MAP LEARNING WITH UNINTERPRETED SENSORS AND EFFECTORS

by

DAVID MARK PIERCE, B.S., B.A., M.S.C.S.

DISSERTATION
Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN
May, 1995

MAP LEARNING WITH UNINTERPRETED SENSORS AND EFFECTORS

Publication No._____

DAVID MARK PIERCE, Ph.D.
The University of Texas at Austin, 1995

Supervising Professor: Benjamin J. Kuipers

This dissertation presents a set of methods by which a learning agent, called a "critter," can learn a sequence of increasingly abstract and powerful interfaces to control a robot whose sensorimotor apparatus and environment are initially unknown. The result of the learning is a rich, hierarchical model of the robot's world (its sensorimotor apparatus and environment). The learning methods rely on generic properties of the robot's world such as almost-everywhere smooth effects of actions on sensory features.

At the lowest level of the hierarchy, the critter analyzes the effects of its actions in order to define control signals, one for each of the robot's degrees of freedom. It uses a generate-and-test approach to define sensory features that capture important aspects of the environment. It uses linear regression to learn action models that characterize context-dependent effects of the control signals on the learned features. It uses these models to define high-level control laws for finding and following paths defined using constraints on the learned features. The critter abstracts these control laws, which interact with the continuous environment, to a finite set of actions that implement discrete state transitions. At this point, the critter has abstracted the robot's world to a finite-state machine and can use existing methods to learn its structure.

Table of Contents

Chapter 1

Introduction

This dissertation addresses the following problem: Given a robot with an uninterpreted sensorimotor apparatus situated in a continuous, static environment, how can a learning agent (henceforth called a "critter") learn descriptions of the structure of the sensorimotor apparatus and environment suitable for prediction and navigation? Here, and in the rest of the dissertation, I make a distinction between the critter and the robot. The robot is a machine (physical or simulated) and the critter is a learning agent that learns how to use that machine. The robot's sensorimotor apparatus is comprised of a set of sensors and effectors. By "uninterpreted," I mean that the critter does not initially know what the robot's sensors are sensing nor how the effectors move the robot about in its environment. From the critter's perspective, the sensorimotor apparatus is represented as a *raw sense vector* $\mathbf{s}$ and a *raw motor control vector* $\mathbf{u}$. The former is a vector of real numbers giving the current values of all of the sensors. The latter is a vector of real numbers, called control signals, produced by the critter and sent to the robot's motor apparatus. The critter's situation is illustrated in Figure 1.1.

This dissertation solves the learning problem by presenting a set of methods that the critter can use to learn (1) a model of the robot's set of sensors, (2) a model of the robot's motor apparatus, and (3) a set of behaviors that allow the critter to abstract the robot's continuous world to a discrete world of places and paths. These methods have been demonstrated on a simulated mobile robot with a ring of distance sensors.

These learning methods comprise a body of knowledge that is given to the critter *a priori*. They incorporate a knowledge of basic mathematics, multivariate analysis, computer vision, and control theory. A list of the learning methods and the *a priori* knowledge that they embody will be given in Section 1.10. The learning methods are domain independent in that they are not based on a particular set of sensors or effectors and do not make assumptions about the structure or even the dimensionality of the robot's environment.

To better understand the problem that the critter faces, imagine yourself in its place. Imagine that you are in front of the control console for a teleoperated robot. At the left of the console is a display giving your sensory input from the robot. All that you will ever see of the environment comes via that vector of numbers. Next to this display is a joystick. You know that this can be used to move the robot through its environment and you know that the robot does not move when the joystick is in its zero position but you have no idea how the joystick affects the robot's motion. Your mission is to develop a model of the robot's environment as well as its sensorimotor interface to that environment. At the lowest level, you need to develop an understanding of the robot's sensorimotor apparatus — you need to know the effects of the control signals on the sense

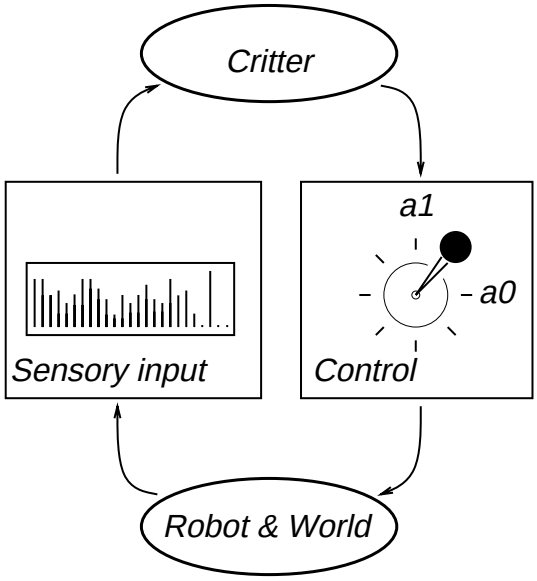


Figure 1.1: The learning problem addressed in this dissertation is illustrated by this interface between a learning agent, called a "critter," and a teleoperated robot in an unknown environment. The critter's problem is to learn a model of the robot and its environment with no initial knowledge of the meanings of the sensors or the effects of the control signals.

vector. At a higher level, you need to understand the robot's environment. The robot could be a mobile robot in an office building or laboratory or it could be a submarine swimming in the ocean. The problem you face is analogous to that faced by a cryptanalyst trying to decipher an enemy's code. Like the cryptanalyst, you need a set of tools to apply to the problem. Developing a set of appropriate tools is the subject of this dissertation. Sections 1.2 through 1.7 introduce a number of these tools (i.e., learning methods) and show how they are used by a critter as it develops an understanding of a robot's world by learning a sequence of increasingly abstract and powerful interfaces to the robot. Section 1.8 gives a formal definition of the general learning problem and solution.

1.1 Example: Learning to control a mobile robot

For concreteness, the learning methods will be illustrated with a particular robot and environment. The robot's world is simulated as a rectangular room of dimensions 6 meters by 4 meters. The room has a number of walls and obstacles in it. The robot itself is modeled as a point. The robot has 29 sensors. Each sensor's value lies between 0.0 and 1.0. Collectively, these define the raw sense vector $\mathbf{s}$, which is the input from the robot to the critter. The first 24 components of the raw sense vector give the distances to the nearest objects in each of 24 directions. These have a maximum value of 1.0 which they take on when the nearest object is beyond one meter away. The sonars are numbered clockwise from the front. The 21st component is defective and always returns a value of 0.2. The 25th component is a sensor giving the battery's voltage, which decreases slowly

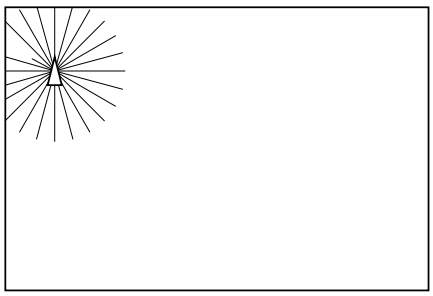


Figure 1.2: The robot that the critter learns to control has 24 distance sensors (one of which is defective), a battery-voltage sensor, and a digital compass. The robot lives in a 6m x 4m rectangular room.

from an initial value of 1.0. The 26th through 29th components comprise a digital compass. The component with value 1 corresponds to the direction (E, N, W, or S) in which the robot is most nearly facing. The robot has a "tank-style" motor apparatus. Its two motor control signals a_0 and a_1 tell how fast to move the right and left treads. Moving the treads together at the same speed produces pure forward or backward motion; moving them in opposition at the same speed produces pure rotation. Moving the treads at different speeds causes the robot to move in a circular arc. The robot and its environment are shown in Figure 1.2.

The critter does not know what any of these sensors or effectors do. The critter only knows that that robot's raw sense vector has 29 elements and its raw motor control vector has two elements.

1.2 Learning a model of the sensory apparatus

The critter's first task is to model the robot's sensory apparatus. The critter develops an understanding of the robot's sensory apparatus by learning new *features*. A *feature*, as defined in this dissertation, is a function over time whose current value is completely determined by the history of current and past values of the robot's raw sense vector. The type of the feature is determined by the type of that function's value. The types of features used in this dissertation include the following: *scalar*, *vector*, *matrix*, *image*, and *field*. The first three types are based on standard mathematical constructs. The image and field features will be defined later. Examples of features are the raw sense vector (a vector feature) and the elements of the raw sense vector (scalar features).

The critter produces new features using *feature generators*. A feature generator is a rule that creates a new feature or set of features based on already existing features.

1.2.1 The group feature generator

A sensory apparatus may contain a structured array of similar sensors. Examples of such arrays are a ring of distance sensors, an array of photoreceptors in a video camera, and an array of touch sensors. The critter uses the *group feature generator* to recognize such arrays of similar sensors. A *group feature* is a vector feature whose elements are all related in some way (e.g., all correspond to sensors in an array of similar sensors).

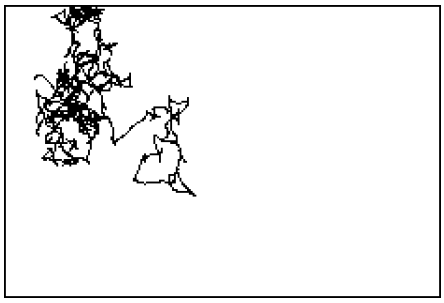


Figure 1.3: Before it has any knowledge of how the sensorimotor apparatus works, the critter chooses random actions, executing each for one second (ten time steps).

The group feature generator is based on the following observation. A well-engineered array of sensors (e.g., a ring a distance sensors) that measure a property (e.g., distance between the robot and a nearby object) that typically varies continuously with position in the array will satisfy the following: if two sensors are physically close together in the array, then they will "behave similarly" in the following two ways: (1) the two sensors' values at each instant in time will be similar and (2) the two sensors' frequency distributions[1] will be similar. Corresponding to these two ways are two distance metrics (examples of matrix features) that are used by the group feature generator. This generator computes these two distance metrics over a period of several minutes while the critter moves the robot using the following strategy (Figure 1.3): choose a random motor control vector; execute it for one second (10 time steps); repeat.[2] The result of the analysis is a set of distance metrics that tell, for each pair of sensors, how similar they are. The group generator uses these metrics to group similar subsets of sensors together.[3] The result is a new interface to the robot illustrated in the Figure 1.4. Notice that the 23 working distance sensors have been isolated from the rest.

1.2.2 The image feature generator

The grouping of the sensors into subgroups is a first step but it tells nothing about the positions of the sensors in the array. This is accomplished by the *image feature generator*. The image feature generator is a rule that takes a group feature and associates a position vector with each element of the group feature in order to produce an *image feature*: a function over time, completely determined by the current and past values of the raw sense vector, whose value at any given time is an *image*. An image is an ordered list of *image-elements*. An image-element is a scalar with an associated position vector. An example of the use of an image feature is to represent the pattern of light intensities hitting the photoreceptors in a camera.

The image feature generator uses the first of the two distance metrics used by the group feature

[1]The frequency distribution of a variable gives, for each of a set of subintervals in the variable's domain, the percentage of time that the variable assumes a value in that subinterval.

[2]My experiments with the critter have shown that this strategy is more effective for efficiently exploring a large subset of the robot's state space than choosing motor control vectors randomly at each time step.

[3]Section 3.1.1 gives the details of this operation.

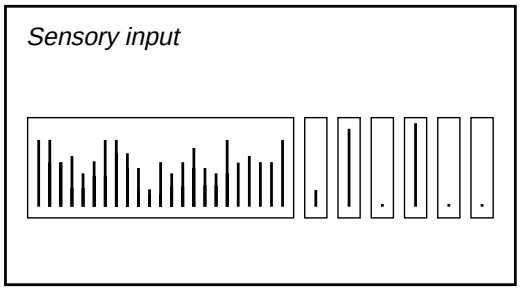


Figure 1.4: Applying the *group generator* is a first step toward modeling the robot's sensorimotor apparatus. The 23-element group feature at the left corresponds to a structured array of sensors and will be given special consideration.

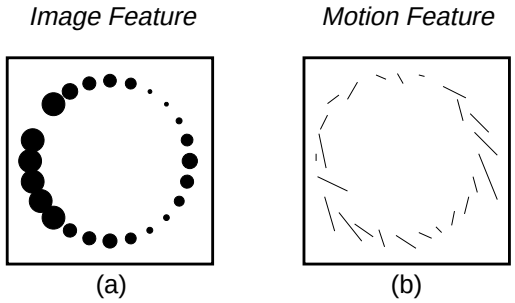


Figure 1.5: (a) The *image feature generator* produces an image feature that represents the physical structure of a group of related sensors. In this example, it is clear from the image feature that the robot's sensors are organized in a ring. Once a sensory array is represented as an image feature, then new features such as local minima, local maxima, and motion may be defined. (b) The motion generator is applied to the image feature to produce the motion feature which uses spatial and temporal derivatives of the image feature to measure the magnitude and direction of the "optical flow" at each point in the image feature.

generator. It uses a mathematical technique called metric scaling and a relaxation algorithm to map the sensors onto a two-dimensional surface in such a way that sensors that are similar to each other according to the distance metric are close together in the resulting 2-D representation.[4] This new representation is illustrated in Figure 1.5a.

1.3 Learning a model of the motor apparatus

The techniques that the critter has used so far have applied to the sensory system but not to the motor apparatus. The effects of the robot's motor control vectors must be defined using the robot's sensory system, its only source of information about its environment. One way to characterize a motor control vector's effect is in terms of motion. Once the image feature is learned, a new tool, the *motion feature generator*, becomes applicable and is used to define a motion feature (Figure 1.5b), an example of a *field* feature. A *field* is an ordered list of *field elements*. A field element is a vector

[4]See Section 3.1.2 for a discussion of the techniques and how they may be used to determine the appropriate number of dimensions for the image.

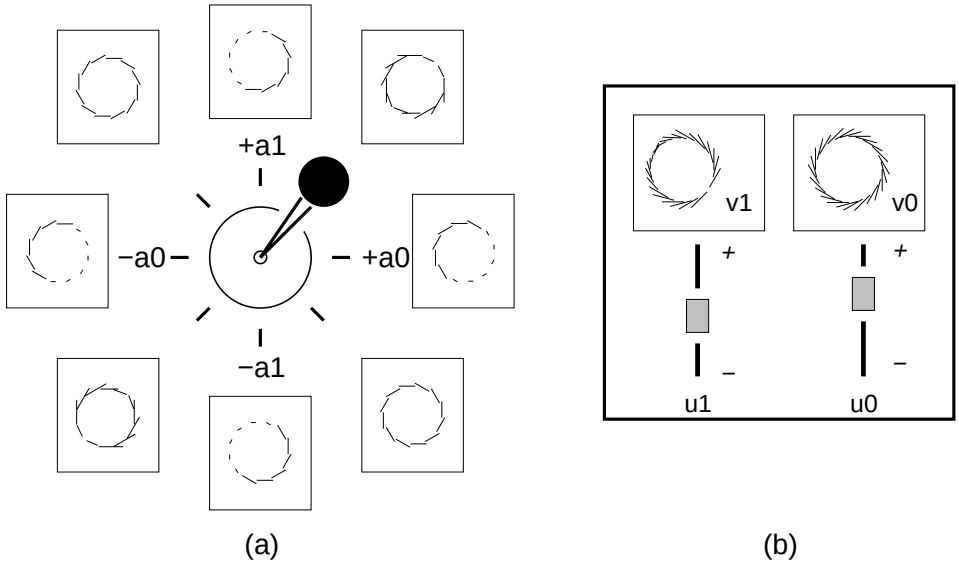


Figure 1.6: The critter defines a new motor control interface that abstracts away the details of the motor apparatus. (a) The robot's motor control signals control the right and left treads of a tank-style motor apparatus. Associated with each motor control vector is an average motion vector field (*amvf*) that characterizes the effect of the motor control vector. (b) The critter uses principal component analysis to discover a basis set of *amvf*'s, one per degree of freedom, and corresponding primitive actions and control signals. For this robot, these signals tell it how fast to turn and advance.

(e.g., a velocity vector) and an associated position vector. Even without knowledge of the physical structure of the environment, it is possible to define motion detectors, using only knowledge of spatial and temporal derivatives which can be defined directly from the image feature.[5] The value of a motion feature corresponds to what researchers in vision call an *optical-flow pattern*.

The critter uses the new motion feature to analyze its motor apparatus using the following steps: (1) The infinite space of all possible motor control vectors is discretized into a finite set of *representative* motor control vectors uniformly distributed over the space of motor control vectors. (2) While randomly executing these representative motor control vectors, the critter maintains an *average motion vector field* (*amvf*) for each of the motor control vectors. The *amvf* for a motor control vector is the average of the motion feature's value over all the times in which the motor control vector is used. The effect of each motor control vector is represented by its *amvf*. Eight of the motor control vectors and their *amvf*'s are shown in Figure 1.6a. (The *amvf*'s in the figure are only illustrative — the actual *amvf*'s are shown in Figure 5.1.) (3) The critter uses principal component analysis[6] to analyze the space of *amvf*'s and identify a set of *principal eigenvectors* that collectively capture the motion effects that the motor apparatus is capable of producing. For the robot of the example, the first principal eigenvector ($\mathbf{v}^0$ in Figure 1.6b) is an optical-flow pattern produced as the result of turning in place and the second eigenvector ($\mathbf{v}^1$ in

[5]The details will be given in Section 2.6.5

[6]Principal component analysis is a multivariate-analysis technique that will be described in Section 5.1.3

the figure) is an optical-flow pattern produced as the result of advancing. 4) The critter identifies *primitive actions*: A motor control vector is identified as a primitive action if the *amvf* for the motor control vector matches a principal eigenvector, in which case the motor control vector can be used to produce motion for one of the robot's degrees of freedom. For the example robot, the first principal eigenvector is matched by the primitive action $(0.7, -0.7)$ which moves the treads in opposite directions resulting in a pure rotation; the second principal eigenvector is matched by the primitive action $(0.7, 0.7)$ which moves the treads together resulting in a pure advancing motion. For the third and subsequent principal eigenvectors, no motor control vector's *amvf* matches. 5) The critter defines a new interface to the motor apparatus using the new control signals u_0 and u_1 that specify the magnitudes of the two primitive actions. The motor control vector is now given by

$$\mathbf{u} = u_0\mathbf{u}^0 + u_1\mathbf{u}^1$$

where $\mathbf{u}^0 = (0.7, -0.7)$ and $\mathbf{u}^1 = (0.7, 0.7)$ are the primitive actions, and u_0 and u_1 are the new control signals that specify how fast to turn and advance respectively. The new interface, illustrated in Figure 1.6b, abstracts away unimportant details of the motor apparatus. The interface looks the same for a tank-style motor apparatus as it does for a synchro-drive robot whose two control signals specify how fast to turn and advance, respectively. The representation of the robot's actions is now based on sensory effects rather than raw motor control signals.

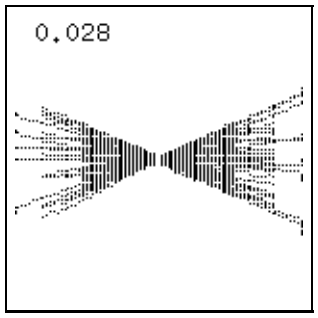
1.4 Discovering local state variables

The learned image feature (Figure 1.5) is important because it can be used by a number of feature generators to produce new features. An example is the motion feature which has already been discussed. Other feature generators use the learned image feature to produce, for example, local minima detectors and local maxima detectors. A more comprehensive list of feature generators will be given in Chapter 3.

The question remains, while generating new features, how does the critter know when it has found one that is useful? What does it mean for a feature to be useful? One component of understanding the robot's world is to be able to know the position of the robot, that is, to know where the robot is in its state space. Consider the example of a mobile robot that lives on a floor of an office building. Its state is completely determined by three variables — two for position (e.g., latitude and longitude) and one for orientation. Since the critter is not given the robot's state variables, the critter must define state variables for itself. A natural way to do this is to use features defined as functions on the sensory system as *local state variables*. These features locally determine the state of the robot. As an example, consider the case of a mobile robot, equipped with distance sensors, in the corner of a room. The position of the robot is locally determined by the minimum distances from the robot to the two corner walls. These distances serve as local state variables. They are local in the sense that they only exist as long as the robot is able to see the walls.

The critter's goal, while generating new features, is to recognize features that are suitable as local state variables. Without going into the details (see Chapter 6), a feature can serve as a local

$\Delta lmin$ vs. $u_0 \Delta t$



$\Delta lmin$ vs. $u_1 \Delta t$

Figure 1.7: The critter uses linear regression to investigate the effect of the control signals on the local-minimum features. It learns that u_0 has no appreciable effect. It also learns (since the standard deviation of $\dot{y}$ is large when u_1 is large) that u_1 does have an effect but that the effect is unpredictable. Context information will be necessary in order to predict the effects of u_1 on the features.

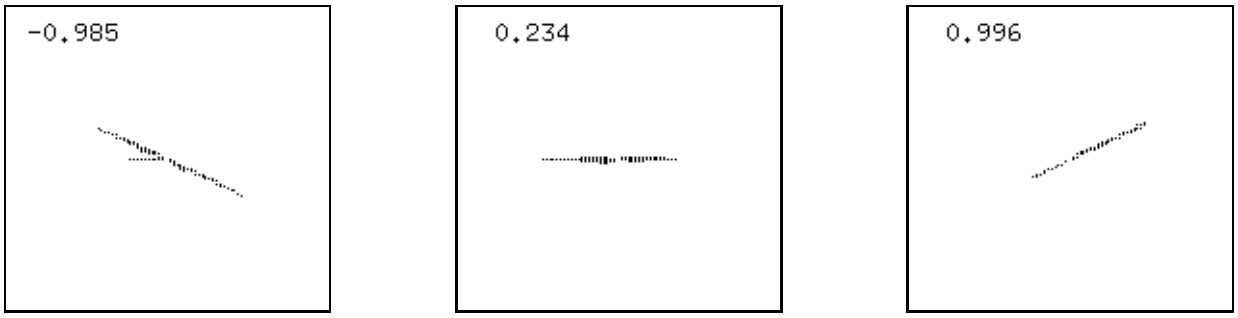
state variable if its derivative with respect to time can be approximated as a linear, nonzero function of the motor control vector. Discovering local state variables involves a process of generate and test. New features are defined using feature generators such as those already described. Testing a feature to confirm that its derivative is a linear, nonzero function of the motor control vector is done when the critter learns the *static action model*, described below.

1.4.1 Learning the static action model.

The *static action model* expresses the effects of the motor control signals on each scalar feature as a function of the current context (i.e., the current state of the robot as reflected by its current sense vector). Consider a *local-minimum* feature (*lmin*) that is derived from the image feature as the value of an element that is less than all of its neighbors in the image (see Section 2.6.5). To model the effects of the control signals on the feature, linear regression is used to test the hypothesis that $\dot{y_i} = m_{ij}\, u_j$ where $\dot{y_i}$ is the derivative of the i^{th} feature's value, m_{ij} is a constant, and u_j is the j^{th} control signal. Linear regression provides both an estimate of m_{ij} and a measure of the validity of the hypothesis. This measure, the *correlation*, will be close to 1 or -1 if the hypothesis is valid, in which case the critter can use the equation $\dot{y_i} = m_{ij}\, u_j$ to know how to set u_j in order to change y_i.

The linear regressions for the two control signals and a local-minimum feature are shown in Figure 1.7. From these, the critter learns that the first control signal u_0 (for turning) has no effect on the feature and that the second control signal u_1 (for advancing) does, but that the effect is not consistent. It may be that the effect of the control signal is context dependent — that in some contexts it will increase the feature's value and in others it will decrease it. To test this hypothesis, the critter breaks the robot's state space up into different contexts and then performs a separate linear regression for each context. In this way, it attempts to model the effect of the control signal on the feature by the equation $\dot{y_i} = m_{ijk}\, u_j$ where the constant m_{ijk} depends on the context.

One way to partition state space into a set of contexts is to choose a feature and divide its range of values into a finite set of intervals, each defining its own context. Using feature x to define



$\dot{y}_i$ vs. u_0, k=0 $\dot{y}_i$ vs. u_0, k=6 $\dot{y}_i$ vs. u_0, k=12

Figure 1.8: Three representative linear regressions used in learning the static action model. A local-minimum feature y_i's position in the image from which it is derived is found to provide valuable context information for predicting the effect of control signal u_1 on the feature's value. Linear regression is used for each context (i.e., each of the 23 possible positions) to determine the relationship between u_1 and y_i. The critter discovers that, for contexts 0 through 5 and 19 through 22, y_i decreases when u_1 is positive. For contexts 7 through 17, y_i increases when u_1 is positive. For contexts 6 and 18, y_i does not change no matter what the value of u_1 is. This information is recorded in the *static action model* for feature y_i.

a set of contexts is appropriate if the value of x is a good predictor of the effect of a given control signal u_j on a given feature y_i. To test the hypothesis that x is a good predictor for the effect of u_j on y_i, linear regression can be used to determine u_j's effect on y_i for each context defined using a subinterval of x's range. If the relationship between u_j and y_i is linear when x's value lies in a certain interval I_k, then the equation $x \in I_k$ defines a region in state space in which y_i can serve as a local state variable.

Testing each of a large set of features to see if they improve the predictability of a control signal's effect is expensive. One heuristic to reduce the expense is to first look at features closely related to the feature being analyzed. In the current example, the critter wants to know how control signal u_1 affects a local-minimum feature y_i. A feature closely related to y_i is the position in the image from which it is defined. The position is already conveniently discretized: there are 23 possible positions from which 23 contexts may be defined. For each of these contexts, the critter performs a context-dependent linear regression to discover the relationship between the control signal u_1 and the local-minimum feature (see Figure 1.8).

The results gained from all of these linear regressions are used to define a static action model for local-minimum features which tells, given the robot's current context, how the control signals affect the features. In the example, the critter learns that for contexts 0 through 5, 7 through 17, and 19 through 22, it can reliably affect the value of a local-minimum feature y_i using control signal u_1. In these contexts, the feature y_i is a local state variable: the derivative of y_i is a linear nonzero function of the motor control vector $\mathbf{u}$. In contexts 6 and 18, the control signal leaves the value of the feature invariant. The learned local state variables are illustrated in Figure 1.9.

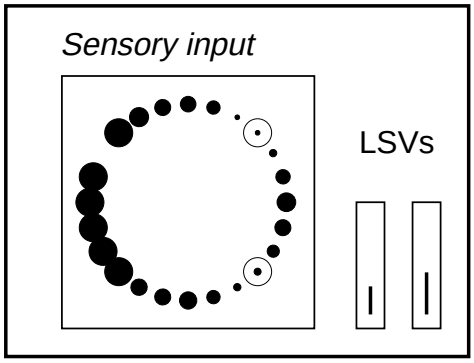


Figure 1.9: A generate-and-test approach is used to learn a set of features that can serve as local state variables. Here, the *local-minimum* generator is used to produce suitable features. From the critter's perspective, these are local minima of an image whose structure is based on intersensor correlations. From our perspective, these are minimum distances to nearby objects.

1.5 Learning behaviors

In order to actively explore its world, the critter needs to learn behaviors for navigation through the environment. The approach the critter takes is to learn robust *path-following behaviors*. The heart of a behavior is a *control law* that tells the robot what motor control vector to use given the current sense vector. The control law for a path-following behavior constrains the robot to move along a well-defined one-dimensional path. This is to ensure that the effect of following the path will be the same each time the critter chooses that path. For example, consider a mobile robot in an office. A behavior that involves moving the robot along the edge of the office while maintaining a constant distance to the wall is a good path-following behavior. It is consistent in that it will always take the robot to the same place, e.g., the place where the wall forms a corner with another wall.

Path-following behaviors are learned in three steps: (1) continuous error signals are defined; (2) behaviors are learned for minimizing the error signals; (3) behaviors are learned for moving while keeping the error signals near zero.

1.5.1 Defining error signals for control laws

The critter's approach to learning path-following behaviors is to first define error signals of the form $e = y^* - y$ where y is a local state variable and y^* is a constant called a *target value*. Without any *a priori* knowledge of the environment, the choice of y^* is arbitrary. The critter chooses a nominal value of $y_i^* = 0.5$ since the features' values range from 0 to 1. The subsequent sections tell how to use these error signals to define path-following behaviors.

1.5.2 Learning homing behaviors

The second step of the process of learning path-following behaviors is to use the static action model to define *homing behaviors*. These are the behaviors that move the robot to a state where a given feature y_i has a desired value y_i^*. These behaviors are used to establish the prerequisites for

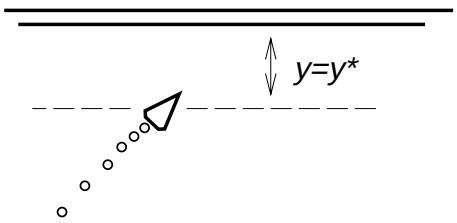


Figure 1.10: An example of a homing behavior. The critter has learned and recorded in its static action model that primitive action $\mathbf{u}^1$ will decrease the value of the feature y_i in any context in which the robot is facing toward the wall. It uses this knowledge to define a control law that moves the robot to a state where $y_i = y_i^*$.

path-following behaviors, behaviors that move the robot while maintaining a feature at its desired value.

For each local state variable y_i and control signal u_j, a homing behavior is defined that uses the control signal to move the local state variable to its target value. The behavior is applicable when the static action model predicts that the control signal can be used to reliably affect the feature's value. The behavior is done when the feature is at its target y_i^*. The behavior's output is given by a simple control law. A homing behavior for moving to a specific distance from a wall is illustrated in Figure 1.10.

1.5.3 Learning path-following behaviors

The previous section presented a method for learning homing behaviors that minimize a given error signal. In this section, a method is presented for moving while minimizing the error signal. The result is a path-following behavior. Learning a path-following behavior involves two steps: 1) learning how to move in the general direction that keeps the error near zero and 2) learning the necessary feedback for error correction to avoid straying off the path defined by the minimum of the error signal.

1. Learning open-loop path-following behaviors. The first step toward learning path-following behaviors is to use the static action model to define *open-loop path-following behaviors*. An open-loop path-following behavior is a preliminary form of path-following behavior that lacks error correction but that is useful for learning the *dynamic action model* which is in turn useful for defining path-following behaviors with error correction. For each local state variable y_i and primitive action $\mathbf{u}^j$, for each context in which $\mathbf{u}^j$ has no effect on y_i according to the static action model, two behaviors are defined that are applicable when y_i is at its target value y^* and the static action model predicts that the primitive action does not affect the feature. The behaviors' outputs are given by

$$\mathbf{u} = \mathbf{u}^\beta + \sum_{\delta \neq j} u_\delta \, \mathbf{u}^\delta$$

where $\mathbf{u}^\beta = \pm \mathbf{u}^j$ and $|u_\delta| \ll 1$. The u_δ components will be used in learning the dynamic action model. The purpose of an open-loop path-following behavior is to allow the critter to learn the

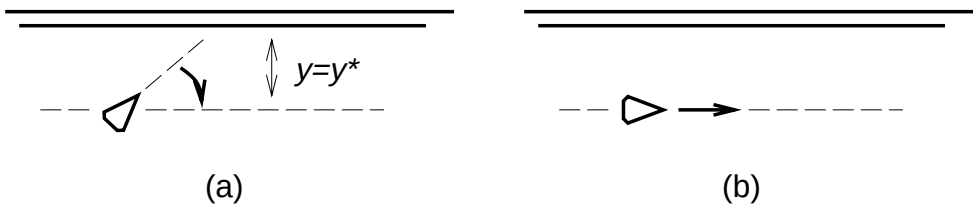


Figure 1.11: Two examples of open-loop path-following behaviors. (a) A behavior based on $\mathbf{u}^0$ (for turning) and constraint $y_i = y_i^*$ is applicable whenever $y_i = y_i^*$ since $\mathbf{u}^0$ never changes the value of y_i. (b) A behavior based on primitive action $\mathbf{u}^1$ (advancing) and constraint $y_i = y_i^*$ is applicable whenever $y_i = y_i^*$ and the robot's heading is parallel to the wall on its left since in this context $\mathbf{u}^1$ maintains the constraint $y_i = y_i^*$.

effects of the other primitive actions on the feature while motor control vector $\mathbf{u}^\beta$ is used. With this knowledge, it will be possible to use these other control signals for error correction.

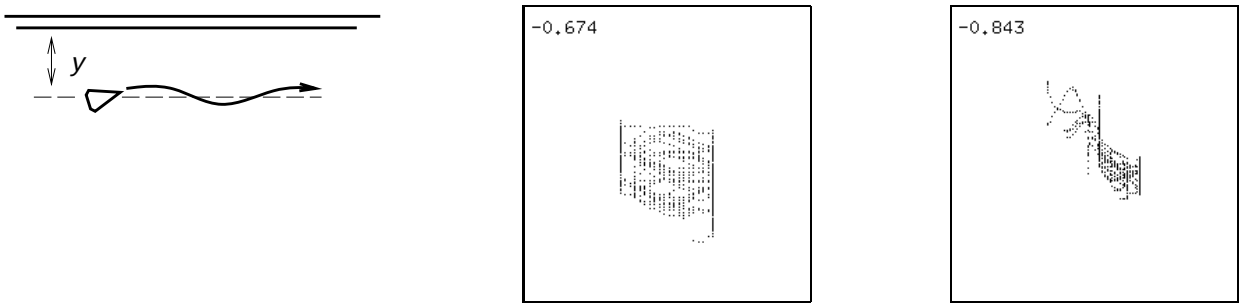
For the robot of the running example, there is an open-loop path-following behavior based on $\mathbf{u}^0$ (for turning) for each local state variable y_i. It is applicable whenever $y_i = y_i^*$ since, according to the static action model, turning has no effect on y_i. There is also an open-loop path-following behavior based on $\mathbf{u}^1$ (for advancing) and each feature y_i for each of the constant contexts 6 and 18. These are contexts in which the critter has a wall on its right or left.

The dynamic action model. Whereas the *static* action model predicts the context-dependent effects of a primitive action on the local state variables, the *dynamic* action model predicts the context-dependent effects of primitive actions on local state variables while an open-loop path-following behavior is being executed. Consider the mobile robot of the example and suppose that there is a wall to its left and that it is facing parallel to the wall (Figure 1.11). In this context, primitive action $\mathbf{u}^1$ (advancing) will maintain the distance to the wall, y_i, invariant. Therefore, the open-loop path-following behavior based on $\mathbf{u}^1$ and y_i will be applicable. While executing this behavior, the effects of other primitive actions (e.g., $\mathbf{u}^0$) can be diagnosed (Figure 1.12). The relationship between u_0 and y_i is discovered using linear regression to test the hypotheses $\dot{y_i} = m_{i0}\, u_0$ and $\ddot{y_i} = m_{i0}\, u_0$. (Recall that $\mathbf{u} = u_0\mathbf{u}^0 + u_1\mathbf{u}^1$.) In this example, the second equation better captures the relationship between u_0 and y_i than does the first. This is because turning changes the robot's direction of motion relative to the wall and this direction determines how fast the robot moves toward or away from the wall as it advances. To summarize this step: for each open-loop path-following behavior, based on motor control vector $\mathbf{u}^\beta = \pm\mathbf{u}^j$ and feature y_i, for each primitive action $\mathbf{u}^\delta \neq \mathbf{u}^j$, linear regression is used to learn the effect of $\mathbf{u}^\delta$ on y_i while the open-loop path-following behavior is running. This effect is represented by the equation

$$y_i^{(n)} = m_{ij\delta n}\, u_\delta$$

where $y_i^{(n)}$ is the n^{th} derivative of y_i and $n \in \{1, 2\}$.[7]

[7]See Section 7.4.2 for the details including the reason why both the first and second derivatives need to be considered.



$\dot{y}_i$ vs. u_0 $\ddot{y}_i$ vs. u_0

Figure 1.12: Learning the dynamic action model for control signal u_0. While an open-loop path-following behavior based on control signal u_1 is running, the critter diagnoses the effect of a small superimposed orthogonal control signal. The motor control vector is computed by the equation $\mathbf{u} = u_1\,\mathbf{u}^1 + u_0\,\mathbf{u}^0$ where $u_1 = 1$ and u_0 varies between -0.1 and 0.1. Linear regression is used to characterize the relationship between u_0 and both $\dot{y}_i$ and $\ddot{y}_i$. The critter discovers that the effect of u_0 on y_i is best characterized by $\ddot{y}_i = m_{i0}\,u_0$.

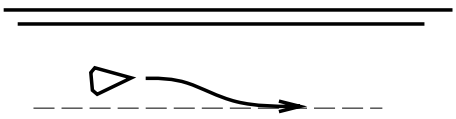


Figure 1.13: Learning closed-loop path-following behaviors. The critter uses the learned dynamic action model to add error correction to an open-loop path-following behavior in order to obtain a closed-loop path-following behavior. In this example, a small turning motion is used to keep the robot on the path as it advances.

2. Learning closed-loop path-following behaviors. The final step in learning path-following behaviors is to add error correction to the open-loop path-following behaviors in order to define closed-loop path-following behaviors. If the robot wanders away from the path, the control law should steer it back to the path. A *closed-loop* behavior is one that receives feedback from the environment in the form of an error signal which it uses to modify its motor control signals so as to minimize the error. The details are given in Section 7.4.3. Consider again the case where the robot is facing parallel to a wall on its left. In this context, the critter knows, because of its static action model, that control signal u_1 will leave feature y_i (the distance to the wall) invariant. Moreover, the critter knows, because of its dynamic action model, how control signal u_0 (turning) affects y_i while u_1 is being taken. Together, this information is sufficient to define a closed-loop path-following behavior that robustly moves the robot along the wall. If y_i goes below its target value (i.e., if the robot gets too close to the wall), then the critter knows to increase the value of u_0 (i.e., to turn right as shown in Figure 1.13). Because of the error correction implemented using control signal u_0, the path-following behavior is robust in the face of noise in the sensorimotor apparatus, small perturbations in the shape of the wall, and even inaccuracies in the action models themselves.

Two closed-loop path-following behaviors are defined for each primitive action $\mathbf{u}^j$ for each subset $\mathbf{y}$ of the set of all local state variables, and for each context in which the static action model predicts that primitive action $\mathbf{u}^\beta$ will maintain the invariant $\mathbf{y} = \mathbf{y}^*$. A behavior based on $\mathbf{u}^i$ and $\mathbf{y}$ is applicable when all of the components of $\mathbf{y}$ are at their target values, i.e., $\mathbf{y} = \mathbf{y}^*$. The behavior

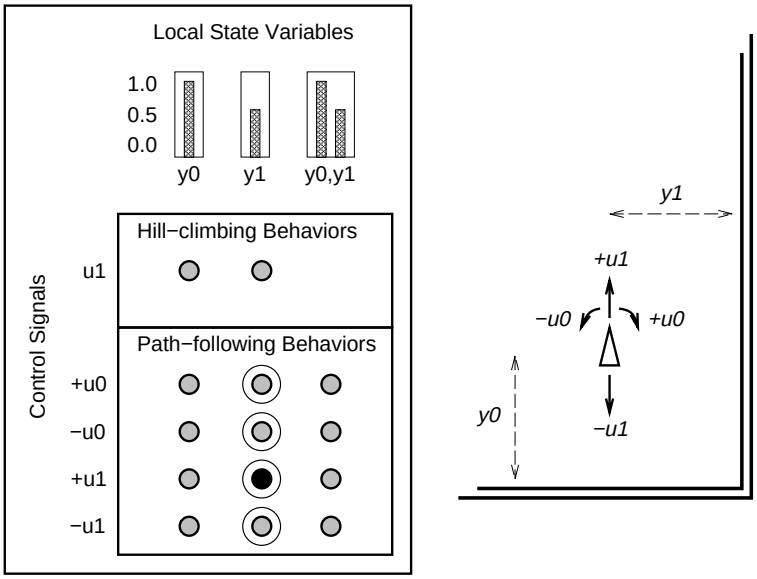


Figure 1.14: The critter defines a high-level interface to the robot using learned homing and path-following behaviors. The state of the control panel corresponds to the state of the robot shown at the right. The local state variables y_0 and y_1 are the distances to the south and east walls, respectively. For each local state variable, a homing behavior is defined using primitive action $\mathbf{u}^1$ (the only primitive action that affects the variables). For each subvector $\mathbf{y}$ of (y_0, y_1) and each primitive action $\mathbf{u}^i$, two path-following behaviors are defined for moving using $\pm\mathbf{u}^i$ while maintaining the invariant $\mathbf{y} = \mathbf{y}^*$. A circle around a behavior button indicates that the behavior is applicable. The path-following behavior based on $+\mathbf{u}^1$ and y_1 is highlighted indicating that it is the behavior currently selected by the critter. Its effect is to move the robot north along the right wall.

is done when a new path-following behavior becomes applicable indicating that the the critter now has a choice — to continue the current path-following behavior or to choose a new one.

For the running example, the set of path-following behaviors contains behaviors for turning in place as well as for following walls. In both cases, the behavior is defined using a base action (turning or advancing) while maintaining an invariant (e.g., a constant distance to the wall).

By learning the path-following behaviors, the critter has defined a higher-level interface to the robot, illustrated in Figure 1.14. The effects of these high-level behaviors are demonstrated in Figure 1.15.

1.6 Defining a discrete sensorimotor apparatus

Beginning with a raw sense vector and a raw motor control vector, the critter has learned a set of local state variables and high-level behaviors. Its next step is to abstract from the continuous sensorimotor apparatus to a discrete sensorimotor apparatus by defining finite sets of *views* and *actions*.

For any given state of the robot, there is a finite set of homing and path-following behaviors. These behaviors are the *actions* of the discrete sensorimotor apparatus. Executing one of the actions

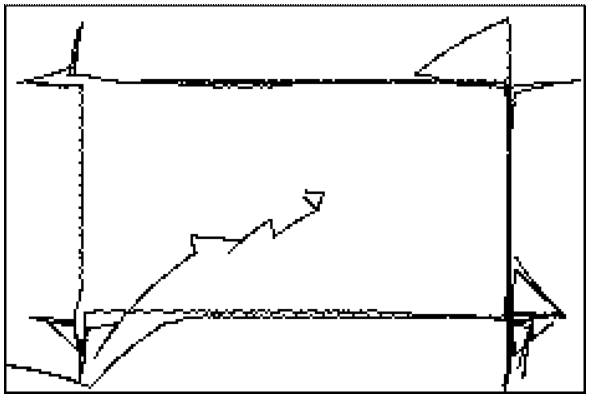


Figure 1.15: A demonstration of the learned homing and path-following behaviors. Here, the critter wanders by randomly choosing applicable path-following behaviors. If no path-following behavior is applicable, it chooses a homing behavior. If no behavior is applicable, it wanders randomly until one becomes applicable.

involves running the corresponding behavior until it terminates. A behavior terminates when the local state variable or variables on which it is based are no longer defined or when a new behavior becomes applicable indicating that the critter may now choose to continue the current behavior or select a new one. The set of states in which actions terminate are named via a mapping from sense vectors to symbols called *views*. This mapping uses a matching predicate that tells whether two sense vectors are similar. If the current sense vector is new then a new view is created and associated with it. If the current sense vector matches one previously seen, it is associated with the same view as the previous sense vector.

The resulting interface is illustrated in Figure 1.16. While a path-following behavior is executing, the interface is undefined. When the behavior terminates, the interface identifies the current view and lists the current set of applicable behaviors.

1.7 Learning the topology of the environment

The critter has made a very important change of representation by abstracting a continuous sensorimotor apparatus to a discrete sensorimotor apparatus with a finite set of sense values and actions. Understanding a continuous world is very difficult but the problem of understanding a discrete world has been extensively studied.

The robot's path-following behaviors constrain its motion to a one-dimensional subspace of the robot's complete state space. This 1-D skeleton is the basis for an abstraction of the robot's environment as a graph (a set of nodes and a set of edges connecting the nodes together). The edges correspond to paths — trajectories in the robot's state space produced by path-following behaviors. The nodes correspond to states where paths terminate, that is, states where a new path-following behavior becomes applicable and the critter stops to choose one of the currently applicable paths. The critter's goal is to construct this graph.

In the case where views uniquely identify states, the problem is straightforward. For each state it sees, the critter keeps track of all the actions applicable at that state. Each time it takes an

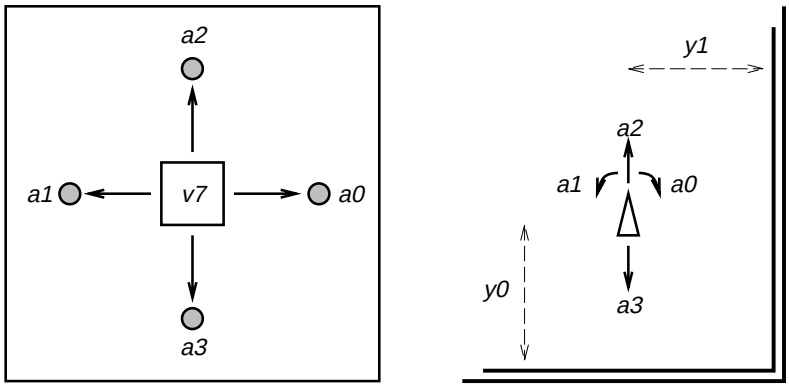


Figure 1.16: The discrete sensorimotor apparatus. The critter abstracts from an infinite space of sense vectors to a finite set of views and from an infinite space of motion trajectories to a finite set of actions. The current *view* is a symbol (v_7 above) that identifies the current sense vector. The set of *actions* (a_0 to a_3 above) is the set of currently applicable path-following behaviors. Selecting a behavior causes it to be executed until it terminates, at which point a new view and action set is presented. For this example, a_0 and a_1 are based on primitive action $\mathbf{u}^0$ and maintain the constraint $y_0 = y_1 = 0.5$; a_2 and a_3 are based on primitive action $\mathbf{u}^1$ and maintain the constraint $y_1 = 0.5$.

action, a_j, that takes it from view v_i to v_k, it adds the edge (v_i, a_j, v_k) to the graph. It continues to explore (intelligently or randomly) until there are no state-action pairs that it has not explored.

In the case that views do not uniquely identify states, a more sophisticated exploration strategy is required. Such strategies will be discussed in Section 9.5.1. They are all based on the following idea: If the current view does not uniquely identify the current state, the critter supplements the current sense vector with the sense vectors of nearby states. With enough information about the surrounding area, the current state can be uniquely identified.

1.8 The abstract-interfaces approach

The preceding scenario illustrates how the critter can learn a model of a robot's sensorimotor apparatus and environment without being given knowledge of the robot's particular set of sensors and effectors. This scenario demonstrates what I call the *abstract-interfaces* approach to learning the model. In this Chapter, Figure 1.1 shows the intial interface between the robot and the critter. Figures 1.4, 1.5, 1.6b, 1.9, 1.14, and 1.16 show a sequence of learned abstract interfaces between the robot and the critter. Each abstract interface gives the critter a new way of understanding the robot's sensory input or of interacting with the robot's environment. The problem and solution, exemplified by the critter learning to understand the mobile robot, are given formally below,

followed by definitions of the italicized terms:

Problem

Given: a robot with an *uninterpreted* sensorimotor apparatus in a *continuous*, *static* environment.

Learn: descriptions of the structure of the robot's sensorimotor apparatus and environment and an abstract interface to the robot suitable for *prediction* and *navigation*.

Solution

Representation: a hierarchical model. At the bottom of the hierarchy are egocentric models of the robot's sensorimotor apparatus. At the top of the hierarchy is a discrete abstraction of the robot's environment defined by a set of discrete views and actions.

Method: a sequence of statistical and evolutionary methods for learning the objects of the hierarchical model.

The sensorimotor apparatus is *uninterpreted* meaning that the critter that is learning how to use the robot has no *a priori* knowledge of the meaning of the sensors, of the structure of the sensory system, or of the effects of the motor's control signals. A *continuous* world (which includes both the robot and its environment) is one whose state can be represented by a vector $\mathbf{x}$ of continuous, real-valued state variables. A *discrete* world, on the other hand, is represented by a finite set of states. The mobile robot of the example lives in a continuous world with three state variables: two for its position (e.g., longitude and latitude) and one for its orientation (i.e., the direction in which it is facing). A *static* world is one whose state does not change except as the result of a nonzero motor control vector. A static world exhibits no inertia. When the motor controls go to zero, the robot comes to an immediate stop. In a static world, there are no active agents (e.g., pedestrians) besides the robot itself.

The critter's goal is to understand its world, that is, to learn a model of it suitable for prediction and navigation. *Prediction* refers to the ability to predict the effects of motor control vectors. *Navigation* refers to the ability to move efficiently from one place to another. These definitions do not apply perfectly to a critter's world: places do not exist *a priori* — they must be discovered or invented by the critter itself. The raw sense vector and the raw motor control vectors are at the wrong level of abstraction for describing the global structure of a world. People do not understand their world in terms of sequences of high-resolution visual images — they use abstractions from visual scenes to places and objects. In order to understand its continuous world, the critter must also use abstractions. Instead of trying to make predictions based on the raw sense vector, it needs to learn high-level features and behaviors. Understanding the world thus requires a hierarchy of features, behaviors, and accompanying descriptions. The hierarchy that the critter uses is called the *spatial semantic hierarchy*. It is defined in the next section.

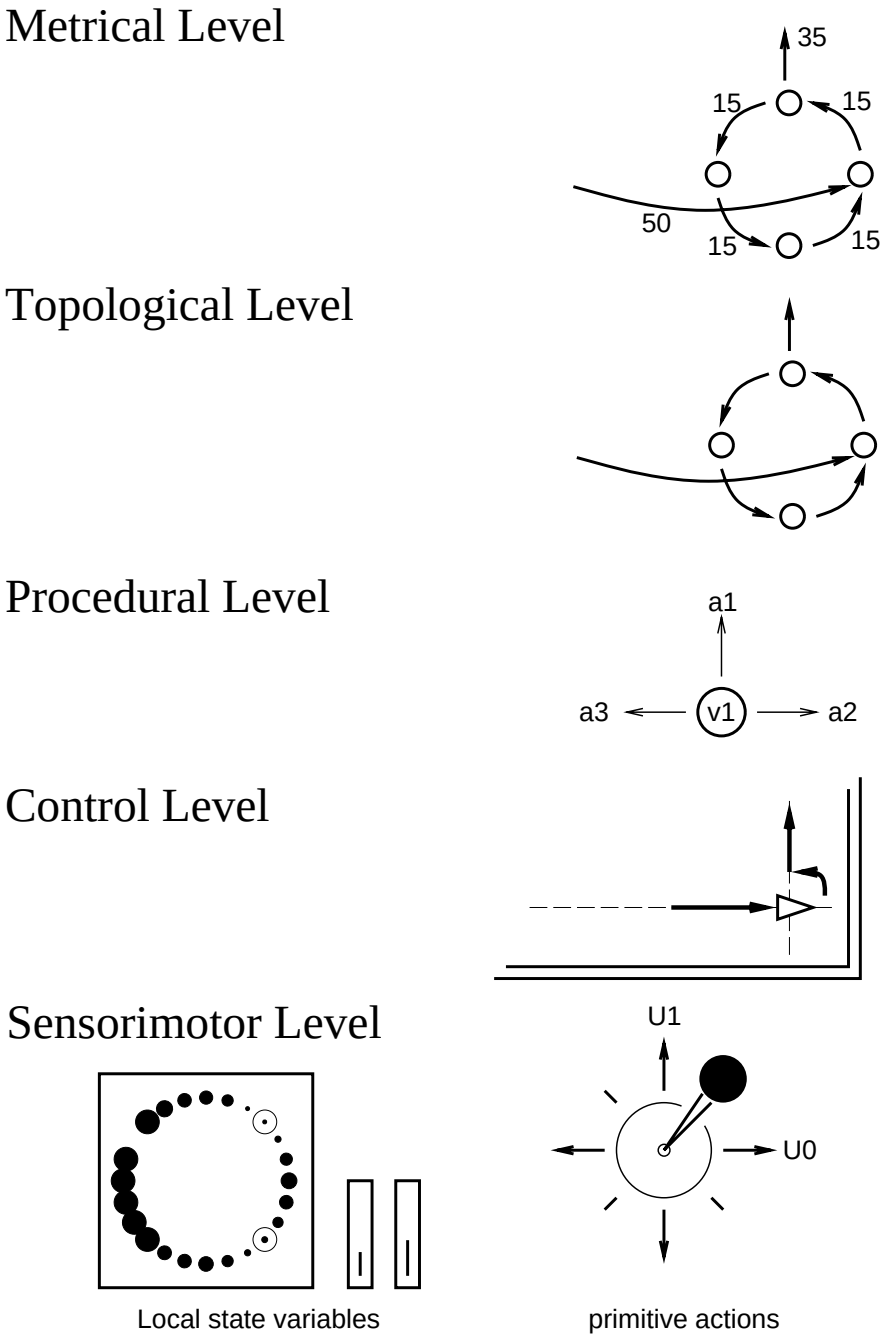


Figure 1.17: By applying a series of operators to an uninterpreted sensorimotor apparatus, the critter learns a sequence of abstract interfaces that define the sensorimotor, control, and procedural levels of the *spatial semantic hierarchy*.

1.9 The spatial semantic hierarchy

The spatial semantic hierarchy (Figure 1.17) is comprised of five levels: sensorimotor, control, procedural, topological, and metrical. At the sensorimotor level, the interface to the robot is defined by the raw sense vector, a set of primitive actions (one for each degree of freedom of the robot), and a set of learned features. At the control level, action models are learned in order to predict the context-dependent effects of motor control vectors on features. Local state variables are learned and behaviors for homing and path-following are defined. The interface to the robot is defined by the set of local state variables, homing behaviors, and path-following behaviors. At the procedural level, sense vectors are abstracted to a finite set of views and behaviors are abstracted to a finite set of actions. The interface gives the current view and the set of currently applicable actions.

The contribution of this dissertation is a set of methods for learning these first three levels. This dissertation's work is complementary to work done by Kuipers and Byun (1988) who have demonstrated a robot and learning agent for which the first three levels of the spatial semantic hierarchy were engineered by hand. Their learning agent learns the topological and metrical levels autonomously. At the topological level, perceptual ambiguities (in which multiple states map to the same view) are resolved and a global representation of the world's structure as a finite-state graph is learned. At the metrical level, the topological map is supplemented with the lengths of paths in terms of time or effort involved in traversing the paths.

By showing how to learn the first three levels of the spatial semantic hierarchy, this dissertation lays the groundwork for building a learning agent that can learn the entire spatial semantic hierarchy using only domain-independent knowledge.

1.10 Learning methods and *a priori* knowledge

The critter can be viewed as an expert system whose goal is to learn (1) a model of a mobile robot's set of sensors, (2) a model of the robot's motor apparatus, and (3) a set of path-following behaviors, without being given domain-specific knowledge of the robot's sensorimotor apparatus and environment. Designing the critter so that it can achieve this goal using domain-independent knowledge is the subject of this dissertation. The following list summarizes this *a priori* knowledge while pointing out which knowledge is used in which step of the critter's learning process.

- The group feature uses a set of distance metrics that explicitly represent different ways to recognize similar sensors.
- The image feature generator uses one of the distance metrics used by the group generator. It also uses metric scaling (a technique from multivariate analysis) and a relaxation algorithm to construct the image feature from the distance metric.
- The motion feature generator uses a method from the field of computer vision to define the motion feature using the learned image feature.
- The method for learning the set of primitive actions uses principal component analysis (another multivariate-analysis technique).

- The method for generating the scalar features that are candidates for local state variables uses a set of feature generators. These generators (described in Chapter 3) use concepts from basic mathematics such as minimum, maximum, and differentiation with respect to time.
- The method for learning the static action model uses linear regression to learn the effects of the primitive actions on the scalar features. Its *a priori* knowledge includes a heuristic that tells which context feature to associate with scalar features derived from image features.
- The method for defining error signals uses the fact that the sensors' values range from 0 to 1.
- The method for learning the homing and path-following behaviors uses linear regression (for the dynamic action model) and two controller templates drawn from control theory.
- The critter is given a set of rules telling it how to choose its motor control vectors. The critter initially chooses motor control vectors randomly, repeating each 10 times. After the homing and open-loop path-following behaviors are learned, it randomly selects from among these behaviors in order to learn the dynamic action model which it uses to define the closed-loop path-following behaviors.

1.11 The importance of the problem

There are a number of reasons for studying the problem of autonomously learning a model of a mobile robot with an uninterpreted sensorimotor apparatus in an unknown environment.

Practical. A practical motivation is a desire to build robots that can autonomously learn to use their sensors and effectors. Robots with the capability of learning a new sensorimotor apparatus will also be able to learn to deal with a faulty sensorimotor apparatus. Such robots will be more robust than ones which are programmed to use a specific apparatus in a specific environment. By developing its own understanding of its own sensorimotor apparatus, the robot is not limited by the designers' ideas of what sensors are important or useful. The designer is relieved of the task of coding representations and behaviors for every new sensorimotor apparatus (which includes taking into account the idiosyncrasies of each particular sensorimotor apparatus).

Philosophical. A philosophical motivation is to identify sources of *structure-revealing* information that are implicit in the behavior of an uninterpreted sensorimotor apparatus. This information can be used to make sense of the sensory information to which a newborn infant or robot is exposed. The learning methods of this dissertation demonstrate a number of examples of sources of structure-revealing information:

- Similarities among sensors (as measured by the distance metrics used by the group and image feature generators) comprise one source of information. This information is based solely on the sequence of those sensors' values over time (as opposed to *a priori* knowledge about the sensors). This information is used to build a model of the structure (i.e., organization and layout as represented by the group and image features, respectively) of a set of sensors.

- The sequence of images produced by the image feature comprise another source of information. This information is used by the motion feature to provide a characterization of the robot's motor control vectors in terms of average motion vector fields. The critter uses these *amvf*'s to build a model of the structure of the robot's motor apparatus. This structure is represented by a set of primitive actions, one for each of the robot's degrees of freedom.

Psychological. A psychological motivation is to learn how to acquire and exploit this structure-revealing information, i.e., to model an organism's perceptual and motor development in a realistic spatial world. Animals' sensory and motor systems are not completely programmed by genetic information. Instead, they are programmed to develop in a certain type of environment. The animals' predispositions together with the information gained by interacting with their environment allow them to learn how to perceive and act. The critter embodies an analogous learning process. The critter's *a priori*, domain-independent knowledge is analogous to the animals' genetic predispositions. The critter's process of using its feature generators to learn a model of its sensorimotor apparatus is analogous to the animals' process of motor and perceptual development.

1.12 Overview

Chapter 2 introduces a language of features and feature generators, the building blocks with which the abstract interfaces are built. Chapter 3 uses this language to define a generate-and-test approach to feature learning. Chapter 4 describes the group and image feature generators in detail and demonstrates their use in learning structural descriptions of two very different sensory apparatuses. Chapter 5 describes the methods for learning a model of a robot's motor apparatus and demonstrates them on two different simulated robots. Chapter 6 describes the method for discovering a set of local state variables. Chapter 7 uses the results of the preceding chapters to define a set of homing and path-following behaviors, the basis for navigation in large-scale space. The learning methods described in Chapters 2 through 7 are used to learn the sensorimotor and control levels of the spatial semantic hierarchy. Chapter 8 describes a number of experiments that demonstrate the generality and some limitations of these learning methods. Chapter 9 shows how the learned homing and path-following behaviors are used to define a discrete abstract interface that allows the critter to treat the robot's environment as a finite-state environment. It then discusses various methods for inferring the structure of finite-state environments. Chapter 9 also discusses the relationships between this and other work.

Chapter 2

A Language of Features

This dissertation addresses the problem of a learning agent, called a "critter" that must develop an understanding of a robot's world to the point where it can intelligently navigate through that world. The solution involves an *abstract interfaces* approach in which the critter defines increasingly sophisticated sensorimotor interfaces to the robot. On the sensory side, this is accomplished by defining new *features*, functions defined on the raw sensory input. On the motor side, this is accomplished by defining new control laws. This chapter presents a language of feature types and operators that the critter will use in subsequent chapters to learn a rich model of the robot's sensorimotor apparatus and environment.

2.1 Introduction

A feature is a mathematical object (that will be formally defined in Section 2.3) intended to measure a property of an object or of the environment that can be detected by a sensory apparatus. Consider a few examples from biological vision. In the human retina are about 125 million photoreceptors (Rosenzweig & Leiman, 1982). Each photoreceptor measures the intensity of light impinging on it. The photoreceptor is a feature detector — the feature it detects is the light intensity at a specific point in the visual image. The retina itself is a structured array of photoreceptors. It can be viewed as a feature detector where the feature is an entire image — a structured array of light intensities. In the visual cortex are motion detectors. Individually, these detectors measure local motion. Collectively, these detectors detect an optical flow pattern, a feature that assigns to each point in the retinal image a vector giving the direction and magnitude of motion. Analogously, there are feature detectors in the visual cortex that respond to oriented edges. Collectively, these detect a pattern of edges, a feature that assigns to each point in the retinal image a vector giving the orientation and magnitude of the edge detected at that point. There are also feature detectors for specific objects. For example, frogs are capable of detecting flies in motion. The fly-spotting feature tells whether a moving fly (or any object that looks like a moving fly) is present and, if so, where it is located with respect to the frog's retinal image.

Another example of a detector is a thermometer that measures ambient air temperature. There is an interesting difference between this feature detector and a frog's fly-spotting detector. The temperature of the room is always defined whereas the fly-spotting feature is defined only when there is a fly present to be seen. In this work, the former type of feature is called *persistent*; the latter type is called *ephemeral*. A feature language should be able to represent both types of features.

These examples illustrate a wide variety of feature detectors, without committing to a precise definition of the word "feature." They demonstrate a number of properties of features that a

language of features should provide:

- Features are detected through the processing of sensory data.
- Features come in many different types, and these types correspond to different mathematical objects such as scalars and vectors.
- Features can be ephemeral or persistent.

A formal definition of "feature" will be given in Section 2.3.

2.2 Goals of feature learning

A language of features is a valuable tool for machine learning. It provides a means for changes of representation, focus of attention, and dimensionality reduction. Moreover, it facilitates the process of programming robots, whether that programming is done by hand or automatically.

2.2.1 Change of representation

Consider the robot, described in Chapter 1, with a ring of distance sensors. Representing this distance information as a vector feature does not capture the important structure of the sensory array. Changing the representation from a vector feature to an *image feature* (see page 27) opens up many possibilities for exploiting the information from the distance sensors.

An image feature represents relationships such as relative positions of image elements. This spatial information is necessary for defining features such as edge detectors that are based on spatial derivatives. Motion detectors use both spatial and temporal derivatives. Chapter 5 will give an example of the use of motion detectors.

2.2.2 Focus of attention

A visual image can be a very large, complex feature, containing much more information than can be attended to all at once. A mechanism is needed to provide focus of attention. Consider the case of a robot with a camera looking at a red object on a white background. When the robot's head moves, the red object's image-relative position changes. In order to focus attention on the object, a mechanism is needed to abstract above the level of the raw visual image and explicitly represent the object. A feature generator called a *tracker* (see Section 3.1.4) will be shown to implement this type of focus of attention. It can produce a new *image-element* feature (see page 27) that represents the object and its position. Unlike the image feature in which each element has a constant position associated with it, the position of the image-element feature will change, tracking the red object as the robot's camera moves.

2.2.3 Dimensionality reduction

In a sensory apparatus, a high-dimensional feature, such as a visual image, may have a great amount of redundancy. For example, consider a pan and tilt camera with 2500 photoreceptors

organized in a 50×50 grid. If it detects a two-dimensional motion vector at each location in the grid, the resulting motion-field feature will have 5000 elements. Yet the camera itself has only two degrees of freedom—it can produce motion in only two directions. Here is an example where dimensionality reduction is clearly appropriate—a 5000-element motion field can be reduced to a two-element motion vector. Chapter 5 will give examples in which *principal component analysis* is used to perform this reduction.

2.2.4 Robot programming

A language of features is very useful in the context of programming robots. A complementary language of behaviors will be presented in Chapter 7. Consider the problem of obstacle avoidance for a mobile robot with a ring of distance sensors in which the goal is to prevent the robot from coming too close to any wall or obstacle. The problem is simplified if the programmer can define features that express distances and directions to obstacles as functions of the raw sensory input. Section 2.5 will show how to apply a sequence of simple operators to the raw sense vector in order to define an image element feature that encodes distances and directions to obstacles.

2.2.5 Automated robot programming

A language that makes programming easier for humans also makes programming easier for computers. If a desired property for a feature can be expressed in the feature language, then a generate-and-test approach can be used to search for a feature that satisfies the property. Chapter 6 will use this method to discover a set of features that can serve as local state variables, the basis for defining control laws for exploration and navigation.

2.3 Features defined

Features are defined in terms of a sensory apparatus which, for the present purposes, can be completely characterized by a function over time giving, for any time t, the value of the raw sense vector at time t. This function is called the *raw sensory feature* and is denoted by $\mathbf{s}$. Here, time is represented by the infinite sequence [0,1,2,...]. If t is the current time, then it is assumed that the feature $\mathbf{s}$ is defined for all time values $\tau \leq t$. For any feature x, the feature *history* $H(x,t)$ is a set of time-value pairs $\{ (\tau, x(\tau)) \mid \tau \in [0..t] \wedge x(\tau) \text{ is defined} \}$. All of the information that is available to the critter from the environment is contained in the *history* of the raw sensory feature. With these preliminary definitions in place, a general definition of "feature" can be given:

> **feature**: a partial function x over the sequence $[0, 1, 2, \ldots]$ whose value at time t, denoted $x(t)$, is completely determined by the history of raw sensory values $H(\mathbf{s}, t)$.

Here, a partial function over time is one that may not be defined for all time values. The possible types for the value $x(t)$ are described in the next section. This definition satisfies the criteria given in Section 2.1: a feature is detected through processing of sensory data (i.e., the raw sensory feature $\mathbf{s}$); a feature can be ephemeral (i.e., a partial function that is not defined at all times); and, as is shown in the next section, features are of many different types.

2.4 Feature types

Features are typed objects. A feature is defined as a function. In general, the type of a function is given by the expression $D \rightarrow R$ where D is the type of the function's domain and R is the type of the function's range. The arrow in the expression is suggestive of the fact that the function is *from* D *to* R. For features, as defined here, D is equal to the sequence $[0, 1, 2, \ldots]$. Since the domain of a feature's function is always the same, the type of the feature's range will be subsequently be used to identify the type of the feature itself.

In mathematical texts, the typeface used to print the name of an object often reflects that object's type. For example, lower-case script letters are used for scalars, lower-case bold-face letters for vectors, and upper case letters for matrices. In order to avoid a proliferation of different type faces, all features will generally be denoted using lower-case script letters. The type of the feature, when it is important, should be clear from context. In some cases, lower-case, bold-face letters will be used when it is important to denote that the feature is a vector. In many cases the type of the feature will not be important. As will be discussed in Section 2.5, many operators are polymorphic — they are defined for many different types of features. There are eight basic feature value types (i.e., types of values that a feature may assume): *scalar*, *vector*, *matrix*, *image element*, *image*, *field element*, *field*, and *histogram*. Other types such as *group* and *focused image* are special cases of other types. Features are implemented on the computer as *objects* in the sense used in object-oriented programming. The set of possible feature types is organized in a hierarchy (see Figure 2.1).

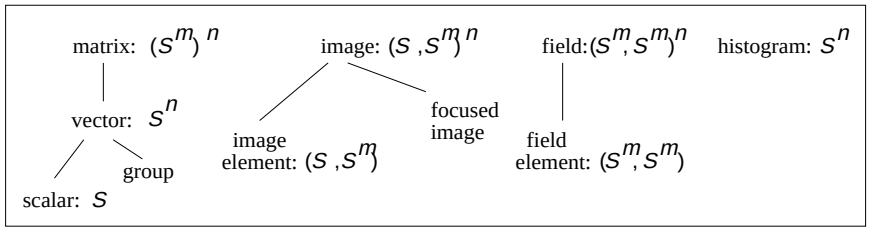


Figure 2.1: The hierarchy of feature types. A scalar is a vector of length 1 and thus inherits all of the operators that apply to vectors. Similarly, a vector is a matrix of dimensions $1 \times n$, an image element is an image of length 1, and a field element is a field of length 1.

The *scalar* (Figure 2.2) is the most basic type — all of the other types are built out of it. The temperature of a room is an example of a scalar feature as is the intensity associated with a single photoreceptor. A scalar v, for the purposes of this dissertation, has two components, both real numbers. The first is called the *value* and is denoted by $v.value$; the second is called the *strength* and is denoted by $v.strength$. In many cases it is the value that is important (e.g., if the strength of the scalar under discussion is always 1) so the expression $v.value$ will often be abbreviated to v when the meaning is unambiguous. The strength of a feature has several purposes. For an ephemeral feature, the strength is 1 when the feature is defined, and 0 otherwise. For structured features such as images, it may be that some elements are more important than others. The strength associated with each element tells how important that element is. For many features, the strength will always

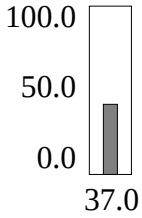


Figure 2.2: A scalar.

be 1 and can be ignored. The set of scalars is denoted by $\mathcal{S}$. The type of a feature will be taken to be the set of all features of that type. Thus the type of a scalar is $\mathcal{S}$.

A *vector* of length n is an n-tuple of scalars (see Figure 2.3). The set of vectors of length n will

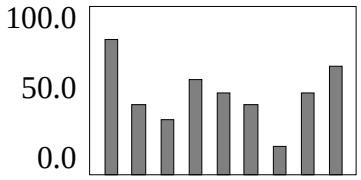


Figure 2.3: A vector of type $\mathcal{S}^9$.

be denoted by $\mathcal{S}^n$. The intensity values of all of the photoreceptors in a retina can be represented as one vector feature. In this case, the positions of the photoreceptors are not encoded in the feature.

A *group feature* is a special type of vector feature in which the elements of the vector are all related in some meaningful way. For example, the set of distance sensors on a mobile robot can be represented as a group feature where the grouping is intended to reflect the fact that all of the elements in the group are of the same type. The raw sensory feature is an example of a vector feature that is not a group feature. One reason to distinguish between vector features and group features is that there are certain feature generators that apply specifically to group features but not to vector features in general.

A *matrix* of dimensions m and n is an n-tuple of vectors of length m (see Figure 2.4). If the

	1	2	3	4	5	6
1	.00	.13	.32	.40	.47	.11
2	.13	.00	.18	.13	.51	.01
3	.32	.18	.00	.26	.32	.06
4	.40	.13	.26	.00	.43	.24
5	.47	.51	.32	.43	.00	.10

Figure 2.4: An example of a matrix of type $(\mathcal{S}^6)^5$.

matrix is viewed as having columns and rows, then n is the number of rows and m is the number columns. The set of all real-valued matrices with n rows and m columns is denoted by $(\mathcal{S}^m)^n$.

An *image element* of dimension m is a pair in which the first component is a scalar and the second is a vector of length m (see Figure 2.5a). The set of m-dimensional image elements is

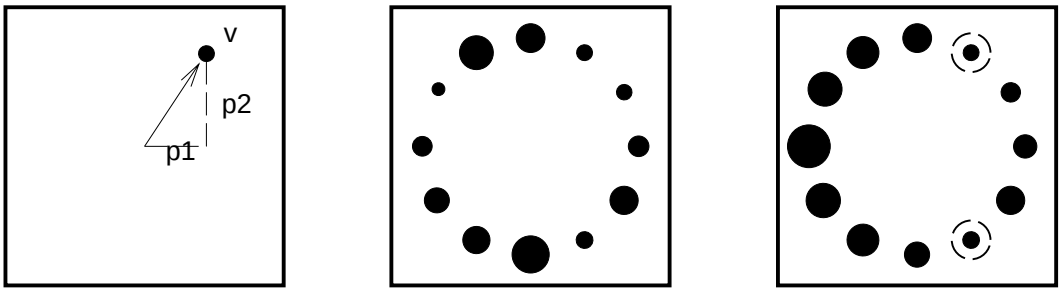


Figure 2.5: a) An example of a two-dimensional image element of type $(\mathcal{S}, \mathcal{S}^2)$. The position vector is represented by the arrow; the value is represented by the size of the disk. b) An example of an image of type $(\mathcal{S}, \mathcal{S}^2)^{12}$, a sequence of image-elements. c) An example of a focused image. The circled elements are local minima and have been assigned a strength of 1.

denoted by $(\mathcal{S}, \mathcal{S}^m)$. An example of such a feature is a photoreceptor with an associated position in the retinal array where the position is represented by a two-dimensional vector.

An *image* of dimension m and length n is an n-tuple of image elements of dimension m (see Figure 2.5b). The set of m-dimensional images of length n is denoted by $(\mathcal{S}, \mathcal{S}^m)^n$. An example of such a feature is a retinal image feature, a set of photoreceptor features in which each photoreceptor has its own associated value and position.

A *focused image* is a special type of image in which the strengths associated with the elements tell which elements are important (see Figure 2.5c). An example of such a feature is an image in which the local minima are identified by being given a strength of 1.

A *field element* of dimension m is a pair in which both components are vectors of length m (see Figure 2.6a). The set of field elements of dimension m will be denoted by $(\mathcal{S}^m, \mathcal{S}^m)$. The local motion feature is an example of this type of feature. The first component of the pair is a vector giving the direction and magnitude of the detected motion. The second component is a vector giving the position in the retina of that motion detector.

A *field* of dimension m and length n is an n-tuple of field elements of dimension m (see Figure 2.6b). The set of m-dimensional fields of length n will be denoted by $(\mathcal{S}^m, \mathcal{S}^m)^n$. An example of a feature of this type is a set of local motion features that collectively detect an optical flow pattern describing the motion detected over the entire retina.

A *histogram* is an approximation of a function of the type $\mathcal{R} \rightarrow \mathcal{R}$, the set of functions from the reals into the reals. The approximation works like this: the domain of the function is limited to an interval $[v_{lo}, v_{hi}]$ instead of the entire interval $\mathcal{R} = (-\infty, \infty)$. This interval is broken up into n subintervals $\{[v_0, v_1), [v_1, v_2), \ldots, [v_{n-1}, v_n]\}$ where $v_0 = v_{lo}$ and $v_n = v_{hi}$. Associated with each subinterval is a value x_i that approximates the value of the function for any value in that

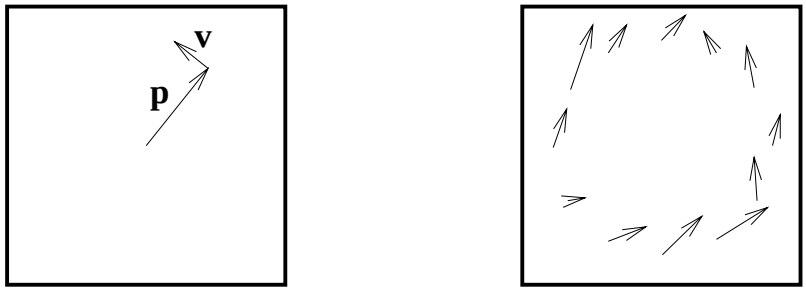


Figure 2.6: a) An example of a field element of type $(\mathcal{S}^2, \mathcal{S}^2)$. The position and value vectors are both represented by arrows. b) An example of a field, a sequence of field-elements, of type $(\mathcal{S}^2, \mathcal{S}^2)^{12}$.

subinterval. The histogram can thus be viewed as a vector of length n — operators that apply to vectors will also apply to histograms (see Section 2.5). A typical use of the histogram is to approximate a probability density function or frequency distribution. For example, x_i could be the number of times that another feature's value was in the range $[v_i, v_{i+1})$. Figure 2.7 illustrates a histogram.

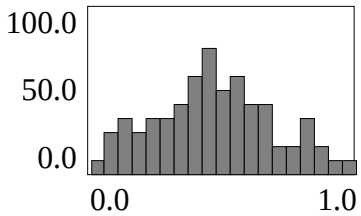


Figure 2.7: An example of a histogram approximating a frequency distribution.

2.5 Feature operators

The language of features has two components. The objects of the language are the features themselves. The tools used to analyze and define new features are the *feature operators*. These two components are illustrated by the human visual system which can be modeled by a hierarchical set of features and operators. In the visual system, information of many different types is represented at different levels, from the retina to the visual cortex. This information can be modeled by a set of features. Processing in the visual system that defines feature values in terms of values of features lower in the hierarchy can be modeled by a set of feature operators. Feature operators will be used in Chapter 3 to define feature generators and testers.

Whereas features are defined to be partial functions over time, feature operators are *functionals*—they map functions into functions or, more specifically, they map features into features. Consider the expression

$$x + y$$

where x and y denote scalar features. This expression denotes a new feature defined in terms of x and y. In this expression, "+" is a feature operator. It is defined by the equation

$$(x + y)(t) = x(t) + y(t)$$

itself an abbreviation for $((x+y)(t)).value = x(t).value + y(t).value$. In these equations, the second "+" is the ordinary one that applies to numbers.

2.5.1 Polymorphism

The preceding example illustrates the concept of *polymorphism*, the use of one operator to denote functions (or functionals) of different types. For example, the + operator applies to scalars and vectors as well as scalar features and vector features. The expression $x + y$ denotes a scalar if x and y are scalars, a scalar feature if x and y are scalar features, a vector feature if x and y are vector features, etc.

2.5.2 Inheritance

As is shown in Figure 2.1, the types of features are organized in a hierarchy. One property of this hierarchy is that a feature inherits the operators of all of its ancestors in the hierarchy. Thus, for example, any operator that applies to matrix features will also apply to vector and scalar features.

2.6 A catalog of operators

The presentation of the feature operators will proceed in a bottom-up fashion. First, operators will be defined that apply to scalar *values*. Then, these operators will be extended in a straightforward way to apply to scalar *features* and additional operators will be introduced that apply to scalar features but not to scalar values. The scalar-feature operators will then be extended so that they apply to vector features and new vector-feature operators will be introduced.[1] The process will be repeated for matrix features, image features, field features, and finally histogram features.

A number of conventions are used in this dissertation. Parentheses serve two purposes. The first is in the shorthand notation of the **eval** operator (Section 2.6.2): $x(t)$ denotes the value of feature x at time t. Application of an operator $\mathbf{f}$ to a feature x is denoted using juxtaposition, not parentheses: $\mathbf{f}\ x$ rather than $\mathbf{f}(x)$. The second use of parentheses is to group terms together to explicitly show how to parse a mathematical expression. For example, the application of operator $\mathbf{f}$ to the feature obtained by applying operator $\mathbf{g}$ to feature x is written $\mathbf{f}(\mathbf{g}\ x)$. Here the parentheses indicate that $\mathbf{g}$ is applied before $\mathbf{f}$.

2.6.1 Scalar-value operators

A scalar has two components: a value and a strength. These components, for scalar v, are denoted by $v.value$ and $v.strength$. The operator **set-strength** is a binary operator. The expression

[1] The new vector-feature operators will apply to the scalar features as well since scalar features are also vector features.

(**set-strength** x y), also written $(x \mid y)$, denotes the scalar defined by

$$\begin{aligned}(x \mid y).value &\stackrel{\text{def}}{=} x.value \\ (x \mid y).strength &\stackrel{\text{def}}{=} y.value.\end{aligned}$$

Mathematical operators (arithmetical operators $+$, $-$, $*$, and $\div$; relational operators $=$, $<$, $\leq$, $>$, and $\geq$; trigonometric operators **sin**, **cos**, and **tan**; and others, including **abs** (absolute value)) are defined on scalars in the following way:

$$(\mathbf{op}\ v^0 \ldots).value \stackrel{\text{def}}{=} \mathbf{op}\ v^0.value \ldots \tag{2.1}$$

$$(\mathbf{op}\ v^0 \ldots).strength \stackrel{\text{def}}{=} \min\{v^0.strength, \ldots\} \tag{2.2}$$

where the second "**op**" is the ordinary operator for numbers. Here the scalar (**op** $v^0 \ldots$) is defined by defining its *value* and *strength* components in terms of ordinary mathematical operators. The ellipsis is a placeholder for the rest of the arguments, if any. Here and in the rest of the dissertation, prefix notation is used for operator application: the operator is given, followed by its arguments. For binary operators, infix notation will sometimes be used when it makes an expression more readable. Thus the expression $+\,v\ w$ will sometimes be written $v + w$.

2.6.2 Scalar-feature operators

Scalar-value operators are extended to scalar-feature operators using the **eval** operator. This operator takes two arguments: a feature x and a time index t. The expression (**eval** x t), also written $x(t)$, denotes the value of feature x at time t. The type of the value is determined by the type of the feature, in this case, scalar. This is the most basic feature operator in the following sense: defining a new feature x means specifying $x(t)$, the value of x at time t, for all values of t. For example, when defining a binary operator **op**, it is necessary and sufficient to define $(x\ \mathbf{op}\ y)(t)$ for all values of t. All of the operators defined on scalar values are extended to scalar features in the following way:

$$(\mathbf{op}\ x^0 \ldots)(t) \stackrel{\text{def}}{=} \mathbf{op}\ x^0(t) \ldots. \tag{2.3}$$

Here the **eval** operator is used to define the feature (**op** $x^0 \ldots$) by defining its value as a function of time in terms of an expression (right-hand side) that was defined in Equations 2.1 and 2.2.

Temporal operators. The operators **tsum**, **tmin**, **tmax**, **delay**, and **ddt** are operators that apply to scalar features but not scalar values. This is because they use history information and thus deal explicitly with time.

$$\begin{aligned}(\mathbf{tsum}\ x)(t) &\stackrel{\text{def}}{=} \sum_{\tau=0}^{t} x(\tau) \\ (\mathbf{tmin}\ x)(t) &\stackrel{\text{def}}{=} \min_{\tau=0}^{t} x(\tau) \\ (\mathbf{tmax}\ x)(t) &\stackrel{\text{def}}{=} \max_{\tau=0}^{t} x(\tau)\end{aligned}$$

$$(\textbf{delay}\ x)(t) \stackrel{\text{def}}{=} x(t-1)$$
$$(\textbf{ddt}\ x)(t) \stackrel{\text{def}}{=} x(t) - x(t-1)$$

Thus the **tsum** operator sums up the values of a feature over time; the **tmin** operator finds the minimum value over time, and the **tmax** operator finds the maximum value over time. The expression (**delay** x) denotes a feature whose value at time t is equal to the value of x at time $t-1$. The expression (**ddt** x), also written $\frac{d}{dt}x$, denotes a feature whose value at time t is equal to the temporal derivative of x at time t.

Statistical operators. The operators **mean** and **sdev** also deal explicitly with time. They compute averages and standard deviations respectively. The expression (**mean** x), also written $(\mu\, x)$, denotes the feature whose value at time t is the average of all of the values of x seen up to time t:

$$(\mu\ x)(t) \stackrel{\text{def}}{=} \frac{\sum_{\tau=0}^{t} x(\tau)\, x(\tau).strength}{\sum_{\tau=0}^{t} x(\tau).strength}$$

The expression (**sdev** x), also written $(\sigma\ x)$, denotes the feature whose value at time t is the standard deviation of x:

$$\left(\frac{\sum_{\tau=0}^{t}(x(\tau))^2\, x(\tau).strength - (\sum_{\tau=0}^{t} x(\tau).strength)[(\mu\ x)(t)]^2}{\sum_{\tau=0}^{t} x(\tau).strength}\right)^{1/2}.$$

2.6.3 Vector-feature operators

Scalar feature operators are extended to vector feature operators using the **index** operator. This operator is first defined for vectors and then extended to vector features. For vector v, (**index** v i), also written v_i, denotes the i^{th} element of the vector, a scalar. For vector feature x, (**index** x i), also written x_i, is defined as a function over time by the equation

$$x_i(t) \stackrel{\text{def}}{=} (x(t))_i.$$

Here the **eval** operator is used to define scalar feature x_i in terms of an expression that involves the **index** operator applied to a vector. Now that the **index** operator is defined for vector features, it can be used to extend all of the scalar-feature operators so that they apply to vector features:

$$(\textbf{op}\ x^0 \ldots)_i \stackrel{\text{def}}{=} (\textbf{op}\ x_i^0 \ldots). \tag{2.4}$$

Here the **index** operator is used to define the feature (**op** $x^0 \ldots$) in terms of an expression that is defined by Equation 2.3 or elsewhere in Section 2.6.2.

The **length** operator returns the number of elements in a vector or vector feature. For feature x, (**length** x) is an integer, not a feature. The implicit assumption is that the length of a feature never changes. The **select** operator is a generalization of the **index** operator. Instead of extracting an element of a vector feature, it extracts a subsequence of elements from a vector feature. The

operator takes two arguments: a feature x of length n and a sequence of nonnegative integers $I = (I_0, I_1, \ldots I_{m-1})$ satisfying $\forall j : 0 \leq I_j < n$. The expression (**select** x I), also written x_I, denotes a new feature of length m defined by the equation

$$(x_I)_j \stackrel{\text{def}}{=} x_{I_j} \text{ for } j = 0, 1, \ldots m-1.$$

Notice that the new feature can be longer than the old one.

Reduction operators. The reduction operators **vsum**, **vmin**, and **vmax** all reduce vector features to scalar features:

$$\begin{aligned}
(\textbf{vsum}\ x)(t) &\stackrel{\text{def}}{=} \sum_i x_i(t) \\
(\textbf{vmin}\ x)(t) &\stackrel{\text{def}}{=} \min_i x_i(t) \\
(\textbf{vmax}\ x)(t) &\stackrel{\text{def}}{=} \max_i x_i(t)
\end{aligned}$$

Thus the **sum** operator sums up the elements of a vector feature; the **min** operator finds the minimum element, and the **max** operator finds the maximum element. For (**vmin** x) [(**vmax** x)], the strength of the output feature is equal to the strength of the minimal [maximal] element of x.

A constructor. The **cat** operator takes a list of vector features as arguments. The expression (**cat** $x^0 \ldots$) denotes a vector feature whose length is equal to the sum of the lengths of the arguments. The value of the feature is a vector obtained by concatenating the values of the input features together.

2.6.4 Matrix-feature operators

A matrix feature is a vector of vector features. Vector-feature operators are extended to matrix features using the **index** operator. For matrix feature x, x_i is a vector feature: $x_i(t)$ is the i^{th} row vector of matrix $x(t)$. All of the operators defined on vector features are extended to matrix features in the following way:

$$(\textbf{op}\ x^0 \ldots)_i \stackrel{\text{def}}{=} (\textbf{op}\ x_i^0 \ldots). \tag{2.5}$$

Here the **index** operator is used to define the feature (**op** $x^0 \ldots$) one element at a time in terms of an expression (right-hand side) that was defined in Equation 2.4 or elsewhere in Section 2.6.3. The i^{th} element of (**op** $x^0 \ldots$) is a vector feature that is obtained by applying the operator to the i^{th} element of x. For operators that are defined for scalar features, this equation can be taken one step further by using a second index:

$$(\textbf{op}\ x^0 \ldots)_{ij} \stackrel{\text{def}}{=} (\textbf{op}\ x_{ij}^0 \ldots). \tag{2.6}$$

Here the feature (**op** $x^0 \ldots$) is defined by applying the operator to each scalar-feature element of matrix feature x.

The **transpose** operator takes one argument, a matrix feature x of type $(\mathcal{S}^m)^n$. The expression (**transpose** x), also written x^T, denotes a feature of type $(\mathcal{S}^n)^m$ and is defined by the equation

$$x^T_{ij} \stackrel{\text{def}}{=} x_{ji}.$$

The **mtimes** operator takes two arguments, matrix features of types $(\mathcal{S}^n)^m$ and $(\mathcal{S}^p)^n$. The expression (**mtimes** x y) denotes a feature of type $(\mathcal{S}^p)^m$ and is defined by the equation

$$(\textbf{mtimes}\ x\ y)_{ij} \stackrel{\text{def}}{=} \sum_{k=0}^{n-1} x_{ik} y_{kj}.$$

The **xprod** (cross-product) operator takes two arguments, vector features of length m and n. The expression (**xprod** x y) denotes a matrix feature of type $(\mathcal{S}^n)^m$ and is defined by

$$(\textbf{xprod}\ x\ y)_{ij} \stackrel{\text{def}}{=} x_i y_j.$$

Similarly, the **xdiff** (cross-difference) operator takes two arguments, vector features of length m and n. The expression (**xdiff** x y) denotes a feature of type $(\mathcal{S}^n)^m$ and is defined by

$$(\textbf{xdiff}\ x\ y)_{ij} \stackrel{\text{def}}{=} x_i - y_j.$$

The **norm** (normalize) operator takes one argument, a matrix feature x. The expression (**norm** x) denotes a matrix feature defined by the equation

$$(\textbf{norm}\ x)_{ij} = x_{ij} / \max_{kl} |x_{kl}|.$$

2.6.5 Image-feature operators

For the purposes of the **length**, **index**, **select**, and **cat** operators, an image feature is treated as a vector of image-element features. The length of an image feature is equal to the number of image elements in the feature. Given image v, v_i denotes an image element, the i^{th} value-position pair of the image. The **select** operator applied to an image feature yields a new image feature comprised of selected image elements of the input feature. The **cat** operator combines a number of image features of the same dimensions into a new image feature whose length is equal to the sum of the lengths of the input image features.

Constructors and selectors. The **image** operator is a constructor that creates an image feature of dimension m and length n from a vector feature of length n and a matrix feature of dimensions $m \times n$. The **val** (for "value") and **pos** (for "position") operators are complementary selectors. For image feature $x = ((v_1, p_1), (v_2, p_2), \ldots, (v_n, p_n)) \in (\mathcal{S}, \mathcal{S}^m)^n$,

$$\begin{aligned} (\textbf{val}\ x) &= (v_1, v_2, \ldots, v_n) \text{ and} \\ (\textbf{pos}\ x) &= (p_1, p_2, \ldots, p_n). \end{aligned}$$

Here, (**val** x) is a vector feature of length m and (**pos** x) is a matrix feature of dimensions $m \times n$. Note that the **val** operator is different from the ".value" operator. The relationship between the **image** operator and the **val** and **pos** operators is given by the equations:

$$\textbf{val}\ (\textbf{image}\ x\ p) \stackrel{\text{def}}{=} x \tag{2.7}$$

$$\textbf{pos}\ (\textbf{image}\ x\ p) \stackrel{\text{def}}{=} p \tag{2.8}$$

$$\tag{2.9}$$

All of the operators defined on vector features are extended to image features in the following way, where the input features must all have the same positions vector:

$$\textbf{op}\ x^0 \ldots = \textbf{image}\ (\textbf{op}\ (\textbf{val}\ x^0) \ldots)\ (\textbf{pos}\ x^0). \tag{2.10}$$

Here the **image**, **val**, and **pos** operators are used to define the feature (**op** x^0) in terms of an expression that was defined in Equation 2.4 or elsewhere in Section 2.6.3. The values vector of (**op** $x^0 \ldots$) is a vector feature that is obtained by applying the operator to the values vectors of $x^0 \ldots$. The positions vector of (**op** $x^0 \ldots$) is the same as the positions vector of the input features.

The local-optima operators. The local-minimum operator **lmin** is analogous to the **min** operator but applies to images. Whereas (**min** x) computes a single global minimum of the elements of a vector, (**lmin** x) computes multiple local minima for an image feature. The expression (**lmin** x) denotes a *focused-image* feature in which the local minima are the elements with strength 1. To determine whether an element is minimal, it is compared with its neighbors — elements whose positions in the image are close to its position. The operator is defined as follows:

$$\begin{aligned}
(\textbf{pos}\ (\textbf{lmin}\ x)) &\stackrel{\text{def}}{=} (\textbf{pos}\ x) \\
(\textbf{val}\ (\textbf{lmin}\ x))_i(t).value &\stackrel{\text{def}}{=} (\textbf{val}\ x)_i(t).value \\
(\textbf{val}\ (\textbf{lmin}\ x))_i(t).strength &\stackrel{\text{def}}{=} \forall j \in (\text{nbrs}\ i) : x_i < x_j \vee (x_i = x_j \wedge i < j) \\
\text{nbrs}\ i &= \{j \mid d_{ij} < \epsilon\} \\
d_{ij} &= \|(\textbf{pos}\ x)_i - (\textbf{pos}\ x)_j\|
\end{aligned}$$

The distance d_{ij} between elements i and j is the Euclidean distance between their associated position vectors. The set (nbrs i) is the set of elements close to i in the image, where the definition of "close" is parameterized by the variable ϵ. The strength associated with element i is 1 if its value is less than that of all of its neighbors, otherwise 0. In the case that two neighboring elements have the same value, the tie goes to the one with the smaller index. The local-maximum operator **lmax** is analogous.

The motion operator. Given image feature x, (**motion** x) denotes a field feature whose elements are local-motion detectors. Each element of the field feature measures the amount of motion detected at the corresponding point in the image.

The detection of motion requires both spatial and temporal information, both of which are provided by an image feature. The spatial information is provided by the positions of the elements of the image; the temporal information is provided by the derivatives of the elements' values with respect to time. A temporal sequence of images, represented as vectors of values and associated positions, can be viewed as an intensity function $E(p,t)$ that maps image positions to values, called intensities, as a function of time. Such a function has both a spatial derivative, $\vec{E}_{\mathbf{p}}$ and a temporal derivative, E_t.[2] The spatial derivative $\vec{E}_{\mathbf{p}}$, also called the *gradient* of E, is a vector in image-position coordinates that gives the direction in which the intensity increases most rapidly.

A large gradient in an image detected by a robot's sensory array corresponds to a detectable property of the environment such as the edge of an object. If the object moves relative to a robot's sensory array (or vice versa), the edges detected in the image will move. This motion will result in a change in intensity. A point in the image with a large gradient will, in the presence of motion, also have a large temporal derivative. This is an informal motivation for the *optical flow* constraint equation (Horn 1986) which defines the optical flow at a point in an image to have magnitude $-E_t/\|\vec{E}_{\mathbf{p}}\|$ and direction $\vec{E}_{\mathbf{p}}$:

$$\mathbf{v} = -\frac{E_t}{\|\vec{E}_{\mathbf{p}}\|}\frac{\vec{E}_{\mathbf{p}}}{\|\vec{E}_{\mathbf{p}}\|} = -\frac{E_t\vec{E}_{\mathbf{p}}}{\|\vec{E}_{\mathbf{p}}\|^2}$$

A problem with this formulation is that if the magnitude of $\vec{E}_{\mathbf{p}}$ is small (or zero), then the calculation will be prone to error (or will be undefined). Since the **motion** operator will be used to measure average motion over time (Chapter 5) and since the measurement of the optical flow is more precise at edges or, in general, when the gradient $\vec{E}_{\mathbf{p}}$is large, we have found it useful to weight the expression using the term $\|\vec{E}_{\mathbf{p}}\|^2$ and measure the value of:

$$\mathbf{v} = -E_t\,\vec{E}_{\mathbf{p}}$$

In most computer vision applications, images are represented as regularly spaced arrays of pixels (picture elements). With such a representation, it is straightforward to define an approximation for the spatial derivative at a point in the image. The images as defined here, however, do not have such a regular structure so a different approach to defining the optical flow field is used. The optical flow measured at element i is taken to be a weighted sum of *local motion vectors* $\mathbf{v}_{ij}$ in the direction from element i to element j where j ranges over all of the elements close to element i (see Figure 2.8). The weight is inversely proportional to the distance between elements i and j. The precise definition of the **motion** operator is given below, where (**pos** x) denotes the vector of positions associated with feature x, and (**val** x) denotes the vector of values associated with feature x.

$$\begin{aligned}
\mathbf{pos}(\mathbf{motion}\ x) &\stackrel{\text{def}}{=} \mathbf{pos}\ x \\
(\mathbf{val}(\mathbf{motion}\ x))_i &\stackrel{\text{def}}{=} \sum_{j\in(\text{nbrs}\ i)} \mathbf{v}_{ij}/\|\mathbf{p}_{ij}\|
\end{aligned}$$

[2] In the description of the **motion** operator, small arrows or bold-face type will be used for clarity to explicitly identify vector variables.

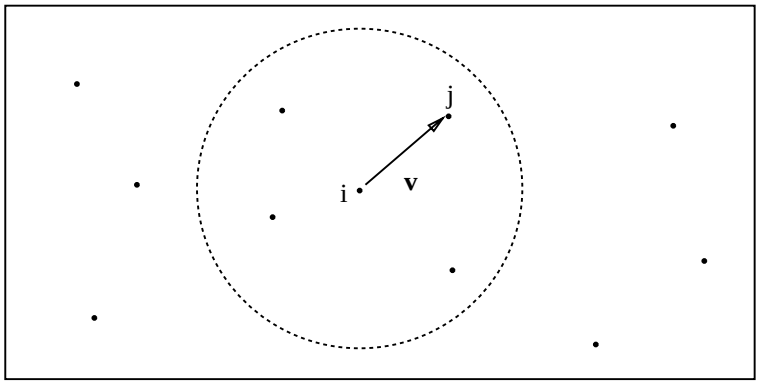


Figure 2.8: The instantaneous motion vector field at element i in an image is taken to be the weighted sum of local motion vectors from element i to other elements in its neighborhood.

$$\begin{aligned}
\mathbf{p}_{ij} &= (\mathbf{pos}\ x_j) - (\mathbf{pos}\ x_i) \\
\mathbf{v}_{ij} &= -E_{t,i}\,\vec{E}_{\mathbf{p},ij} \\
E_{t,i} &= \frac{d}{dt}(\mathbf{val}\ x)_i \\
\vec{E}_{\mathbf{p},ij} &= \frac{((\mathbf{val}\ x)_j - (\mathbf{val}\ x)_i)}{\|\mathbf{p}_{ij}\|}\ \frac{\mathbf{p}_{ij}}{\|\mathbf{p}_{ij}\|}
\end{aligned}$$

Here, $\|\mathbf{p}_{ij}\|$ is the distance in the image between the positions of elements i and j; $E_{t,i}$ is the temporal derivative of the intensity function for element i; and $\vec{E}_{\mathbf{p},ij}$ is the element of gradient $\vec{E}_{\mathbf{p}}$ at element i in the direction toward element j.

2.6.6 Field-feature operators

For the purposes of the **length**, **index**, **select**, and **cat** operators, a field feature is treated as a vector of field-element features. The length of a field feature is equal to the number of field elements in the feature. Given field v, v_i denotes a field element, the i^{th} value-position pair of the field. The **select** operator applied to a field feature yields a new field feature comprised of selected field elements of the input feature. The **cat** operator combines a number of field features of the same dimensions into a new field feature whose length is equal to the sum of the lengths of the input field features.

The **field** operator is a constructor that creates a field feature of dimension m and length n from two matrix features of dimensions $n \times m$. The **val** and **pos** operators are complementary selectors. For field feature $x = ((v_1, p_1), (v_2, p_2), \ldots, (v_n, p_n)) \in (\mathcal{S}^m, \mathcal{S}^m)^n$,

$$\begin{aligned}
(\mathbf{val}\ x) &= (v_1, v_2, \ldots, v_n) \text{ and} \\
(\mathbf{pos}\ x) &= (p_1, p_2, \ldots, p_n).
\end{aligned}$$

Here, $(\mathbf{val}\ x)$ and $(\mathbf{pos}\ x)$ are matrix features of dimensions $m \times n$. The relationship between the

field operator and the **val** and **pos** operators is given by the equations:

$$\textbf{val}\ (\textbf{field}\ x\ p) \stackrel{\text{def}}{=} x \tag{2.11}$$

$$\textbf{pos}\ (\textbf{field}\ x\ p) \stackrel{\text{def}}{=} p \tag{2.12}$$

$$\tag{2.13}$$

All of the operators defined on matrix features are extended to field features in the following way, where the input features must all have the same positions vector:

$$\textbf{op}\ x^0 \ldots = \textbf{field}\ (\textbf{op}\ (\textbf{val}\ x^0) \ldots)\ (\textbf{pos}\ x^0). \tag{2.14}$$

Here the **field**, **val**, and **pos** operators are used to define the feature (**op** $x^0 \ldots$) in terms of an expression that was defined in Equation 2.5 or elsewhere in Section 2.6.4. The values vector of (**op** $x^0 \ldots$) is a vector feature that is obtained by applying the operator to the values vector of x. The positions vector of (**op** $x^0 \ldots$) is the same as the positions vector of the input features.

2.6.7 Histogram-feature operators

Since histograms may be treated as vectors, all of the operators that apply to vector features also apply to histogram features (though the results may not always be meaningful). For histogram feature x, (**length** x) is the number of subintervals in the histogram and x_i is a scalar feature whose value is the value associated with the i^{th} subinterval.

The probability density function operator. The operator **pdf** takes one argument, a scalar feature. The expression (**pdf** x) denotes a feature whose value is a histogram that approximates a probability density function for feature x. The operator has three parameters: n, v_{lo}, and v_{hi}. The histogram approximates the probability density function over the interval $[v_{lo}, v_{hi}]$. This interval is broken up into n subintervals called "class intervals" where the i^{th} class interval is $[v_i, v_{i+1}]$. The feature (**pdf** x) associates with the i^{th} interval the relative frequency with which the value of x falls in that interval. Let

$$b_i(v) = \begin{cases} 1, & \text{if } v \in [v_i, v_{i+1}] \\ 0, & \text{otherwise} \end{cases}$$

Then

$$(\textbf{pdf}\ x)_i(t) \stackrel{\text{def}}{=} \frac{1}{t+1} \sum_{\tau=0}^{t} b_i(x(\tau)).$$

2.6.8 Summary of feature operators

The operators, their definitions, and their types are given in Figure 2.9.

The Feature Language	
type	definition
Scalar-value Operators	
	$(\mathbf{op}\ v^0 \ldots).value = \mathbf{op}\ v^0.value \ldots$
	$(\mathbf{op}\ v^0 \ldots).strength = \mathbf{op}\ v^0.strength \ldots$
Scalar-feature Operators	
$\mathcal{S}^* \to \mathcal{S}$	$(\mathbf{op}\ x^0 \ldots)(t) = \mathbf{op}\ v^0(t) \ldots$
$\mathcal{S} \to \mathcal{S}$	$(\mathbf{tsum}\ x)(t) = \sum_{\tau=0}^{t} x(\tau)$
$\mathcal{S} \to \mathcal{S}$	$(\mathbf{tmin}\ x)(t) = \min_{\tau=0}^{t} x(\tau)$
$\mathcal{S} \to \mathcal{S}$	$(\mathbf{tmax}\ x)(t) = \max_{\tau=0}^{t} x(\tau)$
$\mathcal{S} \to \mathcal{S}$	$(\mathbf{delay}\ x)(t) = x(t-1)$
$\mathcal{S} \to \mathcal{S}$	$(\mu x)(t) = \frac{1}{t+1} \sum_{\tau=0}^{t} x(\tau)$
$\mathcal{S} \to \mathcal{S}$	$(\sigma x)(t) = \frac{1}{t} \left(\sum_{\tau=0}^{t} (x(\tau))^2 - (t+1)[(\mu x)(t)]^2 \right)^{1/2}$
	$(\frac{d}{dt} x)(t) = x(t) - x(t-1)$
Vector-feature Operators	
$(\mathcal{S}^n)^* \to \mathcal{S}^n$	$(\mathbf{op}\ x^0 \ldots)_i = \mathbf{op}\ v_i^0 \ldots$
$\mathcal{S}^n \to \mathcal{S}$	$x_i(t) = (x(t))_i$
$\mathcal{S}^n \to \mathcal{S}^m$	$(x_I)_j = x_{I_j}$
$\mathcal{S}^n \to integer$	$(\mathbf{length}\ x) =$ number of elements in x
$\mathcal{S}^n \to \mathcal{S}$	$(\mathbf{vsum}\ x)(t) = \sum_i x_i(t)$
$\mathcal{S}^n \to \mathcal{S}$	$(\mathbf{vmin}\ x)(t) = \min_i x_i(t)$
$(\mathcal{S}^n)^* \to \mathcal{S}^m$	$\mathbf{cat}\ x =$ see text
Matrix-feature Operators	
$((\mathcal{S}^n)^m)^* \to (\mathcal{S}^n)^m$	$(\mathbf{op}\ x^0 \ldots)_i = \mathbf{op}\ v_i^0 \ldots$
$(\mathcal{S}^n)^m \to (\mathcal{S}^m)^n$	$x_{ij}^T = x_{ji}$
$(\mathcal{S}^n)^m \times (\mathcal{S}^p)^n \to (\mathcal{S}^p)^m$	$(\mathbf{mtimes}\ x\ y)_{ij} = \sum_{k=0}^{n-1} x_{ik} y_{kj}$
$\mathcal{S}^m \times \mathcal{S}^n \to (\mathcal{S}^n)^m$	$(\mathbf{xprod}\ x\ y)_{ij} = x_i\, y_j$
$\mathcal{S}^m \times \mathcal{S}^n \to (\mathcal{S}^n)^m$	$(\mathbf{xdiff}\ x\ y)_{ij} = x_i - y_j$
Image-feature Operators	
$((\mathcal{S}, \mathcal{S}^m)^n)^* \to (\mathcal{S}, \mathcal{S}^m)^n$	$\mathbf{op}\ x^0 \ldots = \mathbf{image}\ (\mathbf{op}\ (\mathbf{val}\ x^0) \ldots)\ (\mathbf{pos}\ x^0)$
$(\mathcal{S}, \mathcal{S}^m)^n \to (\mathcal{S}, \mathcal{S}^m)^n$	$\mathbf{lmin}\ x =$ see text
$(\mathcal{S}, \mathcal{S}^m)^n \to (\mathcal{S}, \mathcal{S}^m)^n$	$\mathbf{lmax}\ x =$ see text
$(\mathcal{S}, \mathcal{S}^m)^n \to (\mathcal{S}^m, \mathcal{S}^m)^n$	$\mathbf{motion}\ x =$ see text
Field-feature Operators	
$((\mathcal{S}, \mathcal{S}^m)^n)^* \to (\mathcal{S}, \mathcal{S}^m)^n$	$\mathbf{op}\ x^0 \ldots = \mathbf{field}\ (\mathbf{op}\ (\mathbf{val}\ x^0) \ldots)\ (\mathbf{pos}\ x^0)$
Histogram-feature Operators	
$\mathcal{S} \to$ histogram	$(\mathbf{pdf}\ x)_i(t) = \frac{1}{t+1} \sum_{\tau=0}^{t} x(\tau) \in [v_i, v_{i+1}]$

Figure 2.9: A summary of the feature language showing a representative set of operators.

Chapter 3

A Generate-and-Test Approach to Feature Learning

The feature operators of the previous chapter define a language that can be used in constructing a perceptual system for a robot. Designing such systems has traditionally been done by hand. The approach taken here is to automate this process. This automation is demonstrated by the critter, a learning agent that learns how to use the robot's sensorimotor apparatus in order to accomplish its goals. For the critter, constructing a perceptual system means learning a set of useful features. This is accomplished as follows: Initially, the critter has only one feature defined, a vector feature called the raw sensory feature. It applies feature *generators* to already existing features in order to produce new features. While this process of searching through the space of possible features proceeds, the critter uses feature *testers* to test the learned features to see if they are useful for achieving its goals.

Section 3.1 describes the language of generators; Section 3.2 describes the language of testers; and Section 3.3 describes ways to control the search process so that it does not become intractable.

3.1 Feature generators

A feature generator has three components: an input type specification, an antecedent, and an output specification. The input type specification is an ordered list of feature types, analogous to the list of parameters for a function in a strongly typed computer language. Most of the generators described in this chapter take a single input feature. The antecedent is a predicate that may provide additional constraints on the set of input features. The output specification is an algorithm that defines the output feature or set of output features in terms of the input feature or features. The output of the generator may be delayed — the output features may be undefined until the input features have been analyzed for a period of time. Any feature operator or legal composition of feature operators may be used to define a generator. A generator is a typed object: its type is given by the types of its input and output features.

Designing a critter requires a good set of generators, just as designing an expert system requires a good set of production rules. With too few generators (or with the wrong generators), the critter will be unable to learn useful new features. With too many generators, the critter will get bogged down in a proliferation of useless features.[1] The size of the search space can be greatly reduced by using using a small set of carefully chosen generators. Strong typing of the generators, meaning that a generator will only be applied to features of the appropriate type, also helps reduce the

[1] Unlike the expert system, the critter has no need for conflict resolution — the features do not interfere with each other.

size of the search space. This chapter describes a representative set of generators that have been sufficient for the robot worlds in which the critter has been tested.

An example of a generator is one whose antecedent matches any vector feature and whose consequent applies the **vmin** operator to the vector feature in order to obtain a new scalar feature. The type of this generator is *vector→scalar*. This generator simply applies a feature operator to an input feature. Some generators, such as the *group generator*, are more complicated. The group generator (described in Section 3.1.1) does not involve a simple operator but instead analyzes a vector feature over a period of time before creating any new features.

For a simple example of the use of the generate-and-test approach to feature learning, consider a critter whose goal is to move a robot to a power source marked with a light beacon. The robot has attached to its front bumper a row of directed photocells that can detect the beacon. The photocell that points most directly at the beacon produces the largest value. The closer the robot is to the beacon, the larger the values returned by the photocells. The critter can learn a number of features that can help it achieve its goal. For example, if x is the vector feature of photocell values, then the feature (**vsum** x), when large, indicates the presence of a power source, and, when maximal, indicates that the robot is at the power source. Applying the image generator (see Section 3.1.2) yields further possibilities. For example, if y is a generated image feature that captures the structure of the array of photocells, then (**pos** (**lmax** y)) denotes a feature that encodes the orientation of the robot relative to the beacon. Various methods for learning homing behaviors can be used (see Section 7.3) in order to achieve the goal of maximizing the learned feature (**vsum** x) and thereby moving the robot to the power source.

The rest of Section 3.1 presents definitions of a number of different generators.

3.1.1 The group generator

The group generator's type is (*vector* → (**list** *group*)) where the keyword **list** indicates that this generator creates multiple output features. The purpose of this generator is to make a first step toward discovering and representing the structure of a sensory apparatus. The group generator decomposes a vector feature into subsets of related elements. These subsets are called *group features* and are useful for defining higher-level features such as image and motion features. Examples of groups of related features are: an array of tactile sensors, a retina of photoreceptors, and a ring of distance sensors. The sensory apparatus described in Section 1.1 was comprised of a ring of distance sensors (one broken), a compass, and a battery-voltage sensor. In this case, the group generator should recognize that the working distance sensors are related and should organize them into a separate group.

The operation of the group generator is based on the observation that spatially adjacent sensors of the same type measure properties of objects that are physically close to each other. Since the world is approximately continuous, adjacent sensors tend to have similar values.[2] This similarity is exploited by the group generator in three steps: (1) definition of one or more intersensor distance metrics, (2) formation of subgroups of sensors that are similar according to all of the distance

[2]Marr and Poggio (1976) exploited this fact in their theory of stereo vision.

metrics, and (3) taking the transitive closure of the similarity relation to form close groups of related sensors. The use of multiple metrics decreases the chance that unrelated sensors will be grouped together. For the implemented group generator, two metrics are used.

1. Definition of distance metrics. The first step is to define a set of metrics, based on the history of values of the input vector feature **s**, each giving a measure of dissimilarity $d_{k,ij}$ between pairs of elements. The metrics are designed so that features that belong together in a sensory array will be similar according to the metrics. For example, the distance between two adjacent distance sensors and the distance between two adjacent photoreceptors in a retina should be small according to each metric.

- The first metric d_1 is based on the principle that in a continuous world, adjacent sensors generally have similar values. The metric is defined as a matrix feature:

$$d_{1,ij}(t) = \frac{1}{t+1} \sum_{\tau=0}^{t} |x_i(\tau) - x_j(\tau)|.$$

  Written in terms of feature operators, the matrix feature is defined by

$$d_1 = \mu(\textbf{abs } (\textbf{xdiff } x\ x))$$

- The second metric d_2 is based on the observation that sensors in a homogeneous array will have similar frequency distributions. (A frequency distribution tells, for each interval in a range, how often the sensor's value falls in that range.) For example, an array of binary touch sensors can be distinguished from an array of photoreceptors by the fact that the different types of sensors have radically different frequency distributions. Binary touch sensors can assume value 0 or 1 whereas photoreceptors can assume any value from a continuous range. $d_{2,ij}$ is proportional to the sum over the distribution intervals of absolute differences in frequency for elements i and j.

$$d_{2,ij} \quad = \quad \frac{1}{2}(\textbf{vsum } (\textbf{abs } ((\textbf{pdf } x_i) - (\textbf{pdf } x_j))))$$

  Here, $(\textbf{pdf } x_i)$ is a histogram feature that approximates the frequency distribution for feature x_i. The distance between two elements will have a maximum value of 1.0 if the frequency distributions of the elements do not overlap at all.

2. Formation of subgroups of similar sensors. The group generator's second step is to use the distance metrics d_k to form subgroups of similar sensors. Elements i and j are similar, written $i \approx j$, if they are similar according to each distance metric d_k:

$$i \approx j \ \text{ iff } \ \forall k : i \approx_k j.$$

The definition of $i \approx_k j$ requires the use of a threshold. One way to define this threshold, that has proven to be more robust than the use of a constant, is this:

$$\epsilon_{k,i} = 2 \min_j \{d_{k,ij}\}.$$

Each element i has its own threshold based on the minimum distance from i to any of its neighbors. Elements i and j are considered similar if and only if both $d_{k,ij} < \epsilon_{k,i}$ and $d_{k,ij} < \epsilon_{k,j}$, that is if j is close to i from i's perspective and vice versa. Combining these constraints gives

$$i \approx_k j \text{ if } d_{k,ij} < \min\{\epsilon_{k,i}, \epsilon_{k,j}\}.$$

3. Formation of closed subgroups. The group generator's third step is to take the transitive closure of the similarity relation to produce the *related-to* relation $\sim$. Consider again the ring of distance sensors. Adjacent sensors tend to be very similar according to the distance metric, but sensors on opposite sides of the ring may be dissimilar (according to d_1) since they detect information from distinct and uncorrelated regions of the environment. In spite of this fact, the entire array of distance sensors should be grouped together. This is accomplished by defining the *related-to* relation $\sim$ as the transitive closure of the *similarity* relation $\approx$. Two elements i and j are *related* to each other, written $i \sim j$, if $i \approx j$ or if there exists some other element k such that $i \sim k$ and $k \sim j$:

$$i \sim j \text{ iff } i \approx j \vee \exists k : (i \sim k) \wedge (k \sim j).$$

The related-to relation $\sim$ is clearly reflexive, symmetric, and transitive and is therefore an equivalence relation. Computing the relation $\sim$ for i and j given the relation $\approx$ is straightforward (e.g., Cormen et al., 1990).

The definition of "group of related features" is given in terms of the $\sim$ relation. The group of all elements of vector feature x that are related to element x_i is given by x_{G_i} where

$$G_i = \{j \mid i \sim j\}.$$

Here, G_i is the set of indices that are related to i, as defined above, and x_{G_i} is a new group feature derived from feature x using the **select** operator (Section 2.6.3). The group generator creates a new group feature for each element of its input feature:

$$\textbf{group } x = \{x_{G_i} \mid 0 \leq i < n\}$$

where n is the length of feature x. Note that $G_i = G_j$ if $i \sim j$ so that the number of output features of the group generator may be less than n.

For the robot described in the introduction (Section 1.1), the raw sensory feature has 29 elements. In order, these are: 24 distance sensors (one of which is defective), a battery-voltage sensor, and a four-element digital compass. The distance metric is computed while the robot wanders randomly for 2,500 steps. For each element of the raw sensory feature, the set of similar elements $\{j \mid i \approx j\}$ is computed and shown below:

(0 1 22 23) (0 1 2 3) (1 2 3) (1 2 3 4 5) (3 4 5) (3 4 5 6) (5 6 7 8) (6 7 8) (6 7 8 9) (8 9 10 11) (9 10 11 12) (9 10 11 12 13) (10 11 12 13) (11 12 13 14) (13 14 15) (14 15 16) (15 16 17) (16 17 18) (17 18 19) (18 19 21) (20) (19 21 22 23) (0 21 22 23) (0 21 22 23) (24) (25) (26) (27) (28)

Notice that neighboring distance sensors are grouped together. For example, the group (0 1 22 23) contains two elements on each side of element 0. The related-to relation $\sim$ is obtained by taking the transitive closure of the relation $\approx$ and is described by the following equivalence classes:

(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23)
(20) *defective*
(24) *battery voltage*
(25) *east*
(26) *north*
(27) *west*
(28) *south*

The distance sensors have all been grouped together into a group containing no other sensors. Further examples of the application of this generator are given in Chapter 4.

3.1.2 The image generator

The image generator takes a group feature x and associates a position with each element thus producing an image feature y. Its type is (*group* $\rightarrow$ *image*). The point of the image generator is to find an assignment of positions to elements that captures the structure of an array of sensors as reflected in the distance metric d_1. The reason that d_1 is used is that it is based on instantaneous differences that should, on average, be small for sensors that are close together in a well-designed sensory array. The distance between the positions of any two elements in the constructed image should be equal to the distance between those elements according to the metric d_1. Expressed mathematically, image y should satisfy

$$\|(\mathbf{pos}\ y_i) - (\mathbf{pos}\ y_j)\| = d_{1,ij}$$

where $(\mathbf{pos}\ y_i)$ is the position vector associated with the i^{th} element in the image and $\|(\mathbf{pos}\ y_i) - (\mathbf{pos}\ y_j)\|$ is the Euclidean distance between the positions of the i^{th} and j^{th} elements.

Finding a set of positions satisfying the above equation is a constraint-satisfaction problem. If the group feature x has n elements, then the metric d_1 provides $n(n-1)/2$ constraints.[3] Specifying the positions of n points in $n-1$ dimensions requires $n(n-1)/2$ variables: 0 for the first point, which is placed at the origin; 1 for the second, which is placed somewhere on the x axis; 2 for the third, which is placed somewhere on the $x-y$ plane; etc. Thus, to satisfy the constraints, n position vectors of dimension $n-1$ are required. Solving for the position vectors given the distance constraints can be done using a technique called *metric scaling* (Krzanowski, 1988).

The problem remains that n points of dimension $n-1$ are inconvenient to use, if not meaningless, for large n. In general, sensory arrays are 1-, 2-, or 3-dimensional objects. What is needed is a method for finding the smallest number of dimensions that are needed to satisfy the given constraints without excessive error, where the error can be defined by the equation

$$E = \frac{1}{2}\sum_{ij}(\|(\mathbf{pos}\ y_i) - (\mathbf{pos}\ y_j)\| - d_{ij})^2.$$

[3]The metric can be represented as a symmetric matrix with zeros on the diagonal. Such a matrix has $n(n-1)/2$ free parameters.

Metric scaling helps by ordering the dimensions according to their contribution toward minimizing the error term.

Ignoring all but the first dimension (i.e., using only the first element of the position vectors), yields a rough description of the sensory array with large error (unless the array really is a one-dimensional object). Using all $n-1$ dimensions yields a description that has zero error but contains a lot of useless information. Statisticians use a graph called a "scree diagram" (Figure 3.1a), which shows the amount of variance in the data that is accounted for by each dimension, to subjectively choose the right number of dimensions. The image generator chooses the number of dimensions to be equal to m where m maximizes the expression $\sigma^2(m) - \sigma^2(m+1)$ where $\sigma^2(m)$ is the variance in the data accounted for by the m^{th} dimension. For the example, $m = 2$. The set of two-dimensional positions found by metric scaling for the group of distance sensors is shown in Figure 3.1b.

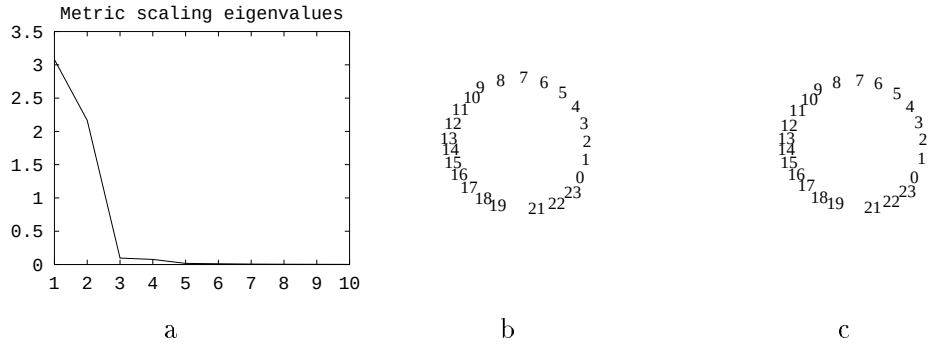


Figure 3.1: Learning a structural model of a ring of distance sensors. (a) The scree diagram shows that the first two dimensions account for most of the variance. (b) Metric scaling is used to assign positions to elements of the group of distance sensors. The 22-dimensional position vectors are projected onto the first two dimensions to produce the representation shown above. (c) A relaxation algorithm is used to find a set of two-dimensional positions for the group of distance sensors that best satisfies the constraints $\|\mathbf{p}_i - \mathbf{p}_j\| = d_{ij}$. Notice the gap corresponding to the defective distance sensor. The element with index 0 corresponds to the robot's forward sensor.

The set of $(n-1)$-dimensional position vectors optimally describe the structure of a group, but when these positions are projected onto a subspace of lower dimensionality, the resulting description is no longer optimal. Elements that were the right distance apart in $n-1$ dimensions will generally be too close together in the two-dimensional projection. To compensate for this, a relaxation algorithm is used to find the best set of positions in a small-dimension space to approximate the given distances in $n-1$ dimensions.

The relaxation algorithm associates an m-dimensional point $\mathbf{p}_i$ with each element i of the group feature, where m is the number of dimensions for the resulting image feature. For the running example, $m = 2$. The algorithm repeats the following loop until no appreciable change occurs:

1. Compute the "force" f_i on each point $\mathbf{p}_i$ from all the other points

$$\begin{aligned} f_i &= \sum f_{ij} \\ \text{where } f_{ij} &= (\|\mathbf{p}_i - \mathbf{p}_j\| - d_{ij})(\mathbf{p}_j - \mathbf{p}_i)/\|\mathbf{p}_j - \mathbf{p}_i\| \end{aligned}$$

2. Move each point $\mathbf{p}_i$ a small distance in the direction of the force acting on it:

$$\mathbf{p}_i \leftarrow \mathbf{p}_i + \eta f_i$$

where η is a small constant.

Dewdney (1987) presents a similar algorithm.

This relaxation algorithm could be used without metric scaling by simply initializing the vector of positions randomly. Metric scaling provides two benefits. It helps decide how many dimensions are needed for the image feature, and it provides a starting point for the relaxation algorithm, decreasing the chance that the algorithm will find a local but not global minimum of the error function. The application of the relaxation algorithm to the group of distance sensors is illustrated in Figure 3.1c.

To summarize, the image generator takes a group feature x and produces an image feature y defined by

$$\begin{aligned} (\textbf{val}\ y_i) &= x_i \\ (\textbf{pos}\ y_i) &= \mathbf{p}_i \end{aligned}$$

where the values for the position vectors are found using metric scaling and a relaxation algorithm so that they approximately satisfy the constraints

$$\|\mathbf{p}_i - \mathbf{p}_j\| = k \sum_t |x_i(t) - x_j(t)|$$

while keeping the dimensionality of the position vectors $\mathbf{p}_i$ small. Further examples of the application of this generator are given in Chapter 4.

3.1.3 The local-optima generators

While any operator can be used to define a generator, some, such as the *local-optima* operators, are more relevant for the purposes of this dissertation and will be given special mention. A local-optima generator takes an image feature as input and produces a focused-image feature. Its type is thus (*image* $\rightarrow$ *focused-image*) The two types of local-optima generators are the local-minima generator, which uses the **lmin** operator, and the local-maxima generator, which uses the **lmax** operator. The local-optima generators provide a method for focus of attention. The feature (**lmin** x) focuses attention on just those elements that are less than all of their neighbors. This generator will be demonstrated in Chapter 6.

3.1.4 The tracker generator

The tracker generator takes a focused-image feature as input and produces a list of image-element features. Its type is thus (*focused-image* $\rightarrow$ (**list** *image-element*)). It uses a special-purpose operator called a *tracker*.

The purpose and operation of the tracker will be described using the example of a robot with a ring of distance sensors. Suppose that the robot is near a corner in a large room. The corner is an

interesting "feature" (in the non-technical sense) of the robot's environment: the presence of the corner is a property of the local region that can distinguish it from other regions in the environment. The corner can be used as a point of reference — the robot's position can be described relative to the corner.

With the tracker, it is possible to define an image-element feature that captures these properties and can be used as a representation of the corner. Let x be the robot's image feature of distance sensors. This feature may have been given to the robot by design, or defined using the image generator. The corner is manifested in feature x as a local maximum, since the distance to the corner is greater than the distance to points on the wall near the corner. Let y be the focused-image feature (**lmax** x). The corner is then represented by an element of y with strength 1. An element of a focused-image feature with strength 1 will be called an *active image element*.

For the active image element $z = (v, \mathbf{p})$ that corresponds to the corner, the value v is the distance to the corner and the position vector $\mathbf{p}$ is an encoding of the direction to the corner from the robot. What is missing from the focused-image representation of the corner is a concept of identity over time. The location of the local maximum in the image will move from one element to another when the robot moves — several different elements will detect the same corner.

The tracker maintains a list of active image-element features in the input focused-image feature. An output image-element feature that exists at time t continues to exist at time $t+1$ if there is a matching active image element in the new focused-image, where two image elements match if their values and positions are approximately equal. Specifically, if an output image-element feature has value $(v_i(t), \mathbf{p}_i)$ at time t, then at time $t+1$, it will have value $(v_j(t+1), \mathbf{p}_j)$ where $(v_j(t+1), \mathbf{p}_j)$ is an active image element in the input image at time $t+1$ satisfying

$$(|v_i(t) - v_j(t+1)| < \epsilon_v) \wedge (i \approx j)$$

if such an active image element exists, where ϵ_v is a small constant and the relation $\approx$ is the same one that was defined on page 41. If more than one exists, the choice of which to use is arbitrary. If none exists, then the output feature ceases to exist.

The tracker implements a form of focus of attention. It abstracts away the small changes in value and position of an active image element in order to produce a feature which tracks an interesting property of the robot's environment. The tracker generator is an example of a generator that produces an *ephemeral* feature (Section 2.1). When the robot moves away from a corner, the strength of the image-element feature that was tracking the corner goes to zero — the feature ceases to exist. An application of this generator for the purpose of defining local state variables is described in Chapter 6.

3.2 Feature testers

The feature testers provide the second half of the generate-and-test approach to feature learning. A feature tester is any feature generator whose output feature is used in some way to characterize a set of one or more input features. The output feature's associated strength is used to tell how much confidence to place in the feature's value. Like the feature generators, the feature testers are typed according to the type of features to which they apply.

Testers come in different types. A Boolean tester produces a Boolean-valued feature. An example is a tester that determines whether an input feature is constant. The tester's output value is 1 as long as the input feature does not change. If the input ever changes, then the tester's output is permanently set to 0. The strength of the tester's output feature increases slowly from zero asymptotically to 1. The longer the input feature remains constant, the higher the tester's confidence that it is indeed a constant feature.

Examples of real-valued testers are **mean** (μ) and **sdev** (σ). The values of ($\mu\, x$) and ($\sigma\, x$) can be used to characterize the range of values that feature x assumes.

An example of a more complex tester is the *correlator* (Section 3.2.3) which characterizes the relationship between two input features while testing the hypothesis that there is a linear relationship between them.

3.2.1 Learning hill-climbing functions

Other work (Pierce & Kuipers, 1991) has demonstrated the use of a number of feature testers to identify features that could be used in the definition of goal-directed behaviors for a mobile robot. In that work, a goal was given *a priori* and a reward signal was provided to the learning agent that told it when it had achieved the goal. The learning agent's task was to learn how to achieve the goal reliably and efficiently. It solved the task using the following two steps:

1. Derive a function defined in terms of the raw sense vector such that this function is maximized at the goal state and is suitable for hill-climbing.

2. Learn a behavior that does gradient ascent on this hill-climbing function.

What is of interest for the current discussion is the first step. The critter applied feature testers including the continuity tester (Section 3.2.2) to its raw sensory feature to determine which sensors were both smoothly varying and useful in predicting the value of the reward signal. These features were then used to define a hill-climbing function which was in turn used to train a reactive behavior for moving efficiently to the goal state.

In that work, the emphasis was on the testing portion of the generate-and-test process of feature learning.[4] In the work described here, the emphasis is on the generation of new features. Features such as the group and image features are an end in themselves — they provide information about the physical structure of a robot's sensory apparatus. The only tester that will be used in the generate-and-test process of learning local state variables (Chapter 6) is the correlator, a type of linear regression tester.

3.2.2 The continuity tester

The continuity tester determines whether a scalar feature varies continuously as the robot moves around in state space. Its value is equal to the standard deviation of the time derivative of the

[4]See the original paper for the descriptions of the testers used.

feature divided by the standard deviation of the feature:

$$C\,x = \sigma(\frac{d}{dt}x)/\sigma(x)$$

where x is the scalar feature being tested. The standard deviation is a statistical measure and as such becomes more reliable with larger sample sizes. To be useful, the continuity tester should be computed while the robot explores a representative subset of its state space.

3.2.3 The linear-regression tester

The linear regression tester can be used to determine whether one scalar feature's value is a good predictor of the value of another scalar feature. Given scalar features x and y, it uses linear regression (see, for example, Press et al., 1988) to determine the best values of m and b in the equation

$$y(t) = m\,x(t) + b + e(t).$$

Here, m and b are scalars and e is the error term. Linear regression finds the values of m and b that minimize the value of $\sum_{\tau=0}^{t}[e(\tau)]^2$. Their values are defined by the equations

$$\begin{aligned} m &= \frac{S_{xx}S_y - S_x S_{xy}}{(t+1)S_{xx} - (S_x)^2} \\ b &= \frac{(t+1)S_{xy} - S_x S_y}{(t+1)S_{xx} - (S_x)^2} \end{aligned}$$

where

$$\begin{aligned} S_x &= \sum_{\tau=0}^{t} x(\tau) \\ S_y &= \sum_{\tau=0}^{t} y(\tau) \\ S_{xx} &= \sum_{\tau=0}^{t} (x(\tau))^2 \\ S_{xy} &= \sum_{\tau=0}^{t} x(\tau)\,y(\tau). \end{aligned}$$

The *linear correlation coefficient*

$$r = \frac{\sum_\tau (x(\tau) - \mu\,x)(y(\tau) - \mu\,y)}{[\sum_\tau (x(\tau) - \mu\,x)^2 \sum_\tau (y(\tau) - \mu\,y)^2]^{1/2}}$$

gives a measure of the strength of the correlation between x and y. If $|r|$ is near one, then the two features are highly correlated. If $|r|$ is near zero, then they are uncorrelated. If two features are highly correlated, then the approximation $y = m\,x + b$ can be used to reliably predict the value of y given a knowledge of the value of x. The linear regression tester will be used in the dynamic action model to model the effects of the robot's actions on its local state variables (Section 7.4.2).

3.2.4 The correlator

The correlator is a specialization of the linear-regression tester that is useful for determining the best value of m in the equation:

$$\dot{y}(t) = m\, x$$

where y and x are scalar features and m is a real-valued constant. The correlator uses an additional input feature, a *reset* signal. Instead of directly applying linear regression to x and $\dot{y}$,[5] the correlator first integrates both sides of the equation, computing the new features

$$\begin{aligned} \Delta y(t) &= y(t) - y(t_0) \\ S_x(t) &= \sum_{\tau=t_0}^{t} x(\tau) \end{aligned}$$

where t_0 is reset to t whenever the reset signal is on (equal to 1). When the reset signal is off, t_0 does not change. The correlator applies linear regression to features S_x and Δy to produce output features m an r. In addition, the correlator produces the feature

$$\gamma = \sigma(\Delta y)/\sigma(S_x),$$

a feature that is useful for testing whether there is a causal relationship between x and y. If γ is small relative to $(\sigma\, y)$, then x does not affect y. Without additional information, the converse is not necessarily true. The correlator will be used in Section 6.2 to characterize the effects of a robot's motor control signals on a set of features.

3.3 Search Control

The generate-and-test process of learning potentially useful features executes the following steps in a continuous loop. Initially, there is only one feature, the *raw sensory feature*. This feature is marked as *new*.

1. Each tester is applied to each new feature to which it is applicable.
2. Each generator is applied to each new feature to which it is applicable.
3. The features that were new are marked as old, and the features just generated are marked as new.

The critter initially explores by randomly choosing motor control vectors. Later (see Chapter 7), it learns more sophisticated behaviors that allow it to explore more intelligently. The generate-and-test process continues until a termination criterion is satisfied. That criterion will depend on the goal of the learning task. It may be defined in terms of a Boolean tester that returns TRUE when it has found an appropriate feature.

[5] The problem with using the signal $\dot{y}$ directly is that any noise or quantization error in feature y is enhanced by the process of differentiation.

For the task of learning local state variables (Chapter 6), the correlator will be used to find features whose derivatives can be approximated by linear functions of the robot's motor control signals. Such features will be used as *local state variables*.

In generate-and-test approaches to learning, controlling the search through a large space of possibilities is an important concern. Without any constraints, the number of features generated on each iteration of the generate-and-test loop may grow exponentially. There are several ways to constrain a search algorithm.

One way is to limit the depth of the search. In the current implementation of the generate-and-test algorithm, it is possible to set a limit on the number of generations of new features that are created.

A second way is to limit the breadth of the search. This method is used in genetic algorithms where population size is constrained to a certain number. This method requires a fitness measure to tell which members of the population are worthy of survival. Such a fitness measure can be defined as a feature tester, though this has not been done for any of the experiments described in this dissertation. This is an instance of a more general approach: to define a Boolean tester that heuristically identifies features to be pruned from the tree of generated features.

A third way to constrain a search space is to limit the branching factor. For the feature-learning problem, this is the average number of new features that are generated for each old feature at each step of the generate-and-test process. The branching factor for the feature learning problem is limited in two ways. First, the number of generators is kept reasonably small. Second, the generators are strongly typed which means that not every generator applies to every feature. Strong typing is enhanced by the use of a well developed hierarchy of feature types. For example, the distinction between a vector feature and a group feature limits the number of features to which an image generator will apply. Vector features and group features are structurally equivalent, but the image generator only applies to the latter.

In the experiments described in this dissertation, the combinatorial explosion of features has not been an issue. The generators form deep but narrow search trees.

Chapter 4

Learning a Model of a Sensory Apparatus

The previous chapters introduced a language that is appropriate for analyzing a robot's sensory apparatus and defining new features. This feature language is useful for solving the problem of modeling the robot's sensorimotor apparatus. The first half of the problem, modeling the sensory apparatus, is described in this chapter. The second half, modeling the motor apparatus, is described in Chapter 5.

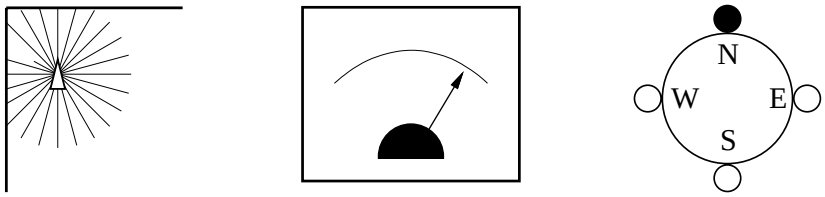
A sensory apparatus may contain a structured array of homogeneous sensors. Examples of such arrays are a ring of distance sensors, a retina of photoreceptors, and an array of touch sensors. This chapter describes experiments in which the group and image generators (Sections 3.1.1 and 3.1.2) are used to learn and represent this structure. In Section 4.1, the generators are demonstrated for the robot and environment described in Chapter 1. In Section 4.2, they are applied to a robot with a simple visual system.

4.1 A robot with distance sensors

This experiment involves a simulated mobile robot with a ring of distance sensors. The goal is to use the group and image generators to define an image feature that captures the physical structure of the sensor ring.

The robot's environment. The robot's world for this experiment is simulated as a rectangular room of dimensions 6 meters by 4 meters. The robot itself is modeled as a point.

The robot's sensory apparatus. The robot's sensorimotor apparatus is illustrated in Figure 4.1. Each sensor's value lies between 0.0 and 1.0. Collectively, the sensors define the raw sense vector **s**. The first 24 elements of the raw sense vector give the distances to the nearest objects in each



distance sensors battery voltage digital compass

Figure 4.1: The robot's sensory apparatus includes a ring of 24 distance sensors, of which one is defective and always returns a value of 0.2; a battery-voltage sensor; and a digital compass that tells which direction the robot is most nearly facing.

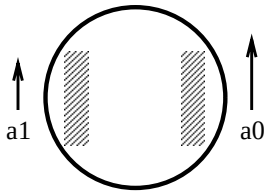


Figure 4.2: The robot has a tank-style motor apparatus with two control signals that tell how fast to move the right and left treads.

of 24 directions. These have a maximum value of 1.0 which they take on when the nearest object is beyond one meter away. The sensors are numbered clockwise from the front. The 21st element is defective and always returns a value of 0.2. The 25th element is a sensor giving the robot's battery's voltage, which decreases slowly from an initial value of one. The 26th through 29th elements comprise a digital compass. The element with value 1 corresponds to the direction (E, N, W, or S) in which the robot is most nearly facing.

The robot's motor apparatus. The robot has a "tank-style" motor apparatus (Figure 4.2). Its two motor control signals a_0 and a_1 tell how fast to move the right and left treads. Moving the treads together produces forward or backward motion; moving them in opposition produces rotation. The robot's maximum speed is 0.25 meters per second. Its maximum rotational speed is 100 degrees per second.

The critter controlling the robot uses the following exploration strategy: choose a random motor control vector; execute it for one second (10 time steps); repeat. Experience has shown that this strategy is more effective for efficiently exploring a large subset of the robot's state space than choosing actions randomly at each time step.

4.1.1 Discovering related sensory subgroups.

The group generator (see Section 3.1.1) can be used to recognize groups of related sensors, for example, the group of distance sensors. The grouping is based on two distance metrics d_k that give measures of dissimilarity between any two elements of vector feature. The first distance metric is defined by the equation

$$d_{1,ij}(t) = \frac{1}{t+1} \sum_{\tau=0}^{t} |x_i(\tau) - x_j(\tau)|.$$

Its value after the robot has explored for five minutes is given in Figure 4.3. The second distance metric is defined by the equation

$$d_{2,ij} = \frac{1}{2}(\mathbf{vsum}\ (\mathbf{abs}\ ((\mathbf{pdf}\ x_i) - (\mathbf{pdf}\ x_j)))).$$

Its value after 5 minutes is shown in Figure 4.4. These distance metrics are used to identify, for each element s_i of feature $\mathbf{s}$, the set of elements that are similar to s_i according to the distance metrics

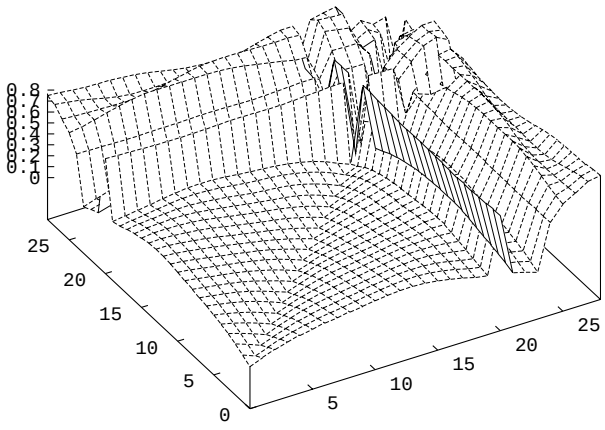


Figure 4.3: The value of distance metric d_1 after the robot has wandered for five minutes. The axes give the values for i and j; the height of the plot gives the value of $d_{1,ij}$. The ij^{th} entry in the matrix is a measure of the dissimilarity between the i^{th} and j^{th} elements of the raw sense vector.

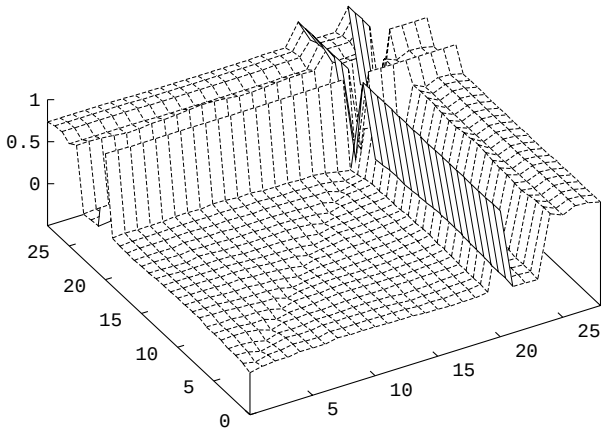


Figure 4.4: The value of distance metric d_2 after the robot has wandered for five minutes.

(0 1 22 23) (0 1 2 3) (1 2 3) (1 2 3 4 5) (3 4 5) (3 4 5 6) (5 6 7 8)
(6 7 8) (6 7 8 9) (8 9 10 11) (9 10 11 12) (9 10 11 12 13) (10 11 12 13)
(11 12 13 14) (13 14 15) (14 15 16) (15 16 17) (16 17 18) (17 18 19)
(18 19 21) (20) (19 21 22 23) (0 21 22 23) (0 21 22 23) (24) (25) (26)
(27) (28)

Figure 4.5: For each element of the raw sense vector, the set of similar elements is shown.

(see Section 3.1.1 for the details). These sets of elements are shown in Figure 4.5. Treating these sets as equivalence classes and taking the transitive closure yields the groups of related elements shown in Figure 4.6.

(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23)
(20) *defective*
(24) *battery voltage*
(25) *east*
(26) *north*
(27) *west*
(28) *south*

Figure 4.6: By taking the transitive closure of the groups of similar elements, a list of related subgroups is produced. The working distance sensors have all been grouped together.

4.1.2 A structural model of the sensory apparatus.

The group generator identified seven groups of related sensors. The largest group is a candidate for application of the image generator (Section 3.1.2). The image generator's first step is to apply metric scaling to the distance metric d_1. The value of this distance metric is shown in Figure 4.7. Metric scaling produces the scree diagram of Figure 4.8a indicating that the sensory array is best modeled as a two-dimensional object. Metric scaling assigns positions to each element of the group feature. Projecting these positions onto the first two dimensions produces the mapping shown in Figure 4.8b. The set of positions produced by metric scaling is improved using a relaxation algorithm so that the distances in the resulting image more closely match distance metric d_1. The resulting set of positions is shown in Figure 4.8c. The result of the experiment is a structural description of the robot's sensory apparatus (Figures 4.6 and 4.8) that will be used in Chapter 5 to analyze the robot's motor apparatus.

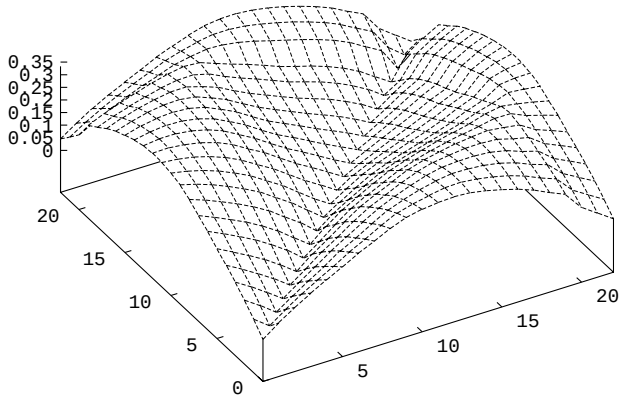
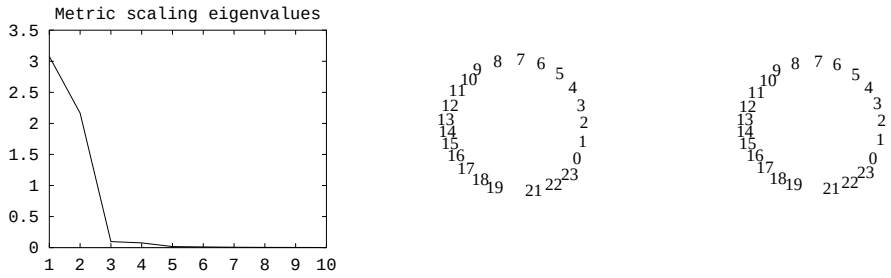


Figure 4.7: This figure shows the intersensor distances $d_{1,ij}$ for all of the distance sensors. It is this matrix that the image generator uses to construct the image feature that captures the structure of the ring of distance sensors.



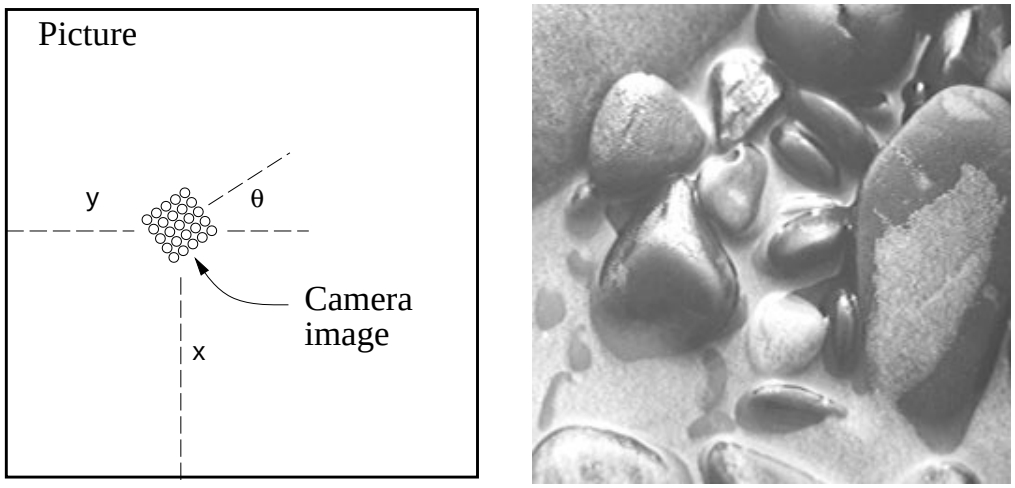
a b c

Figure 4.8: (a) The metric-scaling scree diagram for the group of distance sensors indicates that the sensors are organized in a two-dimensional array. (b) The two-dimensional projection of the set of positions produced by metric scaling for the group of distance sensors provides an initial approximation of the ring structure of the array of distance sensors. (c) The final set of positions is produced using a constraint-satisfaction relaxation algorithm that uses the previous set of positions as initial values.

4.2 The roving eye

This experiment involves a more fanciful, simulated robot called a "roving eye." It's primary sensory array is a retina of photoreceptors.

The robot's environment. This robot is a simulation of a small camera mounted on the movable platform of an X-Y plotter, pointing down at a square picture 10 centimeters on a side. The camera sees one square centimeter of the picture at a time. The robot has three degrees of freedom (translation in two directions and rotation) and its state space is described by three state variables (two for position and one for orientation). The robot's structure is shown in Figure 4.9a. The



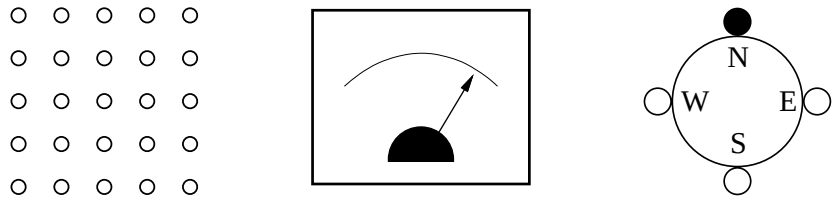
a b

Figure 4.9: (a) The robot is a "roving eye" that can see a 1 centimeter wide image that is part of a picture that is 10 centimeters wide. (b) The picture used for the roving-eye experiment is a close-up view of the Oregon coast.

actual picture used is shown in Figure 4.9b.

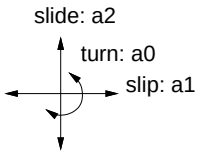
The robot's sensory apparatus. The sensory apparatus is as before except that the ring of distance sensors has been replaced by a 5 by 5 retinal array looking down on a picture. The sensory apparatus is illustrated in Figure 4.10.

The robot's motor apparatus. The motor apparatus is illustrated in Figure 4.11. Unlike the robot of the previous experiment, this robot has three degrees of freedom instead of two. The three elements (control signals) of the motor control vector are *rotate*, *slip* (for motion to the left or right), and *slide* (for motion forward or backward). Positive values of the first turn it counterclockwise (up to 100 degrees per second) and negative values clockwise. The second determines how fast the robot advances (up to 2.5 centimeters per second). Negative values move it backwards. The third determines how fast the robot moves to the right (for positive values) or left (for negative values).



retinal array battery voltage digital compass

Figure 4.10: The roving eye's sensorimotor apparatus includes a 5 by 5 retinal array of photoreceptors. The image on the retina corresponds to a 1-centimeter square area of the 10-centimeter square picture. The sensory apparatus also includes the battery-voltage sensor and the digital compass.



turn-slip-slide

Figure 4.11: The robot is able to move forward and backward, left and right, and can rotate in either direction. The robot's maximum speed is 2.5 centimeters per second. Its maximum rotational speed is 100 degrees per second.

4.2.1 Discovering related sensory subgroups.

The experimental results for this experiment parallel those of the previous experiment. The value of the distance metrics used in discovering groups of related sensors are shown in Figures 4.12 and 4.13.

As before, these distance metrics are used to identify, for each element s_i of feature $\mathbf{s}$, the set of elements that are similar to s_i. These sets of elements are shown in Figure 4.14. Treating these sets as equivalence classes and taking the transitive closure yields the groups of related elements shown in Figure 4.15.

4.2.2 A structural model of the sensory apparatus.

The group generator identified six groups of related sensors. Again, the largest group is a candidate for application of the image generator. The image generator's first step is to apply metric scaling to the distance metric d_1. The value of this distance metric is shown in Figure 4.16. Metric scaling produces the scree diagram of Figure 4.17a indicating that the sensory array is best modeled as a two-dimensional object. The set of positions produced by metric scaling (Figure 4.17b) is improved so that the distances in the resulting image more closely match distance metric d_1 (Figure 4.17c).

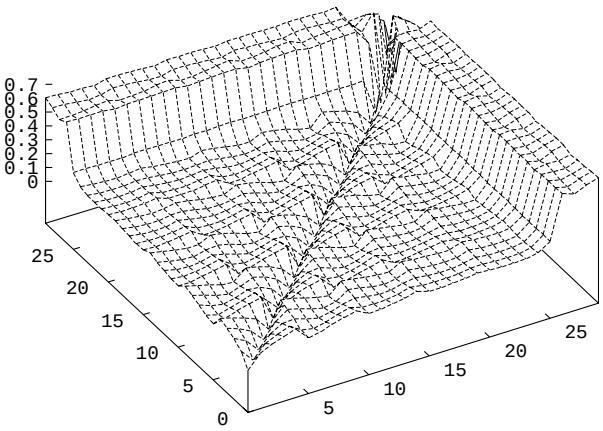


Figure 4.12: The value of distance metric d_1 after the robot has wandered randomly for five minutes.

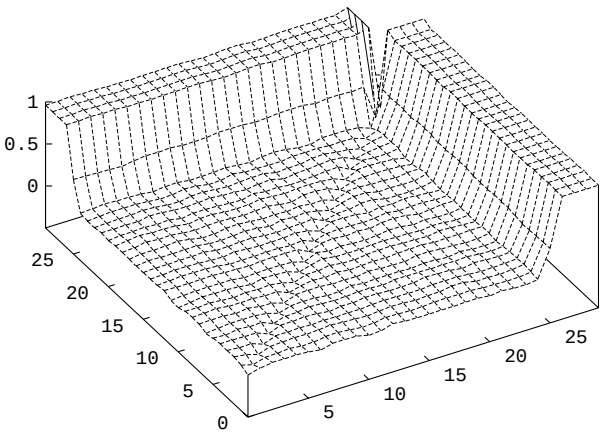


Figure 4.13: The value of distance metric d_2 after the robot has wandered randomly for five minutes.

(0 1 6 10 11 20 22 23)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 17 20 21 22 23)
(1 2 3 4 5 7 9 11 12 17)
(1 2 3 4 5 6 7 8 9 11 12 21 23)
(1 2 3 4 6 7 8 9 11 12 14 20 23)
(1 2 3 5 6 7 8 10 11 12 13 17 20 23)
(0 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 22 23)
(1 2 3 4 5 6 7 8 9 11 12 13 14 15 16 17 20 21 22 23)
(1 3 4 5 6 7 8 9 11 12 13 14 16 17 20)
(1 2 3 4 6 7 8 9 10 11 12 13 14 17 18 19 23 24)
(0 1 5 6 9 10 11 13 14 15 20 23)
(0 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16 17 18 19 20 21 22 23 24)
(1 2 3 4 5 6 7 8 9 12 13 14 15 16 17 20 21 22)
(1 5 6 7 8 9 10 11 12 13 14 15 16 17 18 20 21 23 24)
(1 4 6 7 8 9 10 11 12 13 14 16 17 18 19 23 24)
(6 7 10 11 12 13 15 16 17 20 21 22 23)
(6 7 8 11 12 13 14 15 16 17 20 21 22 23)
(1 2 5 6 7 8 9 11 12 13 14 15 16 17 18 20 21 22 23 24)
(6 9 11 13 14 17 18 19 20 23 24)
(6 9 11 14 18 19 24)
(0 1 4 5 6 7 8 10 11 12 13 15 16 17 18 20 22 23)
(1 3 7 11 12 13 15 16 17 21 22 23)
(0 1 6 7 11 12 15 16 17 20 21 22 23)
(0 1 3 4 5 6 7 9 10 11 13 14 15 16 17 18 20 21 22 23 24)
(9 11 13 14 17 18 19 23 24)
(25)
(26)
(27)
(28)
(29)

Figure 4.14: For each element of the raw sense vector, the set of similar elements is shown.

(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24)
(25) *battery voltage*
(26) *east*
(27) *north*
(28) *west*
(29) *south*

Figure 4.15: The photoreceptors have all been grouped together.

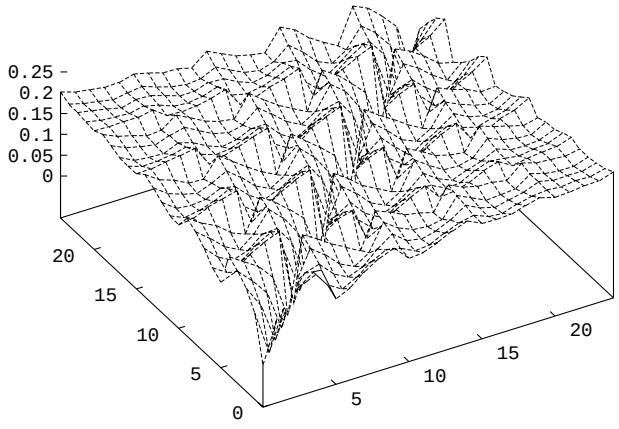


Figure 4.16: The metric d_1 is represented as a matrix. This figure shows the intersensor distances for all of the photoreceptors. It is this matrix that the image generator uses to construct the image feature that captures the structure of the retinal array.

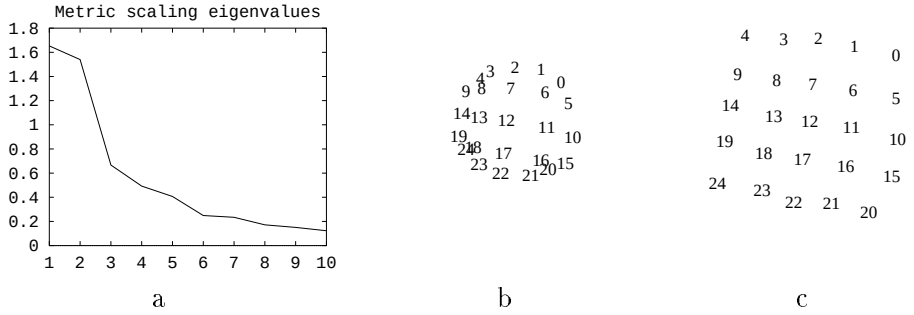


Figure 4.17: (a) The metric-scaling scree diagram for the group of photoreceptors indicates that the sensors are organized in a two-dimensional array. (b) The two-dimensional projection of the set of positions produced by metric scaling for the group of photoreceptors provides an initial approximation of the grid structure of the array of photoreceptors. (c) The final set of positions are produced using a constraint-satisfaction relaxation algorithm that uses the previous set of positions as initial values.

Chapter 5

Learning a Model of a Motor Apparatus

Recall that the critter's goal is to understand a robot's sensorimotor apparatus and environment well enough to navigate through that environment. The previous chapter presented a method for learning a model of the robot's sensory apparatus. This chapter presents a method for learning a model of the robot's motor apparatus. Modeling the motor apparatus means modeling the effects of the robot's control signals.

5.1 The learning method

The effects of the robot's control signals must be defined using the robot's sensory system, the critter's only source of information about its environment. One way to characterize a control signal's effect is in terms of motion. Once the image feature is defined, the **motion** feature operator (page 34) becomes applicable. The image feature makes it possible to define spatial attributes of the sensory input in terms of the locations of sensors in the image. With spatial attributes, it is possible to define spatial as well as temporal derivatives, so motion features can be defined, even without knowledge of the physical structure of the environment. The critter uses the new motion feature to analyze its motor apparatus using the following steps:

1. **Discretize the space of motor control vectors.** The robot's infinite space of motor control vectors is discretized into a finite set of representative vectors, $\{\mathbf{u}^i\}$.

2. **Compute average motion vector fields** (*amvf*'s). The critter repeatedly executes each representative motor control vector many times in different locations and measures the average value of the resulting motion feature. It is this average value that characterizes the effect of that control vector.

3. **Apply principal component analysis.** The set of computed *amvf*'s is a representation of the effects that the motor apparatus is capable of producing. Principal component analysis is used to decompose this set into a basis set of *principal eigenvectors*, a set of representative *amvf*'s from which all *amvf*'s may be produced by linear combination.

4. **Identify primitive actions.** Each principal eigenvector is matched against the *amvf*'s produced by the representative control vectors to find a control vector that produces that effect or its opposite. Such a motor control vector, if it exists, is identified as a *primitive action* and can be used to produce motion for one of the robot's degrees of freedom.

5. **Define a new abstract interface.** For each degree of freedom, a new control signal is defined that allows the critter to specify the amount of motion for that degree of freedom.

The result of the learning is a new abstract interface to the robot comprised of a new set of control signals, one per degree of freedom of the robot. The new interface hides the details of the motor apparatus. For example, whether a mobile robot's motor apparatus uses tank-style treads or a synchro-drive mechanism, the learned interface will present the critter with two control signals: one for rotating and one for advancing. These learned control signals will be used in Chapters 6 and 7 to further characterize the robot's motor apparatus using the *static* and *dynamic action models*.

Steps 1 through 5 are explained in detail in the next five sections using the robot and environment described in Chapter 1 as an example. Section 5.2 describes additional experiments.

5.1.1 Discretize the space of motor control vectors

The choice of the set of representative motor control vectors must satisfy two criteria: first, they must adequately cover the space of possible *motor control vectors* so that the space of possible *effects* (*amvf*'s) will be adequately represented. Second, the distribution of motor control vectors must be dense enough so that, given a desired effect (e.g., an *amvf* that corresponds to one of the robot's degrees of freedom), a motor control vector that produces that effect can be found.

The approach taken here is to use a set of unit motor control vectors uniformly distributed over the space of unit motor control vectors. (A unit vector has a magnitude of 1 where its magnitude is equal to the square root of the sum of squares of its components.) For two- and three-dimensional spaces of motor control vectors, respectively, 32 and 100 vectors have been found to be adequate. For the 2-D case, it is easy to find a set of vectors that are uniformly distributed on the unit circle. The i^{th} of n vectors has value $(\cos(2\pi i/n), \sin(2\pi i/n))$. For the 3-D case, finding a set of vectors that are uniformly distributed on the unit sphere is more complicated. The solution used here involves the same relaxation algorithm that was used by the image generator (Section 3.1.2). The vectors are constrained to lie on the unit sphere (i.e., to have magnitude 1), and the target distance between any pair of points is much larger than 2. The resulting configuration of vectors is analogous to a collection of electrons on a charged sphere — each vector is as far from its neighbors as possible. These vectors are used as the representative motor control vectors for sampling the continuous space of average motion vector fields. This method generalizes to any dimension.

5.1.2 Compute average motion vector fields

The learning method of this chapter is based on the observation that the representation of the space of motor control vectors is arbitrary whereas the space of effects is meaningful. The former depends on the details of the robot's motor apparatus whereas the latter is based on sensory perception. In the experiments described here, the effect of a motor control vector is the average motion vector field that it produces.

The hard work in defining *amvf*'s has already been done in Section 2.6.5 where the motion operator was defined. Using that operator, the definition of the *amvf* associated with the i^{th} representative motor control vector $\mathbf{u}^i$ is

$$amvf_i = \mu\left((\textbf{motion}\ x) \mid (=\ \mathbf{u}\ \mathbf{u}^i)\right)$$

where x is the image feature that has already been learned (Section 4.1.2) and $\mathbf{u}$ is the motor control vector used to control the motor apparatus. Recall that the μ operator computes the weighted average of a feature where the weight is given by that feature's strength. The expression above computes the average of the motion feature for all time steps during which $\mathbf{u}^i$ was taken. Examples are shown in Figure 5.1. These are obtained after the critter has wandered for 20 minutes using the exploration strategy of randomly choosing a representative motor control vector and executing it for one second (ten time steps).

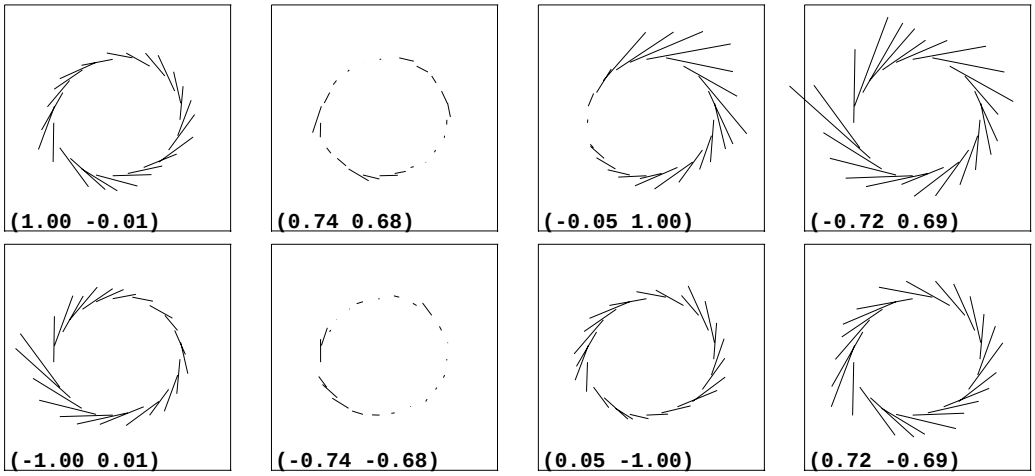


Figure 5.1: Examples of average motion vector fields (*amvf*'s) and their associated motor control vectors. An *amvf* associates an average local motion vector with each position in the image (see Figure 4.8). The examples in this section were all produced in an experiment involving the robot and environment described in Section 4.1.

5.1.3 Apply principal component analysis

The goal of this step is to find a basis set for the space of effects of the motor apparatus, i.e., a set of representative motion vector fields from which all of the motion vector fields may be produced by linear combination. This type of decomposition may be performed using principal component analysis. (See Mardia et al. (1979) for an introduction and Oja (1982) for a discussion of how a neural network can function as a principal component analyzer.)

Principal component analysis of a set of values for a variable $\mathbf{y}$ produces a set of orthogonal unit vectors $\{\mathbf{v}^i\}$, called *eigenvectors*, which may be viewed as a basis set for the variable $\mathbf{y}$. The i^{th} principal component of $\mathbf{y}$ is the dot product of $\mathbf{y}$ and eigenvector $\mathbf{v}^i$. In practice, $\mathbf{y}$ may be approximated as a linear combination of the first few eigenvectors while throwing the remaining ones away. The principal components are ordered according to their standard deviations. This means that the first eigenvector accounts for the most variance in the set of observed values for $\mathbf{y}$, and so forth. The application of principal component analysis to a two-dimensional variable is illustrated in Figure 5.2.

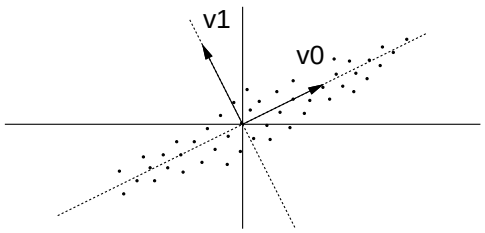


Figure 5.2: Principal component analysis applied to a two-dimensional random variable. The two basis vectors $\mathbf{v}^0$ and $\mathbf{v}^1$ are shown. Each dot represents one sample value of the variable.

Principal component analysis may be performed using a technique called *singular value decomposition* (Press et al., 1988) which identifies the eigenvectors and computes the standard deviation of each principal component. The relative magnitudes of the standard deviations tell how important each eigenvector is for the purposes of approximating the sample values for $\mathbf{y}$.

To perform the analysis, the sample values of $\mathbf{y}$ are organized as the rows of matrix Y. In the experiment with the robot with the ring of 23 working distance sensors, the sample values of $\mathbf{y}$ are the *amvf*'s produced by the 32 representative motor control vectors, each having 46 components.[1] The singular value decomposition of Y is

$$Y_{m\times n} = R_{m\times n} W_{n\times n} V_{n\times n}$$

where W is a diagonal matrix whose elements are the *singular values* of Y and the rows of V are the desired eigenvectors. Matrix R tells how to express each sample vector $\mathbf{y}$ as a linear combination of the eigenvectors. The singular value associated with an eigenvector gives a measure of that vector's importance in terms of explaining variation in the input set of *amvf*'s. Here, $m = 32$, the number of average motion vector fields, and $n = 46$, the number of components in each *amvf*. The equation below makes explicit the relationship among the four matrices. The decomposition orders the singular values according to magnitude, with the largest in the upper left corner.

$$\begin{bmatrix} \mathbf{y}^0 \\ \vdots \\ \mathbf{y}^{m-1} \end{bmatrix} = \begin{bmatrix} \mathbf{r}^0 \\ \vdots \\ \mathbf{r}^{m-1} \end{bmatrix} \begin{bmatrix} w_0 & & \\ & \ddots & \\ & & w_{n-1} \end{bmatrix} \begin{bmatrix} \mathbf{v}^0 \\ \vdots \\ \mathbf{v}^{n-1} \end{bmatrix}$$

This equation shows how each of the *amvf* vectors $\mathbf{y}^i$ is written as a linear combination of the eigenvectors in V:

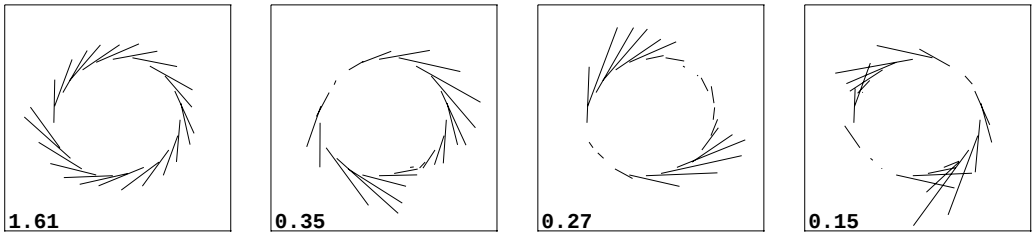
$$\mathbf{y}^i = \sum_{j=0}^{n-1} r_{ij} w_j \mathbf{v}^j$$

[1]The motion feature has 23 elements, one per element of the image on which it is based. Each element of the motion feature is a vector with two components for a total of 46.

Thus the row vectors of V form a basis set for the space of *amvf*'s. The *amvf*'s may be approximated by throwing away all but the most important basis vectors. Thus, for example, vector $\mathbf{y}^i$ may be approximated by

$$\mathbf{y}^i \approx r_{i0} w_0 \mathbf{v}^0 + r_{i1} w_1 \mathbf{v}^1,$$

keeping only the first two eigenvectors. The first four eigenvectors obtained in the experiment are shown in Figure 5.3.



$\mathbf{u}^0$= (-0.66 0.74) $\mathbf{u}^1$= (0.74 0.68)

Figure 5.3: The first four eigenvectors and the standard deviations of the associated principal components for the space of average motion vector fields. The first corresponds to a pure rotation motion and the second corresponds to a forward translation motion. (The top-left elements in these diagrams are associated with the robot's front sensor s_0). The robot's motor apparatus can produce the first two effects directly using the motor control vectors shown.

5.1.4 Identify primitive actions

In the previous step, principal component analysis was used to determine a basis set of effects for the motor apparatus, namely, the set of eigenvectors. The goal of this step is to discover which motor control vectors can be used to produce those effects. This is accomplished by matching the eigenvectors with the *amvf*'s of all of the representative motor control vectors. The matching involves computing the angle between each eigenvector and each *amvf*. An angle near zero indicates that the *amvf* is similar to the eigenvector. An angle near 180 degrees indicates that the *amvf* is similar to the opposite of the eigenvector. If any *amvf*'s match the i^{th} eigenvector to within 45 degrees, then motor control vector $\mathbf{u}^{i+}$ is defined to be the motor control vector whose *amvf* is most collinear with the i^{th} eigenvector and $\mathbf{u}^{i-}$ is defined to be the motor control vector whose *amvf* is most antilinear. The definitions of control laws (Chapter 7) assume that the robot's motor apparatus is linear, implying that $\mathbf{u}^{i+} = -\mathbf{u}^{i-}$. In the case that $\mathbf{u}^{i+} \approx -\mathbf{u}^{i-}$, they can be approximated by plus and minus $\mathbf{u}^i$, respectively, where $\mathbf{u}^i \stackrel{\text{def}}{=} \frac{1}{2}(\mathbf{u}^{i+} - \mathbf{u}^{i-})$. Subsequently, this will be used as the definition of the i^{th} primitive action. The values of $\mathbf{u}^i$ are given in Figure 5.3.

To conclude, here is the characterization of the robot's motor apparatus after this step: The motor apparatus has two degrees of freedom (from the perspective of the distance-sensor image). The motor control vector $\mathbf{u}^0$ can be used for motion for the first degree of freedom and the motor control vector $\mathbf{u}^1$ can be used for motion for the second degree of freedom.

5.1.5 Define a new abstract interface

The goal of this step is to define a new interface to the robot that abstracts away the details of the motor apparatus. For each of the robot's degrees of freedom, a new control signal is defined for producing motion along that degree of freedom. Negative values of the control signal will move the robot in the opposite direction. For the robot of the example, there will be two control signals, one for turning (left and right) and one for advancing (forward and backward). The effect of the control signals is defined by the following equation:

$$\mathbf{u} = u_0\mathbf{u}^0 + u_1\mathbf{u}^1$$

where u_0 and u_1 are the new control signals and $\mathbf{u}^0$ and $\mathbf{u}^1$ are the primitive actions corresponding to the first two principal eigenvectors.

The above definition of the control signals assumes that the motor apparatus can be approximated as linear, i.e., that the effect of the sum of two motor control vectors is equal to the sum of the effects of each motor control vector individually. This assumption holds for all of the experiments described in this dissertation. Even if the assumption does not hold, the results of the analysis of primitive actions is still useful. Instead of composing primitive actions simultaneously, they may be composed over time by alternately executing one primitive action then another. Suppose, for example, that the robot has four discrete actions corresponding to the four cardinal directions and that only one action may be taken at a time. The analysis of primitive actions will still correctly identify these four actions as primitive actions. However, the definition of the abstract interface would be more complicated. The critter would have to recognize that the motor apparatus is not linear and define an interface that decomposed complex actions such as "move northeast" into sequences of primitive actions. This extension is not currently implemented.

5.2 Additional experiments

The techniques for characterizing the effects of the motor apparatus have been applied to two additional simulated robots. The first is similar to the "tank-style" robot described in Section 5.1 except that the motor apparatus is now a synchro-drive robot base. The second robot is the roving eye.

5.2.1 The synchro-drive robot

The robot's environment is the same as that for the robot in Section 5.1. The synchro-drive gives the robot the same capabilities as the tank-style robot, but the control signals are interpreted differently. Instead of specifying how fast to move the left and right treads, the control signals specify how fast to turn and advance, respectively.

The critter's learning proceeds as in the experiment of Section 4.1. After learning the image feature, it applies the motion generator to that feature to obtain the motion feature. The critter explores by randomly selecting one of 32 representative motor control vectors, executing it for one second, and repeating. While exploring, it computes average values of the motion feature resulting

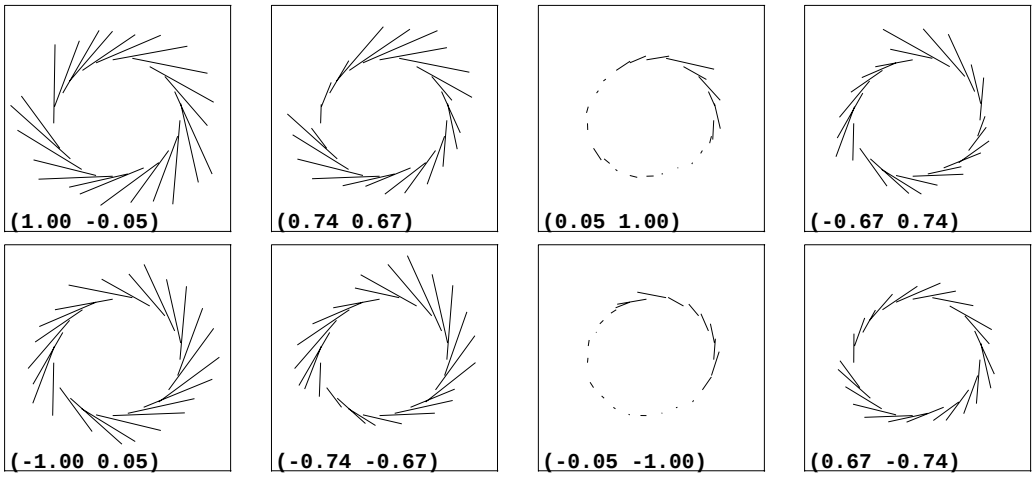
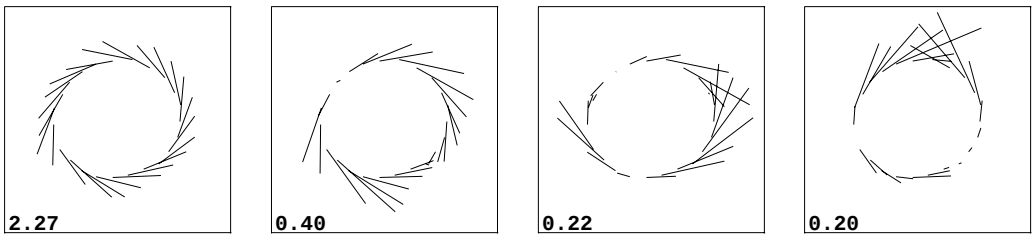


Figure 5.4: Examples of average motion vector fields and their associated motor control vectors for the synchro-drive robot with distance sensors.

from each of the 32 motor control vectors, producing a set of 32 *amvf*'s. Eight of these are shown in Figure 5.4.

Applying principal component analysis to the 32 *amvf*'s, the critter obtains a set of eigenvectors ordered by their importance in explaining the variance in the set of amvf's. The first four eigenvectors are shown in Figure 5.5. The first two eigenvectors correspond to rotation and advancing respectively. The values of the two primitive actions $\mathbf{u}^0$ and $\mathbf{u}^1$ are shown in the figure. The third and subsequent eigenvectors do not correspond to any motor control vectors.



$\mathbf{u}^0$= (-0.96 -0.14) $\mathbf{u}^1$= (0.05 1.00)

Figure 5.5: The first four eigenvectors and standard deviations for the synchro-drive robot with distance sensors. The first corresponds to a pure rotation motion and the second corresponds to a pure translation motion. The robot's motor apparatus can produce the first two effects directly using the motor control vectors shown.

5.2.2 The roving eye

The robot's environment and sensorimotor apparatus for this experiment are the same as in Section 4.2. The motor apparatus has three control signals, for rotating (clockwise and counterclockwise), for advancing (forward and backward), and for moving sideways (left and right).

Examples of average motion vector fields are shown in Figure 5.6. The first four eigenvectors for

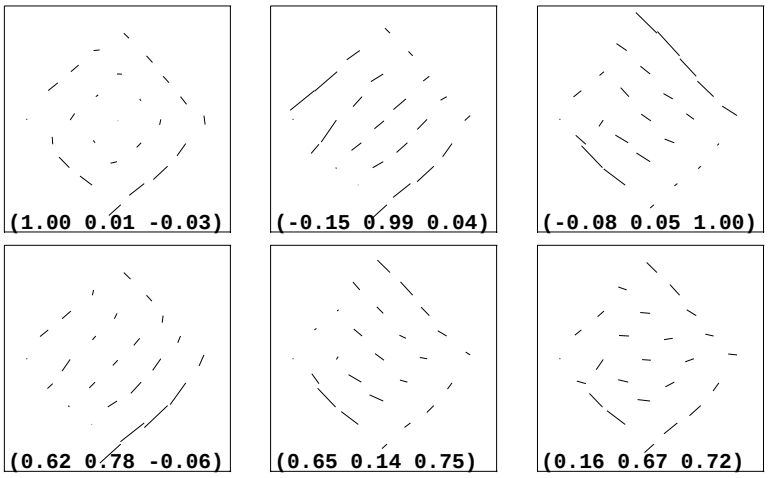


Figure 5.6: Examples of average motion vector fields and their associated motor control vectors for the roving eye.

this robot are shown in Figure 5.7. The first three eigenvectors correspond to forward motion, sideways motion, and rotation respectively. The values of the three primitive actions $\mathbf{u}^0$, $\mathbf{u}^1$, and $\mathbf{u}^2$ are shown in the figure. The fourth, and subsequent, eigenvectors do not correspond to any motor control vectors. To conclude, here is the characterization of this robot's motor apparatus:

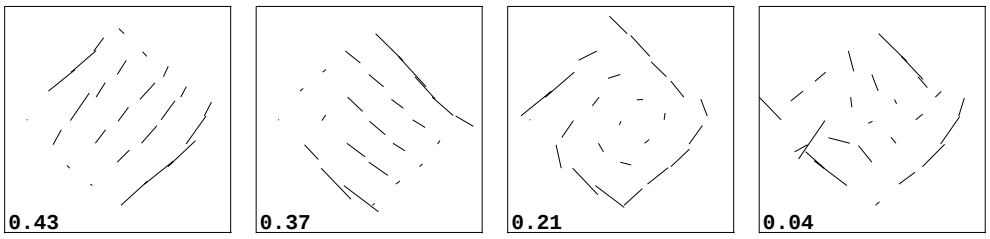


Figure 5.7: The first four eigenvectors and standard deviations for the roving eye.

The motor apparatus has three degrees of freedom (from the perspective of the distance-sensor image). The motor control vector $\mathbf{u}^0$ can be used for motion for the first degree of freedom, the motor control vector $\mathbf{u}^1$ can be used for motion for the second degree of freedom, and the motor control vector $\mathbf{u}^2$ can be used for motion in the third degree of freedom.

5.3 Conclusions

The abstract interface defined by this step abstracts away the details of the motor apparatus. The synchro-drive and tank-style robots demonstrate two different motor apparatuses with identical capabilities. The learned abstract interface, since it is based on sensory effects rather than motor

control signals, is the same for both: it abstracts away the details of the motor apparatus, providing a new set of control signals, one for each of the robot's degrees of freedom.

The learning method of this section is based on the observation that the representation of the space of motor control vectors is arbitrary whereas the space of effects is meaningful. The former depends on the details of the robot's motor apparatus whereas the latter is based on sensory perception. In the experiments described here, the sensor-based effect of a motor control vector is represented as an average motion vector field. An interesting line of research would be to explore alternative methods for representing the effects of motor control vectors. In place of the *amvf*, one might use the temporal derivative of the image feature (or of any feature, for that matter).

The learning methods described in this chapter build on the sensory image structure learned in the previous chapter. The result is a new abstract interface whose control signals will be used in Chapter 7 to define behaviors for navigation.

Chapter 6

Local State Variables

The *state* of a dynamic system (e.g., the robot in its environment) is a description of the system that captures everything that is important about the history of the system with respect to the goal of predicting the future behavior of the system. For a robot in a deterministic environment, the effects of future actions are completely determined by the state of the robot and environment. In a continuous world, the state of a robot is represented by a state vector whose elements are called state variables. For a mobile robot, these variables typically describe its position and orientation with respect to an absolute coordinate system.

For the critter that is trying to understand a robot's world, an important goal is to be able to determine the robot's current state at any time. This involves first defining a representation of the state. For a robot in a continuous world, this means defining an appropriate set of state variables and then computing their values. Since the critter does not have access to the robot's state variables, it uses features defined as functions on the robot's sensory input as *local state variables*. These features locally determine the state of the robot and will be used to define behaviors for exploration and navigation.

For example, consider the case of the mobile robot with distance sensors in the corner of a room. The position of the robot is locally determined by the minimum distances from the robot to the two corner walls. These distances serve as local state variables. They are local in the sense that they only exist as long as the robot is able to see the walls.

The definition of local state variable is given below:

> Let $\hat{\mathbf{u}}$ be the vector of control signals u_j. A scalar feature y_i is a *local state variable* if the effect of the control signals on y_i can be approximated locally by
>
> $$\dot{y_i} = \mathbf{m}_i \cdot \hat{\mathbf{u}} \;\; (= \sum_j m_{ij}\, u_j) \tag{6.1}$$
>
> where $\mathbf{m}_i$ is nonzero.

Determining whether a feature is a local state variable while learning the context-dependent value of $\mathbf{m}_i$ is the job of the static action model (Section 6.2).[1]

Local state variables are analogous to state variables in the following sense. If x is a state variable, then the constraint $\dot{x} = 0$ reduces the dimensionality of the robot's state space by one. If y is a *local* state variable, then the constraint $\dot{y} = 0$ reduces the dimensionality of the robot's

[1] The critter's assumptions about the robot and its environment imply that, when analyzing the effect of $\hat{\mathbf{u}}$ on y_i, the relationship $\dot{y_i} = \mathbf{m}_i\, \hat{\mathbf{u}}$ is the only one to consider. With more relaxed assumptions, other possibilities such as $y_i = \mathbf{m}_i\, \hat{\mathbf{u}}$ or $\ddot{y_i} = \mathbf{m}_i\, \hat{\mathbf{u}}$ must be considered.

motor control vector space by one.[2] In other words, the constraint reduces the robot's degrees of freedom by one. Since the critter does not have access to the robot's state space, it defines local state variables using its knowledge of motor control vector space to which it does have access.

An important feature of local state variables is that they are controllable: feature y_i may be moved to a target value y_i^* using a simple control law. This fact will be exploited in the definition of the homing behaviors (Section 7.3). The discovery of local state variables has two components: generating new features (Section 6.1), and testing each feature to see if it satisfies the definition of local state variable (Section 6.2).

6.1 Generating new features

If a sensory system does not directly provide useful features, it may be possible to generate features that are useful. The generate-and-test approach described in Chapter 3 is demonstrated in the following experiment using the tank-style mobile robot in which the critter learns new scalar features that are better candidates for local state variables than are the elements of the raw sense vector. The test portion of the method is performed when the static action model is learned.

6.1.1 A set of feature generators

The following feature generators are used to produce new features as candidates for local state variables.

- **splitter** takes a vector feature of length n and produces n scalar features.
- **vmin**, **vmax**, and **vsum** apply to vector features of length greater than 1. They provide three different ways to reduce a vector feature to a scalar feature.
- **group** and **image** identify useful structure in the sensory apparatus. Group and image features are not scalar features and thus will not be able to serve as local state variables, but they do serve as the basis for higher-level features that may turn out to be useful.
- **lmin** (local-min) and **lmax** (local-max) apply to image features. They produce *focused-image* features (image features in which elements have associated strengths as well as values) that focus attention on particular properties of an image, e.g., local minima or maxima.
- **tracker** applies to focused-image features and produces *image-element* features (single value-position pairs). From the focused image produced by the **lmin** generator, the **tracker** generator produces one image-element feature for each local minimum in the image. The tracker implements a form of focus of attention, abstracting away small changes in value and position of an image element in order to produce a feature which tracks an interesting property of the robot's environment such as the minimum distance to a nearby object.
- **val** extracts a scalar value feature from an image-element feature.

[2] If $\dot{y} = 0$, then by Equation 6.1, $\hat{\mathbf{u}}$ must lie in the subspace perpendicular to vector $\mathbf{m}_i$.

6.1.2 An experiment

In this experiment, the critter explores by randomly choosing unit motor control vectors and executing them for one second (10 time steps) each. Figure 6.1 gives a list of the scalar features produced by the generators. Each of these will be tested (Section 6.2) to see if it can serve as a local state variable.

s-vmin	*s-vmax*	s_0	s_1
s_2	s_3	s_4	s_5
s_6	s_7	s_8	s_9
s_{10}	s_{11}	s_{12}	s_{13}
s_{14}	s_{15}	s_{16}	s_{17}
s_{18}	s_{19}	s_{20}	s_{21}
s_{22}	s_{23}	s_{24}	s_{25}
s_{26}	s_{27}	s_{28}	*s-g0-vmin*
s-g0-vmax	*s-g0-im-lmin-tr-val*	*s-g0-im-lmax-tr-val*	

Figure 6.1: A list of generated scalar features. The names of the features reflect their derivation as a sequence of generators applied to the raw sensory feature s (g=group, im=image, tr=tracker).

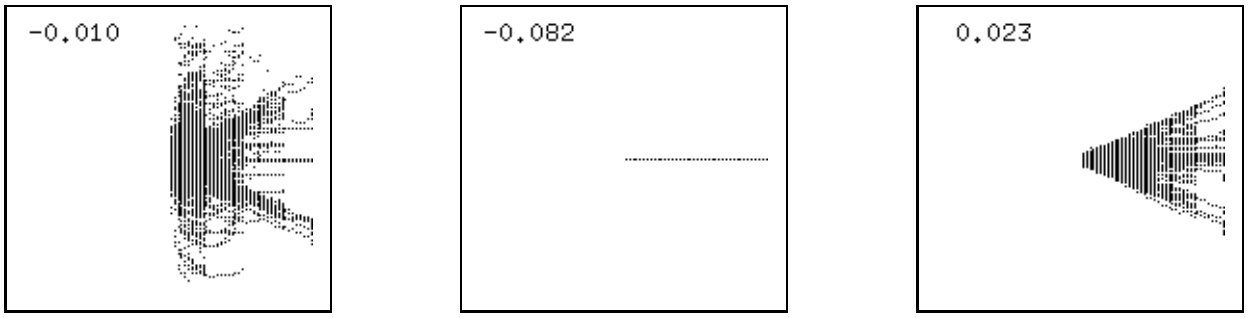
6.2 Testing features: The static action model

The purpose of the static action model is to predict the behavior of each scalar feature. The learning of the static action model for a feature proceeds in three steps. In the first step, the critter tries to predict the behavior of the feature without taking into account which primitive action is being used. If it fails, then it tries to predict the behavior of the feature as a function of the action being taken. If this fails for a primitive action, then the critter tries to predict the context-dependent effect of that action on the feature. If a feature is both action dependent and predictable, then it can serve as a local state variable. With the information contained in the static action model, it is a simple matter to define homing behaviors for moving the robot so that the local state variable moves toward its target value.

6.2.1 An action-independent model

The first step toward modeling the behavior of a feature y_i is to see if it is possible to predict its behavior independently of the motor control vector being used. The critter explores by repeatedly choosing a primitive action and executing it for one second (ten time steps). It analyzes the behavior of the feature using a correlator (Section 3.2.4). This produces a set of statistics based on the plot of the feature's value as a function of time (Figure 6.2). The coordinate for the horizontal axis is $\Delta t = t - t_0$ where t_0 is the last time the motor control vector changed. The vertical axis gives $\Delta y_i = y_i(t) - y_i(t_0)$.

Recall that the correlator produces the statistics m_i, r_i, and γ_i. The value of m_i is the slope of the line that best fits the set of $(\Delta t, \Delta y_i)$ points. The value of r_i is the correlation between variables



Δs_0 vs. Δt Δs_{24} vs. Δt *lmin* vs. Δt

Figure 6.2: Plots of Δy_i vs. Δt for three features. Whenever a new motor control vector is used, Δy_i and Δt are reset to 0. These are used to see if it is possible to predict the behavior of the feature independently of the motor control vector. Here, *lmin* is short for *s-g0-im-lmin-tr-val*. The numbers shown are the correlations between Δy_i and Δt.

Δy_i and Δt. The value of γ_i is the ratio of the standard deviations of Δy_i and Δt. It is a measure of how fast the feature changes as a function of time. A number of properties are defined in terms of these statistics. The feature is **constant** if $\gamma_i < 0.001$. It is **increasing** if $r_i > 0.6$ and **decreasing** if $r_i < -0.6$. It is **predictable** if any of these properties holds. Otherwise, it is unpredictable and the critter will try to predict the behavior of the feature using an action-dependent model.

For the running example, the features *s-vmin*, *s-vmax*, s_{20} (the broken distance sensor), s_{24} (the battery voltage), and *s-g0-vmax* are all diagnosed as **constant** and are thus not suitable for use as local state variables. The rest are candidates for the next step in the learning of the static action model.

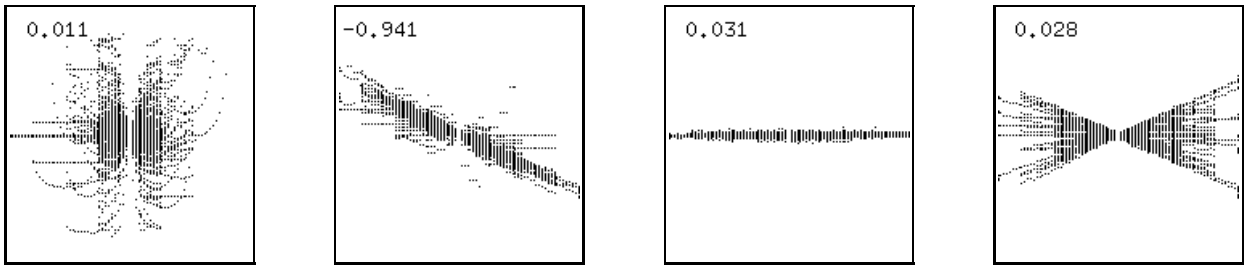
6.2.2 An action-dependent model

If the previous step failed to produce a model that predicts the behavior of a feature y_i, then the critter uses one correlator for each primitive action to analyze its effect on the feature. In this case, the correlator characterizes the relationship between $u_j \Delta t$ and Δy_i where Δt and Δy_i are defined as before. The critter continues to explore by randomly selecting primitive actions and executing them for a second at a time. It computes the statistics m_{ij} (the slope of the line that best fits the set of $(u_j \Delta t, \Delta y_i)$ points), r_{ij} (the correlation between $u_j \Delta t$ and Δy_i), and γ_{ij} (the ratio of the standard deviations of $u_j \Delta t$ and Δy_i). A feature is labeled **constant** for control signal u_j if $\gamma_{ij} < \gamma_i / 4$. The properties **increasing**, **decreasing**, and **predictable** for control signal u_j are defined as before. For each predictable feature-control signal pair, a rule of the form

$$\boxed{\dot{y_i} = m_{ij}\, u_j}$$

is added to the static action model. If a feature is predictable for all of the primitive actions, then the feature itself is predictable.

For the running example (Figure 6.3), all of the distance sensors are found to be unpredictable for the first primitive action (rotating). The effect of u_1 is to decrease features s_0, s_1, s_2, s_3, and s_{23}; to increase features s_9 through s_{14}. Its effect is unpredictable for features s_4–s_8, s_{15}–s_{19}, s_{21},



Δs_0 vs. $u_0 \Delta t$ Δs_0 vs. $u_1 \Delta t$ $\Delta lmin$ vs. $u_0 \Delta t$ $\Delta lmin$ vs. $u_1 \Delta t$

Figure 6.3: Plots of Δy_i vs. $u_j \Delta t$ for two features and two primitive actions. These are used to see if it is possible to predict the behavior of the feature as a function of the motor control vector.

and s_{22}. The discrete compass sensors s_{25} through s_{28} are unpredictable for u_0 and constant for u_1. The features *s-g0-vmin* and *s-g0-im-lmin-tr-val* (a.k.a. *lmin*) are constant for u_0 and unpredictable for u_1. Feature *s-g0-im-lmax-tr-val* is unpredictable for both primitive actions.

6.2.3 A context-dependent model

If u_j has an effect on y_i that is unpredictable, then the critter tries to find a partition of sensory space into a discrete set of contexts so that the relationship can be approximated by a linear equation for each context.[3] In general, a *context feature* z_{ij}, for local state variable y_i and control signal u_j, is an integer-valued feature that takes on a finite set of values. This set defines a partition of the robot's state space into a finite set of contexts defined by the predicates $z_{ij} = k$. One way to define a context feature is to first choose a feature x and divide its range of values into a finite set of intervals, $\{I_k\}$, where each interval defines its own context. The context feature is then defined by $z_{ij} = k$ iff $x \in I_k$. Using feature x to define a set of contexts is appropriate if the value of x is a good predictor of the effect of the control signal u_j on the feature y_i. To test the hypothesis that x is a good predictor for the effect of u_j on y_i, a correlator can be used to determine u_j's effect on y_i for each context defined by the predicate $z_{ij} = k$.

Testing each of a large set of features to see if they improve the predictability of a control signal's effect is expensive. The proposed alternative is to use heuristics to guide the search for relevant features to use in defining contexts. For example, it makes sense to first look at features that are closely related to the feature being analyzed, in the sense that they are close together in the tree of features produced by the generate-and-test process.

Currently, only one such heuristic is implemented: if a feature is based on the value of an element of an image, then use the position of that element to define the context. Since there is a discrete set of possible positions for an image-element feature, it is trivial to break the space of possible positions into a discrete set of contexts. For example, in the case of the *lmin* and *lmax* features, there are 23 possible positions and these can be used to break sensory space into a partition of 23 contexts each defined by the predicate $z_{ij} = k$ where z_{ij} is an integer feature whose value is between 0 and 22 and identifies the position associated with the local minimum or maximum.

[3] This approach is analogous to Drescher's marginal attribution (Drescher, 1991).

For each context $z_{ij} = k$, a correlator is used to try to predict the effect of u_j on y_i given that the robot is in that context. The critter continues to explore randomly while computing the statistics m_{ijk}, r_{ijk}, and γ_{ijk}. The properties **constant** , **increasing** , **decreasing** , and **predictable** are defined as before. For each predictable context, a rule of the form

$$\boxed{\dot{y_i} = m_{ijk}\, u_j, \text{ if } z_{ij} = k}$$

is added to the static action model. If m_{ijk} is 0, then the predicate $z_{ij} = k$ defines a "constant context" (which will be useful for defining path-following behaviors). If the primitive action's effect on the feature is predictable for every context, then the feature is predictable for that action.

For the running example, the only features with associated context features are *lmin* and *lmax*.

- *lmin* is already predictable (constant) for control signal u_0.
- The effect of u_1 on *lmin* is predictable for every context. Its effect is to decrease *lmin* for contexts 0–5 and 19–22, and to increase it for contexts 7–17. For contexts 6 and 18 (in which the robot's heading is parallel to the wall), *lmin* is constant (see Figure 6.4).
- The effect of u_0 on *lmax* is unpredictable for almost every context.
- The effect of u_1 is to decrease *lmax* for contexts 0–5 and 20–22 and to increase it for contexts 8–16. The effect is unpredictable for contexts 6, 7, 17, and 18.

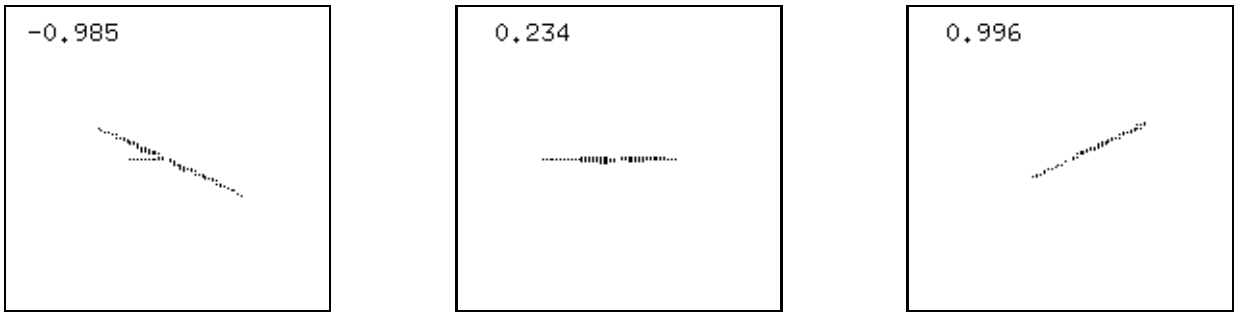


Figure 6.4: Example plots of Δy_i vs. $u_1 \Delta t$ for the *s-g0-im-lmin-tr-val* feature for three different contexts. These are used to see if it is possible to predict the behavior of the feature as a function of the motor control vector and the current context.

At this point the only feature that is both predictable and action-dependent (and is thus a local state variable) is *lmin*. Its behavior can be modeled by the equation $\dot{y_i} = m_{i1k}\, u_1$ where k is the current value of the context feature z_{ij}, which represents the location of the local minimum in the image feature. The feature *lmin* was produced by the **tracker** generator. This generator actually produces multiple *lmin* features, one for each local minimum in the input image feature. The number of local state variables will depend on the robot's location. There will be two local state variables in the neighborhood of a corner, three in the neighborhood of a T-intersection, but just one if only a single wall is within range.

Chapter 7

Learning Behaviors

The result of the critter's learning so far is an abstract interface that includes a model of the robot's sensorimotor apparatus. The model of the *sensory* apparatus is the description of its physical structure represented primarily by the positions of the components of the learned image feature. The model of the *motor* apparatus is the set of learned primitive actions that tells the critter how many degrees of freedom it has, and how to produce motion for each. In addition to the sensorimotor model, the critter has learned a set of local state variables.

The critter's ultimate goal is to abstract the continuous world of the robot to a *cognitive map* by which the world is viewed as a set of recognizable places with well-defined paths connecting them. This requires that the critter learn behaviors for moving the robot through its state space. Moreover, these behaviors must be repeatable in the sense that executing a behavior from a given initial state will always move the robot to the same final state.

This chapter discusses the problem in detail and shows how to learn a suitable set of *homing* and *path-following* behaviors, using the results of the preceding chapters, specifically, the set of primitive actions (Chapter 5) and the set of local state variables (Chapter 6). Chapter 9 will show how to use these learned behaviors to define the discrete abstraction.

Path-following behaviors are learned in three steps: (1) continuous error signals are defined; (2) behaviors are learned for minimizing the error signals; (3) behaviors are learned for moving while keeping the error signals near zero.

7.1 Defining error signals for control laws

The critter's approach to exploration, mapping, and navigation uses path-following behaviors in which the robot moves while maintaining an error signal near zero. An example of a path-following behavior based on an error signal involves a person walking down a corridor. The error signal is $e = (y^* - y)$ where y is the distance from the person to the right side of the corridor (left in Britain) and y^* is a constant that depends on the person, his mood, and the number of other people in the corridor. The error signal is used in a control law for moving along the corridor. If the error is positive, the person moves to the left (away from the wall) while walking; if it is negative, he moves to the right. The control law is efficient and repeatable. By using the control law, the person reliably follows an efficient path from one end of the corridor to the other.

A second example involves CADR (for "constant angle, decreasing range"), a rule familiar to missile designers. If the angle to an airplane relative to a missile is constant and the distance between them is decreasing, then the missile's path will reliably lead it to the airplane (a fact worth remembering when trying to beat a train to an intersection). In this case, the error signal

is $e = \theta^* - \theta$ where θ is the angle to the airplane relative to the missile and θ^* is a target angle between $-90°$ and $90°$.

In these examples, y and θ are local state variables. The critter's approach to defining path-following behaviors is to first define error signals of the form $e = y^* - y$ for each local state variable y.[1] The subsequent sections tell how to use these error signals to define path-following behaviors. But first, a definition of "behavior" will be given in the next section that will make it clear exactly what is required when defining a new behavior.

7.2 Anatomy of a behavior

So far in this chapter, behaviors and control laws have been treated as if they were synonymous. In fact, a control law is just one component of a behavior. This section gives a definition of *behavior* and shows exactly what will be required when defining path-following behaviors. A behavior is an object with four components, called *output*, *app*, *done*, and *init*.

The *output* component is a function that returns a vector of motor control signals. The *app* component is a scalar function whose value indicates whether the behavior is currently applicable. The value of this function may be zero (indicating that the behavior is not applicable) or one (indicating that the behavior is applicable) or some number in between (indicating a certainty less than 100% that the behavior is applicable). The *done* signal is a Boolean function that tells when the behavior has finished. For example, a behavior used to minimize an error signal will be done when the error is close enough to zero. The *init* signal is an input signal that tells the behavior to initialize itself (in case it has any internal state information that needs to be reset).

The four components just described are required for any behavior. They define the "public interface" to a behavior. In addition to these, the behavior may have a set of inputs (e.g., sensory features on which the *output*, *app*, and *done* signals are based) and a set of sub-behaviors. The general structure of a behavior is shown in Figure 7.1.

The definition of behavior was designed to support the definition of hierarchies of behaviors. Figure 7.2 shows how a complex behavior is built using a **seq** behavior and two sub-behaviors. Any behavior, since it must honor the protocol shown in Figure 7.1, may be used as a sub-behavior. At any given time, the critter will have one behavior identified as the top-level behavior. The output of this behavior is sent directly to the robot's motor apparatus. The critter behaves by continuously calling the top-level behavior's output function and sending the result to the motor apparatus.

To summarize, a behavior is more than a control law. When defining a behavior, it will be necessary to define the *app*, *done*, and *init* functions as well as the *output* function (the control law itself).

7.3 Learning homing behaviors

The purpose of a *homing behavior* is to move an error signal toward zero so that path-following behaviors based on that error signal will be applicable. While it would be possible to

[1] Choosing an optimal target value y^* for a feature y is beyond the scope of this dissertation. The implemented critter chooses a value equal to half the feature's maximum value.

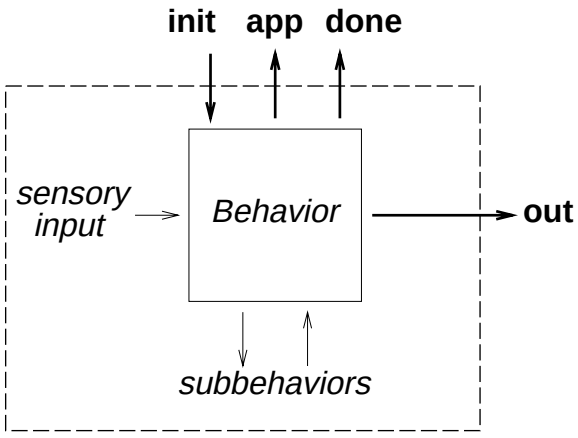


Figure 7.1: A schematic diagram of a behavior. The four signals on the outside of the dotted rectangle define the "public interface" to the behavior object. Every behavior must provide these four functions.

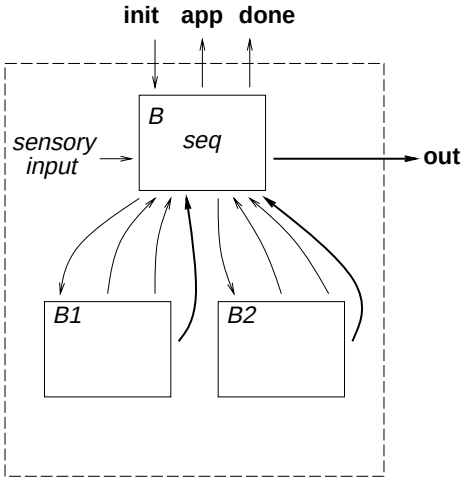


Figure 7.2: An example of a complex behavior. When the **seq** behavior is initialized (by calling its *init* function), it initializes its first sub-behavior which becomes its current sub-behavior. When the current sub-behavior is done, the next sub-behavior becomes current. When the last sub-behavior has had its turn and is done, the **seq** (for "sequence") behavior is done. When the **seq** behavior's *output* (or *app*) function is called, it calls the *output* (or *app*) function of its current sub-behavior and returns what it returns.

use reinforcement-learning methods to learn a homing behavior given an error signal (e.g., Pierce & Kuipers, 1991; Lin & Hanson, 1993), most of the relevant learning has already been done. The homing behavior can be defined as an instance of the generic, domain-independent control law in Figure 7.3, drawing on the knowledge in the static action model.

For each local state variable y_i and control signal u_j, a homing behavior is defined for reducing the error $e = y_i^* - y_i$. It is applicable in every context $z_{ij} = k$ for which the static action model includes a rule of the form $\dot{y}_i = m_{ijk} u_j$ where m_{ijk} is nonzero. It is done when the error is zero. Its output is given by a simple control law. The definition is based on the partition of sensory space used by the static action model to characterize the effects of u_j on y_i. This partition is described by the set of contexts $\{k\}$. The components of the homing behavior (*app*, *output*, and *done*) are defined for each possible context k (Figure 7.3). A homing behavior that the critter learns for the

For each context $z_{ij} = k$,

$$\begin{aligned} app(k) &= \max\{0,\ 2|r_{ijk}| - 1\} \\ output(k) &= u_{ijk}\, \mathbf{u}^j \\ done &\equiv \frac{|y_i^* - y_i|}{y_i^*} < 0.1 \end{aligned}$$

where

$$\begin{aligned} u_{ijk} &= \frac{2\zeta\omega}{m_{ijk}}\, e_i + \frac{\omega^2}{m_{ijk}} \int e_i\, dt \\ e_i &= y_i^* - y_i. \end{aligned}$$

Figure 7.3: A homing behavior is defined for each local state variable y_i and for each primitive action $\mathbf{u}^j$ to achieve the goal $y_i = y_i^*$. The applicability and output are defined as functions of the current context as defined by the context feature z_{ij}. The applicability has a maximum value of 1.0 if the correlation between u_j and $\dot{y}_i$ has a magnitude of 1.0 and a minimum value of zero if the correlation has a magnitude of 0.5 or less. The output is given by a proportional-integral (PI) control law with parameters $\zeta = 1.0$, $\omega = 0.05$ (see Kuo 1982) that minimizes the difference between y_i and y_i^*. The behavior is done when this difference is close to zero. The *init* function resets the value of the integral of the error to zero.

mobile robot is illustrated in Figure 7.4.

7.4 Learning path-following behaviors

The previous section presented a method for learning homing behaviors that minimize a given error signal. In this section, a method is presented for moving while minimizing the error signal. The result is a path-following behavior. Learning a path-following behavior involves two steps: 1) learning how to move in the general direction that keeps the error near zero and 2) learning the necessary feedback for error correction to avoid straying off the path defined by the minimum of the error signal.

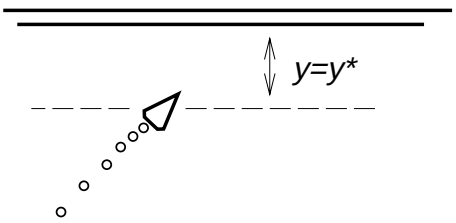


Figure 7.4: An example of a homing behavior for the mobile robot with distance sensors and tank-style motor apparatus. The critter's static action model predicts that in this context the second primitive action $\mathbf{u}^1$ will decrease the value of local state variable y_i. This information is used in the definition of a homing behavior that is (a) applicable in this context, (b) uses primitive action $\mathbf{u}^1$ to move the robot so as to minimize the error $e = y_i^* - y_i$, and (c) is done when $y_i \approx y_i^*$.

For the corridor-following example, motion along the path is produced by walking. Error correction involves moving away from the wall when it gets too close or moving toward it if it gets too far away. For the missile example, the motion is provided by the missile's jet engine; error correction is provided by its fins which turn the missile so as to maintain the desired angle of the target relative to the missile.

The critter's solution for defining path-following behaviors is to use its static action model to determine which primitive action to use to provide motion along a path and to learn and use a *dynamic action model* to tell how to use the remaining primitive actions to provide error correction.

7.4.1 Learning open-loop path-following behaviors

The static action model does not give the critter enough information to define *closed-loop* path-following behaviors with error correction, but it does give the critter enough information to define *open-loop* path-following behaviors.[2] An open-loop path-following behavior lacks error correction but is useful for learning the dynamic action model which is in turn useful for defining path-following behaviors with error correction. Recall that the static action model identifies constant contexts $z_{ij} = k$ in which primitive action $\mathbf{u}^j$ has no effect on local state variable y_i.

For each local state variable y_i and primitive action $\mathbf{u}^j$, for each constant context $z_{ij} = k$, two open-loop behaviors are defined, one for each direction of motion. The behaviors' outputs are given by

$$\mathbf{u} = \mathbf{u}^\beta + \sum_{\delta \neq j} u_\delta \, \mathbf{u}^\delta$$

where $\mathbf{u}^\beta = \pm \mathbf{u}^j$ and $|u_\delta| \ll 1$. The u_δ components will be used in learning the dynamic action model. The purpose of an open-loop path-following behavior is to allow the critter to learn the effects of the *orthogonal* control signals on the feature while motor control vector $\mathbf{u}^\beta$ is used.[3] With this knowledge, it will be possible to use the other control signals for error correction. The definition

[2] In a closed-loop control law, an error signal is used as feedback to determine a motor control vector that minimizes that error.

[3] The primitive actions are orthogonal to each other in the sense that their *amvf*'s are orthogonal to each other (see Section 5.1.3).

of open-loop path-following behaviors is summarized in Figure 7.5. A behavior is done when the

$$
\begin{aligned}
app &\equiv \frac{|y_i^* - y_i|}{y_i^*} < 0.1 \ \wedge \ z_{ij} = k \\
output &= \mathbf{u}^\beta + \sum_{\delta \neq j} u_\delta \mathbf{u}^\delta \\
done &\equiv \frac{|y_i^* - y_i|}{y_i^*} > 0.4 \\
&\quad \vee \text{(a new behavior becomes applicable)}
\end{aligned}
$$

Figure 7.5: An open-loop path-following behavior is defined for each local state variable y_i, for each primitive action (or opposite) $\mathbf{u}^\beta$, and for each constant context $z_{ij} = k$. The predicate $z_{ij} = k$ defines a constant context if it implies that $\mathbf{u}^\beta$ maintains y_i constant according to the static action model. The behavior is applicable when the error signal $y_i^* - y_i$ is small. The output has two components: a base motor control vector and a small orthogonal component. During the learning of the dynamic action model, the orthogonal component changes every 3 seconds. Only one of the u_δ's is nonzero at a time. The behavior is done when the error signal is too large or a new behavior becomes applicable.

robot strays too far off the path or when a new behavior becomes applicable indicating that the critter has a choice to make: to continue the current behavior or start a new one.

For the mobile robot of the running example, there is an open-loop path-following behavior based on $\mathbf{u}^0$ (for turning) for each local state variable y_i (see Figure 7.6a). It is applicable whenever

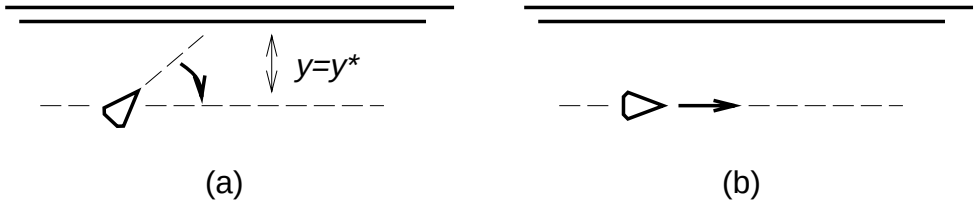


Figure 7.6: Two examples of open-loop path-following behaviors. (a) A behavior based on $\mathbf{u}^0$ (for turning) and constraint $y_i = y_i^*$ is applicable whenever $y_i = y_i^*$ since $\mathbf{u}^0$ never changes the value of y_i. (b) A behavior based on primitive action $\mathbf{u}^1$ (advancing) and constraint $y_i = y_i^*$ is applicable whenever $y_i = y_i^*$ and the robot's heading is parallel to the wall on its left (i.e., $z_{ij} = 18$) since in this context $\mathbf{u}^1$ keeps the error $e = y_i^* - y_i$ near zero.

$y_i = y_i^*$ since, according to the static action model, turning has no effect on y_i. There is also an open-loop path-following behavior based on $\mathbf{u}^1$ (for advancing) for each feature y_i (see Figure 7.6b). It is applicable when the robot is facing parallel to the object being detected by y_i (that is, when context feature z_{ij} has value 6 or 18). Figure 7.10b shows a trace of the behavior of the robot that results as the critter uses its learned open-loop path-following behaviors to explore the robot's environment.

7.4.2 The dynamic action model

The *static* action model predicts the context-dependent effects of a control signal on the local state variables. The *dynamic* action model predicts the context-dependent effects of control signals on the local state variables while an open-loop path-following behavior is being executed.

The dynamic action model tells, for each open-loop path-following behavior, the effect of each orthogonal action (each primitive action other than the path-following behavior's base action), on the local state variable that is used in the definition of the path-following behavior's error signal. To learn the dynamic action model, an exploration behavior is used that randomly chooses applicable homing and open-loop path-following behaviors. A behavior runs until it is no longer applicable, or a new path-following behavior becomes applicable. Linear regression is used to learn the relationships between the orthogonal actions u_δ and the features y_i in the context of running the open-loop path-following behavior based on feature y_i, motor control vector $\mathbf{u}^\beta = \pm\mathbf{u}^j$, and context $z_{ij} = k$. While it is running, linear regressors test the hypotheses $\dot{y}_i = m_{ijk\delta 1}\, u_\delta$ and $\ddot{y}_i = m_{ijk\delta 2}\, u_\delta$ by computing the correlations $r_{ijk\delta n}$ between u_δ and $y_i^{(n)}$. If $r_{ijk\delta 1} > r_{ijk\delta 2}$ and $|r_{ijk\delta 1}| > 0.6$, then the rule

$$\boxed{\dot{y}_i = m_{ijk\delta 1}\, u_\delta, \quad \text{if} \;\; z_{ij} = k \;\wedge\; \mathbf{u} = \pm\mathbf{u}^j + u_\delta \mathbf{u}^\delta}$$

is added to the dynamic action model. Otherwise, if $|r_{ijk\delta 2}| > 0.6$, then the rule

$$\boxed{\ddot{y}_i = m_{ijk\delta 2}\, u_\delta \quad \text{if} \;\; z_{ij} = k \;\wedge\; \mathbf{u} = \pm\mathbf{u}^j + u_\delta \mathbf{u}^\delta}$$

is added to the dynamic action model. Otherwise, the relationship between u_δ and y_i is either zero or unpredictable.[4]

Suppose that the mobile robot of the running experiment has a wall to its left and that its heading is parallel to the wall (Figure 7.6b). In this context, primitive action $\mathbf{u}^1$ (advancing) will maintain the distance to the wall, y_i, constant ($m_{ijk} = 0$). Therefore, the open-loop path-following behavior based on $\mathbf{u}^1$ and y_i will be applicable. While executing this behavior, the effects of other control signals (i.e., u_0) can be diagnosed. In this example, u_0 affects the second derivative of the feature: $\ddot{y}_i = m_{i1k0,2}\, u_0$. This is because turning changes the robot's direction of motion relative to the wall and this direction determines how fast the robot moves toward or away from the wall as it advances. Examples of the linear regressors used to learn the dynamic action model for the robot of the running example are illustrated in Figure 7.7.[5]According to the dynamic action model, $\mathbf{u}^0$ has a predictable effect on y_i while any of the open-loop behaviors based on $\mathbf{u}^1$ is executing. For the open-loop path-following behaviors based on $\mathbf{u}^0$, the effect of $\mathbf{u}^1$ on y_i is unpredictable.

[4]For the dynamic action model, it is necessary to consider both first and second derivatives of the features. Informally, this is because $\mathbf{u}^\delta$ may affect the derivative of m_{ij} in the equation $\dot{y}_i = m_{ij}\, u_j$, that is, $\dot{m}_{ij} = m_{j\delta}\, u_\delta$. Together, these give $\ddot{y}_i = \dot{m}_{ij}\, u_j = m_{j\delta}\, u_\delta\, u_j = m_{ij\delta 2}\, u_\delta$, using the product rule and the fact that u_j is constant for a path-following behavior.

[5]The linear regressors operate on filtered versions of y_i and u_j to remove noise that would otherwise hide the relationship between the signals. The signals are filtered using a moving average taken over several seconds.

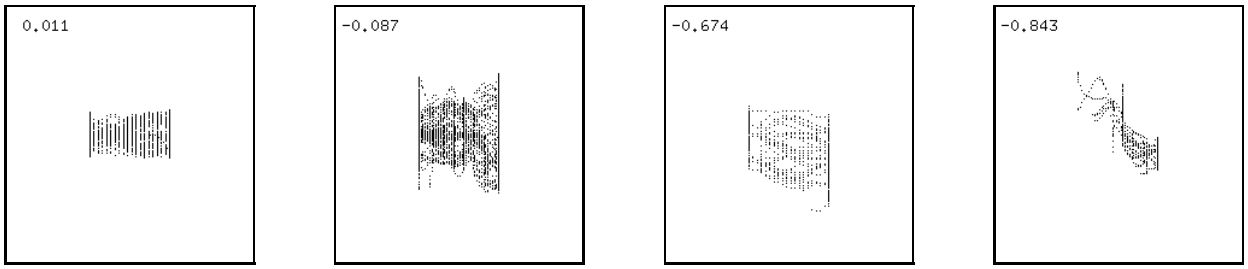


Figure 7.7: Plots illustrating the relationships measured by the linear regressors used in learning the dynamic action model. The first two plots show the effect of $\mathbf{u}^1$ (advancing) on $\dot{y_0}$ and $\ddot{y_0}$ respectively while an open-loop path-following behavior based on $\mathbf{u}^0$ is executed. Here y_0 is one of the local state variables (instances of *lmin*) produced by the tracker generator. The second two plots show the effect of $\mathbf{u}^0$ (turning) on $\dot{y_0}$ and $\ddot{y_0}$ respectively while an open-loop path-following behavior based on $\mathbf{u}^1$ is executed in context $z_{0,1} = 6$. This is the context in which the robot heading parallel to a wall on its right. The labels under the plots give the values of i, j, k, δ, and n, where n is the number of the derivative of y_i being tested.

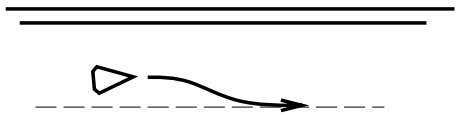


Figure 7.8: Defining closed-loop path-following behaviors. The critter uses the dynamic action model to add error correction to an open-loop path-following behavior in order to obtain a closed-loop path-following behavior. In this example, a small turning motion is used to keep the robot on the path as it advances.

7.4.3 Learning closed-loop path-following behaviors

The final step in learning path-following behaviors is to add error correction to the open-loop path-following behaviors in order to define closed-loop path-following behaviors. A *closed-loop* behavior is one that receives feedback from the environment in the form of an error signal which it uses to modify its motor control signals so as to minimize the error. Consider again the case where the robot is facing parallel to a wall on its left. In this context, the critter knows, because of its static action model, that primitive action $\mathbf{u}^1$ will leave feature y_i (the distance to the wall) constant. Moreover, the critter knows, because of its dynamic action model, how control signal u_0 (turning) affects y_i while $\mathbf{u}^1$ is being taken. Together, this information is sufficient to define a closed-loop path-following behavior that robustly moves the robot along the wall. If y_i goes below its target value (i.e., if the robot gets too close to the wall), then the critter knows to increase the value of u_0 (i.e., to turn right as shown in Figure 7.8). Because of the error correction implemented using control signal u_0, the path-following behavior is robust in the face of noise in the sensorimotor apparatus, small perturbations in the shape of the wall, and even inaccuracies in the action models themselves.

A closed-loop path-following behavior is defined using the generic template in Figure 7.9 for each constraint $\mathbf{y} = \mathbf{y}^*$, for each primitive action or opposite $\mathbf{u}^\beta = \pm\mathbf{u}^j$, and for each constant context $\mathbf{z} = \mathbf{k}$. The predicate $\mathbf{z} = \mathbf{k}$ (where $\mathbf{z}$ is a vector of context features z_{ij} and $\mathbf{k}$ is a vector of context values k_i) defines a constant context if for each $z_{ij} \in \mathbf{z}$ and $k_i \in \mathbf{k}$, $z_{ij} = k_i$ defines

a constant context for y_i and $\mathbf{u}^j$ according to the static action model. The variable $r_{ijk\delta n}$ is the correlation between u_δ and $y_i^{(n)}$ while motor control vector $\mathbf{u}^\beta$ is used in context k. The behavior is applicable when all of the components of $\mathbf{y}$ are near their target values (i.e., $\mathbf{y} \approx \mathbf{y}^*$) and when $\mathbf{z} = \mathbf{k}$ indicating that the static action model predicts that motor control vector $\mathbf{u}^\beta$ will keep the error vector $\mathbf{y}^* - \mathbf{y}$ near zero. The behavior is done when a new path-following behavior becomes applicable indicating the the critter now has a choice — to continue the current path-following behavior or to choose a new one.

$$
\begin{aligned}
app &\equiv \forall y_i \in \mathbf{y} : \left(\frac{|y_i^* - y_i|}{y_i^*} < 0.1 \right) \wedge \forall z_{ij} \in \mathbf{z} : (z_{ij} = k_i) \\
output &= \mathbf{u}^\beta + \sum_{\delta \neq j} u_\delta \, \mathbf{u}^\delta \\
done &= \exists y_i \in \mathbf{y} : \left(\frac{|y_i^* - y_i|}{y_i^*} > 0.4 \right) \\
&\quad \vee \text{(a new behavior becomes applicable)}
\end{aligned}
$$

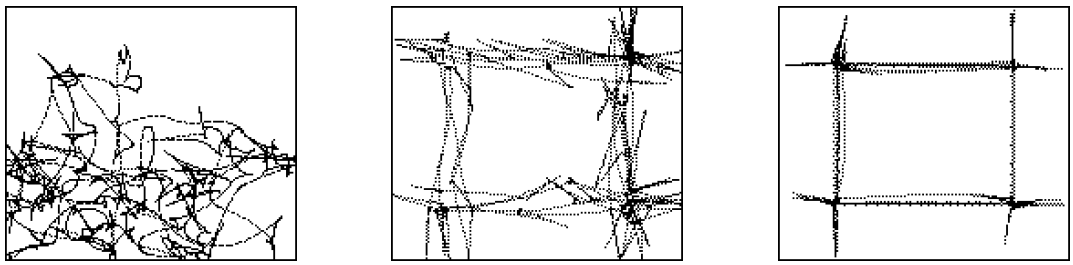
where

$$
\begin{aligned}
u_\delta &= \sum_{y_i \in \mathbf{y}} u_{\delta i} \\
u_{\delta i} &= \frac{2\zeta\omega}{m_{ijk\delta 1}} e_i + \frac{\omega^2}{m_{ijk\delta 1}} \int e_i \, dt \qquad \text{if } |r_{ijk\delta 1}| \geq |r_{ijk\delta 2}|, 0.6 \\
u_{\delta i} &= \frac{\omega^2}{m_{ijk\delta 2}} e_i + \frac{2\zeta\omega}{m_{ijk\delta 2}} \dot{e_i} \qquad \text{if } |r_{ijk\delta 2}| > |r_{ijk\delta 1}|, 0.6 \\
u_{\delta i} &= 0, \quad \text{otherwise} \\
e_i &= y_i^* - y_i .
\end{aligned}
$$

Figure 7.9: Definition of a closed-loop path-following behavior. Here, $\mathbf{y}$ is a vector of local state variables y_i; $\mathbf{y}^*$ is the corresponding vector of target values; $\mathbf{u}^\beta = \pm\mathbf{u}^j$ is one of the primitive actions or their opposites; $\mathbf{z}$ is a vector of context features z_i, one for each local state variable y_i; and $\mathbf{k}$ is the corresponding vector of context values k_i. The equation $\mathbf{z} = \mathbf{k}$ defines a context in which $\mathbf{u}^\beta$ maintains $\mathbf{y}$ constant according to the static action model. The values of $m_{ijk\delta n}$ and $r_{ijk\delta n}$ are taken from the dynamic action model. Simple PI and PD (proportional-derivative) controllers are used depending on whether the primary effect of $\mathbf{u}^\delta$ is on $\dot{y_i}$ or $\ddot{y_i}$, respectively. Again, ζ=1.0, ω=0.05.

For the example robot, the set of path-following behaviors contains behaviors for turning in place as well as for following walls. For the behavior based on $\mathbf{u}^1$ (advancing), the effect of the orthogonal primitive action $\mathbf{u}^0$ on the local state variables is predictable and thus it can be used for error correction. For the behaviors based on $\mathbf{u}^0$(turning), no error correction is used since the effect of $\mathbf{u}^1$ is unpredictable.[6] Figure 7.10 shows the behavior of the robot at three different stages

[6] An extension to the current implementation is to learn a context-dependent dynamic action model for each open-loop path-following behavior. In this way the effect of $\mathbf{u}^1$ could become predictable and the action could be used for



(a) (b) (c)

Figure 7.10: Exploring a simple world at three levels of competence. (a) The robot wanders randomly while learning a model of its sensorimotor apparatus. (b) The robot explores by randomly choosing applicable homing and open-loop path-following behaviors based on the static action model while learning the dynamic action model. (c) The robot explores by randomly choosing applicable homing and closed-loop path-following behaviors based on the dynamic action model.

as the critter learns the set of path-following behaviors. In Chapter 9, the path-following behaviors learned in this chapter will be used as the basis for an exploration and mapping strategy that allows the critter to develop a discrete abstraction of the robot's continuous world.

7.5 Discussion

This chapter presents a general solution to the problem of learning path-following behaviors. The general solution is comprised of three steps: 1) learning a set of error signals, 2) learning a behavior to minimize each error signal, and 3) learning behaviors for motion while keeping the error signals near zero.

This chapter also gives a particular solution to the problem of learning path-following behaviors that exploits the knowledge already learned about the robot's sensorimotor apparatus, specifically the set of primitive actions and the set of local state variables. If the critter had direct access to the robot's state variables, it could use them to define control laws for path-following. Since it does not, it uses *local* state variables instead. These are scalar features whose derivatives can be approximated as a linear function of the robot's motor control signals. They are useful for defining behaviors because the critter knows how to use the motor control signals to control them.

7.5.1 Handling context dependence

The approach taken by the critter is to use a knowledge of control theory to allow it to directly define local control strategies once it has learned a set of action models characterizing the effects of the motor control signals on the local state variables. An alternative approach to learning local control strategies is to use reinforcement learning (e.g., Lin & Hanson 1993). This approach taken here has several advantages over reinforcement learning. First, it is faster since no additional learning is necessary once the action models are learned. Second, it handles context dependence at the feature level instead of the behavior level. This fact is important because the critter can

error correction in a context-dependent control law.

learn the effects of motor control signals on multiple features simultaneously, whereas it is only possible to train one behavior at a time. Consider three example behaviors: following the right wall, following the left wall, and following a corridor with walls on both sides. The reinforcement-learning approach would require that the robot be trained separately for the three different contexts. The action-model approach, on the other hand, allows the critter to immediately define all three behaviors once the action models are learned since they take into account the context dependence of the effects of the motor control signals on the distance-to-wall features.

In the Chapter 9, the path-following behaviors learned in this chapter will be used as the basis for an exploration and mapping strategy that allows the critter to develop a discrete abstraction of the robot's continuous world.

Chapter 8

Additional Experiments

The previous chapters have demonstrated a set of learning methods that a critter may use to learn the sensorimotor and control levels of the spatial semantic hierarchy. The purpose of this chapter is to describe a number of experiments (in addition to those described in the previous chapters) that demonstrate the generality and some limitations of the methods for learning the sensorimotor and control levels.

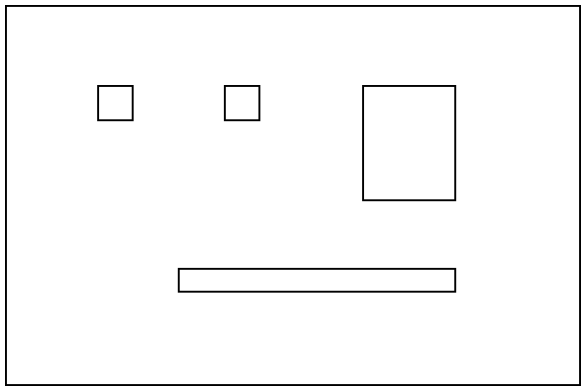
8.1 Overview

The learning methods will first be demonstrated for the mobile robot in a cluttered room. Then, to demonstrate that the learned model of the sensorimotor apparatus does not apply only to the particular environment in which the model was learned, the critter will be transferred to a new, T-shaped environment after its control-level learning has been erased. Here it will re-learn the control level and demonstrate a set of learned path-following behaviors. Finally, to demonstrate that the learning of the control level does not apply only to the particular environment in which it was learned, the critter will be transferred to an empty room where it will again demonstrate the learned path-following behaviors.

Sections 8.5 through 8.7 will describe three experiments in which various of the learning methods failed and explain why they failed. Section 8.5 will describe an experiment in which the image feature generator fails to produce a ring-shaped representation of the structure of the ring of distance sensors. Section 8.6 will describe an experiment in which the critter fails to discover any local state variables. Section 8.7 will describe an experiment in which the critter fails to learn any path-following behaviors. Finally, Section 8.8 will summarize the set of learning methods and identify a number of ways in which they can be improved.

8.2 A cluttered room

The environment used in this experiment is shown below. It is a rectangular room with a number of obstacles in it. The room has dimensions six meters by four meters.

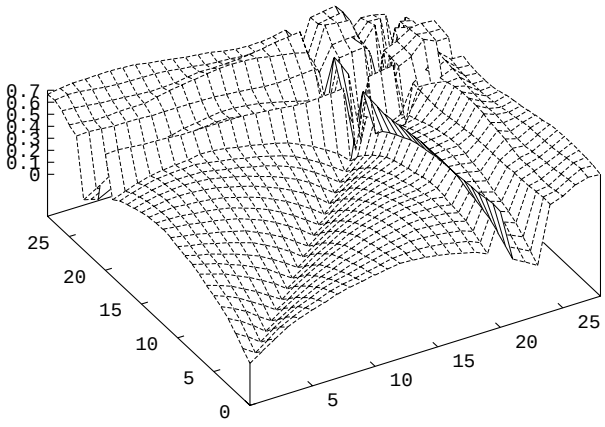


The simulated mobile robot used throughout this chapter is the same as that described earlier in the dissertation. The robot's raw sense vector has 29 components. The first 24 components of the raw sense vector give the distances to the nearest objects in each of 24 directions. These have a maximum value of 1.0 which they take on when the nearest object is beyond one meter away. The sensors are numbered clockwise from the front. The 21st component is defective and always returns a value of 0.2. The 25th component is a sensor giving the robot's battery's voltage, which decreases slowly from an initial value of one. The 26th through 29th components comprise a digital compass. The component with value 1 corresponds to the direction (E, N, W, or S) in which the robot is most nearly facing.

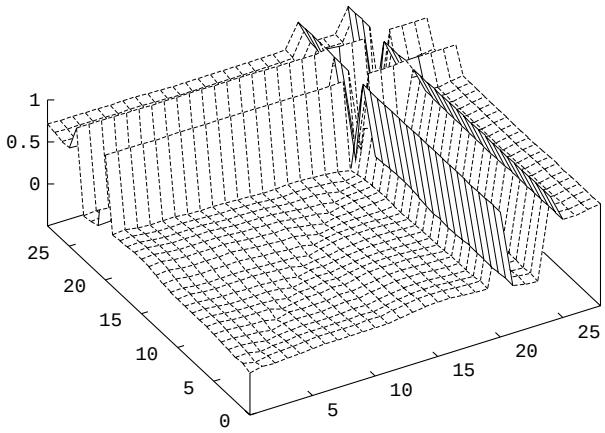
The robot has a "tank-style" motor apparatus. Its two motor control signals a_0 and a_1 tell how fast to move the right and left treads. Moving the treads together produces forward or backward motion; moving them in opposition produces rotation. The robot's maximum speed is 50 centimeters per second. Its maximum rotational speed is 50 degrees per second.

8.2.1 Modeling the sensory apparatus

The first step in modeling the robot's sensory apparatus is to apply the group feature generator. Recall that the group feature generator uses two distance metrics. The value of the first distance metric, d_1, after the robot has wandered for 20 minutes, is shown below:



The value of distance metric d_2 after the robot has wandered for 20 minutes is shown below:



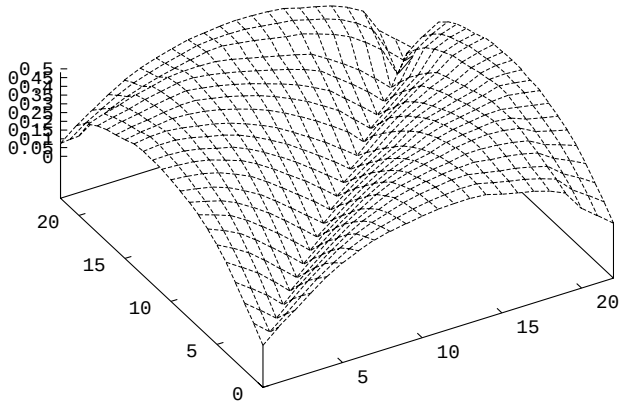
For each component of the raw sense vector, the set of similar components is shown.

(0 1 22 23) (0 1 2) (1 2 3) (2 3 4) (3 4 5) (4 5 6) (5 6 7) (6 7 8 9) (7 8 9)
(7 8 9 10) (9 10 11) (10 11 12) (11 12 13 14) (12 13 14) (12 13 14 15)
(14 15 16) (15 16 17 18) (16 17 18) (16 17 18 19) (18 19) (20) (21 22)
(0 21 22 23) (0 22 23) (24) (25) (26) (27) (28)

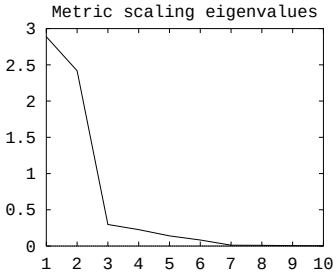
By taking the transitive closure of the similarity relation defined by the above sets of similar components, a list of related groups is produced. Notice that the working distance sensors have all been grouped together.

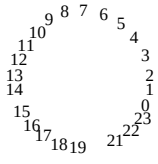
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23)
(20)
(24)
(25)
(26)
(27)
(28)

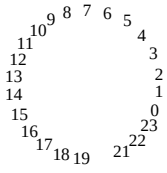
The second step in modeling the robot's sensory apparatus is to apply the image feature generator. Recall that this feature generator uses the distance metric d_1 to find a representation capturing the physical layout of the sensors that have been grouped together by the group feature generator. For the sensors in the group, the value of distance metric d_1 is shown below. This metric was obtained while the critter wandered randomly for 40 minutes.



The outputs of the metric scaling and relaxation algorithm are shown below. Notice that the image feature generator has faithfully reconstructed the physical layout of the set of distance sensors.







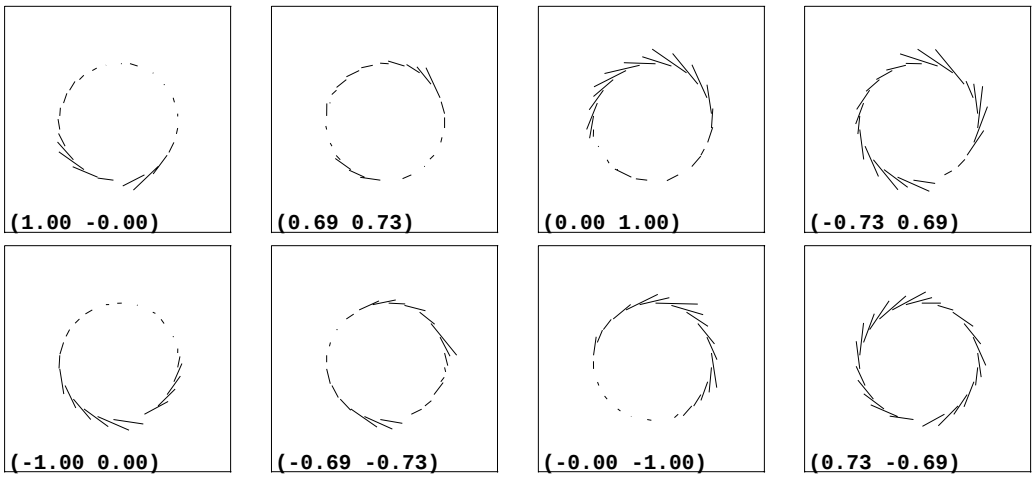
8.2.2 Modeling the motor apparatus

The image feature learned in the previous step serves as input to the motion feature generator which produces a motion feature. The value of the motion feature at any instant represents the

motion of the robot at that instant and can be used to characterize the effect of the motor control vector currently being used.

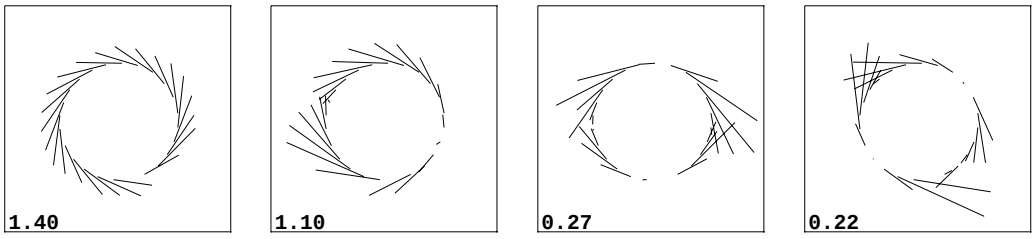
The first step in modeling the robot's motor apparatus is to characterize the effects of each of a large set of representative motor control vectors. Here, 100 representative motor control vectors of unit magnitude are chosen uniformly from the space of all possible motor control vectors. The effect of a motor control vector is represented by an *average motion vector field* (*amvf*), computed as the average of the motion feature's value over all time steps during which that motor control vector was used.

Eight example *amvf*'s and their associated motor control vectors are shown below. These were obtained while the critter wandered for 60 minutes, repeatedly choosing a representative motor control vector at random and executing it for one second (ten time steps).



Principal component analysis is used to analyze the set of 100 *amvf*'s and identify a basis set of effects represented as *principal eigenvectors*. By matching these eigenvectors against the 100 *amvf*'s, it is possible to discover which motor control vectors can be used to produce which effects. Motor control vectors that match are identified as *primitive actions*.

The first four eigenvectors for the space of average motion vector fields are shown below. The first corresponds to a pure rotation motion and the second corresponds to a pure translation motion.



$\mathbf{u}^0$= (-0.706 0.707) $\mathbf{u}^1$= (0.728 0.683)

The two motor control vectors identified as primitive actions are shown above under the two principal eigenvectors. None of the other eigenvectors match any of the *amvf*'s. The critter concludes that the robot's motor apparatus has two degrees of freedom and that the above primitive actions can be used to produce motion for each degree of freedom.

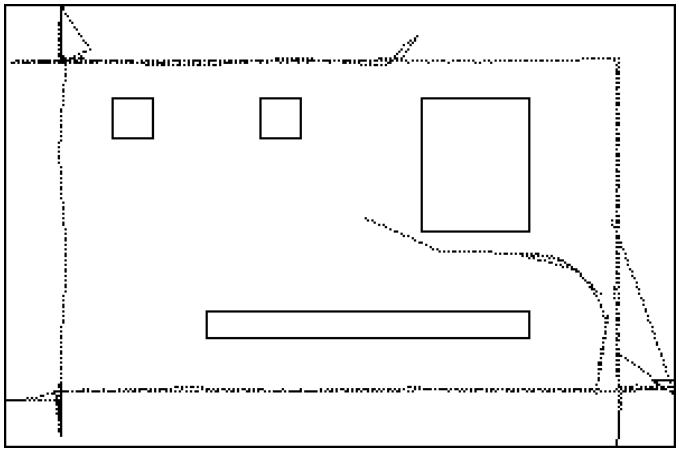
8.2.3 Learning behaviors

After learning a model of the robot's sensorimotor apparatus, the critter learns a set of homing and path-following behaviors. These behaviors comprise its acquired knowledge at the control level of the spatial semantic hierarchy. Its first step is to find a set of *local state variables* — features that are both action dependent and predictable. To find a set of local state variables, the critter first generates a set of scalar features (candidate local state variables) and then analyzes them using linear regression to produce the *static action model*.[1]

For this experiment, the result of the analysis is the identification of the set of local-minimum features as local state variables.[2] These are the only generated features that are identified as both action dependent and predictable.

The information in the static action model is used to define a set of homing behaviors and open-loop path-following behaviors. The critter experiments with the open-loop path-following behaviors while learning a *dynamic action model* that predicts how the primitive actions can be used for error correction.

In this experiment, the critter discovers that it can use the turning primitive action for error correction while executing an open-loop path-following behavior based on the advancing primitive action. It uses this information to define closed-loop path-following behaviors. The following picture shows a trace of a random exploration behavior demonstrating the critter's learned behaviors.



8.3 Re-learning the behaviors in a T-shaped room

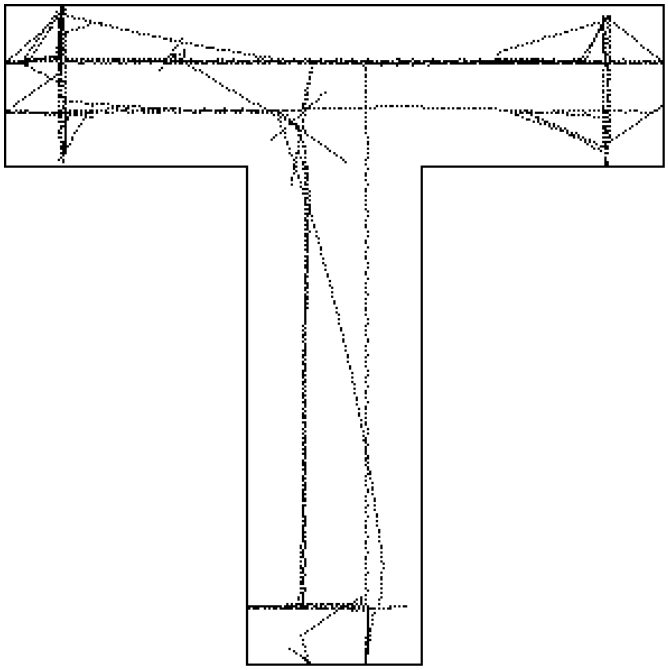
For this experiment, the robot was moved from the cluttered room to a T-shaped environment and the critter's control-level learning (i.e., static action model, dynamic action model, and learned behaviors) was erased. Its task was to begin with an intact model of the robot's sensorimotor apparatus and learn an appropriate set of homing and path-following behaviors. The environment used in this experiment consists of two corridors connected to form a T. The corridor forming the

[1] See Chapter 6 for a full description of the set of feature generators used and the set of features produced during the generate-and-test process.

[2] These features are produced by applying the composition of the following operators to the raw sense vector **s**: **group**, **image**, **lmin**, **tracker**, and **val**.

top of the T is 6 meters long and 1.5 meters wide. The shorter corridor is 4.5 meters long and 1.5 meters wide.

The critter successfully learns the control-level of the spatial semantic hierarchy. The picture below shows a trace of a random exploration behavior demonstrating the learned behaviors.



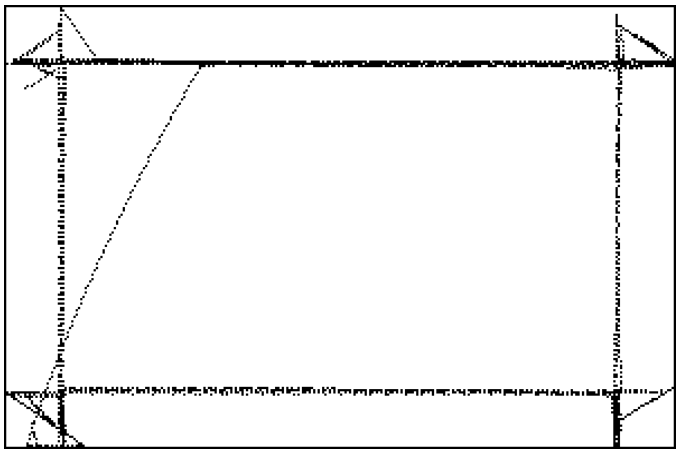
Path-following behaviors based on the advancing primitive action produce the straight-line trajectories that are parallel to the walls. Path-following behaviors based on the turning primitive action leave the robot in the same place while changing the robot's heading. The homing behaviors based on the advancing action produce most of the rest of the trajectories shown in the picture. A few of the trajectories are produced by a random wandering behavior that is used whenever none of the other behaviors are applicable.[3]

This experiment demonstrates that both the set of features and the model of the sensorimotor apparatus that were learned in the first environment are applicable in the second environment.

8.4 Using the behaviors in an empty room

For this experiment, the robot was moved from the T-shaped environment to an empty rectangular room (of dimensions 6 meters by 4 meters). The critter's model of the robot's sensorimotor apparatus and its set of learned behaviors was left intact. The following trace of a random exploration behavior demonstrates that the learned behaviors do not apply only to the environment in which they were learned.

[3]The exploration behavior selects its sub-behaviors stochastically and occasionally selects the random wandering behavior even when other behaviors are applicable.



8.5 A long and narrow room

This experiment demonstrates an instance in which the image feature generator fails to produce a ring-shaped representation of the structure of the ring of distance sensors. The environment used in this experiment is a long, narrow, rectangular room. The room is six meters long and one half meter wide. This environment is intended to confuse the image feature generator. Since the room is so narrow, the values of distance sensors on opposite sides of the ring will be often be similar: If a sensor is detecting the distance to one of the long walls of the room, then the sensor opposite to it will be detecting the distance to the wall on the opposite side of the room. Both sensors will produce a small value (less than 0.5). On the other hand, if a sensor is returning a large value, then there is a good chance that the sensor opposite to it will also be returning a large value.

If opposite sensors return similar values, on average, then the image feature generator will place them close together in the image feature. It seems unlikely then, that the resulting image feature will capture the ring structure of the array of distance sensors.

8.5.1 Modeling the sensory apparatus

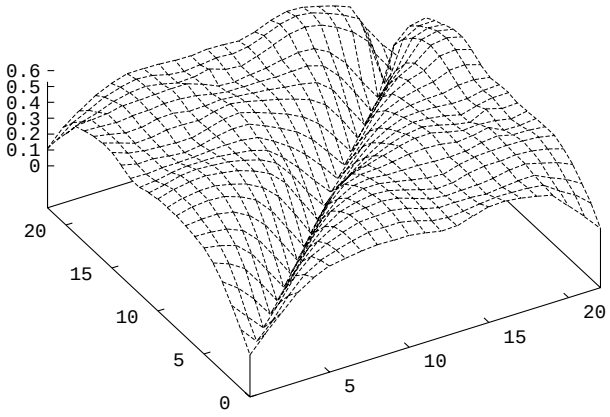
For each component of the raw sense vector, the set of similar components is shown.

(0 1 23) (0 1 2) (1 2 3) (2 3 4) (3 4 5) (4 5 6) (5 6 7) (6 7 8) (7 8 9)
(8 9 10) (9 10 11) (10 11 12 13) (11 12 13 14) (11 12 13 14 15)
(12 13 14 15) (13 14 15 16) (15 16 17) (16 17 18) (17 18 19) (18 19)
(20) (21 22) (21 22 23) (0 22 23) (24) (25) (26) (27) (28)

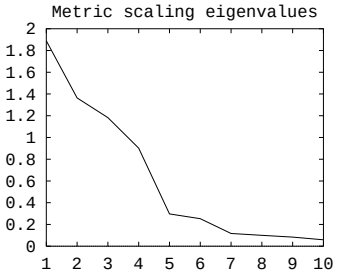
The groups of related sensors, produced by taking the transitive closure of the sets of similar sensors, are shown below. The distance sensors have again all been grouped together.

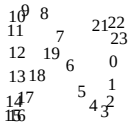
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23)
(20)
(24)
(25)
(26)
(27)
(28)

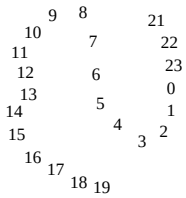
The intersensor distances $d_{1,ij}$ for all of the sensors that have been grouped together by the group feature generator are shown below. Notice that the shape of the plot is qualitatively different from those seen previously.



The outputs of the metric scaling and relaxation algorithm are shown below.







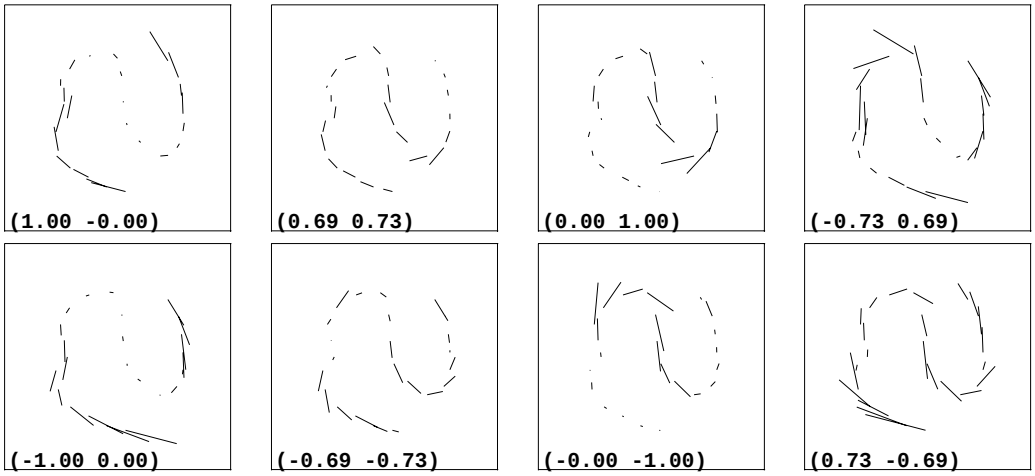
According to the metric-scaling scree diagram on the left, the structure of the array of sensors is best captured by a four-dimensional representation. The middle figure below shows the projection onto two dimensions of the set of points generated by the metric-scaling algorithm. The figure on the right shows the results of the relaxation algorithm.[4] Notice that sensors that are adjacent in

[4]The metric-scaling algorithm, the relaxation algorithm, and the definition of the image and motion features can all handle images of arbitrary dimension. However, in the current implementation, I have constrained the image feature to be two-dimensional. A goal for future research is to remove this artificial constraint and test the methods on sensory arrays that are genuinely three-dimensional.

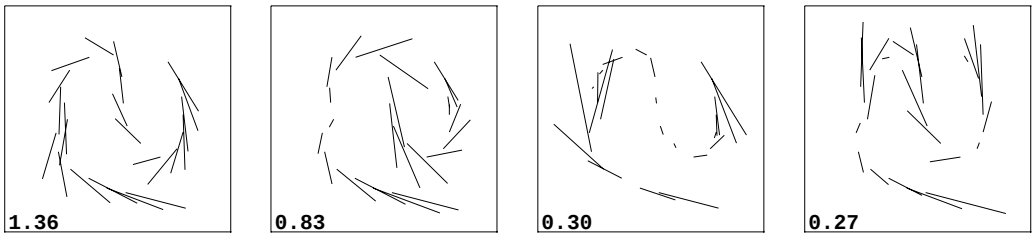
the ring of sensors are close together in the image. [5]

8.5.2 Modeling the motor apparatus

Eight example *amvf*'s and their associated motor control vectors are shown below.



The first four principal eigenvectors for the space of average motion vector fields are shown below. The method actually identifies the turning motor control vector correctly. The second primitive action is primarily an advancing action but has a significant turning component to it. The method erroneously identifies a third primitive action.



$\mathbf{u}^0$= (-0.723 0.680) $\mathbf{u}^1$= (-0.532 -0.837) $\mathbf{u}^2$= (-0.877 -0.481)

8.6 A circular room

This experiment demonstrates an instance in which the critter fails to discover any local state variables. The robot's environment is a circular room 3 meters in diameter.

8.6.1 Modeling the sensory apparatus

The sets of similar sensors identified by the group feature generator are shown below.

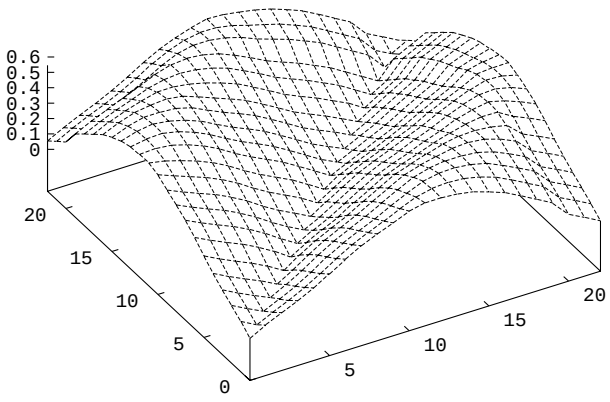
[5] Though the results are not shown here, I have also run the relaxation algorithm for this distance metric in three dimensions. In that case the resulting pattern of sensors resembles the pattern of stitching on a baseball.

(0 1 23) (0 1 2) (1 2 3) (2 3 4) (3 4 5) (4 5 6) (5 6 7) (6 7 8)
(7 8 9) (8 9 10) (9 10 11) (10 11 12) (11 12 13) (12 13 14) (13 14 15)
(14 15 16) (15 16 17) (16 17 18) (17 18 19) (18 19) (20) (21 22)
(21 22 23) (0 22 23) (24) (25) (26) (27) (28)

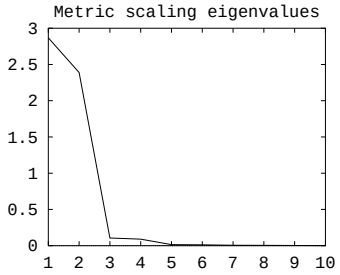
The groups of related sensors, produced by taking the transitive closure of the sets of similar sensors, are shown below:

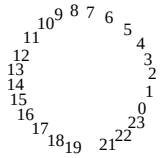
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23)
(20)
(24)
(25)
(26)
(27)
(28)

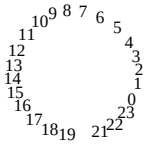
The value of distance metric d_1 (used by the image feature generator) for the sensors in the group is shown below.



The outputs of the metric scaling and relaxation algorithm are shown below.

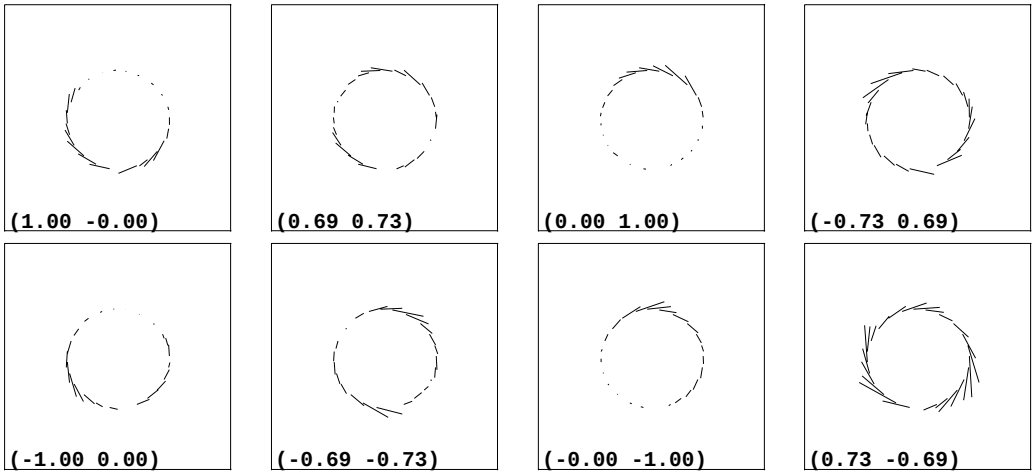




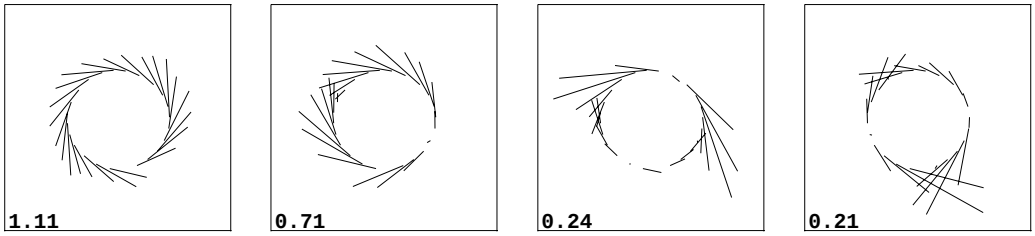


8.6.2 Modeling the motor apparatus

Eight example *amvf*'s and their associated motor control vectors are shown below.



The first four principal eigenvectors for the space of average motion vector fields are shown below.



$\mathbf{u}^0$= (-0.746 0.659) $\mathbf{u}^1$= (0.765 0.632)

At this stage, the critter has identified two primitive actions, shown above with the two principal eigenvectors.

8.6.3 Learning behaviors

In this experiment, the critter fails to discover any local state variables. In order for a feature to be a local state variable, it must be both action-dependent and predictable. For a feature to be predictable, the effects of the primitive actions on the feature must be known for all possible contexts.

In the rectangular and T-shaped environments, the local-minimum features (which give distances from the robot to nearby objects or walls) were identified as local state variables. Here is a summary of what was learned by the critter (and represented in the static action model) for the robot in the rectangular environment:

- The first primitive action (turning) does not affect the local-minimum features. The effects of the primitive action are thus predictable for all contexts.
- The effect of the second primitive action (advancing) is context dependent:

- When the robot is facing toward a wall, the primitive action reliably decreases the value of the local-minimum feature.
- When the robot is facing away from a wall, the primitive action reliably increases the value of the local-minimum feature.
- When the robot is facing parallel to the wall (in either direction), the primitive action leaves the value of the feature constant.

For this experiment (with the circular environment), the critter's learned static action model is identical to that described above, but with the following exception: When the robot is facing parallel to the wall, the effect of the second primitive action on the local-minimum feature is unpredictable. Here is an explanation for the difference. When facing parallel to a straight wall, a robot can move for many steps without changing the distance to the wall significantly. This is why it is possible for the linear regression tester that analyzes the effect of the primitive action to conclude that its effect is, to a good approximation, zero in this context. In the circular world, on the other hand, the robot can only advance a few steps without changing the distance to the wall. The only conclusion that the critter is able to draw from the linear regression tester is that the effect of advancing is unpredictable in this context.

8.7 A small room

This experiment demonstrates an instance in which the method for learning primitive actions identifies primitive actions whose effects are combinations of rotation and translation motions. This is in contrast to the experiment in Section 8.2 in which the primitive actions corresponded to pure rotation and translation motions. The reason for this difference and the consequences for the control-level learning methods will be given at the end of this section.

The environment used in this experiment is a small rectangular room. The room is 80 centimeters long and 60 centimeters wide. Since the range of the distance sensors is one meter, this means that all of the walls will always be within sensor range, no matter where the robot is.

8.7.1 Modeling the sensory apparatus

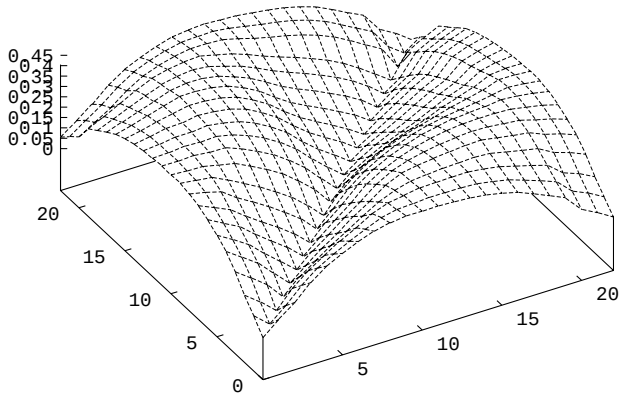
The sets of similar sensors identified by the group feature generator are shown below.

(0 1 23) (0 1 2 23) (1 2 3) (2 3 4 5) (3 4 5 6) (3 4 5 6 7) (4 5 6 7 8)
(5 6 7 8) (6 7 8 9) (8 9 10) (9 10 11) (10 11 12 13) (11 12 13)
(11 12 13 14) (13 14 15) (14 15 16 17) (15 16 17 18) (15 16 17 18)
(16 17 18 19) (18 19) (20) (21 22) (21 22 23) (0 1 22 23) (24) (25)
(26) (27) (28)

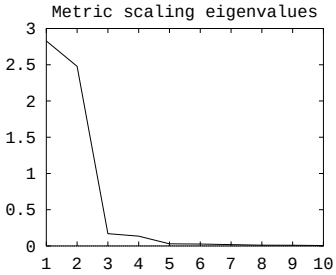
The groups of related sensors, produced by taking the transitive closure of the sets of similar sensors, are shown below:

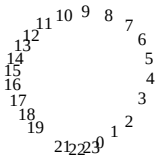
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23)
(20)
(24)
(25)
(26)
(27)
(28)

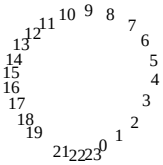
The value of distance metric d_1 (used by the image feature generator) for the sensors in the group is shown below.



The outputs of the metric scaling and relaxation algorithm are shown below.

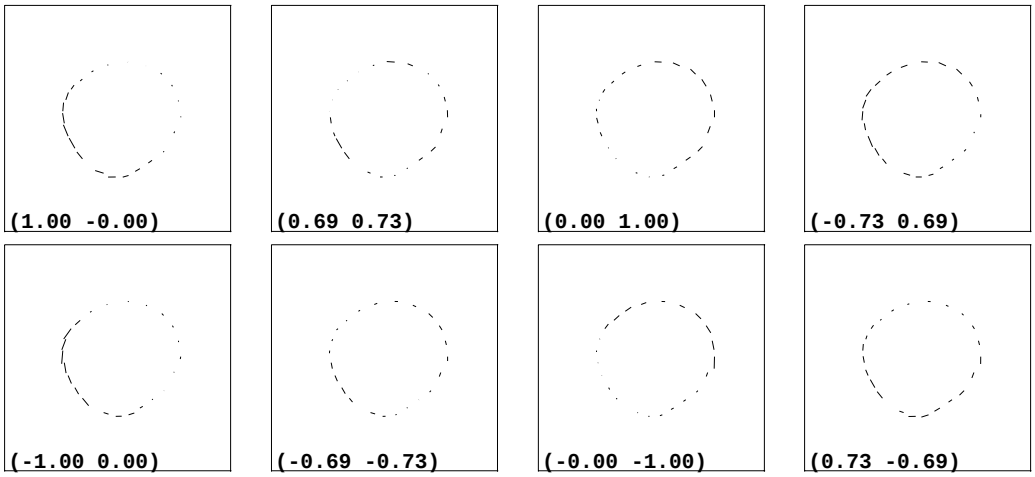




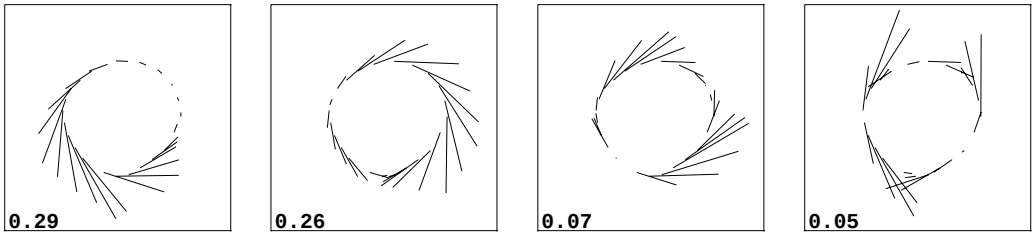


8.7.2 Modeling the motor apparatus

Eight example *amvf*'s and their associated motor control vectors are shown below.



The first four eigenvectors and the standard deviations of the associated principal components for the space of average motion vector fields are shown below.



$\mathbf{u}^0$= (-0.965 0.185) $\mathbf{u}^1$= (0.062 -0.996)

The set of primitive actions identified in this experiment is different from the set of primitive actions identified in the experiment described in Section 8.2. In the earlier experiment, the motor control vectors identified as primitive actions corresponded to pure turning and advancing motions, respectively. In this experiment, the motor control vectors identified as primitive actions correspond to combinations of turning and advancing motions. The rest of this section will explain why this happened. Section 8.7.3 will then describe how the different set of primitive actions affects the results of the control-level learning methods.

Recall how principal component analysis works. Given a set of input vectors $\{\mathbf{x}^i\}$, principal component analysis produces a basis set of unit vectors called eigenvectors satisfying the following: The first eigenvector $\mathbf{v}^0$ is chosen so that the variance of $\mathbf{x}^i \cdot \mathbf{v}^0$ (where $\mathbf{x}^i$ ranges over the set of input vectors) is maximized. Each subsequent eigenvector $\mathbf{v}^j$ is chosen so that it is orthogonal to all of the preceding eigenvectors and maximizes the variance of $\mathbf{x}^i \cdot \mathbf{v}^j$. Less formally, $\mathbf{v}^0$ accounts for most of the variation in the input set of vectors, $\mathbf{v}^1$ accounts for most of the remaining variation, etc. The preceding description of principal component analysis explains why the experiment in Section 8.2 identified a motor control vector for turning as the first primitive action. It is because the magnitude of the effect of turning, defined as the magnitude of the corresponding *amvf*, is significantly larger than the magnitude of the effect of advancing.

By contrast, in the small room, the magnitude of the effect of turning is similar to the magnitude

of the effect of advancing. The following explains why. The apparent motion[6] of an object relative to the robot while the robot advances is greater if the object is close to the robot than if the object is far away. For a turning motion, the distance from the robot to the object does not affect the apparent motion. In the small room, the walls are always close to the robot. Thus the apparent motion resulting from advancing (relative to the apparent motion resulting from turning) is greater in the small room than in the large room. For the particular room and robot of this experiment, the magnitudes of the effects of turning and advancing turn out to be similar.

The consequence of this is that the first principal eigenvector produced by principal component analysis will not necessarily correspond to a pure turning motion. In the small room, the magnitudes of the effects of all of the unit motor control vectors are similar. The result is that small differences in the values of the measured *amvf*'s may make a substantial difference in the values of the principal eigenvectors. In this particular case, the result of the analysis is a pair of primitive actions that are combinations of turning and advancing.

It is important to note that, although the eigenvectors do not correspond to pure turning and advancing motions, they do span the space of all motions that the robot can produce. The learning method has produced a valid model of the motor apparatus and has correctly concluded that the robot's motor apparatus has two degrees of freedom.

8.7.3 Learning behaviors

This section demonstrates that the success of the control-level learning methods can depend on the set of primitive actions identified by principal component analysis.

The features that are identified as local state variables in this experiment are s_0, s_1, s_{11}, and s_{12}. The first two measure distances to objects in front of the robot; the second two measure distances to objects behind the robot. These are the only features that are both action dependent and predictable. To understand why, consider the effects of the learned primitive actions. The first primitive action ($\mathbf{u}^0$= (-0.965 0.185)) moves the robot backwards while turning it counterclockwise; the second primitive action ($\mathbf{u}^1$= (0.062 -0.996)) moves the robot backwards while turning it clockwise. Both of the primitive actions move the robot in gently curving arcs. The effects of both primitive actions are roughly similar to the effects of simply moving backwards. Both of the primitive actions reliably increase the values of features s_0 and s_1 and both reliably decrease the values of features s_{11} and s_{12}. The effects of the primitive actions on the other features are either zero or unpredictable (e.g., for both the local-minimum and local-maximum features, the primitive actions are predictable for some contexts and unpredictable for others).

The critter successfully defines homing behaviors for moving the local state variables to their target values, but since there are no contexts in which any of the primitive actions leave any of the local state variables constant, no path-following behaviors are defined. Since it is clearly possible (as previous experiments have demonstrated) for the robot to define path-following behaviors based on the local-minimum features, this experiment points out a limitation of the currently implemented method for learning local state variables and path-following behaviors. Section 8.8.3 will describe

[6]By "apparent" motion, I mean the representation produced by the motion feature as opposed to the actual motion of the robot which cannot be directly detected by the critter.

alternative methods for learning path-following behaviors that do not depend on the particular set of learned primitive actions.

8.8 Discussion

Section 8.8.1 will summarize the sequence of methods used to learn the sensorimotor and control levels of the spatial semantic hierarchy. Section 8.8.2 will then identify a number of limitations of the methods (i.e., ways in which the methods can fail). Section 8.8.3 will discuss a number of possible improvements to the particular set of learning methods. Finally, Section 8.8.4 will identify a general approach, of which the particular approach presented in the dissertation is one example.

8.8.1 Summary of learning methods

The learning methods used by the critter to learn the sensorimotor and control levels of the spatial semantic hierarchy are summarized below. For each method, I identify the source of information used by that method, and describe the representations or objects produced by the method.

1. **Modeling the sensory apparatus**. The group and image feature generators exploit the fact that if two sensors are close together in a array of sensors that adequately samples a continuous property of the environment, then those sensors will, on average, produce similar values. The source of information is the sequence of values produced by the sensors while the critter wanders by choosing motor control vectors randomly. The group and image features produced by the generators represent the structure of the sensory apparatus.

2. **Representing motion**. The motion feature generator exploits temporal and spatial information in the image feature to produce a motion feature that represents the instantaneous motion of the robot.

3. **Modeling the motor apparatus**. This learning method exploits the information in the motion feature while the critter wanders randomly. The method uses principal component analysis to produce a basis set of effects and corresponding primitive actions.

4. **Generating candidate local state variables**. A set of feature generators is used to develop a tree of learned features. Some of the generators apply a simple operator to an existing feature to produce a new feature. Others (e.g., the group and image feature generators) first analyze an input feature's behavior over time in order to produce a new feature. Any scalar feature produced by this method is a candidate local state variable.

5. **Discovering local state variables**. Linear regression is used to try to predict the effects of the primitive actions on the features. Features that are predictable in this way, and hence controllable, are identified as local state variables. The source of information is the sequence of scalar-feature values produced while the critter experiments with its primitive actions. The results of the linear regressions comprise the static action model.

6. **Defining homing behaviors**. The source of information for this step is the static action model produced in the previous step. Behaviors are defined, using the information in the static action model, that can be used to move a local state variable to a target value.

7. **Defining open-loop path-following behaviors**. The static action model is the source of information for this step as well. It is used to identify ⟨*local state variable, primitive action, context*⟩ triples where the primitive action maintains the local state variable constant given that the robot is in the context. Behaviors are produced that can be used to move the robot while maintaining a local state variable at its target value.

8. **Learning a dynamic action model**. The source of information for this step is the sequence of values of the local state variables while the critter experiments with its open-loop path-following behaviors. While one primitive action is used for an open-loop path-following behavior, linear regression is used to try to predict the effects of the other primitive actions on the local state variable on which the path-following behavior is based. The results of the linear regressions comprise the dynamic action model.

9. **Defining closed-loop path-following behaviors**. The source of information for this step is the dynamic action model. Primitive actions whose effects are predictable are used to provide error correction, turning the open-loop path-following behaviors into closed-loop path-following behaviors.

8.8.2 Failure modes

The preceding list gives a number of learning methods that can be used by the critter to learn the first three levels of the spatial semantic hierarchy. This section describes a number of ways in which the methods can fail.

Modeling the sensory apparatus. If there is no structured array of sensors, then the group feature generator will produce only small or singleton groups and the image feature generator will not apply. If there is an array of sensors but the sensors do not adequately sample a continuous property of the environment, then the group and image features will fail to produce a representation of the structure of those sensors. For example, if there are only four distance sensors, then the values of adjacent sensors may not be similar enough for the group feature generator to group them together. If the environment is large and the critter does not adequately explore the environment before applying the group and image feature generators, then the measured inter-sensor distance metrics may not accurately reflect the structure of the sensory apparatus. For the robot with distance sensors, an environment with large open spaces can be problematic – the robot might spend a great deal of time with no objects in sensor range.

Representing motion. If no image feature is produced, then of course the motion feature generator will not apply. If the robot's motion is so fast that successive image-feature values are unrelated, then the motion feature will fail to produce meaningful results.

Modeling the motor apparatus. The matching process that identifies primitive actions (i.e., motor control vectors whose *amvf*'s match the principal eigenvectors) can fail to correctly identify

a primitive action if the *amvf*'s have not converged (i.e., if the critter has not wandered long enough and the values of the amvf's are still fluctuating with time).

Generating candidate local state variables. The discovery of local state variables may fail if the language of features and feature generators is not general enough. In such a case, none of the generated scalar features would satisfy the definition of local state variable. On the other hand, if the language of features and generators is too general, the critter will quickly become bogged down in a combinatorial explosion of mostly useless features.

In this dissertation, I identified a small set of feature generators that are appropriate for a robot with a rich sensorimotor apparatus and then demonstrated that they are sufficient for a particular set of environments and sensorimotor apparatuses.

Learning action models. The critter will fail to correctly learn the static and dynamic action models if it does not explore long enough for the linear-regression calculations to converge. In the case that the critter must learn the relationships between a motor control vector and a feature for a large number of contexts, the method requires that the critter experiment with the motor control vector in each of those contexts.

Learning path-following behaviors. As was mentioned in Section 8.7.3, the learning of path-following behaviors can depend on the set of learned primitive actions. If none of the primitive actions can be used to maintain any of the local state variables constant, then no path-following behaviors will be learned.

8.8.3 Future work

Section 8.8.2 identified a number of ways in which the learning methods can fail. This section provides suggestions for improvements to the learning methods.

Improved feature testers. One way that several of the learning methods can fail is by jumping to a conclusion prematurely. For example, if the group generator uses a distance metric before the distance metric has converged, then the output of the generator may be incorrect. If primitive actions are identified before the *amvf*'s have converged, then the model of the motor apparatus may be incorrect.

In these examples, the distance metrics and the *amvf*'s are examples of *feature testers* — features that are used to characterize other features. A solution to the problem of drawing premature conclusions is to have each feature tester tell when its output is meaningful. It can do this by providing a measure of confidence in addition to its output value. For example, the confidence level for a tester might be close to 1 if the tester's output is stable (changing slowly) and close to 0 if the tester's output is still fluctuating.

For a linear regression tester, the confidence level should be a function of the set of inputs it has received. Consider, for example, how the static action model uses linear regression testers. It uses a separate linear regression tester for each $\langle$*feature, primitive action, context*$\rangle$ triple. If the robot is never in a given context, then the confidence level for any linear regression tester based on that context should be zero. The confidence level for a linear regression tester might be defined in

terms of the *90% confidence interval*[7] for the correlation betweeen the input variables. The smaller the confidence interval, the greater the tester's confidence level.

An improved static action model. The critter uses the static action model to define a set of open-loop path-following behaviors — behaviors that move the robot while maintaining a local state variable constant. In the current implementation, open-loop path-following behaviors are based on primitive actions. If a primitive action maintains a local state variable constant, according to the static action model, then it can be used as the "base action" for a path-following behavior. Using only primitive actions as base actions is a limitation of the current implementation. The method for learning path-following behaviors would be improved if the static action model could predict the effects of arbitrary motor control vectors, not just the primitive actions. With a more comprehensive static action model, more path-following behaviors could be defined. This would solve the problem seen in Section 8.7.3 where no path-following behaviors were learned.

One approach to improving the static action model would be to discretize the space of all motor control vectors into a set of representative motor control vectors and then to learn models of all of these instead of just the primitive actions. Another approach would be to use a neural network (e.g., Jordan & Rumelhart, 1992) to learn to predict the context-dependent effects of arbitrary actions. The network could then serve as the static action model and could be used to find base actions for path-following behaviors.

Reinforcement learning. It may be possible to use reinforcement learning to learn homing and path-following behaviors without the need for the primitive actions or explicit action models. An advantage of such an approach is that it does not presume that a particular model of the sensorimotor apparatus has been learned. A disadvantage is that it is only possible to train one behavior at a time whereas it is possible to learn action models for a large number of features simultaneously.

8.8.4 A general approach

The sequence of steps described in Section 8.8.1 provide a particular solution to the learning problem addressed by this dissertation. This particular solution is an instance of the more general solution outlined below:

1. Apply a generate-and-test algorithm to produce a set of scalar features.
2. Try to learn how to control the generated scalar features. Those that can be controlled are identified as local state variables.
3. Define homing behaviors — behaviors that move a local state variable to a target value.
4. Define path-following behaviors — behaviors that move the robot while keeping a local state variable at its target value.

Clearly, there are other ways to instantiate the above sequence of steps besides the specific set of learning methods identified in this dissertation. Future work will involve both improving the current set of methods and identifying alternate paths to the solution.

[7] See, for example, Krzanowski, 1988, p. 415.

8.8.5 Conclusion

The set of learning methods presented here does not represent the final word on the problem of learning to use an uninterpreted sensorimotor apparatus. Instead it is one path to the goal.

Moreover, the learning methods are interesting in their own right. Each one identifies a source of information available through experimentation with an uninterpreted sensorimotor apparatus, and each provides a method for exploiting that information to give the critter a new way of understanding the robot's sensory input or of interacting with the robot's environment.

Chapter 9

From Continuous World to Discrete World

As a result of its learning so far, the critter has made the transition from raw senses and motor control vectors to local state variables and high-level behaviors. Its final step is to define a new abstract interface that abstracts the continuous world of a robot to a discrete world whose structure can be modeled by a finite-state automaton. This interface, called the *discrete abstract interface*, comprises the critter's knowledge at the procedural level of the spatial semantic hierarchy. The purpose of the discrete abstract interface is to set the stage for learning the topological level of the spatial semantic hierarchy.

Section 9.1 puts the discrete abstract interface in context by describing a two-level model of a robot's state space. The first level of the model corresponds to the learning that has already been done at the sensorimotor and control levels of the spatial semantic hierarchy. The second level of the model corresponds to the topological level. Section 9.2 defines the discrete abstract interface in terms of the set of behaviors that the critter has already learned. Section 9.3 demonstrates the discrete abstract interface for the robot in the rectangular room. Section 9.4 explains how to use the discrete abstract interface as the basis for learning a topological model of the robot's environment. Section 9.5 describes related work dealing with the problem of learning a model of an environment. Section 9.6 discusses a number of relationships between the work described in this dissertation and related work. Finally, Section 9.7 summarizes the set of learning methods used by the critter as it learns the first three levels of the spatial semantic hierarchy.

9.1 A two-level model of state space

In general, a model of an environment requires (1) knowledge of the environment's state space, that is, the set of possible states, (2) the ability to recognize the current state, and (3) the ability to navigate through state space. These three components are illustrated by two very different examples — the configuration space of a six-degree of freedom robot arm and the two-dimensional world of a hypothetical urban-dwelling mobile robot. These examples illustrate two different levels at which an agent can model its environment.

Consider first the example of a robot arm with a manipulator at the end of it and a video camera mounted above the arm. The state space (1) of the robot arm is the six-dimensional continuous space of Euclidean geometry. There are three state variables for position and three for orientation. Recognition of the current state (2) is accomplished using the camera which determines the position and orientation of the manipulator. Navigation through state space (3) requires both a knowledge of the structure of the state space and a knowledge of how the joint motors affect the state variables. A description of the structure of the state space should include the bounds of the state variables

and the fact that the three orientation variables "wrap around" (i.e., 360 degrees is equal to 0 degrees).

In a simple system like the robot arm, it is fairly clear how to model the system. For a more complex example, consider how the urban-dwelling mobile robot might model the city in which it lives. The state space (1) can be modeled as a set of interesting places — home, office, and hardware store, for example, together with a representation of their locations relative to each other. Recognition of the current state (2) requires either a sophisticated visual system or a set of beacons that identify the interesting places. The ability to navigate (3) requires the knowledge of how to move from any place to any other.

Small-scale vs. large-scale space. The examples of the robot arm and mobile robot illustrate two types of space which have been called *small-scale space* and *large-scale space*, respectively (Kuipers and Levitt, 1988). Small-scale space refers to the state space that is directly accessible. In the case of the robot arm, the entire state space is small-scale. For the mobile robot moving along a road, small-scale space is the part of the world that is directly visible. Large-scale space is a state space whose structure is at a larger scale than can be directly perceived. Whether a state space is small or large scale depends on the observer. For a person in a car, a city is a large-scale space. For a person in a plane flying over the city, the city is a small-scale space.

9.1.1 The critter's model of small-scale space

The critter needs to model the robot's world at both levels. For small-scale space, it models the state space (1) using the set of currently defined local state variables. The current state (2) is determined by the current values of the local state variables. The critter's knowledge of how to navigate (3) is contained in its static and dynamic action models. These tell how to use the set of primitive actions to affect the local state variables. The purpose of this chapter is to show how to learn a model of large-scale space.

9.1.2 The critter's model of large-scale space

One possibility for representing large-scale space is to use *global state variables* that uniquely determine the robot's state. In general, people and robots do not have access to global state information. While global state variables might be obtainable by integrating motion over time, the results will be error prone due to cumulative error. In some cases, it may be possible to have direct access to global state information. For example, a global positioning system (GPS) and a compass can be used to uniquely determine the current state of a person driving through a city. A disadvantage of using global state variables is that they do not give positions relative to other objects in the environment. For a person driving through a city, the position of the car with respect to the road is much more relevant than its longitude and latitude.

Instead of trying to use a set of continuous global state variables to model both small- and large-scale space, the critter models small-scale space using continuous local state variables and large-scale space using a finite-state automaton (to be defined formally in Section 9.2). This two-level model allows the critter to be both graceful and discrete: The continuous model of small-scale

space provides the critter with enough information to define control laws for smooth motion along paths. The discrete model of large-scale space allows the critter to efficiently reason about large-scale motions. It can plan paths between distant places by applying a shortest-path algorithm to its topological representation of places and paths.

Learning the the finite-state automaton involves two steps: (1) defining a *discrete abstract interface*, and (2) inferring the structure of the underlying automaton defined by this interface. The knowledge gained by the critter after the first step is analogous to the knowledge a driver has when he first arrives in a new city — he knows how to explore and remember landmarks but he has not yet learned a map.

The next section gives a formal definition of "finite-state automaton" and defines the abstract interface that allows the critter to view the robot's world as a finite-state automaton whose structure is unknown.

9.2 The discrete abstract interface

A *finite-state automaton* is defined as a tuple $(Q, V, A, \delta, \gamma)$ where Q is a finite set of *states*, V is a finite set of output symbols which I will call *views*, A is a finite set of *actions*, δ is the next-state map, which maps state-action pairs to next states, and γ is the output map, which maps states to views.[1]

A *deterministic finite-state automaton* (DFSA) is an FSA in which δ and γ are functions. This means that the current state and action uniquely determine the next state and that the current state uniquely determines the view. If at time t, the automaton is in state $q(t)$ and action $a(t)$ is selected, then at time $t+1$, the automaton will be in state $\delta(q(t), a(t))$ and the output will be given by $\gamma(q(t+1)) \in V$.

A *nondeterministic finite-state automaton* (NDFSA) is an FSA in which δ and γ are general mappings. A given state and action may nondeterministically map to multiple next states and a given state may map nondeterministically to multiple views.

Given the tuple $(Q, V, A, \delta, \gamma)$ for a deterministic finite-state automaton, the input/output behavior of the automaton is easily predicted. Going the other way—inferring the tuple $(Q, V, A, \delta, \gamma)$ given only the input/output behavior—is a classic machine-learning problem. In that case, the finite-state automaton is a black box: the internal workings are not directly observable. At time t, the output view $v(t)$ is observed and the set of legal actions $A_{q(t)} \subset A$ is presented. Selection of an action $a \in A_q$ moves the automaton to its next state at which point a new view $v(t+1)$ and set of legal actions $A_{q(t+1)}$ is presented.

An abstraction of the robot's world as a finite-state automaton may be defined by giving implicit definitions for the values of Q, V, A, δ, and γ:

- Q is the set of states in which path-following behaviors terminate.[2]

[1] Some definitions also include q_0, the automaton's state when it is initialized.

[2] I have made the simplifying assumption that the resulting automaton is finite state. It is possible to imagine robot worlds in which the resulting automaton has an infinite number of states (e.g., an infinitely big office building with a random layout of rooms and corridors). In such an environment, the discrete abstract interface is still well defined, but it would not be possible to model the behavior of the interface using a finite-state automaton.

- V is the set of all *views*, where a view $v(q)$ is a symbol that is uniquely identified by the sense vector seen at state q. Different states may map to the same view. With each view v_i is an associated raw sense vector s_i. Upon entering a new state, the current raw sense vector $s(t)$ is compared with all of those remembered for views previously encountered. If $s(t)$ matches s_i, then $v(t) = v_i$. Otherwise, a new view v_j is created with associated raw sense vector $s_j = s(t)$.[3]

- The set of actions is given by $A = \bigcup_{q \in Q} A_q$ where A_q is the set of behaviors applicable in state q.

- The next-state mapping, δ, is defined by the set of all possible (q, a, q') triples where action a takes the robot from state q to q'.

- The output mapping, γ, is defined by the set of all possible pairs (q, v) where v is the view sensed by the robot when it is in state q.

The critter cannot directly recognize states, meaning that it cannot directly tell whether its current state is the same as one it has seen earlier. However, it can detect the current set of actions, A_q, and it can detect the current view, $v(q)$. The current view and the current set of applicable actions define the abstract interface that allows the critter to view the robot's world as a finite-state automaton. With this interface, the critter does not observe state transitions directly, it observes schemas of the form (v, a, v'). General methods for learning the model using the abstract interface will be described in Section 9.5. Section 9.4 will describe how to learn the model given the simplifying assumption that views uniquely identify states.

9.2.1 Reducing nondeterminism

Inferring the structure of an finite-state automaton (e.g., Rivest & Schapire, 1987, 1993), while an expensive task, is much less expensive than inferring the structure of an NDFSA (Dean et al., 1992, 1993). For this reason, it is advantageous to use actions that are as constrained as possible.

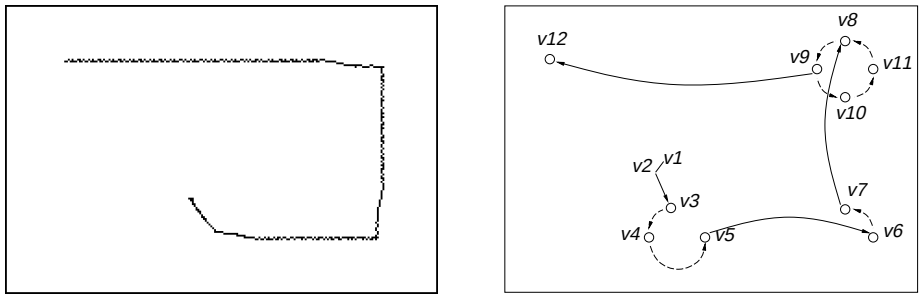
The discrete abstract interface supports the ability to select constrained actions by identifying four different types of actions, from completely nondeterministic to as deterministic as possible.

1. A *wandering* behavior is always included in A_q for each state q to guarantee that there is at least one action available. This behavior randomly chooses a motor control vector and executes it for one second. It is the least deterministic.

2. An *underconstrained* path-following behavior is more deterministic since only one of the robot's degrees of freedom is used to produce motion. An underconstrained path-following behavior is one for which not all of the control signals are used for motion or error correction.

3. A *homing* behavior is still more deterministic since it consistently moves a single local state variable to its target value.

[3] In the implementation, the vectors s_i and s_j match if they differ component-wise by less than a small constant.



a b

Figure 9.1: A demonstration of the discrete abstract interface. Here, it is used (under human control) to select appropriate behaviors to drive the robot around the room. At each step, the interface provides the view name (e.g., *v1*) that identifies the current state, and a finite set of applicable homing and path-following behaviors. The dotted arrows represent behaviors based on left turn motor control vectors ($u_0 > 0$). The solid arrows represent behaviors based on forward advance motor control vectors ($u_1 > 0$). During this exploration, the robot identifies the 12 unique views shown in the figure on the right.

4. A *one degree of freedom* (1-DOF) path-following behavior is the most deterministic. A 1-DOF path-following behavior is one for which one control signal is used to produce motion along the path and *all* of the other control signals are used for error correction.

A critter exploring its world using the discrete abstract interface can take advantage of these distinctions in the following way: It can first explore by only selecting the most deterministic behaviors. If it succeeds in mapping its world in this way, then it can use a more relaxed exploration strategy and thereby possible discover parts of its world that it would otherwise miss. Suppose, for example, that the critter begins exploring a large rectangular room with a box in the center that is not within sensor range when the robot is near a wall. By initially favoring 1-DOF behaviors, the critter will explore and map the perimeter of the room. When it succeeds at this, it can try randomly wandering and thereby discover the box in the center of the room.

9.3 A demonstration of the discrete abstract interface

This interface abstracts from continuous time to discrete time. While a path-following behavior is executing, the interface is undefined. When the behavior terminates, the interface identifies the current view and lists the current set of applicable behaviors. Figure 9.1 demonstrates this interface. Initially ($v1$), no wall is within sensor range and the only available action is the wandering behavior. When the wandering behavior terminates ($v2$), a homing behavior is applicable. Selecting this behavior leads to view $v3$ where two path-following behaviors based on u_0 (turning) are applicable. Selecting the first leads to view $v4$. Selecting it again leads to view $v5$. At this point, two 1-DOF path-following behaviors based on u_1 (advancing) are applicable. Choosing the first leads to $v6$. The figure shows the behavior of the robot during a user-guided exploration that leads it to $v12$. The rest of the exploration around the room (not shown) eventually returns the robot to view $v6$ (which is correctly recognized as $v6$).

9.4 Learning the topology of the environment

The critter has made a critical change of representation by abstracting a continuous sensorimotor apparatus to a discrete abstract interface with a finite set of sense values and actions. Understanding a continuous world is very difficult but the problem of understanding a discrete world has been extensively studied.

The robot's path-following behaviors constrain its motion to a one-dimensional subspace of the robot's complete state space. This 1-D skeleton is the basis for an abstraction of the robot's environment as a graph (a set of nodes and a set of edges connecting the nodes together). The edges correspond to paths — trajectories in the robot's state space produced by path-following behaviors. The nodes correspond to states where paths terminate, that is, states where a new path-following behavior becomes applicable and the critter stops to choose one of the currently applicable paths. The critter's goal is to construct this graph.

In the case where views uniquely identify states, the problem is straightforward. The critter keeps track, for each state it has seen, of all the actions applicable at that state. Each time it takes an action, A_j, that takes it from view V_i to V_k, it adds the edge (V_i, A_j, V_k) to the graph. It continues to explore (intelligently or randomly) until there are no state-action pairs that it has not explored.

In the case that views do not uniquely identify states, a more sophisticated exploration strategy is required. Such strategies are generally based on the following idea: If the current view does not uniquely identify the current state, the critter supplements the current sense vector with the sense vectors of nearby states. With enough information about the surrounding area, the current state can be uniquely identified.

Metrical information can be added to the topological representation by recording the time it takes to traverse each path. With this information, navigation including shortest-path planning is possible.

9.5 Related work

The work mentioned in this section deals with the problem of learning a model of an environment. A complete model of an environment is a description that is sufficient for predicting the input/output behavior of the environment, i.e., for predicting the sensory input that will be received from the environment in response to any sequence of actions. In some cases, learning a complete model is impractical, in which case a partial model may be learned.

9.5.1 Inferring the structure of finite-state worlds

As the name implies, a finite-state world has a finite number of discrete states. This is in contrast to a continuous world in which the state space is infinite and is represented by a vector of continuous state variables. The task of inferring the structure of a finite-state environment is the task of finding a finite-state automaton that accurately captures the input-output behavior of the environment.

Passive learning. In a passive-learning scenario, the learning agent does not choose its actions. Instead, it is passively given instances of the environment's input/output behavior. An instance is a sequence of actions and resulting sensory feedback from the environment. It has been shown that finding the smallest automaton consistent with a given set of instances is NP-complete (Angluin 1978, Gold, 1978).

Active learning. With active learning, in which the agent actively chooses its actions, the problem becomes tractable. The problem of inferring the structure of a finite-state environment is greatly simplified if there is a *reset* operation, which allows the agent to return to the initial state at any time, simply by clicking its heels together. Angluin (1987) gives a polynomial-time algorithm using active experimentation and passively received counterexamples. When it finds a complete model that explains all of the instances it has seen so far, it presents the model to an oracle which must then either confirm that the model is correct, or give a counterexample — an instance that is inconsistent with the hypothesized model.

Situated learning. Rivest & Schapire (1993) improve on Angluin's algorithm and give a version that does not require the reset operation, an operation that is unavailable to robots situated in the real world. They also describe a class of environments, called *permutation environments*, that do not require the counterexample-producing oracle.

Marker-based exploration. The algorithms mentioned so far all produce the smallest finite-state automaton that is consistent with the input/output behavior observed. There is no way for the learning agent to know whether the model accurately reflects the structure of the environment. As an extreme example, consider an environment with many states in which the sensory output is the same for every state. The model learned for such an environment will have only one state.

If, however, the learning agent is able to "mark" a state that it has visited, and sense the marker the next time it visits that state, then it is possible to infer the complete structure of a finite-state world in polynomial time (Dudek et al., 1991). The algorithm uses a finite number of markers (at least one) that can be dropped and picked up. The more markers the algorithm has, the more efficient it is.

Exploration in spatial environments. Kuipers' TOUR model (1978) is a method for modeling discrete spatial environments based on a theory of cognitive maps. A spatial world in this context is a deterministic finite-state automaton with additional assumptions that (1) the states correspond to positions in a two-dimensional coordinate frame and (2) the actions allow for turning in place and advancing.

Exploration in stochastic environments. Work has also been done in extending these methods to stochastic environments. In a stochastic environment, either one or both of the mappings δ or γ (defined on page 110) are stochastic. Dean et al. (1992) have extended Rivest and Schapire's theory to handle stochastic FSA's. They assume that actions are deterministic (i.e., that δ is deterministic)

but that the output function mapping states to senses is probabilistic. The key to their method is "going in circles" until the uncertainty washes out. Dean, Basye, and Kaelbling (1993) give an excellent review of learning techniques for a variety of stochastic automata. Bachrach (1992) has demonstrated a connectionist implementation of one of Rivest and Schapire's learning algorithms which works well in some stochastic environments.

Learning partial models. The algorithms for inferring the structure of arbitrary finite-state worlds are very expensive in terms of computational complexity and thus do not scale up to environments with many states. An alternative approach is to settle for less than a perfect model.

Learning the mapping δ means learning a set of tuples of the form (q_i, a_j, q_k) where q_i and q_k are states and a_j is an action. Knowing that the environment is in state q_i means knowing everything there is to know about the environment's current state. This is not how learning agents in the real world work. Humans, animals, and robots in the real world can never know the complete state of the world — they must be content with partial knowledge.

Drescher's constructivist approach (1991) acknowledges this fact. In his system, the environment is modeled in terms of *schemas*, tuples of the form *(context, action, result)*. The context and result are conjunctions of (possibly negated) *items*. An item is a discrete state variable whose set of possible values is {*on, off, unknown*}. The action is any event that can affect the state of the world. It can be primitive (one of the agent's innate actions), external (not produced by the agent), or composite (a higher-level procedure that achieves a specific result). Learning new schemas is accomplished via the *schema mechanism* which employs a statistical learning method called *marginal attribution*. Since schemas emphasize sensory effects of actions rather than state transitions, they are ideal for representing partial knowledge in stochastic worlds with many states.

9.5.2 Inferring the structure of continuous worlds

Map-based approaches. Much of the work that has been done in learning models of continuous worlds works by abstracting from a continuous environment to a discrete environment. This is the idea behind the spatial semantic hierarchy (Section 1.9). Kuipers and Byun (1988, 1991) demonstrate an engineered solution to the continuous-to-discrete abstraction problem for the NX robot. The target abstraction is the TOUR model (Kuipers, 1977, 1988). NX's *distinctive places* correspond to discrete states and its *local control strategies* implement the turn and travel actions. These constructs have to be manually redesigned in order to apply to a robot with a different sensorimotor apparatus.

Kortenkamp & Weymouth (1994) have engineered a similar solution on a physical robot. They supplement sonar information with visual information, thereby improving the robot's place-recognition ability. Instead of using locally distinctive places, they use *gateways*, "places that mark the transition between one space in the environment and another space."

Mataric (1991, 1992) has used the subsumption architecture (Brooks, 1986) to implement a novel control system for her robot Toto. In Toto, the representation of the map is fully integrated into the reactive control system. What is interesting for the present discussion is that the nodes in the learned map correspond not to distinct places, but to landmarks that are detectable over time

as the robot moves. Instead of using corners and intersections, Toto uses left walls, right walls, and corridors as its landmarks.

Lin and Hanson (1993) use a physical robot, called Ratbot, with 16 sonar sensors and 16 infrared sensors to demonstrate learning of a topological map of locally distinctive places. Their work is inspired by the work of Byun and Kuipers, but they use reinforcement learning[4] to train the local control strategies, rather than engineering them by hand. The target behaviors (e.g., corridor following) are specified by a human teacher. For example, when learning the corridor-following behavior, the robot is rewarded when it moves along the corridor without running into obstacles.

The approach taken in this dissertation is complementary to that of Lin and Hanson. They specify the desirable behaviors by defining appropriate reward signals and then letting the robot learn on its own how to gain the rewards. The critter, on the other hand, specifies its own target behaviors, eliminating the need for the human teacher. It does this by first learning a set of local state variables and then using them to define a set of error signals. Homing and path-following behaviors are then specified as behaviors that minimize the error signals or move the robot while maintaining them near zero. All of this is accomplished in a domain-independent manner — the robot does not need to be given any knowledge about corridors or corridor-following behaviors.

Once the error signals are defined, there are a number of ways in which the homing and path-following behaviors might be learned. Reinforcement learning is one approach.[5] The approach used in this dissertation is to learn static and dynamic action models that characterize the effects of actions on the local state variables and then to use these models to directly define the homing and path-following behaviors. This approach does require that the critter have a basic understanding of control theory, but the required knowledge is domain independent. It would be interesting to combine the approaches used by Lin & Hanson's Ratbot and the implemented critter to produce a learning method that uses neither domain-dependent knowledge nor a knowledge of control theory. The error signals would be defined as for the critter and then a neural-net version of Q learning would be used to learn the local control strategies based on those error signals. The control laws would be implemented as mappings from sensory inputs to motor control signals. If the sensory inputs include the error signals, their derivatives, and their integrals, then the set of control laws that can be defined in this way includes the PI and PD control laws used by the implemented critter.

As was mentioned in Section 7.5.1, the approach taken by the critter handles context-dependent effects at the feature level instead of the behavior level. This is important because the critter can learn the effects of motor control signals on multiple features simultaneously, whereas it is only possible to train one behavior at a time.

Rule-based approaches. The map-based approaches produce a discrete representation of an environment that is based on the finite-state automaton: places correspond to states and path-following behaviors to actions. An alternative approach is to model the behavior of the continuous

[4]The reinforcement-learning algorithm is a neural-network version of Q learning (Watkins 1989, Lin 1993).

[5]An earlier version of the critter actually did use reinforcement learning to learn homing behaviors (Pierce & Kuipers, 1991)

environment in terms of rules. This is analogous to Drescher's approach where state transitions are replaced by schemas — rules of the form *(context, action, result)*. This approach is taken by Shen & Simon (1989). Their LIVE system "is an extension of the GPS problem solving framework with a learning component that creates and learns rules through environmental exploration" (page 676). A key component of the system is the set of constructors used to produce new rules and features. Examples of these constructors are predicates ($=$, $>$, etc.), functions ($+$, $\times$, etc.), and logical connectives ($\wedge$, $\neg$, and $\exists$, etc.). The approach taken by Shen & Simon and other constructive inductionists (e.g., Matheus, 1991; Hiraki, 1994) is complementary to the approach presented in this dissertation and there is the possibility of a rich cross fertilization by combining the approaches.

The critter's learning capabilities could be extended using principles from constructive induction. Currently, the critter tries to understand its world in terms of a map of discrete places and paths. In the real world, there is a great deal of structure that is more appropriately described with rules (e.g., "If I flip the switch, the light goes on.") than with a map of places and paths.

Likewise, the constructive-induction learning systems could benefit by having a richer set of feature generators. Figure 9.2 illustrates a hierarchy of learned features culminating in the local-minimum features that were used by the critter as local state variables. For a robot with a rich, structured sensory system such as a ring of distance sensors, I believe that the feature generators used by the critter are more appropriate than arithmetical primitives and logical connectives. The latter are very general but lead to a combinatoric explosion of features. Ideally, a learning agent should have both powerful generators and general-purpose generators. Such a learning agent will require a good set of heuristics for deciding when each generator is appropriate.

9.6 Discussion

9.6.1 Learning in spatial environments

The critter has learned a discrete abstract interface that transforms a continuous environment into a discrete, finite-state environment. This is in contrast to the NX robot whose learned symbolic procedures corresponded to the primitives of the TOUR model. Since the TOUR model was specifically designed to represent structure in spatial worlds, it produces a more concise representation than does the critter's discrete abstract interface. For example, a distinctive place at a corner of a room is a single node in NX's topological map, whereas the same corner is represented as four distinct states for the critter, one for each of four directions that the robot can face. Thus a concise representation is sacrificed for the sake of generality. Perhaps this is just as well — the fact that a corner is represented as a single node is a reflection of the robot programmer's bias. From the critter's perspective, there may be no reason to view the corner as a single place. With an additional level of abstraction, it may be possible to have it both ways — to represent a corner as a single distinctive place at one level and four distinctive states at a lower level. The grouping of four states into one may be justified from the critter's perspective if, say, the effort required to move from one of the states to any of the others is small compared to the effort required to move from one corner to another. A hierarchical representation such as this is probably appropriate in any case. For example, if the NX robot lived in a larger environment, it would be to its advantage

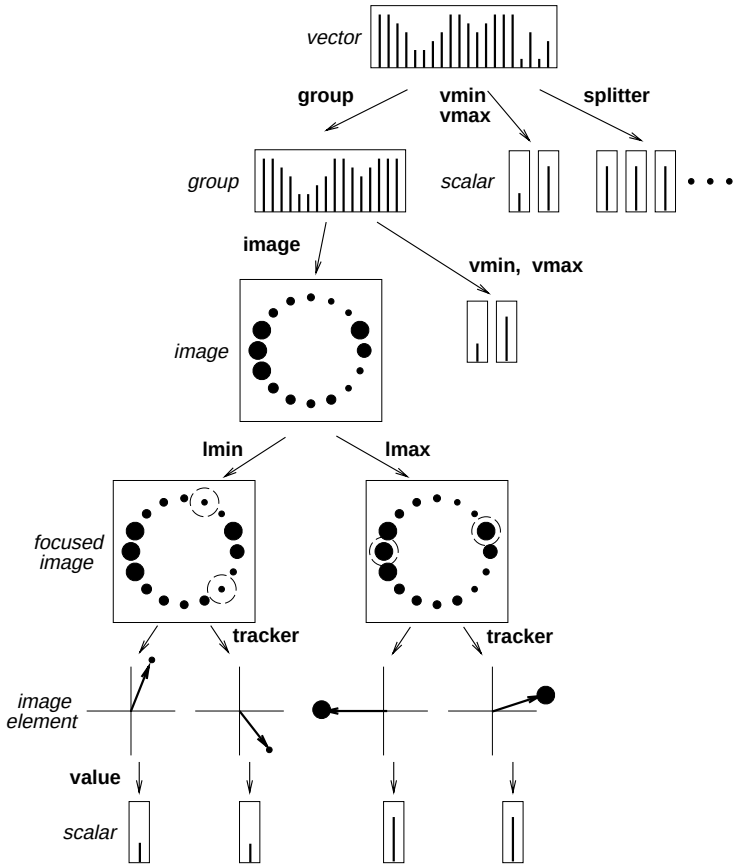


Figure 9.2: The hierarchy of features and generators in the critter's generate-and-test feature-learning process used to produce candidate local state variables.

to use a hierarchical topological map instead of a single one-level map with thousands of nodes in it.

9.6.2 Reinforcement learning

Reinforcement learning works very well for discrete state spaces with a small number of discrete actions (e.g., Sutton, 1990). When applied to continuous environments, the robot's action space is discretized (e.g., Barto et al., 1983; Lin, 1993). This dissertation presents an alternative approach — to learn general-purpose behaviors for navigation. With a set of such behaviors, reinforcement-learning techniques will be more effective. An infinite space of motor control vectors has been reduced to a small set of actions, and each action moves the robot through a much longer trajectory, making it easier to achieve rewards that require motion through large-scale space.

In fact, with the right set of behaviors and a topological map, reinforcement learning may not even be necessary. If a goal is associated with a node in the map, then planning can be used to immediately produce a shortest path to the goal. This point is relevant to the definition of Drescher's *composite actions* which are not trained but instead use a form of backward chaining.

9.7 Conclusions

9.7.1 Changes of representation

Each abstract interface that the critters learns provides a change of representation. At the sensorimotor level, the group and image feature generators exploit structure reflected in intersensor correlations to produce the image feature which is radically different from the raw sense vector. The learned set of primitive actions produces a new representation of the robot's motor capabilities that is grounded in sensory effects.

At the control level, behaviors and features are learned that are no longer purely egocentric. Whereas the primitive actions are grounded in sensory effects averaged over time, the definitions of the homing and path-following behaviors are grounded in the structure of the external environment as reflected by the current values of the local state variables.

At the procedural level, continuous state space is reduced to a one-dimensional space of states and trajectories that can be represented as a finite set of nodes and edges in a topological map.

9.7.2 Use of domain-independent knowledge

In the engineered solutions to the robot-mapping problem (Kuipers and Byun, Mataric, Kortenkamp et al.), specific knowledge of the robot's environment is used to define the control laws. Here, the critter uses domain-independent knowledge. The group and image generators are based on techniques from multivariate analysis and are applicable to any sensory array that detects structured information from an almost-everywhere continuous environment. The control laws are based on control-theoretic principles that make no assumptions about the details of the sensorimotor apparatus, only that each controlled feature's temporal derivative can locally be approximated as a linear function of the motor control signals. Moreover, this information need not be provided by the robot designer – it is discovered when the static action model is learned.

9.7.3 Summary

This dissertation has presented a method for learning a cognitive map of a continuous world in the absence of domain-dependent knowledge of the critter's sensorimotor apparatus or of the structure of its world. The learning methods are summarized below and in Figure 9.3. Chapter 4 showed how to use the **group** and **image** feature generators to learn a structural model of a sensory apparatus. They exploit the fact that, in a well engineered array of sensors sampling an almost-everywhere continuous property of the environment, the layout of the sensors may be reconstructed based on intersensor similarities. Chapter 5 showed how to use this structural knowledge to first define motion detectors and then use them to characterize the capabilities of a motor apparatus using a set of primitive actions, one for each of the robot's degrees of freedom. Chapter 6 showed how to recognize local state variables — scalar features whose derivatives can be approximated by context-dependent linear functions of the motor control signals. The effects of the primitive actions on the local state variables are captured by the static action model. Chapter 7 showed how to use the static action model to define homing and open-loop path-following behaviors, how to learn a dynamic action model to predict the effects of the primitive actions on the local state variables while open-loop path-following behaviors execute, and how to use the dynamic action model to define robust, closed-loop path-following behaviors. Finally, Chapter 9 has shown how to use the homing and path-following behaviors to define a discrete abstract interface that allows the critter to abstract its continuous world to a finite-state world.

By using the finite-state automaton as the target abstraction, the critter inherits a powerful set of methods for inferring the structure of its world. In the process of developing this abstraction, this work has produced methods for modeling a sensorimotor apparatus, for learning useful features, and for characterizing the effects of actions on features in two ways: The static action model captures first-order effects useful for defining homing behaviors and for deciding when an action leaves a feature invariant. The dynamic action model captures second-order effects useful for error correction in robust, path-following control laws.

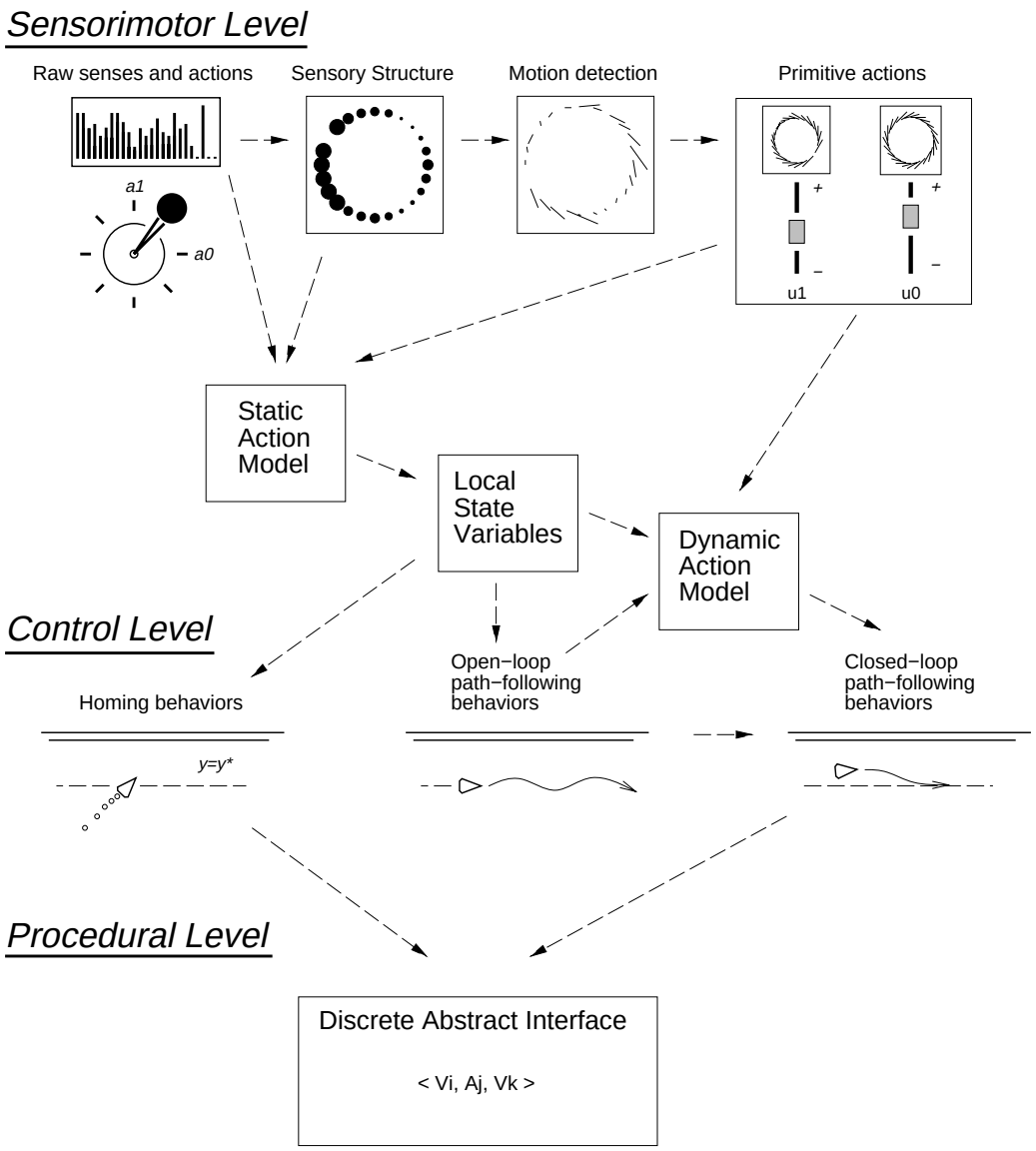


Figure 9.3: A graphical summary of the learning methods used in this dissertation, showing the objects learned at each of the first three levels of the spatial semantic hierarchy.

BIBLIOGRAPHY

[Angluin, 1978] Angluin, D. (1978). On the complexity of minimum inference of regular sets. *Information and Control*, 39:337–350.

[Angluin, 1987] Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106.

[Bachrach, 1992] Bachrach, J. R. (1992). *Connectionist Modeling and Control of Finite State Environments*. PhD thesis, University of Massachusetts.

[Barto et al., 1983] Barto, A. G., Sutton, R. S., and Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:834–846.

[Brooks, 1986] Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23.

[Cormen et al., 1990] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*. Cambridge, MA: MIT Press/McGraw-Hill Book Company.

[Dean et al., 1992] Dean, T., Angluin, D., Basye, K., Engelson, S., Kaelbling, L., Kokkevis, E., and Maron, O. (1992). Inferring finite automata with stochastic output functions and an application to map learning. In *Proceedings, Tenth National Conference on Artificial Intelligence*, pages 208–214. San Jose, CA: AAAI Press/MIT Press.

[Dean et al., 1993] Dean, T., Basye, K., and Kaelbling, L. (1993). Uncertainty in graph-based map learning. In Connell, J. H. and Mahadevan, S., editors, *Robot Learning*, pages 171–192. Boston: Kluwer Academic Publishers.

[Dewdney, 1987] Dewdney, A. K. (1987). Computer recreations: Diverse personalities search for social equilibrium at a computer party. *Scientific American*, pages 112–115.

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

[Dudek et al., 1991] Dudek, G., Jenkin, M., Milios, E., and Wilkes, D. (1991). Robotic exploration as graph construction. *IEEE Transactions on Robotics and Automation*, 7(6):859–865.

[Gold, 1978] Gold, E. M. (1978). Complexity of automaton identification from given data. *Information and Control*, 37:302–320.

[Hiraki, 1994] Hiraki, K. (1994). Abstraction of sensory-motor features. In *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Lawrence Erlbaum Associates.

[Horn, 1986] Horn, B. K. P. (1986). *Robot Vision*. Cambridge, MA: MIT Press.

[Jordan and Rumelhart, 1992] Jordan, M. I. and Rumelhart, D. E. (1992). Forward models: Supervised learning with a distal teacher. *Cognitive Science*, 16:307–354.

[Kortenkamp and Weymouth, 1994] Kortenkamp, D. and Weymouth, T. (1994). Topological mapping for mobile robots using a combination of sonar and vision sensing. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*.

[Krzanowski, 1988] Krzanowski, W. J. (1988). *Principles of Multivariate Analysis: A User's Perspective*. Oxford Statistical Science Series. Oxford: Clarendon Press.

[Kuipers, 1977] Kuipers, B. J. (1977). Representing knowledge of large-scale space. Tech. Report TR-418, MIT Artificial Intelligence Laboratory, Cambridge, MA. Doctoral thesis, MIT Mathematics Department.

[Kuipers, 1978] Kuipers, B. J. (1978). Modeling spatial knowledge. *Cognitive Science*, 2:129–153.

[Kuipers, 1988] Kuipers, B. J. (1988). The tour model: A theoretical definition. Tech. Report AI88-78, AI Laboratory, University of Texas at Austin, Austin, Texas.

[Kuipers and Byun, 1988] Kuipers, B. J. and Byun, Y.-T. (1988). A robust, qualitative method for robot spatial learning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-88)*, pages 774–779.

[Kuipers and Byun, 1991] Kuipers, B. J. and Byun, Y.-T. (1991). A robot exploration and mapping strategy based on a semantic hierarchy of spatial representations. *Journal of Robotics and Autonomous Systems*, 8:47–63.

[Kuipers and Levitt, 1988] Kuipers, B. J. and Levitt, T. S. (1988). Navigation and mapping in large-scale space. *AI Magazine*, 9(2):25–43.

[Kuo, 1982] Kuo, B. C. (1982). *Automatic Control Systems*. Englewood Cliffs, N.J.: Prentice-Hall, Inc. 4 edition.

[Lin, 1993] Lin, L.-J. (1993). *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA.

[Lin and Hanson, 1993] Lin, L.-J. and Hanson, S. J. (1993). On-line learning for indoor navigation: Preliminary results with RatBot. In *NIPS93 Robot Learning Workshop*.

[Mardia et al., 1979] Mardia, K. V., Kent, J. T., and Bibby, J. M. (1979). *Multivariate Analysis*. New York: Academic Press.

[Marr and Poggio, 1976] Marr, D. and Poggio, T. (1976). Cooperative computation of stereo disparity. *Science*, 194:283–287.

[Mataric, 1991] Mataric, M. J. (1991). Navigating with a rat brain: A neurobiologically-inspired model for robot spatial representation. In Meyer, J.-A. and Wilson, S. W., editors, *From Animals to Animats: Proceedings of The First International Conference on Simulation of Adaptive Behavior*, pages 169–175. Cambridge, MA: MIT Press/Bradford Books.

[Mataric, 1992] Mataric, M. J. (1992). Integration of representation into goal-driven behavior-based robots. *IEEE Transactions on Robotics and Automation*, 8(3):304–312.

[Matheus, 1991] Matheus, C. J. (1991). The need for constructive induction. In Birnbaum, L. A. and Collins, G. C., editors, *Machine Learning: Proceedings of the Eighth International Workshop (ML91)*, pages 173–177. San Mateo, CA: Morgan Kaufmann Publishers, Inc.

[Oja, 1982] Oja, E. (1982). A simplified neuron model as a principal component analyzer. *Journal of Mathematical Biology*, 15:267–273.

[Pierce and Kuipers, 1991] Pierce, D. and Kuipers, B. (1991). Learning hill-climbing functions as a strategy for generating behaviors in a mobile robot. In Meyer, J.-A. and Wilson, S. W., editors, *From Animals to Animats: Proceedings of The First International Conference on Simulation of Adaptive Behavior*, pages 327–336. Cambridge, MA: MIT Press/Bradford Books. Also University of Texas at Austin, AI Laboratory TR AI91-137.

[Press et al., 1988] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1988). *Numerical Recipes in C*. Cambridge University Press.

[Rivest and Schapire, 1987] Rivest, R. L. and Schapire, R. E. (1987). A new approach to unsupervised learning in deterministic environments. In Langley, P., editor, *Proceedings of the Fourth International Workshop on Machine Learning*, pages 364–375. Irvine, CA.

[Rivest and Schapire, 1993] Rivest, R. L. and Schapire, R. E. (1993). Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347.

[Rosenzweig and Leiman, 1982] Rosenzweig, M. R. and Leiman, A. L. (1982). *Physiological Psychology*. Lexington, Massachusetts: D. C. Heath and Company.

[Shen and Simon, 1989] Shen, W.-M. and Simon, H. A. (1989). Rule creation and rule learning through environmental exploration. In *Proceedings IJCAI-89*, pages 675–680.

[Sutton, 1990] Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In Porter, B. W. and Mooney, R. J., editors, *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224. Morgan Kaufmann Publishers, Inc.

[Watkins, 1989] Watkins, C. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge.

VITA

David Mark Pierce was born at a missionary hospital in Rwanguba, Zaïre (then Belgian Congo) on May 29, 1963. He is the son of Evelyn Joy and Donald Roy Pierce. Growing up in a missionary family gave him the opportunity to visit over 30 countries in Africa, Europe, and North and Central America. In 1975, his family returned to the United States permanently. After surviving an American high-school education at West Albany High School, David successfully pursued degrees in Electrical Engineering and Mathematics at Oregon State University, graduating with highest honors. His college education included a term abroad, in Avignon, France. In 1985, he entered graduate school at the University of Texas at Austin, supported by a three-year fellowship from the Office of Naval Research and several teaching and research assistantships. In 1988, he received the degree of Masters of Science in Computer Science. The highlight of his graduate career came in October of 1994 when he married Evelyn Tumlin. David has published papers in the Proceedings of The First International Conference on Simulation of Adaptive Behavior, the Proceedings of the 1991 IEEE International Conference on Robotics and Automation, the Proceedings of the Eighth International Workshop on Machine Learning, and the Proceedings of the Twelfth National Conference on Artificial Intelligence. David currently lives in Austin and works at Georgetown Systems.

Permanent address: 12166 Metric Blvd. # 270
Austin, TX 78758