Golay Walks

A fair bit of time was spent familiarizing myself with Golay codes, and developing the code that produced the Golay walk. See **golay.c**, and **mindist.c** for representative samples.

The resulting walk was incorporated into atree_rand_walk in **atree.c**, being automatically selected if the width of the walk was 23 and the number of steps times the stride equaled 214 (the number of steps on the Golay walk). Note that a fixed walk is necessary; the ALNs are trained relative to a given walk. If we save the ALN, we must either save the walk, or, as in our case, be able to reproduce it. My personal feeling is that good walks should be developed off-line and used when appropriate.

7.4 Utilities

7.4.1 fold

fold takes two adjacent journal output lines from the dynamics and appends one onto the end of the other. So lines 1 and 2 get appended, then line 2 and 3 get appended, etc. This makes each line output contain the value of all 64 variables before and after each time step (for a total of 128 values per line).

7.4.2 stride

stride goes through the massive data files produced by fold and extracts every n'th line, where n is a parameter. It can also be told to stop after m lines of output, where m is a second, optional parameter.

7.4.3 right

right was used originally to delete all but the 4 rightmost columns of numbers in the output produced by **lf**, which was producing 130+ numbers per line (all input 64 variables, their quantization step, the actual and expected output and associated quantization steps). This broke both **awk** and **nawk** which were used by **histogram** to get the values reported in section 6.4 - *Results.*

I decided it was more efficient to just modify **lf** to not write out the input variables. I also discovered a UNIX utility, **cut**, that can perform similar duties.

7.4.4 make.worm.lf

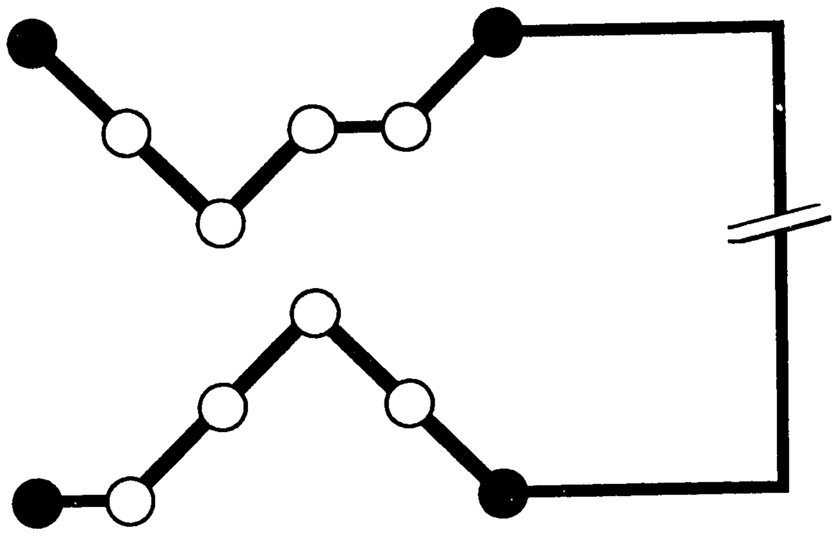
This shell script builds a file called **worm.lf**, that contains the input for **lf**. It sticks the quantization information and the tree description at the top of the file, uses **stride** to produce the training data, sticks some **lf** lines in to mark the start of the test data, and finally uses **stride** to produce the test data. It can be quickly configured by changing some shell variables at the start of the file.

7.4.5 squash

This utility takes an ALN that has been saved by atree, compresses it and converts it to a fast_tree, and then writes it out.

7.4.6 mindist

This utility performs an exhaustive search to find the closest the Golay walk comes to intersecting itself. (The answer, alas, is 3.) A more careful walk between the sphere centers could probably increase this distance. (I used a thermometer code to connect the sphere centers found by the algorithm in section 5.11 - *Golay Walk*. While the sphere centers are at least Hamming distance 7 apart, a random walk between the centers can, and did, veer towards another point on the walk.) One way to improve the walk would be to choose different connecting walks between the two pairs of Golay spheres involved, then iterating until the minimum distance is at a desirable level.


How the Golay walk can come so close to intersecting through poor choice of connecting steps

8 Conclusions

In this thesis, adaptive logic networks were shown to be capable of reasonable predictions of motion using a moderate (5 megabytes) amount of memory. Since the original tree size was 8192 leaves on a balanced tree, if implemented in hardware, each tree would take a maximum of 13 propagation delays to compute. In section 5.15 - *Hardware for Decoding Walks*, a circuit was referenced that can decode a result in twice as many cycles as the width of the codewords. Since our codewords are 23 bits long, the resulting system could take on the order of $13+23\cdot2 = 59$ times the speed of an **and** or **or** gate's propagation delay.

Along the way we discovered two useful walks for use in coding (section 5.12 - *Helical Walks*), a fast method for decoding a class of useful walks over $\{0,1\}^{23}$ (section 5.14 - *Using Golay Codes to Optimize Decoding*), and a computationally and memory efficient method of representing ALNs on sequential machines (section 7.3.1 - *Fast-Trees*).

Given more opportunity to work on this program, I would like to finish installing the ALN predictor into the dynamics program, switch to helical walks for the coding, and recast the dynamics routines using quaternions instead of rotation matrices.

9 Bibliography

[Ande88] Andersson, R. L. 1988. *A Robot Ping-Pong Player: Experiment in Real-Time Intelligent Control,* Cambridge: Massachusetts: The MIT Press.

[Arms79] Armstrong, W. W. Gecsei, J. 1979. Adaption Algorithms for Binary Tree Networks, in *IEEE Transactions on Systems, Man, and Cybernetics,* Vol 9, no. 5, May 1979. pp 276-285.

[Arms85] Armstrong, W. W., Green, M. 1985. The Dynamics of Articulated Rigid Bodies for Purposes of Animation, in *Proceedings of Graphics Interface 85.* pp 407-415.

[Arms90a] Armstrong, W. W., Liang, J., Lin, D., Reynolds, S. 1990. *Experiments using Parsimonious Adaptive Logic,* Technical Report TR 90-30. Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada.

[Arms90b] Armstrong, W. W. 1990. *Lazy Learning in Logic Networks,* Draft version June 15, 1990.

[Berl68] Berlekamp, E. R. 1968. *Algebraic Coding Theory.* New York: McGraw Hill.

[Blum87] Blume, C., Jakob, W., Favaro, J. 1987. *PasRo: Pascal and C for Robots, second Extended Edition.* New York: Springer-Verlag.

[Boch74] Bochmann, G. V., Armstrong, W. W. 1974. Properties of Boolean Functions with a Tree Decomposition, in *BIT 14.* pp 1-13.

[Donn87] Donner, M. D. 1987. *Real-Time Control of Walking.* Boston: Birkhäuser.

[Dwel90] Dwelly, A. 1990. *An Implementation of Adaptive Logic Networks.* Draft version November 11, 1990.

[Fost76] Foster, C. C. 1976. *Content Addressable Parallel Processors*. New York: Van Nostrand Reinhold.

[Gell75] Gellert, W., Küstner, H., Hellwich, M., Kästner (Eds) 1975. *The VNR Concise Encyclopedia of Mathematics*. New York: Van Nostrand Reinhold.

[Gesc83] Geschke, C. G. 1983. A System for Programming and Controlling Sensor-Based Robot Manipulators. In Lee, C. S. G., Gonzalez, R. C., Fu, K. S. (Eds), *Tutorial on Robotics, second edition 1986*, Los Angeles: California: IEEE Computer Society Press. pp 560-566.

[Gold85] Goldman, R. 1985. *Design of an Interactive Manipulator Programming Environment*, Ann Arbour, Michigan: UMI Research Press.

[Gruv84] Gruver, W. A., Soroka, B. I., Craig, J. J., Turner, T. L. 1984. Industrial Robot Programming Languages: A Comparative Evaluation. In Lee, C. S. G., Gonzalez, R. C., Fu, K. S. (Eds), *Tutorial on Robotics, second edition 1986*. Los Angeles, California: IEEE Computer Society Press. pp 455-475.

[Guez88] Guez, A., Selinsky, J. 1988. A Trainable Neuromorphic Controller. In *Journal of Robotic Systems*, Volume 5, Number 4. pp 363-388.

[Harn75] Harnett, D. L. 1975. *Introduction to Statistical Methods, Second Edition*. Don Mills, Ontario: Addison-Wesley.

[Hech90] Hecht-Nielsen, R. 1990, *Neurocomputing*, Don Mills, Ontario: Addison-Wesley.

[Koza90] Kozakiewicz, C., Ogiso, T., Miyake, N. P. 1990. Calibration Analysis of a Direct Drive Robot, in *Proceedings of IROS'90 IEEE International Workshop on Intelligent Robots and Systems'90*. pp 213-220.

[Lake90] Lake, R. M. 1970. *Dynamic Motion Control of an Articulated Figure*. Masters Thesis, University of Alberta, April 1990.

[Lieb77] Lieberman, L. I., Wesley, M. A. 1977. AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly. In Lee, C. S. G., Gonzalez, R. C., Fu, K. S. (Eds), *Tutorial on Robotics, second edition 1986*. Los Angeles: California: IEEE Computer Society Press. pp 540-552.

[Loza83] Lozano-Pérez, T. 1983. Robot Programming. In Lee, C. S. G., Gonzalez, R. C., Fu, K. S. (Eds), *Tutorial on Robotics, second edition 1986*. Los Angeles: California: IEEE Computer Society Press. pp 455-475.

[MacW77] MacWilliams, F. J., Sloane, N. J. A. 1977. *The Theory of Error Correcting Codes*, New York: North Holland.

[Meis90] Meisel, W. S. 1990. Parsimony in Neural Networks. In *Proceedings IJCNN-90-WASH-DC*, volume 1. pp 443-446.

[Mel90] Mel, B. W. 1990. *Connectionist Robot Motion Planning: A Neurally-Inspired Approach to Visually-Guided Reaching*. Toronto: Academic Press, Inc.

[Mill88] Miller, G. S. P. 1988. The Motion of Snakes and Worms. In *SIGGRAPH '88 Conference Proceedings*. pp 169-178.

[Mits85] Mitsuishi, M., Shimoyama, I., Miura, H. 1985. Development of Concurrency Oriented Language "COL". In *Proceedings of the '85 International Conference on Advanced Robotics*. pp 87-94.

[Moha88] Mohamed, A. S., Armstrong, W. W. 1988. Measuring Learning Progress in Intelligent Autonomous Robots, *Journal of Robotic Systems*, Volume 5, Number 6. pp 583-607.

[Moha90] Mohamed, A. S. 1990. *A Neural Trees Architecture for Rule-Based Control of Motion*, (unpublished).

[Mujt82] Mujtaba, M. S., Goldman, R., Binford, T. 1982. The AL Robot Programming Language. In Lee, C. S. G., Gonzalez, R. C., Fu, K. S. (Eds), *Tutorial on Robotics, second edition 1986*. Los Angeles: California: IEEE Computer Society Press. pp 530-539.

[Naka85] Nakano, M., et al 1985. TL-10: A Programming Language for Assembly Robots. In *Journal of Robotic Systems*, Volume 2, Number 3. pp 277-288.

[Raib86] Raibert, M. H. 1986. *Legged Robots That Balance*. Cambridge, Massachusetts: The MIT Press.

[Shim84] Shimano, B. E., Geschke, C. C., Spaling III, C. H. 1984. VAL-II: A New Robot Control System for Automatic Manufacturing. In Lee, C. S. G., Gonzalez, R. C., Fu, K. S. (Eds), *Tutorial on Robotics, second edition 1986*. Los Angeles, California: IEEE Computer Society Press. pp 476-490.

[Shoe85] Shoemake, K. 1985. Animating Rotation with Quaternion Curves, *Proceedings of SIGGRAPH 85*, vol 19, no. 3. pp 245 - 254.

[Shri89] Shriver, B. 1989. *Artificial Neural Systems.* (Video Notes from an International IEEE Broadcast.) Los Alamitos, California: IEEE Computer Society Press.

[Taka81] Takase K., Paul, R. P. 1981. A Structured Approach to Robot Programming and Teaching. In Lee, C. S. G., Gonzalez, R. C., Fu, K. S. (Eds), *Tutorial on Robotics, second edition 1986.* Los Angeles: California: IEEE Computer Society Press. pp 514-529.

[Tayl82] Taylor, R. H., Summers, P. D., Meyer, J. M. 1982. AML: A Manufacturing Language. In Lee, C. S. G., Gonzalez, R. C., Fu, K. S. (Eds), *Tutorial on Robotics, second edition 1986.* Los Angeles: California: IEEE Computer Society Press. pp 491-513.

Appendices

I Complete Training Results

Quantity	*Mean*	*Std Dev*	*Std Err*	*5% Trim*	*Median*	*IQR*	*Max*
$a^r[1][x]$	.9567	3.8785	.0409	.4937	.0000	1.0000	123.0000
$a^r[1][y]$	1.3177	5.1049	.0538	.6738	1.0000	1.0000	127.0000
$a^r[1][z]$	.6953	3.4881	.0368	.3332	.0000	1.0000	151.0000
$a^r[2][x]$	1.5458	4.8937	.0516	.9333	1.0000	1.0000	164.0000
$a^r[2][y]$	2.8976	9.3316	.0984	1.3563	1.0000	2.0000	168.0000
$a^r[2][z]$	1.2856	5.5183	.0582	.6627	1.0000	1.0000	187.0000
$a^r[3][x]$	.9876	3.9478	.0416	.5246	.0000	1.0000	146.0000
$a^r[3][y]$	2.3504	9.1580	.0965	.8891	.0000	1.0000	194.0000
$a^r[3][z]$	.9398	4.1653	.0439	.4772	.0000	1.0000	134.0000
$f^r_E[1][x]$	1.0074	4.0569	.0428	.5957	.0000	1.0000	171.0000
$f^r_E[1][y]$	1.4591	5.9348	.0626	.6352	.0000	1.0000	147.0000
$f^r_E[1][z]$	1.4899	9.5086	.1002	.6569	1.0000	1.0000	208.0000
$f^r_E[2][x]$	2.3297	7.9880	.0842	1.0864	1.0000	2.0000	169.0000
$f^r_E[2][y]$	1.4849	5.6262	.0593	.8546	1.0000	1.0000	166.0000
$f^r_E[2][z]$	4.0970	15.2223	.1605	1.4733	1.0000	2.0000	181.0000
$f^r_E[3][x]$	.7422	2.8522	.0301	.4389	.0000	1.0000	161.0000
$f^r_E[3][y]$	1.4091	5.4494	.0574	.7932	1.0000	1.0000	130.0000
$f^r_E[3][z]$	.9499	4.3894	.0463	.6036	1.0000	1.0000	171.0000
$g^r_E[1][x]$	2.1521	8.9944	.0948	1.0720	1.0000	2.0000	175.0000
$g^r_E[1][y]$	1.0352	5.1535	.0543	.6065	1.0000	1.0000	190.0000
$g^r_E[1][z]$	2.5898	8.6725	.0914	1.2805	1.0000	2.0000	135.0000
$g^r_E[2][x]$	1.6510	6.2307	.0657	.8709	1.0000	1.0000	186.0000
$g^r_E[2][y]$	2.3092	8.8115	.0929	1.0235	1.0000	2.0000	154.0000
$g^r_E[2][z]$	2.5918	9.2039	.0970	1.1065	1.0000	2.0000	150.0000
$g^r_E[3][x]$	1.3683	4.9512	.0522	.8756	1.0000	1.0000	205.0000
$g^r_E[3][y]$	.7540	2.7485	.0290	.4857	.0000	1.0000	156.0000
$g^r_E[3][z]$	1.2713	4.6701	.0492	.7786	1.0000	1.0000	123.0000

Quantity	*Mean*	*Std Dev*	*Std Err*	*5% Trim*	*Median*	*IQR*	*Max*
$g^r[2][x]$	1.1401	.9347	.0099	1.0737	1.0000	1.0000	10.0000
$g^r[2][y]$	1.3799	1.2452	.0131	1.2780	1.0000	1.0000	21.0000
$g^r[2][z]$	1.4026	1.5112	.0159	1.2377	1.0000	2.0000	31.0000
$g^r[3][x]$	1.2654	1.4396	.0152	1.1293	1.0000	2.0000	69.0000
$g^r[3][y]$	.8130	1.3346	.0141	.6785	1.0000	1.0000	82.0000
$g^r[3][z]$	1.5663	1.6206	.0171	1.3957	1.0000	2.0000	21.0000
roll[1]	.9010	3.3505	.0353	.7488	1.0000	1.0000	211.0000
pitch[1]	.8601	6.1298	.0646	.5415	.0000	1.0000	213.0000
roll[2]	.5246	3.2197	.0339	.3900	.0000	1.0000	212.0000
pitch[2]	1.1103	8.1942	.0864	.6654	1.0000	1.0000	213.0000
$\omega^r[1][x]$	2.4372	8.8949	.0938	1.3742	1.0000	2.0000	193.0000
$\omega^r[1][y]$	2.5804	11.8330	.1247	.9584	1.0000	1.0000	190.0000
$\omega^r[1][z]$	1.3923	1.7540	.0185	1.2020	1.0000	2.0000	69.0000
$\omega^r[2][x]$	1.1567	4.3201	.0455	.8043	1.0000	1.0000	199.0000
$\omega^r[2][y]$	1.0654	6.1574	.0649	.5857	1.0000	1.0000	178.0000
$\omega^r[2][z]$	1.0896	1.6230	.0171	.9591	1.0000	2.0000	95.0000
$\omega^r[3][x]$	1.4718	6.1435	.0648	.9116	1.0000	1.0000	190.0000
$\omega^r[3][y]$	1.0384	4.4601	.0470	.4953	.0000	1.0000	103.0000
$\omega^r[3][z]$	1.3601	1.9175	.0202	1.2133	1.0000	2.0000	109.0000
$\dot{\omega}^r[1][x]$	1.7654	5.9808	.0630	.8331	1.0000	1.0000	127.0000
$\dot{\omega}^r[1][y]$	1.5493	5.3569	.0565	.8032	1.0000	1.0000	113.0000
$\dot{\omega}^r[1][z]$	2.4266	8.2928	.0874	1.1459	1.0000	2.0000	147.0000
$\dot{\omega}^r[2][x]$	2.5163	8.3645	.0882	1.1831	1.0000	2.0000	202.0000
$\dot{\omega}^r[2][y]$	1.5250	5.6957	.0600	.7959	1.0000	1.0000	133.0000
$\dot{\omega}^r[2][z]$	3.6872	10.5746	.1115	1.9826	1.0000	2.0000	190.0000
$\dot{\omega}^r[3][x]$	1.8177	5.4913	.0579	.9964	1.0000	1.0000	119.0000
$\dot{\omega}^r[3][y]$	.9749	3.2439	.0342	.6127	.0000	1.0000	162.0000
$\dot{\omega}^r[3][z]$	3.5831	9.8990	.1043	1.9688	1.0000	2.0000	166.0000
$v^r[1][x]$	2.4616	9.6765	.1020	1.0393	1.0000	2.0000	156.0000
$v^r[1][y]$	1.2039	4.[illegible]	.0436	.8238	1.0000	1.0000	151.0000
$v^r[1][z]$	1.2160	6.1453	.0648	.5004	.0000	1.0000	130.0000
$v^r[2][x]$	.8421	4.9815	.0525	.5556	1.0000	1.0000	198.0000
$v^r[2][y]$	.9913	2.5479	.0269	.7832	1.0000	1.0000	130.0000
$v^r[2][z]$	1.6767	8.7233	.0920	.6068	1.0000	1.0000	174.0000
$v^r[3][x]$	1.5726	7.7803	.0820	.9511	1.0000	2.0000	180.0000
$v^r[3][y]$	.8927	1.5956	.0168	.7425	1.0000	1.0000	78.0000
$v^r[3][z]$	.6168	3.1632	.0333	.3644	.0000	1.0000	130.0000

II Code Listings

The code listings for this thesis are contained on a Macintosh 800K diskette. If you did not get a diskette with this thesis, contact the author.