A Framework for Programming Embedded Systems: Initial Design and Results

Sebastian Thrun

October 1998
CMU-CS-98-142

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes CES, a proto-type of a new programming language for robots and other embedded systems, equipped with sensors and actuators. CES contains two new ideas, currently not found in other programming languages: support of computing with uncertain information, and support of adaptation and teaching as a means of programming. These innovations facilitate the rapid development of software for embedded systems, as demonstrated by two mobile robot applications.

This research is sponsored in part by DARPA via AFMSC (contract number F04701-97-C-0022), TACOM (contract number DAAE07-98-C-L032), and Rome Labs (contract number F30602-98-2-0137). The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of DARPA, AFMSC, TACOM, Rome Labs, or the United States Government.

Keywords: Artificial intelligence, embedded system, machine learning, mobile robots, probabilistic reasoning, programming languages, robotics, teaching

1 Introduction

This paper introduces CES, a new language for programming robots. CES, which is short for *C for Embedded Systems*, supports the development of adaptable code: Instructing robots in CES interleaves phases of conventional programming and training, in which the code is improved through examples. CES also supports computation with uncertain information, which is often encountered in embedded systems.

To date, there exist two complementary methodologies for programming robots, which are usually pursued in isolation: conventional programming and learning, which includes teaching and trial-and-error learning. Undoubtedly, the vast majority of successful robots are programmed by hand, using procedural languages such as C, C++, or Java. Robots, their tasks, and their environments are usually complex. Thus, developing robotic software usually incurs substantial costs, and often combines coding, empirical evaluation, and analysis.

Recently, several researchers have successfully substituted inductive learning for conventional program development so that they could *train* their software to perform non-trivial tasks. For example, Pomerleau, in his ALVINN system, trained an artificial neural network to map camera images to steering directions for an autonomous land vehicle [83, 82]. After approximately 10 minutes of training, his system provided a remarkable level of skill in driving on various types of roads and under a wide range of conditions. Coding the same skill manually is difficult, as the work by Dickmanns and his colleagues has shown [23]. This example demonstrates that adaptable software, if used appropriately, may reduce the design time of robotic software substantially. In our own work, we recently employed neural networks for sensor interpretation and mapping tasks [105, 107], which, among other aspects, led to a mobile robot that successfully navigates in densely crowded public places [12]. As argued in [105], the use of neural networks led to a significant speed-up in software development; it also provided an enhanced level of flexibility in that the robot could easily be retrained to new conditions, as demonstrated at a recent AAAI mobile robot competition [11, 106].

The importance of learning in robotics has long been recognized. However, despite an enormous research effort in this direction, learning has had little impact on robotics. This is partially because most of the research on robot learning is focused on the design of general-purpose learning algorithms, which keep the amount of task-specific knowledge at a minimum. For example, virtually all robotics research on reinforcement learning (e.g., [1, 64, 61, 103, 16]) and evolutionary computation (e.g., [27, 62, 94]) seeks to establish algorithms that learn the entire mapping from sensors to actuators from scratch. Consequently, this field often resorts to narrow assumptions, such as full observability of the environment's state, or an abundance of training data. The current best demonstrations of reinforcement learning in robotics solve relatively simple tasks, such as collision avoidance, coordination of legged locomotion, or visual servoing.

This paper advocates the integration of conventional programming and learning. We argue that conventional programming and tabula rasa learning are just two ends of a spectrum, as shown in Figure 1. Both ends are ways of instructing robots, characterized by unique strengths and weaknesses. Conventional programming is currently the preferred way to make robots work, as it is often relatively straightforward to express complex structures and procedures in conventional program code. As the above examples suggest, however, certain aspects of robot software are

Programming $\longleftrightarrow$ **Teaching**

Figure 1: To data, there are two complimentary ways to instruct robots: conventional programming and teaching (learning). Most existing robots live close to one of the two extremes. This research seeks to integrate both, so that robots can be instructed using a mixture of programming and teaching.

easier to program through teaching and other means of learning. It is therefore a desirable goal to integrate both conventional programming and learning to find ways to develop better software with less effort.

This paper presents a proto-type of a new programming language, called CES, designed to facilitate the development of adaptable software for embedded systems. Programming in CES interleaves conventional code development and learning. CES allows programmers to leave "gaps" in their programs, in the form of function approximators, such as artificial neural networks [91]. To fill these gaps, the programmers train their program, by providing examples of the desired program behavior or by letting the program learn from trial-and-error. To bridge the gap between a program's behavior and the parameters of a function approximator, CES possesses a built-in credit assignment mechanism. This mechanism adjusts the parameters of the function approximators incrementally, so as to improve the program's performance.

CES differs from conventional programming languages in a second aspect relevant to embedded systems, in that it provides the programmer with the ability to compute with *uncertain information*. Such information often arises in robotics, since sensors are noisy and limited in scope, imposing intrinsic limitations on a robot's ability to sense the state of its environment. CES provides new data types for representing probability distributions. Under appropriate independence assumptions, computing with probability distributions is analogous to computing with conventional data types, with the only difference that CES's probabilistic variables may assume multiple values at the same time. The probabilistic nature of these variables provides robustness.

CES is an extension of C, a highly popular programming language. The choice to base CES on C seeks to retain the advantages of C while offering the concepts of adaptable software and probabilistic computation to programmers of embedded systems. Throughout this paper, we will assume that the reader is already familiar with C. The remainder of this paper describes the two major extensions in CES: probabilistic computation and learning. Both ideas are interrelated, as the particular learning mechanism in CES relies on the probabilistic nature of the variables. The paper also describes in some depth the development of an example program for a gesture-driven mail delivery robot, illustrating how conventional programming and teaching are closely integrated in CES. It also shows that by using CES's probabilistic constructs, sensor data can be processed in more robust (and more natural) ways. Finally, the paper briefly documents how an existing mobile robot localization algorithm, called BaLL, can be programmed in 58 lines, replacing a 5,000 line implementation in conventional C [104].

The reader should notice that CES is currently not implemented as described in this article, i.e., there exists no interpreter or compiler. The empirical result have been obtained with an implemented function library that is functionally equivalent to CES, but which differs syntactically. Thus, the results reported here should be viewed as a proof-of-concept only.

2 Probabilistic Computation in CES

This section describes CES's probabilistic data types, operators, and functions. The key idea for handling uncertainty is to allow variables to take on more than just one value, and to describe the probability of each value by a probability distribution. For example, a variable `close_to_obstacle` might simultaneously take on both values `yes` and `no`, each value being weighted by a numerical likelihood. Under appropriate independence assumptions, computing with probability distributions is analogous to computing with conventional values. Drawing on this analogy, this section describes the major probabilistic data types, operators and functions. Towards the end of this section, we will introduce three mechanisms that lack a direct analogy in the land of conventional programming: convolved data types, the `probloop` command, and the Bayes operator.

2.1 Probabilistic Data Types

CES uses methods from probability theory to represent and process uncertain information. Uncertain information is represented by a collection of new data types

```
probchar
probint
probfloat
```

which parallel existing data types in C: `char`, `int`, `float`. These new data types will be referred to as *probabilistic data types*. Notice that each numerical data type in C possesses a corresponding probabilistic data type in CES, called the *dual*.

These new data types are used to declare variables that represent *probability distributions* over values. For example, a variable declared `probint` specifies, for every possible integer value x, a probability that the value of the variable is x:

$$\begin{array}{l}\Pr(x=0)\\ \Pr(x=1)\\ \Pr(x=-1)\\ \Pr(x=2)\\ \Pr(x=-2)\\ \qquad\vdots\end{array}$$

According to Kolmogarov's axioms of probability, each of these values must be non-negative, and they must sum up to 1. These properties are guaranteed by the language.

There is a close correspondence between probabilistic data types and their (conventional) duals. Whereas conventional numerical data types are used to represent single values, probabilistic data types represent probability distributions over such values. One might think of the probabilistic data types as generalizations of conventional data types which enable a variable to take on multiple values at the same time. For example, if x is an integer with value 2, this corresponds to the special case of a probabilistic variable where $\Pr(\mathtt{x} = 2) = 1$ and all other values of x take on probabilities of 0. As will be shown below, there is a close correspondence between computing with values and computing with probability distributions over values. There also exist straightforward ways for merging conventional and probabilistic data.

Of course, representing probability distributions for probabilistic variables whose duals can take more than a handful of values can be extremely memory-intense. For example, more than $4 \cdot 10^9$ numbers are needed to specify an arbitrary probability distribution over all floating point values at 4 byte resolution. The statistical literature offers many compact representations, such as mixtures of Gaussians [22], piecewise constant functions [13], Monte-Carlo approximations [44, 50], trees [8, 71], and other variable-resolution methods [77]. In our current implementation all probability distributions are represented by piecewise constant density functions. The granularity of this function can be determined by the programmer, by setting the system-level variable `prob_dist_resolution`, whose default is 10.

2.2 Constants

CES offers a variety of ways to assign distributions to probabilistic variables. The statement

```
x = 2.4;
```

assigns a Dirac distribution to x whose probability is centered on 2.4, that is

$$\Pr(\mathtt{x}) = \begin{cases} 1 & \text{if } \mathtt{x} = 2.4 \\ 0 & \text{if } \mathtt{x} \neq 2.4 \end{cases} \tag{1}$$

Finite probability distributions can be specified through lists. Lists consist of tuples composed of a number (event) and its probability. For example, the assignment

```
x = { {1, 0.5}, {2, 0.3}, {10, 0.2} };
```

assigns the following distribution to the probabilistic variable x:

$$\Pr(\mathtt{x}) = \begin{cases} 0.5 & \text{if } \mathtt{x} = 1 \\ 0.3 & \text{if } \mathtt{x} = 2 \\ 0.2 & \text{if } \mathtt{x} = 10 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

CES possesses definitions for commonly used probability distributions. The statement

```
x = UNIFORM1D(0.0, 10.0);
```

initializes a probabilistic variable x with a one-dimensional uniform distribution over the interval $[0; 10]$. The statement

```
x = NORMAL1D(0.0, 1.0);
```

assigns to x a normal distribution with mean 0.0 and variance 1.0.

While the predefined constants in CES cover a large number of common distributions, certain distributions cannot be specified directly. As described in turn, distributions can be combined using various arithmetic operations. An alternative way for initializing probabilistic variables is the `probloop` command, which will be described further below.

2.3 Arithmetic Operations

Arithmetic with probabilistic data types is analogous to conventional arithmetic in C. For example, let us assume that `x`, `y` and `z` are three probabilistic variables of the type `probint`, and `x` and `y` possess the following distributions:

$$\Pr(\mathtt{x}) \;=\; \begin{cases} 0.5 & \text{if } \mathtt{x}=0 \\ 0.5 & \text{if } \mathtt{x}=3 \\ 0 & \text{otherwise} \end{cases} \qquad\qquad \Pr(\mathtt{y}) \;=\; \begin{cases} 0.1 & \text{if } 0 \le \mathtt{y} < 10 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Then the statement

```
z = x + y;
```

generates a new distribution, whose values are all possible sums of `x` and `y`, and whose probabilities are the products of the corresponding marginal probabilities:

$$\Pr(\mathtt{z}) \;=\; \begin{cases} 0.05 & \text{if } 0 \le \mathtt{z} < 3 \\ 0.1 & \text{if } 3 \le \mathtt{z} < 10 \\ 0.05 & \text{if } 10 \le \mathtt{z} < 13 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

$$\tag{5}$$

Thus, arithmetic operations are performed on the domain (e.g., the floating-point values), not on probability distributions.

It is important to notice that CES makes an implicit *independence assumption* between different right-hand operands of the assignment. More specifically, when computing `z`, CES assumes that `x` and `y` are stochastically independent of each other. The issue of independence in CES will be revisited in Section 2.7.

2.4 Type Conversion

Just as in C, CES provides mechanisms for type conversions. The most interesting conversions are between conventional and probabilistic variables. Suppose `x` is declared as a `float`, and `y` is declared as a `probfloat`. The statement

```
y = (probfloat) x;
```

assigns to `y` a Dirac distribution whose probability mass is centered on the value of `x`

$$\Pr(\mathtt{y}) \;=\; \begin{cases} 1 & \text{if } \mathtt{y}=\mathtt{x} \\ 0 & \text{if } \mathtt{y}\neq\mathtt{x} \end{cases} \tag{6}$$

The inverse statement,

```
x = (float) y;
```

assigns to `x` the *mean* of the distribution `y`. CES offers a collection of alternative functions that convert probabilistic variables to numerical values (floats):

```
mean( );
ml( );
median( );
variance( );
```

As the names suggest, `mean()` computes the mean, `ml()` the maximum likelihood value, `median()` the median, and `variance()` the variance of a probabilistic variable.

Probabilities of individual values of probabilistic variables (or ranges thereof) can be accessed by the library function `prob`. This function accepts a logical expression as input, and computes the probability of the expression for the probabilistic variable at hand. For example, the statement

```
p = prob(x < value);
```

assigns to `p` the probability that `x` is smaller than `value`. Here `p` must be of the type `float`, `x` must be a probabilistic variable and `value` must be its (non-probabilistic) dual.

2.5 Truncation and Inversion

Probabilistic truncation removes low-probability values from a probabilistic variable. Truncation is a library function in CES:

```
x = probtrunc(&y, bound);
```

Truncation first identifies the value with the largest probability in `y`. It then sets to zero all probabilities of values, whose current probability is smaller than `bound` times the largest probability in `y`. The `bound` is of the type `float` and should lie between 0 and 1. For example, the following code segment

```
probfloat x, y;
y = { {1, 0.5}, {2, 0.3}, {3, 0.1}, {4, 0.1} };
x = probtrunc(&y, 0.4);
```

generates the probability distribution

$$\Pr(\mathtt{x}) \quad = \quad \begin{cases} 0.625 & \text{if } \mathtt{x} = 1 \\ 0.375 & \text{if } \mathtt{x} = 2 \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

In this example, the largest probability in `y` is 0.5; thus, `probtrunc` removes all values whose probability is smaller than $0.4 \cdot 0.5 = 0.2$. In our example, the values are 3 and 4 are removed, since their probability is 0.1, which is smaller than 0.2. Normalization of the remaining probability values (for 1 and 2) leads to the distribution specified above.

Truncation is useful to remove low-likelihood values from future consideration, thereby speeding up the computation. In situations where most events are unlikely but not impossible, truncation can often reduce the computation time by several orders of magnitude while only marginally changing the result.

Another useful probabilistic operation is *inversion*. Let x be a probabilistic variable that represents some probability distribution

$$Pr(x) \tag{8}$$

Then the function

```
inverse(x);
```

computes the *inverse* of $Pr(x)$:

$$Pr(x)^{-1} \tag{9}$$

If $Pr(x) = 0$ for some x, then the inverse is undefined.

2.6 Probabilistic Arrays

In preparation for the experimental results discussed further below, let us briefly present a less obvious example: a CES program for averaging distributions over the same domain. Let Pr_i with $i = \{1, 4\}$ denote four different distributions over the same domain x. Then

$$Pr(x) \quad = \quad \frac{1}{4} \sum_{i=1}^{4} Pr_i(x) \tag{10}$$

is their average. In CES, averaging can be expressed as follows. Let

```
probfloat pri[4];
```

represent those four distributions. Then the code segment

```
probfloat pr;
probint   index = {{0, 0.25}, {1, 0.25}, {2, 0.25}, {3, 0.25}};

pr = pri[index];
```

assigns to `pr` the average of the four distributions `pri[]`.

2.7 Independence in CES

When computing with probabilistic variables, CES makes implicit independence assumptions between different probabilistic variables. Consider, for example, a statement of the type

```
z = x - y;
```

CES assumes that x and y are *independent*, that is, CES assumes that their joint distribution is the product of the marginal distributions:

$$\mathrm{Pr}(\mathtt{x}, \mathtt{y}) \quad = \quad \mathrm{Pr}(\mathtt{x})\, \mathrm{Pr}(\mathtt{y}) \tag{11}$$

It is important to notice that the independence assumption is necessary. Without it, results of statements like the one above are usually ill-defined. To demonstrate this point, let us assume that x and y are identically distributed:

$$\Pr(\mathtt{x} = i) \;=\; \Pr(\mathtt{y} = i) \;=\; \begin{cases} 0.5 & \text{if } i = 0 \\ 0.5 & \text{if } i = 1 \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

If x and y are independent,

$$\Pr(\mathtt{z}) \;=\; \begin{cases} 0.25 & \text{if } \mathtt{z} = -1 \\ 0.5 & \text{if } \mathtt{z} = 0 \\ 0.25 & \text{if } \mathtt{z} = 1 \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

but if $\mathtt{x} = \mathtt{y}$ (hence x and y are dependent),

$$\Pr(\mathtt{z}) \;=\; \begin{cases} 1 & \text{if } \mathtt{z} = 0 \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

The reader will quickly notice that (13) and (14) are not equivalent; thus, in the absence of the independence assumption (or a similar assumption) assignments in CES are not well-defined.

The independence assumption in CES, together with the fact that CES does not possess an inference mechanism of the type used in Bayes networks [81, 41], has important consequences. These might not appear obvious at first. Consider, for example, the following statement:

```
z = x - x;
```

If the initial conditions are as specified in (12), the result is the distribution given in (14). The slightly more complicated sequence of statements

```
y = x;
z = x - y;
```

however, leads to the distribution specified in (13). This is because when executing the second instruction, the variables x and y are assumed to be independent. Statements like the ones above specify assignments, not constraints on probability distributions (as in Bayes networks). Consequently, CES does not keep track of the implicit dependence between x and y arising from the first assignment when computing the second. The relation between CES and Bayes networks, a popular but quite different framework for computing with probabilistic information, will be discussed towards the end of this paper.

While the independence assumption is essential for computational efficiency, sometimes it is too strict. In the next two sections, mechanisms will be described that allow the programmer to explicitly maintain dependencies. One is called *compounding* and permits the creation of multi-dimensional probabilistic variables that describe the full joint distribution of more than one variable. Another is the *probloop* command, which makes it possible to trace dependencies correctly through sequences of statements.

2.8 Compounds

Compounds are data structures that enable the programmer to compute with multi-dimensional probability distributions. Their syntax is equivalent to that of `struct` in conventional C. The following declaration

```
compound {
  probfloat a, b;
  probint   c;
} x;
```

creates a variable `x` that models a three-dimensional probability distribution. The first two dimensions of `x` are real-valued, whereas the third is integer. The marginal distributions of compounded variables can be accessed by the dot-operator. For example, `x.b` refers to the marginal distribution of `x` projected onto `b`

$$\mathtt{x.b} \quad = \quad \Pr(\mathtt{x} = \mathtt{b}) \quad = \quad \int\int \Pr(\mathtt{x} = \langle \mathtt{a}, \mathtt{b}, \mathtt{c} \rangle)\, d\mathtt{a}\, d\mathtt{c} \tag{15}$$

Compounding variables results in allocating memory for the full joint distribution. It is generally advisable to keep the dimension of compounded variables small, as the size of the joint distribution space grows exponentially with its dimension (the number of probabilistic variables).

Compounded probabilistic variables can be used just like (one-dimensional) probabilistic variables, as long as all other operands are of the same type and dimension. Accessing the marginals requires additional computation, since they are not memorized explicitly.

2.9 The `probloop` Command

Often, it is necessary to access individual events covered by a probabilistic variable. The most powerful tool for processing probabilistic information is the `probloop` command. This command enables the programmer to handle probabilistic variables just like regular ones, by looping over all possible values.

The syntax of the `probloop` command is as follows:

`probloop`(*var-list-in*; *var-list-out*) *program-code*

where *var-list-in* and *var-list-out* are lists of probabilistic variables separated by commas, and *program-code* is regular CES code. Variables may appear in both lists, and either list may be empty.

The `probloop` command interprets the variables in *var-list-in* as "input" probability distributions. It executes the *program-code* with all combinations of values for the variables in this list, with the exception of those whose probabilities are zero. Inside the *program-code*, the types of all variables in *var-list-in* and *var-list-out* are converted to their non-probabilistic duals. The *program-code* can read values from the variables in *var-list-in*, and write values into probabilistic variables in *var-list-out*. For each iteration of the loop, CES caches two things: The probability of the combination of values (according to the probabilistic variables in the *var-list-in*), and the effect of the *program-code* on the probabilistic variables in *var-list-out*. From those, it constructs new probability distributions for all probabilistic variables in the *var-list-out*. The body of `probloop` command may not change the value of variables other than those listed in *var-list-out* or declared locally, inside the `probloop`.

The `probloop` command is best illustrated with an example. Consider the following program:

```
probint x, y, z;

x = {{1, 0.2},  {2, 0.8}};
y = {{10, 0.5}, {20, 0.5}};

probloop(x, y; x, z){
  if (10 * x - 1 > y)
    z = 1;
  else{
    z = 0;
    x = 5;
  }
}
```

Since `x` and `y` are specified in the *var-list-in*, the `probloop` instruction loops through all combinations of values for the variables `x` and `y`, with the exception of those whose probability is zero. There are exactly four such combinations: $\langle 1, 10\rangle$, $\langle 1, 20\rangle$, $\langle 2, 10\rangle$, and $\langle 2, 20\rangle$. For all those combinations, the *program-code* is executed and the result, which according to the *var-list-out* resides in `x` and `z`, is cached along with the probability assigned to values assigned to `x` and `y`:

$$\begin{array}{lllll} \mathtt{x}=1 & \mathtt{y}=10 & \longrightarrow & \mathtt{z}=0 & & \text{with probability } \Pr(\mathtt{x}=1,\mathtt{y}=10)=0.1 \\ \mathtt{x}=1 & \mathtt{y}=20 & \longrightarrow & \mathtt{z}=0 & & \text{with probability } \Pr(\mathtt{x}=1,\mathtt{y}=20)=0.1 \\ \mathtt{x}=2 & \mathtt{y}=10 & \longrightarrow & \mathtt{z}=1 & \mathtt{x}=5 & \text{with probability } \Pr(\mathtt{x}=2,\mathtt{y}=10)=0.4 \\ \mathtt{x}=2 & \mathtt{y}=20 & \longrightarrow & \mathtt{z}=0 & & \text{with probability } \Pr(\mathtt{x}=2,\mathtt{y}=20)=0.4 \end{array}$$

Upon completion of all iterations, the results are converted into a probability distribution for the variables mentioned in the *var-list-out*: `z` and `x`.

$$\Pr(\mathtt{z}) \;=\; \begin{cases} 0.6 & \text{if } \mathtt{z}=0 \\ 0.4 & \text{if } \mathtt{z}=1 \\ 0 & \text{otherwise} \end{cases} \qquad \Pr(\mathtt{x}) \;=\; \begin{cases} 1 & \text{if } \mathtt{x}=5 \\ 0 & \text{otherwise} \end{cases} \tag{16}$$

Notice that in this example, the probability of assigning a value to `x` is only 0.4. CES automatically normalizes the probabilities, so that each resulting probability distribution integrates to 1.

The `probloop` command can be applied to more complex constructs, such as loops and recursion. For example the following code segment

```
probint x, y;
int i;

x = {{1, 0.7}, {2, 0.3}};

probloop(x; y){
```

```
    y = 0;
    for (i = 0; i < x; i++)
      y = y + x + i;
}
```

generates the probability distribution

$$\Pr(y) = \begin{cases} 0.7 & \text{if } y = 1 \\ 0.3 & \text{if } y = 5 \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

Notice that in this example, the actual number of iterations `x` is a probabilistic variable. Thus, the number of iterations is varies, depending on the value of `x` inside the `probloop`.

The `probloop` command also provides a solution to the problem raised in Section 2.7. There, the side effect of CES's independence assumption in sequences of statements such as

```
y = x;
z = x - y;
```

was discussed. The following piece of code generates the result that appears to be intuitively correct and specified in (14):

```
probloop(x; y, z){
  y = x;
  z = x - y;
}
```

The `probloop` command enables the programmer to manipulate individual elements of probabilistic variables. Inside a `probloop`, CES keeps track of all implicit probabilistic dependencies arising from the variables specified in the *var-list-in*. The `probloop` command provides a sound way to use probabilistic variables in commands such as `for` loops, `while` loops, and `if-then-else`. Its major limitation lies in its computational complexity, which grows exponentially with the number of variables in *var-list-in*. If *var-list-in* contains variables of the type `probint` or `probfloat`, it is usually impossible to loop through all values. Here `probloop` samples the variable in predefined sampling intervals, as specified in the system-level variable `prob_dist_resolution`. The efficiency of the `probloop` command can be further increased by truncation, as described above.

The reader may notice that every single-line assignment is equivalent to a probloop command, in which probabilistic variables on the left hand-side appear in the *var-list-in*, and all probabilistic variables on the right hand side are in the *var-list-out* list. For example, the following two statements,

```
y = (x * z) - 2 * x + y;
```

and

```
probloop(x, y, z;  y)
  y = (x * z) - 2 * x + y;
```

are equivalent.

2.10 The Bayes Operator for Multiplying Distributions

CES features a special operator for integrating probabilistic variables, which does not possess a dual in conventional C. This operator is denoted # and called *Bayes operator*. It multiplies two or more probability distributions in the following way: Let y and z be two probabilistic variables that represent distributions (denoted $\Pr_{\mathtt{y}}$ and $\Pr_{\mathtt{z}}$) over the same domain. Then the statement

```
x = y # z;
```

assigns to x the product distribution

$$\Pr_{\mathtt{x}}(a) \quad = \quad \eta \Pr_{\mathtt{y}}(a) \Pr_{\mathtt{z}}(a) \tag{18}$$

for all a in the domain of y and z. Here η is a normalizer that ensures that the left-hand expression integrates to 1. If $\Pr_{\mathtt{y}}\Pr_{\mathtt{z}}(a) = 0$ for all a, the result of this statement is undefined.

The Bayes operator is useful for integrating conditionally independent sensor information using Bayes rule (hence the name). Suppose we want to estimate a quantity x, and suppose we have two different sources of evidence, y and z. For example, x could be the proximity of an obstacle to a mobile robot, y could be an estimate obtained from sonar sensors, and z the estimate obtained with a laser range finder. The statement

```
x = y # z;
```

integrates the probabilistic variables y and z (called: evidence variables) into a single distribution x, in the same way information is integrated in Kalman filters [47], dynamic belief networks [18, 92], and various other AI algorithms that deal with conditionally independent probability distributions.

Let us make this more formal. Suppose we want to estimate a quantity d from a set of n sensor readings, denoted $s_1, s_2, \ldots, s_n$. Now let us suppose we know already how to estimate d based on a single sensor datum, and the problem is to integrate the results from multiple sensor data into a single, consistent estimate of d.

In the language of probability theory, we are facing the problem of computing the conditional probability $\Pr(d|s_1, \ldots, s_n)$. Using Bayes rule, this probability can be expressed as

$$\Pr(d|s_1, \ldots, s_n) \quad = \quad \frac{\Pr(s_n|d, s_1, \ldots, s_{n-1}) \Pr(d|s_1, \ldots, s_{n-1})}{\Pr(s_n|s_1, \ldots, s_{n-1})} \tag{19}$$

Under the assumption that different sensor readings are *conditionally independent given* d, which is often referred to as the independence-of-noise assumption and which is written

$$\Pr(s_i|d, s_j) \quad = \quad \Pr(s_i|d) \qquad \text{for } i \neq j, \tag{20}$$

the desired probability can be expressed as

$$\Pr(d|s_1, \ldots, s_n) \quad = \quad \frac{\Pr(s_n|d) \Pr(d|s_1, \ldots, s_{n-1})}{\Pr(s_n|s_1, \ldots, s_{n-1})}. \tag{21}$$

The denominator does not depend d and hence is a constant. The desired expression can be rewritten as

$$\Pr(d|s_1, \ldots, s_n) \quad = \quad \alpha \Pr(s_n|d) \Pr(d|s_1, \ldots, s_{n-1}) \tag{22}$$

with an appropriate normalizer α. Suppose the probabilistic variables $\mathtt{x} = \Pr(d|s_1, \ldots, s_{n-1})$ and $\mathtt{y} = \Pr(s_n|d)$. Then the statement

```
x = x # y;
```

assigns to x the probability $\Pr(d|s_1, s_2)$.

Sometimes, one is given probabilistic evidence of the type $\Pr(d|s_i)$, instead of $\Pr(s_i|d)$ as assumed above. Applying Bayes rule to $\Pr(s_n|d)$ in Equation (22) yields

$$\Pr(d|s_1, \ldots, s_n) \quad = \quad \alpha \, \frac{\Pr(d|s_n) \, \Pr(s_n)}{\Pr(d)} \, \Pr(d|s_1, \ldots, s_{n-1}) \tag{23}$$

which, since $\Pr(s_n)$ does not depend on d, can be transformed to

$$\Pr(d|s_1, \ldots, s_n) \quad = \quad \beta \, \frac{\Pr(d|s_n)}{\Pr(d)} \, \Pr(d|s_1, \ldots, s_{n-1}) \tag{24}$$

with appropriate normalizer β. Induction over n yields

$$\Pr(d|s_1, \ldots, s_n) \quad = \quad \gamma \, \Pr(d) \, \prod_{i=1}^{n} \frac{\Pr(d|s_i)}{\Pr(d)} \tag{25}$$

with appropriate normalizer γ. Using the function `inverse`, the incremental update equation (24) can be realized using the Bayes operator:

```
x = x # y # inverse(z);
```

where x represents $\Pr(d|s_1, \ldots, s_{n-1})$, y represents $\Pr(d|s_n)$, and z represents the "prior" distribution $\Pr(d)$. If $\Pr(d)$ is uniform, this term can be omitted since it has no effect. $\Pr(d)$ can also be approximated using data, by averaging y as described in Section 2.9.

The # operator is specifically useful when integrating information over time. For example, suppose the subroutine `obstacle_proximity` computes a distribution over possible obstacle distances based on sensor data recorded by a mobile robot. Iterative application of the recursive assignment

```
dist = dist # obstacle_proximity(sensor_data);
```

computes the conditional probability of the obstacle distance, conditioned on all past sensor readings. Here the variables `dist` and the subroutine `obstacle_proximity` are both assumed to be of the type `probfloat`.

As noted above, the # operator is identical to the evidence integration step in Kalman filters [47], dynamic belief networks [18, 92], and various other approaches dealing with the integration of probabilistic information (e.g., [72, 18]). Using the #-operator to integrate probability distributions is only justified under a conditional independence assumption: y and z have to be *conditionally independent* given the true value of x. If y and z are both measurements of a variable x, then this assumption can be interpreted as an assumption that the *noise* when measuring x is independent across multiple measurements.

3 Learning in CES

Having described the probabilistic data types, operators, and functions in CES, we will now return to the issue of integrating conventional programming and teaching. Programming in CES is an

parameter name	type	default	description
`step_size`	float	0.01	step size for gradient descent
`momentum`	float	0.9	momentum
`params_init_range`	float	1.0	initial parameter range
`min_gradient, max_gradient`	float	–	bounds for gradient size (if defined)
`learning_flag`	int	1	learning flag

Table 1: Control parameters for function approximators in CES.

activity that interleaves *conventional code development* and *learning*. For example, when programming a mobile robot in CES, the programmer might start with a basic level of functionality, leaving certain parameterized "gaps" in his program. He might then train the robot by examples, thereby providing the information necessary to "fill" these gaps. Afterwards, the programmer might resume the code development and implement the next level of functionality, followed by additional training and programming phases.

The division of conventional programming and programming learning/teaching is flexible in CES, and typically depends on their relative difficulties and merits. However, programming and teaching are not symmetric, as programming must always precede teaching. CES's built-in learning mechanism is only capable of changing parameters of function approximators specified by the programmer. It does not modify or create program code directly.

3.1 Function Approximation in CES

CES possesses pre-defined, parameterized function approximators whose parameters can be modified based on examples. These function approximators are parameterized; their parameters are estimated when training a CES program.

The declaration

`fa` *fa-name* `();`

creates such a function approximator called *fa-name*, where *fa-name* adheres to the same conventions as function names in C. Function approximators possess three groups of parameters: (1) adjustable parameters, (2) user-definable control parameters, and (3) internal control parameters. The first group of parameters is modified automatically by CES when training a program. If the programmer chooses to use a neural network, these parameters are the weights and biases of the network. The second group of parameters are control parameters which can be set by the programmer. Currently, CES offers the control parameters listed in Table 1, which can be modified if the initial default parameters are inappropriate, using the command `faset`:

`faset(&`*fa-name*`,` *param-name*`,` *value*`);`

Here *fa-name* denotes of the name of the function approximator, *param-name* the name of the parameter according to Table 1, and *value* the desired parameter value. For example, the code segment

```
fa myfa;
faset(&myfa, step_size, 0.5);
```

sets the step size for the function approximator `myfa` to 0.5. The third group of parameters are internal control parameters which specify the input and output dimension of the function approximator and the nature of the representation (probabilistic or conventional). These parameters are configured initially, using the function `faconfigure`:

`faconfigure(` &*fa-name*, *type*, *input-dim*, *input-type*,
output-dim, *output-type*, *additional-params*);

Here `myfa` refers to the function approximator to be configured. The field *type* specifies the type of the function approximator. It may currently be one of the following: `NEURONET`, `RADIALBASIS`, `POLYNOMIAL`, or `LINEAR`. The dimensions of the input and output spaces are specified by the fields *input-dim* and *output-dim*. The fields *input-type* and *output-type* specify the types of these variables (which may have an impact on the number of parameters, as will be discussed below). Finally, the field *additional-params* may contain additional parameters for the specific function approximator, such as the number of hidden units in a neural network.

For example, the function call

```
faconfigure(&myfa, NEURONET, 3, FLOAT, 2, PROBFLOAT, 10);
```

configures `myfa` to be a neural network with 3 input, 2 output and 10 hidden units, where the inputs are conventional floats and the outputs are probfloats.

Function approximators must be configured before using them. Once a configured, a function approximator can be used just like a conventional function, e.g.:

```
y = myfa(x);
```

This statement uses the function approximator `myfa` to map `x` to `y`. The variables `x` and `y` can be vectors of floats or compounds of probfloats. In both cases, they must have the length/dimension specified in the `faconfigure` command. The outputs of function approximators are always in $[0;1]^{\textit{output-dim}}$.

3.2 Training

The parameters of function approximators are adjusted by minimizing the error between *actual* and *desired* values, using gradient descent. Desired values are set using the symbol “<–”, called the *training operator*. For example, the statement

```
y <- ytrue;
```

specifies that the desired value for the variable `y` is `ytrue` (at the current point of program execution). The training operator uses CES's built-in credit assignment mechanisms to change the parameters of all function approximators who contributed to `y` (and whose learning flag is set) by a small amount in the direction that minimizes the deviation (error) between `y` and `ytrue`.

The training operator permits the combination of different variable types (probabilistic and non-probabilistic), but the variables must share the same dimensionality. The induced error metric, which is the basis for the parameter change, depends on the variable types:

$$\begin{aligned}
E &= (\mathtt{y}-\mathtt{x})^2 && \text{if } \mathtt{x} \text{ and } \mathtt{y} \text{ non-probabilistic} \\
E &= \int (\mathtt{y}-\mathtt{x})^2 \Pr(\mathtt{x})\, d\mathtt{x} && \text{if } \mathtt{x} \text{ probabilistic, } \mathtt{y} \text{ non-probabilistic} \\
E &= \int (\mathtt{y}-\mathtt{x})^2 \Pr(\mathtt{y})\, d\mathtt{y} && \text{if } \mathtt{x} \text{ non-probabilistic, } \mathtt{y} \text{ probabilistic} \\
E &= \int\int (\mathtt{y}-\mathtt{x})^2 \Pr(\mathtt{x}) \Pr(\mathtt{y})\, d\mathtt{x}\, d\mathtt{y} && \text{if } \mathtt{x} \text{ and } \mathtt{y} \text{ probabilistic}
\end{aligned} \tag{26}$$

A key feature of CES is that the programmer does not have to provide target signals directly for the output of each function approximator. Instead, it suffices to provide target signals for some variable(s) whose values depend on the parameters of the function approximator. For example, the following code might, with appropriate training signals, instruct a mobile robot to turn parallel to a wall.

```
float     sonars[24];
probfloat turn, angle;
float     target_turn;
fa        mynet();

faconfigure(&mynet, NEURONET, 24, FLOAT, 1, PROBFLOAT, 10);
angle =  mynet(sonars) * M_PI;
turn  =  angle - (0.5 * M_PI);
turn  <- target_turn;
```

The programmer specifies, on an example-by-example basis, the amount and direction that the robot should turn to be parallel to a wall. Here we assume that this value is stored in the variable `target_turn`. Such target values are used to modify the parameters of `mynet`, thereby modifying the function that maps sonar measurements to `angle`. Here we assume that sonar scans are available in the variable `sonars`, and `M_PI` is the numerical constant π.

CES updates parameters by gradient descent. To solve the problem of credit assignment, every time a variable is updated CES also computes gradients of its value(s) with respect to all relevant function approximator parameters (e.g., weights of neural networks). More specifically, each value that depends on a function approximator is annotated by a gradient field that measures the dependence of this value on the parameters of this function approximator. The chain rule of differentiation enables CES to propagate gradients just like values (and probabilities). CES detects if a parameter influences the value of a variable more than once and sums up the corresponding gradients. When a training operator is encountered, the error is evaluated, its derivative is computed, and the chain rule applied to update the parameters of all contributing function approximators. This credit-assignment mechanism is a version of gradient descent, similar to the real-time Backpropagation algorithm [35, 112], where gradients are propagated through CES program code. Gradients are only propagated for variables whose `learning_flag` is set (c.f., Section 3.1).

3.3 The Importance of Probabilities for Learning

Probabilistic computation is a key enabling factor for the learning mechanism in CES. Conventional C code is usually not differentiable. Consider, for example, the statement

```
if (x > 0) y = 1; else y = 2;
```

where x is assumed to be of the type `float`. Obviously,

$$\frac{\partial \mathtt{y}}{\partial \mathtt{x}} = 0 \qquad \text{with probability 1.} \tag{27}$$

Consequently, program statements of this and similar types will, with probability 1, alter all gradients to zero, gradient descent will not change the parameters, and no learning will occur.

Fortunately, the picture changes if probabilistic variables are used. Suppose both x and y are of the type `probfloat`. Then the same statement becomes differentiable with non-zero gradients:

$$\frac{\partial \Pr(\mathtt{y}=1)}{\partial \Pr(\mathtt{x}=a)} = \begin{cases} 1 & \text{if } a > 0 \\ -1 & \text{if } a \le 0 \end{cases} \tag{28}$$

$$\frac{\partial \Pr(\mathtt{y}=2)}{\partial \Pr(\mathtt{x}=a)} = \begin{cases} 1 & \text{if } a \le 0 \\ -1 & \text{if } a > 0 \end{cases} \tag{29}$$

Notice that none of the gradients are zero. Probabilistic CES programs are essentially differentiable. This observation is crucial. The use of probabilistic computation is a necessary component of the learning approach in CES, not just an independent component of CES. Without it, the current credit assignment mechanisms would fail in most cases. In particular, CES's learning mechanism fails when conventional variables are used in conjunction with non-differentiable statements such as `if-then-else` (see the literature on automatic program differentiation [5, 37, 88] for alternatives).

3.4 Function Approximation with Probabilistic Variables

Usually, function approximators are not used for probabilistic variables; instead, their inputs and outputs correspond to conventional (non-probabilistic) variables. This section explains how function approximators are used for probabilistic variables in CES.

If the *input* to a function approximator is a probabilistic variable, the function approximator is run for every combination of input values (at a user-defined resolution), and the output is weighted by the probability of the input vector and averaged. This is similar, though not identical, to the `probloop` command.

If the *output* of a function approximator is probabilistic, the picture becomes more problematic. Function approximators output values, not probabilities; these outputs might not integrate to 1. CES solves this dilemma by converting outputs into inputs, and interpreting the (normalized) output of the function approximator as the desired probability. More specifically, suppose x is the input and y is the output of a function approximator. Let m be the dimension of x and n the dimension of y. CES considers the function approximator as a function from $\Re^{m+n}$ to $[0;1]$. The

(a) (b)

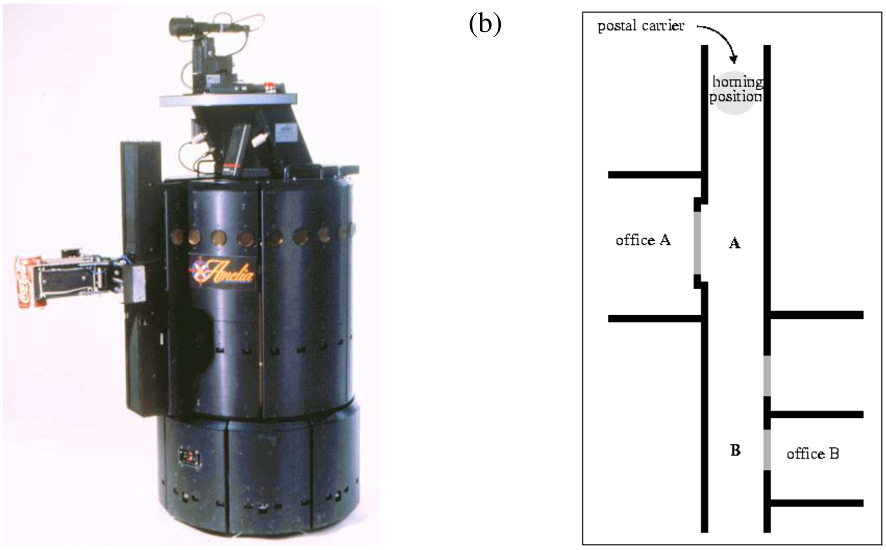

Figure 2: (a) AMELIA, the Real World Interface B21 robot used in our research. (b) Schematics of the robot's environment.

probability of y is given by

$$\Pr(\mathtt{y}|\mathtt{x}) = \frac{f(\mathtt{x},\mathtt{y})}{\int f(\mathtt{x},\bar{\mathtt{y}})\; d\bar{\mathtt{y}}} \tag{30}$$

Thus, to compute the distribution for y, CES loops over all possible values in y, just as if the input were probabilistic. The computation of the denominator is a side-product of computing $f(x, y)$ for every output value in y.

Just as in the `probloop` command, CES samples the function f at a user-specified resolution if probabilistic variables are involved. Once trained, the resolution can easily be changed to sample continuous-valued distributions at different granularities. In learning mode, gradients of the input variables with respect to other function approximator parameters, if any, are propagated through the function approximator using the chain rule of differentiation.

4 Programming a Mail Delivery Robot in CES

This section illustrates the development of an example program in CES. Its purpose is to demonstrate how robust software can be developed in CES with extremely little effort. Therefore, instead of just presenting the final result, emphasis is placed on describing the process of software development, which involves both conventional coding and training.

The robot is shown in Figure 2a. It is equipped with a color camera, an array of 24 sonar sensors, and wheel encoders for odometry. Odometry measurements are incremental, consisting of the relative change of heading direction (the *rotation*) and the distance traveled (the *translation*).

Figure 3: Positive (top row) and negative (bottom row) examples of gestures.

To keep the computational overhead manageable, camera images are automatically sub-sampled to a resolution of 10 by 10 pixels. The robot is controlled by directly setting its translational and rotational velocities.

The performance task, which will be programmed in this section, is the task of mail delivery in the office environment shown in Figure 2b. The task requires a collection of skills. When the robot does not carry any mail, it has to wait in a pre-specified location (called the *home position*) for the postal carrier to arrive. Every day, the carrier hands over mail to the robot for local delivery. Mail might be available for one or both of two possible destinations, A and B, as shown in Figure 2b. To inform the robot of the nature of the mail, the carrier instructs the robot using gestures: If mail has to be delivered to location A, he raises his left hand; If he wants the robot to go to location B, he raises his right hand; If mail is available for both locations, he raises both hands. Figure 3 shows examples of such gestures, along with some negative training examples. The robot then moves to the corresponding location(s), stops, and gives an acoustic signal, so that people in adjacent offices know that the robot is there and can pick up their mail. When all mail has been delivered, the robot returns to its home position. While in motion, the robot has to avoid collisions with obstacles such as walls and with people that might step in its way.

For the robot to perform this task, it has to be able to recognize gestures from camera images. It has to navigate to the target destinations and stop at the appropriate place. The environment is ambiguous. The only distinguishing feature is the door niche next to location A. This makes it difficult to recognize the other target locations and the homing position. In addition, the corridor is populated by people, which often corrupt sensor readings.

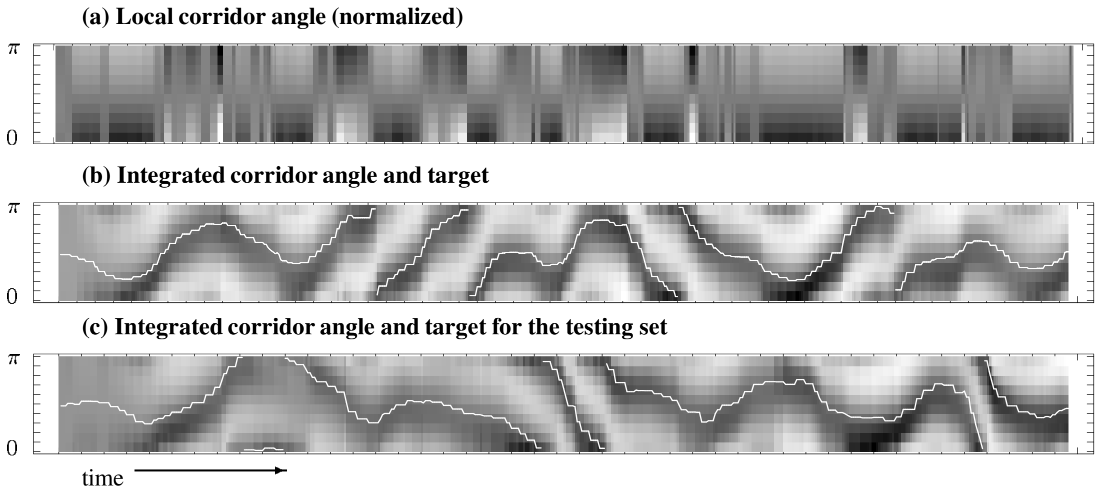

Figure 4: Estimating the corridor angle: Diagram (a) shows the local estimate, extracted from a single sensor reading. Diagram (b) shows the integrated corridor angle, obtained as described in the text. The solid white line in (b) depicts the labels with which the program is trained. Diagram (c) shows the performance over a testing set. In all diagrams, the horizontal axis corresponds to time, and the vertical axis to the angle of the corridor; the darker a value, the higher its likelihood.

Our program will exploit the fact that the robot operates on a single corridor and does not have to enter offices. To program the mail delivery robot in CES, we will first develop a localization routine. This routine recognizes the robot's x-y location and its heading direction in a global Cartesian coordinate frame based on sonar scans. We will then develop code for navigating to a goal location (specified in x-y-coordinates). Finally, we will develop software for recognizing gestures from camera images, along with a scheduler for coordinating the various required activities.

The reader should notice that all results reported here are based on a prototype implementation of CES as a function library. This prototype is functionally equivalent to the language described here, but it uses a somewhat different syntax, as declarations and operations involve function calls. Statements in our implemented version, however, can be translated one-by-one into CES.

4.1 Corridor Angle

Let us begin by developing code for recognizing one of the most obvious feature in the environment: the angle of the corridor relative to the robot. This angle, denoted α, lies in $[0; \pi]$. Due to the symmetry of the corridor, the corridor angle alone is insufficient to determine the global heading direction; however, knowledge of α is useful, as it reduces the space of heading directions to two possibilities, leaving open only which end of the corridor the robot is facing.

The following code, with the appropriate training, tracks the corridor angle α:

```
A-01:   fa          net_sonar();
A-02:   probfloat   alpha, alpha_local, prob_rotation;
A-03:   float       alpha_target;
A-04:   float       scan[24];
```

```
A-05:   struct     { float rotation, transl; } odometry_data;
A-06:
A-07:   alpha = UNIFORM1D(0.0, M_PI);
A-08:   faconfigure(&net_sonar, NEURONET, 24, FLOAT, 1, PROBFLOAT, 5);
A-09:
A-10:   for (;;){
A-11:     GET_SONAR(scan);
A-12:     alpha_local = net_sonar(scan) * M_PI;
A-13:     alpha = alpha # alpha_local;
A-14:
A-15:     GET_ODOM(&odometry_data);
A-16:     prob_rotation = (probfloat) odometry_data.rotation
A-17:        + NORMAL1D(0.0, 0.1 * fabs(odometry_data.rotation));
A-18:     alpha += prob_rotation;
A-19:     if (alpha <   0.0) alpha += M_PI;
A-20:     if (alpha >= M_PI) alpha -= M_PI;
A-21:
A-22:     GET_TARGET(&alpha_target);
A-23:     alpha <- alpha_target;
A-24:   }
```

Functions of the type `GET_xxx` are part of the robot application interface and will not be discussed any further. Line numbers have been added for the reader's convenience.

The most important variable is `alpha`, a probabilistic variable that keeps an up-to-date estimate of the corridor angle. In line `A-07`, `alpha` is initialized uniformly, indicating that initially the robot is unaware of its orientation. The angle `alpha` is modified upon two types of information: sonar scans and odometry. Upon querying its sonar sensors (line `A-11`), the robot uses a function approximator to convert a sensor scan into a "local" estimate of the corridor angle, called `alpha_local` (line `A-12`). In the code above, this function approximator is a neural network called `net_sonar`, with the topology specified in line `A-08`. Subsequently, in line `A-13`, the local estimate of α is integrated into `alpha` using the Bayes operator.

The robot's odometry is queried in line `A-15`. Naturally, a rotation by `odometry_data.rotation` causes the corridor angle to change by about the same amount. However, robot odometry is erroneous. To accommodate errors in the perceived rotation, line `A-16` converts the measurement into a probabilistic variable and adds a small Gaussian term. In line `A-18`, it adds its value to the current value of `alpha`. This addition reflects the programmer's knowledge that a rotation, measured by the robot's shaft encoders, causes the wall orientation to change accordingly. Since after executing the summation in line `A-18`, the new value of `alpha` might not lie any longer in $[0;\pi]$, lines `A-19` and `A-20` normalize `alpha` accordingly.

The program is trained using sequences of measurements (sonar and odometry), for which the corridor angle is labeled manually. Line `A-22` retrieves the label from the training database, and line `A-23` imposes the target signal for the estimate `alpha`. CES's built-in credit assignment mechanism modifies the parameters of the neural network `p_sonar` so as to maximize the accuracy of the variable `alpha`. Notice that training signals do not directly specify the output of the network; instead, they constrain the values of `alpha`, which is a function of the network's outputs.

We successfully trained the network with a 3-minute-long sequence of measurements, during which the robot was joy-sticked through the corridor. The dataset contained 317 measurements (sonar and odometry), which we labeled by hand in less than 10 minutes. Approximately half the data was used for training and the other half for cross-validation (early stopping). Figure 4a shows the value of `alpha_local` for the training set after training. Here the vertical axis corresponds to different values of `alpha_local`, the horizontal axis depicts time, and the grey-level visualizes the probability distribution: the darker a value, the higher its likelihood. The value of `alpha`, as computed in line `A-18`, is shown in Figure 4b. The solid white line in this figure corresponds to the labels. After an initial localization phase, the variable `alpha` tracks the angle well. Figure 4c shows the tracking performance using the cross-validation set, illustrating that the data is sufficient to training the program to track α.

It is interesting to notice that both sonar and odometry data are needed for tracking α. Without sonar data, the robot could never determine the initial angle; thus would be unable to track α. However, as Figure 4a illustrates, `alpha_local` does not produce accurate estimates of α; based on it alone, the robot would not be able to track α either. Thus, both sources of information are needed, along with the built-in geometric model that relates odometry to wall angle.

4.2 Heading Direction

Next, we will extend our program to compute the heading direction of the robot, called θ, which differs from the corridor angle in that it is defined over $[0; 2\pi]$, not just $[0; \pi]$ as is α. We will exploit the fact that

$$\alpha \quad = \quad \theta \, \mathrm{MOD} \, \pi, \tag{31}$$

that is, the corridor angle is the heading direction modulo the information regarding which end of the corridor the robot is facing. Because global localization in highly symmetric environments is challenging, we will make the assumption that initially the robot always faces the same end of the corridor.

The following pieces of code, inserted into the above program as specified, use `alpha_local` to compute `theta_local`, which in turn is used to compute an estimate `theta` of heading direction θ:

```
--- following A-05 ---
  B-01:   probfloat theta_local, theta;
  B-02:   probint   coin = {{0, 0.5}, {1. 0.5}};

--- following A-08 ---
  B-03:   theta = UNIFORM1D(0.0, M_PI);

--- following A-13 ---
  B-04:   probloop(alpha_local, coin; theta_local)
  B-05:     if (coin)
  B-06:       theta_local = alpha_local;
  B-07:     else
  B-08:       theta_local = alpha_local + M_PI;
  B-09:   theta = theta # theta_local;

--- following A-19 ---
  B-10:   theta += prob_rotation;
  B-11:   if (theta <  0.0)        theta += 2.0 * M_PI;
```

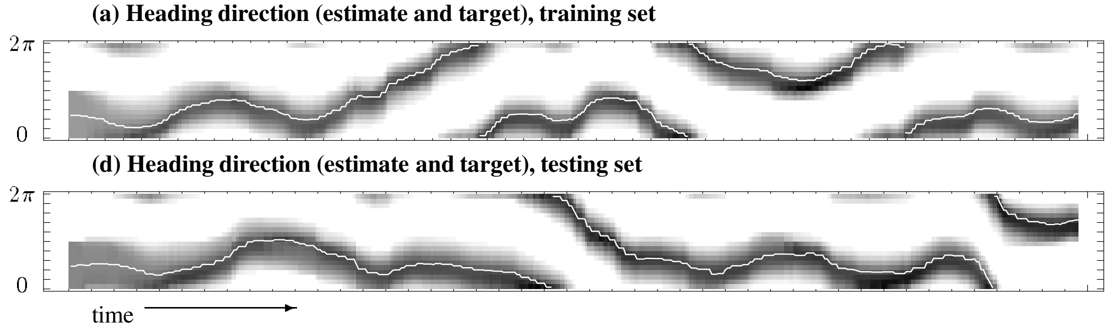

Figure 5: Plot of the heading direction (in $[0; 2\pi]$ over the training and the testing set. The robot accurately tracks the heading direction.

```
B-12:   if (theta >= 2.0 * M_PI) theta -= 2.0 * M_PI;
B-13:   theta = probtrunc(theta, 0.01);
```

The "trick" here lies in the probabilistic variable `coin`, which maps the probabilistic variable `alpha_local` from $[0; \pi]$ to $[0; 2\pi]$. More specifically, the `probloop` command (lines `B-04` to `B-09`) copies the probability in `alpha_local` probabilistically to `theta_local`, so that the distribution of `theta_local` in $[0; \pi]$ and in $[\pi; 2\pi]$ are both equal in shape to the distribution of `alpha_local`.

The sense of the global heading direction is obtained through appropriate initialization. Line `B-03` confines the actual heading direction to the initial interval $[0; \pi]$, thereby ruling out $[\pi; 2\pi]$. When updating `theta` (just like `alpha`), the robot considers only one of the two possible global heading directions—the other one is not considered since its initial likelihood is zero. To avoid the problem of the robot slowly losing its direction (an effect called *leaking*), the variable `theta` is truncated at regular intervals (line `B-13`). No further training is required at this point.

Figure 5 shows the new program in action. Plotted there is the heading direction θ for the dataset used above (training and cross validation run), annotated with the hand-labeled, global heading direction. In both cases, `theta` is initially confined to the interval $[0; \pi]$. The program quickly determines the heading direction and then tracks it accurately over the entire dataset. The traces in Figure 5 and a collection of other experiments, some of which lasted 30 minutes or more, suggest that the program is capable of determining and tracking the heading direction indefinitely—despite the fact that the environment is highly ambiguous and populated. We did not observe a single failure of the localization approach. Notice that only 37 lines of code were required, along with 13 minutes of data collection and labeling (and a few more minutes for function fitting).

4.3 Estimating x and y

In environments such as the one considered here, the two most informative sonar readings are the ones pointing west and east (c.f., Figure 2b). This is because sonar readings that hit a wall at a right angle maximize the chances of measuring the correct distance; whereas those hitting a wall

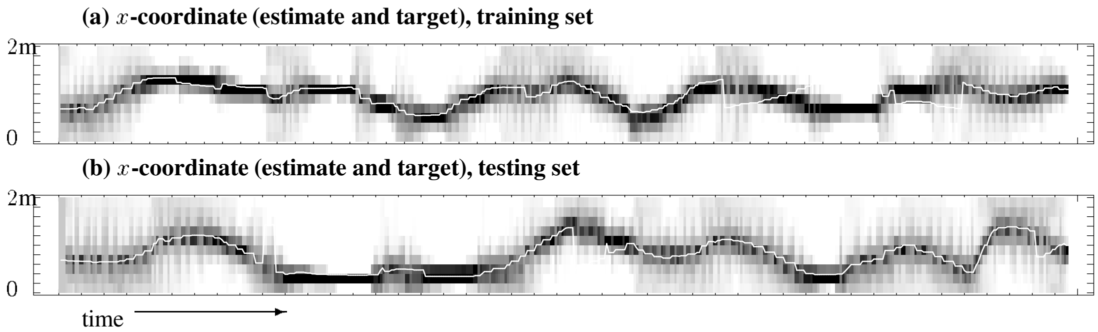

Figure 6: Estimating the x-coordinate. In both data sets, some of the data is partially mislabeled. Nevertheless, the program recovers an accurate estimate.

at a steep angle are often reflected away. To extract these sensor readings, one has to translate robot-centered coordinates to world-coordinates.

This operation is straightforward, as we have now an estimate of the robot's heading direction:

```
--- following B-03 ---
  C-01:   compound { probfloat east, west; } new_sonar;
  C-02:   int i, j;

--- following B-09 ---
  C-03:   probloop(theta; new_sonar){
  C-04:     i = (int) (theta / M_PI * 12.0);
  C-06:     j = (i + 12) % 24;
  C-07:     if (scan[i] < 300.0) new_sonar.east = scan[i];
  C-08:     if (scan[j] < 300.0) new_sonar.west = scan[j];
  C-09:   }
```

The two sonar readings extracted here are probabilistic variables, as the heading direction θ is not known exactly either. Notice that our loop filters out sonar readings more than 3 meters long, which in a 2.5 meters-wide corridor are bound to be specular reflections (and therefore uninformative).

With the new, world-centered sonar measurements it is now straightforward to design CES code for estimating the x and y-coordinates of the robot:

```
--- following C-02 ---
  D-01:   fa          net_x(), net_y();
  D-02:   probfloat   x, x_local, y, y_local, prob_transl;
  D-03:   float       x_target, y_target;

--- following A-08 ---
  D-04:   x = X_HOME; y = Y_HOME;
  D-05:   faconfigure(&net_x, NEURONET, 2, PROBFLOAT, 1, PROBFLOAT, 5);
  D-06:   faconfigure(&net_y, NEURONET, 2, PROBFLOAT, 1, PROBFLOAT, 5);

--- following C-10 ---
  D-07:   x_local = net_x(new_sonar);
  D-08:   y_local = net_y(new_sonar);
```

```
  D-09:   x = x # x_local;
  D-10:   y = y # y_local;

--- following B-13 ---
  D-11:   prob_transl = (probfloat) odometry_data.transl
  D-12:     + NORMAL1D(0.0, 0.1 * fabs(odometry_data.transl));
  D-13:   x = x + prob_transl * cos(theta);
  D-14:   y = y + prob_transl * sin(theta);
  D-15:   x = probtrunc(x, 0.01);
  D-16:   y = probtrunc(y, 0.01);

--- following A-23 ---
  D-17:   GET_TARGET(&x_target);
  D-18:   x <- x_target;
  D-19:   GET_TARGET(&y_target);
  D-20:   y <- y_target;
```

This code is largely analogous to the code for extracting the corridor angle. As there, we use neural networks to extract local x and y estimates from our newly computed sonar readings, using the same training set as above (but different labels). Local information is integrated using the Bayes operator. The robot is told its initial position, called `X_HOME` and `Y_HOME`.

While the estimation of y is largely based on odometry (the network `net_y` does not return much useful information), the estimation of x is close to the actual sensor readings. Figure 6 shows the estimation of x for the two runs, illustrating that our program can accurately track the robot's position.

4.4 Navigation to Goal Points

With our probabilistic estimates of where the robot is at any point in time, it is now straightforward to implement a function that makes the robot move to arbitrary target locations in the corridor. The following code segment, inserted as indicated, makes the robot move to arbitrary locations specified by the two variables `x_goal` and `y_goal`:

```
--- following D-03 ---
  E-01:   float x_goal, y_goal, t, v, theta_goal, theta_diff;
  E-02:   probfloat trans_vel, rot_vel;

--- following D-16 ---
  E-03:   probloop(theta, x, y, x_goal, y_goal; trans_vel, rot_vel){
  E-04:     theta_goal = atan2(y - y_goal, x - x_goal);
  E-05:     theta_diff = theta_goal - theta;
  E-06:     if (theta_diff < -M_PI) theta_diff += 2.0 * M_PI;
  E-07:     if (theta_diff >  M_PI) theta_diff -= 2.0 * M_PI;
  E-08:
  E-09:     if (theta_diff < 0.0)
  E-10:        rot_vel =  MAX_ROT_VEL;
  E-11:     else
  E-12:        rot_vel = -MAX_ROT_VEL;
  E-13:
  E-14:     if (fabs(theta_diff) > 0.25 * M_PI)
  E-15:        trans_vel = 0;
  E-16:     else
```

```
E-17:         trans_vel = MAX_TRANS_VEL;
E-18:     }
E-19:
E-20:     v = (float) rot_vel;
E-21:     t = (float) trans_vel;
E-22:     SET_VEL(t, v);
```

Here the function SET_VEL is used to set the robot's velocity, and the constants MAX_TRANS_VEL and MAX_ROT_VEL specify the maximum translational and rotational velocities.

In lines E-04 to E-07, this code segment computes (and stores in theta_diff) the difference between the robot's heading direction theta and the relative angle to the goal, theta_goal. It then implements a bang-bang controller: The robot always attempts to rotate full speed towards the goal, as specified in lines E-09 to E-11. If the deviation theta_diff is larger than 45 degrees in magnitude (line E-14), the robot does not move forward at all; otherwise, its translational velocity is set to its maximum value.

This code segment computes two probabilistic variables, trans_vel and rot_vel, which assign probabilities to either of their respective motion commands. These probabilistic variables are converted into conventional floats by assigning their likelihood-weighted means, as specified by the type conversions in lines E-20 and E-21. These values are fed to the robot's motors. The resulting controller is not a bang-bang controller; instead, it delivers smooth control whose magnitude depends on the degree as to which the above conditions are assumed to hold true (c.f., [28]). For example, if 50% of the probability in rot_vel suggests a rotational velocity of MAX_ROT_VEL and the other 50% suggests −MAX_ROT_VEL, as is often the case when the robot's heading direction is aligned with the goal, the likelihood-weighted average v will be 0, and the robot will not rotate at all.

The current code works well in empty hallways, but it does not avoid collisions with unexpected obstacles, such as humans. A simple-minded, reactive collision avoidance mechanism, which checks the two frontal sonar sensors and makes the robot stop is something comes too close, is easily designed by inserting the following code before the final motion command:

```
--- following E-20 ---
  F-01:   if (sonar[0] < 15.0 || sonar[23] < 15.0) t = 0.0;
```

This code makes the robot stop if an obstacle comes close. Since only the forward motion is disabled, the robot can still turn—preventing it from getting stuck when it is too close to a wall.

4.5 Gesture Interface

Gestures are recognized by the robot's cameras. In our application domain, the gesture interface must be robust to lighting changes, changes of the viewpoint, daylight changes, and variations in the postal carrier's clothes. The carrier is assumed to be cooperative, in that he poses himself at about the same distance to the camera, so that the hands appear at roughly the same position in the camera's field of view (see Figure 3).

The following, extremely brief piece of CES code turns out to suffice for recognizing gestures:

```
--- following E-02 ---
  G-01:   fa          net_left(), net_right();
  G-02:   float       image[300];
```

		actual gesture			
		none	left hand	right hand	both hands
recognized as	none	84	–	1	–
	left hand	2	14	–	–
	right hand	1	–	26	–
	both hands	–	–	–	10

Table 2: Gesture recognition results, measured on an independent testing set.

```
  G-03:   probint    gesture_left, gesture_right;

--- following D-06 ---
  G-04:   faconfigure(&net_left,  NEURONET, 300, PROBFLOAT, 1, PROBINT, 5);
  G-05:   faconfigure(&net_right, NEURONET, 300, PROBFLOAT, 1, PROBINT, 5);

--- following D-16 ---
  G-06:   GET_IMAGE(image);
  G-07:   gesture_left  = net_left(image);
  G-08:   gesture_right = net_right(image);

--- following D-20 ---
  G-09:   GET_TARGET(&target_left);
  G-10:   gesture_left <- target_left;
  G-11:   GET_TARGET(&target_right);
  G-12:   gesture_right <- target_right;
```

This code segment uses neural networks to map camera images into a probabilistic variable that indicates the likelihood that a gesture was shown. It is trained by labeled data.

We trained the code using a training set of 199 images, 115 of which are used for training and 84 for cross-validation (early stopping). This dataset was collected in approximately $3\frac{1}{2}$ minutes, and labeled in approximately 5 minutes. After training, gestures were recognized by thresholding the likelihood:

```
if ((float) gesture_left > 0.5)
   printf("Left hand up.\n");
if ((float) gesture_right > 0.5)
   printf("Right hand up.\n");
```

This interface yielded 100% accuracy on the training set, and 97.6% accuracy on the cross-validation set. These numbers have to be taken with a grain of salt, as both portions of the dataset participated in the training process. To validate these recognition rates, we collected and hand-labeled another dataset, consisting of 138 images. This dataset was collected on a different day, with the postal carrier wearing different clothes. The results obtained for this independent evaluation set, summarized in Table 2, confirm the high reliability of the gesture interface. Here the overall accuracy was 97.83%, with a false-positive rate of 3.45% and a false-negative rate of 1.96%. All false-positive cases were highly ambiguous, involving arm motion similar to the corresponding gesture.

4.6 Scheduling

Finally, a scheduler is required to coordinate the delivery requests and the final return to the home position. This is achieved by the following code, which is wrapped around the navigation code as indicated.

```
--- following G-03 ---
  H-01:   int        num_goals = 0, active_goal;
  H-02:   struct     { float x, y, dir; } stack[3];

--- following G-08 ---
  H-03:   if (num_goals == 0){                  /* nothing scheduled? */
  H-04:     if ((float) gesture_left > 0.5){    /* left hand gesture? */
  H-05:       stack[num_goals  ].x   = X_A;     /* then: schedule A */
  H-06:       stack[num_goals  ].y   = Y_A;
  H-07:       stack[num_goals++].dir = 1.0;
  H-08:     }
  H-09:     if ((float) gesture_right > 0.5){    /* right hand gesture? */
  H-10:       stack[num_goals  ].x   = X_B;      /* then: schedule B */
  H-11:       stack[num_goals  ].y   = Y_B;
  H-12:       stack[num_goals++].dir = 1.0;
  H-13:     }
  H-14:     if (num_goals > 0){                  /* any gesture? */
  H-15:       stack[num_goals  ].x   = X_HOME;   /* then: schedule return */
  H-16:       stack[num_goals  ].y   = Y_HOME;
  H-17:       stack[num_goals++].dir = -1.0;
  H-18:       active_goal = 0;                   /* start here */
  H-19:     }
  H-20:   }
  H-21:
  H-22:   else if (stack[active_goal].dir *      /* reached goal? */
  H-23:            ((float) y - stack[active_goal].y) > 0.0){
  H-24:     SET_VEL(0, 0);                       /* stop robot */
  H-25:     active_goal = (active_goal + 1) % depth;
  H-26:     if (active_goal)                     /* mail stop? */
  H-27:       for (HORN(); !GET_BUTTON(); );     /* blow horn and wait */
  H-28:     else
  H-29:       num_goals = 0;                     /* done, restart */
  H-30:   }
  H-31:
  H-32:   else{                                  /* approaching goal? */
  H-33:     x_goal = stack[active_goal].x;
  H-34:     y_goal = stack[active_goal].y;

--- following E-22 ---
  H-35:   }
```

This scheduler uses the variable `stack` to memorize a list of goal positions in response to a gesture. The statement in line `H-03` ensures that gestures are only accepted when the robot is not already delivering mail. In lines `H-04` to `H-19`, the robot checks whether a gesture has been spotted, and adds the corresponding destination into its stack, followed by the home position. If the robot is moving, it first checks whether a goal location has been reached. This is done in line `H-22`, which checks if the robot's x-coordinate has crossed the goal's x coordinate—the

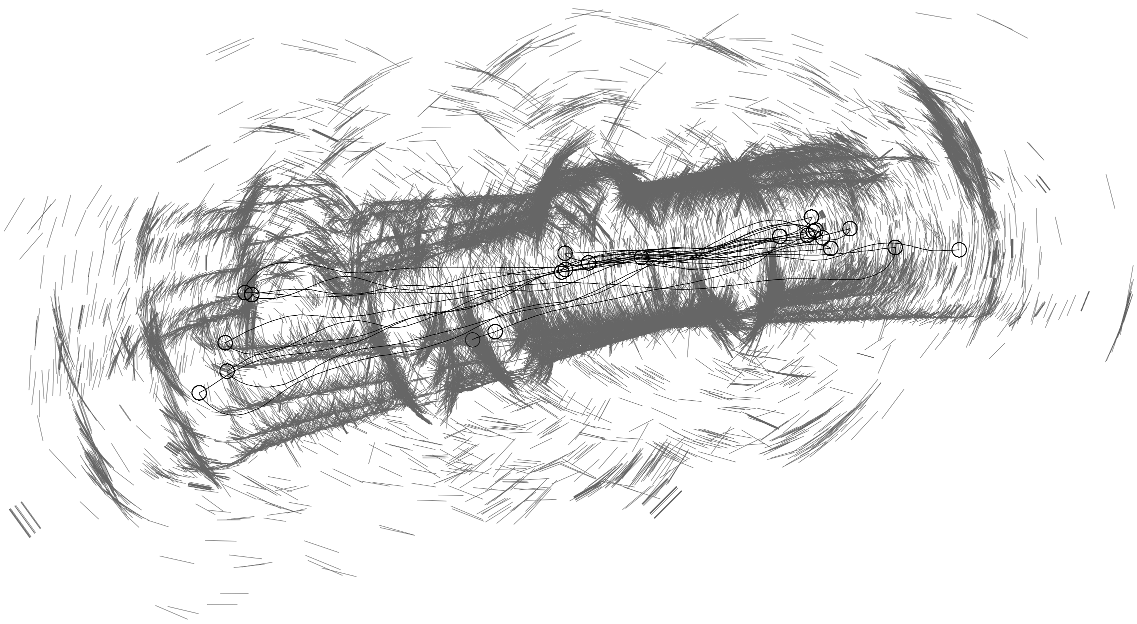

Figure 7: Plot of the robot trajectory (raw odometry) during 8 consecutive runs, in which AMELIA successfully delivers 11 pieces of mail. Shown here are also the raw sonar measurements. The robot reaches the various destination points within 1 meter accuracy, despite the rather significant error in the robot's odometry.

y-coordinate is ignored here. If a destination has been reached, the counter `active_goal` is incremented and, if the location is not the final stop (the home position), the horn is activated and the robot waits for a person to push a button (line `H-27`). Otherwise, it simply empties the stack (line `H-29`), at which point the delivery is completed. Finally, line `H-32` is activated when none of the conditions above are met, in which case the active goal is given to the navigation software for determining an appropriate motion command.

4.7 Results

Table 3 shows the complete CES program with minor reordering of the variable declarations. This program is only 144 lines long, but together with the appropriate training it suffices for the control of a gesture-driven mail delivery robot, all the way from raw sensor readings to motor controls.

In practice, the program proved extremely reliable when delivering mail in a populated corridor. Figure 7 shows raw data collected during eight delivery missions, during which AMELIA (correctly) delivered 11 pieces of mail. As the figure suggests, the raw odometry is too inaccurate to reliably track the robot's position. The figure also illustrates the noise in the sonar measurements, partially caused by total reflection, and partially caused by people walking close to the robot. Nevertheless, during this and other testing runs, the program tracked the position reliably (the error was always below 1 meter), and it successfully delivered the mail to the correct recipients.

```
main(){

  /*********** Declarations ***********/

  fa        net_sonar(), net_x(), net_y(), net_left(), net_right();
  probfloat alpha, alpha_local, prob_rotation;
  probfloat theta_local, theta, trans_vel, rot_vel;
  probfloat x, x_local, y, y_local, prob_transl;
  probint   coin = {{0, 0.5}, {1. 0.5}};
  probint   gesture_left, gesture_right;
  compound  { probfloat east, west; } new_sonar;
  float     alpha_target, scan[24], image[300];
  float     x_target, y_target, x_goal, y_goal, t, v;
  float     theta_goal, theta_diff;
  struct    { float rotation, transl; } odometry_data;
  struct    { float x, y, dir; } stack[3];
  int       i, j, num_goals = 0, active_goal;

  /*********** Initialization ***********/

  alpha = UNIFORM1D(0.0, M_PI);
  theta = UNIFORM1D(0.0, M_PI);
  faconfigure(&net_sonar, NEURONET, 24, FLOAT, 1, PROBFLOAT, 5);
  faconfigure(&net_x,     NEURONET, 2, PROBFLOAT, 1, PROBFLOAT, 5);
  faconfigure(&net_y,     NEURONET, 2, PROBFLOAT, 1, PROBFLOAT, 5);
  faconfigure(&net_left,  NEURONET, 300, PROBFLOAT, 1, PROBINT, 5);
  faconfigure(&net_right, NEURONET, 300, PROBFLOAT, 1, PROBINT, 5);
  x = X_HOME; y = Y_HOME;

  /*********** Main Loop ***********/

  for (;;){

    /*======== Localization ========*/

    GET_SONAR(scan);
    alpha_local = net_sonar(scan) * M_PI;
    alpha = alpha # alpha_local;
    probloop(alpha_local, coin; theta_local){
      if (coin)
        theta_local = alpha_local;
      else
        theta_local = alpha_local + M_PI;
      theta = theta # theta_local;
    probloop(theta; new_sonar){
      i = (int) (theta / M_PI * 12.0);
      j = (i + 12) % 24;
      if (scan[i] < 300.0) new_sonar.east = scan[i];
      if (scan[j] < 300.0) new_sonar.west = scan[j];
    }
    x_local = net_x(new_sonar);
    y_local = net_y(new_sonar);
    x = x # x_local;
    y = y # y_local;

    GET_ODOM(&odometry_data);
    prob_rotation = (probfloat) odometry_data.rotation
       + NORMAL1D(0.0, 0.1 * fabs(odometry_data.rotation));
    alpha += prob_rotation;
    if (alpha <   0.0) alpha += M_PI;
    if (alpha >= M_PI) alpha -= M_PI;
    theta += prob_rotation;
    if (theta <  0.0)        theta += 2.0 * M_PI;
    if (theta >= 2.0 * M_PI) theta -= 2.0 * M_PI;
    theta = probtrunc(theta, 0.01);
    prob_transl = (probfloat) odometry_data.transl
      + NORMAL1D(0.0, 0.1 * fabs(odometry_data.transl));
    x = x + prob_transl * cos(theta);
    y = y + prob_transl * sin(theta);
    x = probtrunc(x, 0.01);
    y = probtrunc(y, 0.01);

    /*======== Gesture Interface, Scheduler ========*/

    GET_IMAGE(image);
    gesture_left  = net_left(image);
    gesture_right = net_right(image);
    if (num_goals == 0){
      if ((float) gesture_left > 0.5){
        stack[num_goals  ].x   = X_A;
        stack[num_goals  ].y   = Y_A;
         stack[num_goals++].dir = 1.0;
      }
      if ((float) gesture_right > 0.5){
        stack[num_goals  ].x   = X_B;
        stack[num_goals  ].y   = Y_B;
        stack[num_goals++].dir = 1.0;
      }
      if (num_goals > 0){
        stack[num_goals  ].x   = X_HOME;
        stack[num_goals  ].y   = Y_HOME;
        stack[num_goals++].dir = -1.0;
        active_goal = 0;
      }
    }
    else if (stack[active_goal].dir *
             ((float) y - stack[active_goal].y) > 0.0){
      SET_VEL(0, 0);
      active_goal = (active_goal + 1) % depth;
      if (active_goal)
        for (HORN(); !GET_BUTTON(); );
      else
        num_goals = 0;
    }
    else{

      /*========== Navigation ==========*/

      x_goal = stack[active_goal].x;
      y_goal = stack[active_goal].y;
      probloop(theta, x, y, x_goal, y_goal;
               trans_vel, rot_vel){
        theta_goal = atan2(y - y_goal, x - x_goal);
        theta_diff = theta_goal - theta;
        if (theta_diff < -M_PI) theta_diff += 2.0 * M_PI;
        if (theta_diff >  M_PI) theta_diff -= 2.0 * M_PI;
        if (theta_diff < 0.0)
          rot_vel =  MAX_ROT_VEL;
        else
          rot_vel = -MAX_ROT_VEL;
        if (fabs(theta_diff) > 0.25 * M_PI)
          trans_vel = 0;
        else
          trans_vel = MAX_TRANS_VEL;
      }
      v = (float) rot_vel;
      t = (float) trans_vel;
      if (sonar[0] < 15.0 || sonar[23] < 15.0) t = 0.0;
      SET_VEL(t, v);
    }

    /*========== Training ==========*/

    GET_TARGET(&alpha_target);
    alpha <- alpha_target;
    GET_TARGET(&x_target);
    x <- x_target;
    GET_TARGET(&y_target);
    y <- y_target;
    GET_TARGET(&target_left);
    gesture_left <- target_left;
    GET_TARGET(&target_right);
    gesture_right <- target_right;
  }
}
```

Table 3: The complete CES implementation of the mail delivery program.

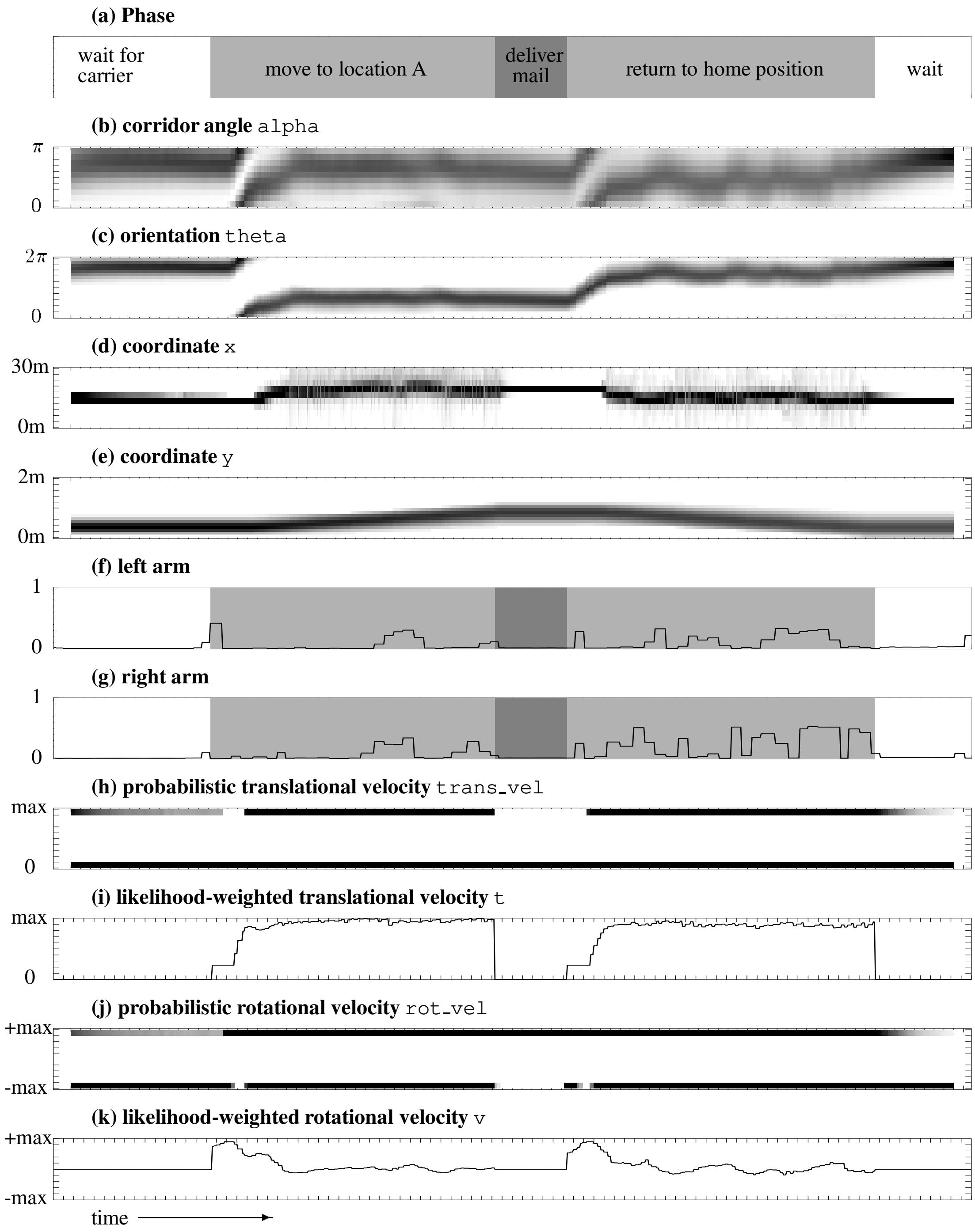

Figure 8: Plot of the key variables during a successful mail delivery. See text.

Figure 8 shows the major variables during a single mail delivery. In this example, the postal carrier lifts the left arm, the robot moves to location A, delivers its mail, and returns. In all diagrams, the horizontal axis corresponds to the time. Figure 8a illustrates the different phases involved: waiting for the carrier, moving to location A, delivering mail, moving back, and waiting. The mail delivery is triggered by the left arm gesture, as shown in Figure 8f. Figures 8c-e illustrate the position estimates, which accurately track the robot's position during that run. The velocity profile is shown in 8i&k, demonstrating that the control is rather smooth. When the robot is in motion, it moves at approximately 20 cm/sec.

5 CES Implementation of BaLL

Table 4 shows a CES implementation of the BaLL algorithm [104], a recent extension of the popular Markov localization algorithm [12, 13, 45, 49, 75, 97]. BaLL, which is short for Bayesian Landmark Learning, is a probabilistic algorithm for mobile robot localization. A key feature of the BaLL algorithm is its ability to select its own landmarks, and to learn functions for their recognition. It does this by minimizing the expected (Bayesian) error in localization. BaLL was originally implemented in C. An estimated 5,000 of its 13,000 lines of code were dedicated to the basic algorithm; the other ones are concerned with graphics and the robot interface. The implementation in Table 4 implements BaLL in 58 lines; a reduction by two orders of magnitude. 56 of these lines implement the basic Markov localization algorithm, and two the extension that enables BaLL to select its own landmarks.

Since the algorithm and its motivation is described in detail elsewhere [104], and since Markov localization generalizes the localization approach described in the previous section, we will only briefly describe it here. Markov localization maintains a probabilistic belief (distribution) of the robot's position. This belief is stored in the three-dimensional variable `pose` which is declared in line 14 and whose type is defined in line 5:

```
05:   typedef compound { probfloat x, y, theta; } pose_type;
14:   pose_type  pose, pose_prime;
```

The pose belief is updated upon two events: perception (something is observed) and robot motion.

1. **Perception.** Observations are compared to a map of the environment, to produce a momentary estimate as to where the robot might be (just based on the one observation). This momentary estimate is then incorporated into the robot's belief via Bayes rule—which is the mathematically correct way if the world is Markov [85].

   In the implementation shown in Table 4, the map is represented by a data set read from a file

   ```
   23:   LOAD_DATA(&data, &num_data, &min_x, &max_x, &min_y, &max_y);
   ```

   Here we assume that `LOAD_DATA` is a library function provided by the robot application interface. The reference map, which associates landmarks to x-y-θ positions, is initialized by running the landmark detecting network.

   ```
   24:   for (n = 0; n < num_data; n++){
   25:     data[n].landmark = p(data[n].sensor_data);
   ```

```
main(){
  /*********** Part 1: Declarations ***********/

  typedef struct   { float     x, y, theta; } target_type; /* pose targets */
  typedef compound { probfloat x, y, theta; } pose_type;   /* poses, x-y-theta space */
  struct {   int         new_episode_flag;                 /* 1, if new episode */
             int         episode_num;                      /* number of episode */
             float       sensor_data[164];                 /* sensor data */
             float       rotation, transl;                 /* motion command */
             target_type target;                           /* hand-labeled pose */
             probbool    landmark;          } *data;       /* landmark observation */
  int        num_data;                                     /* size of the data set */
  float      min_x, max_x, min_y, max_y;                   /* bounds on robot pose */
  pose_type  pose, pose_prime;                             /* pose estimates */
  probfloat  prob_rotation, prob_trans;                    /* motion command + noise */
  probbool   landmark;                                     /* landmark present? */
  probfloat  prob_transl, prob_rotation;                   /* motion estimate */
  fa         p();                                          /* network for landmarks. */

  /*********** Part 2: Initialization ***********/

  faconfigure(&p, NEURONET, 164, FLOAT, 1, PROBFLOAT, 5);  /* initialize network */
  LOAD_DATA(&data, &num_data, &min_x, &max_x, &min_y, &max_y);
  for (n = 0; n < num_data; n++)
    data[n].landmark = p(data[n].sensor_data);            /* initialize map */

  /*********** Part 3: Localization and Training ***********/

  for (;;)
    for (n = 0; n < num_data; n++){                       /* go over the data */
      if (data[n].new_episode_flag)                       /* new episode? */
        pose = UNIFORM3D(min_x, max_x, min_y, max_y,      /* then pose unknown */
                         0.0, 2.0 * M_PI);
      landmark = p(data[n].sensor_data);                  /* find landmarks */
      probloop(; pose_prime)
        for (k = 0; k < num_data; k++)                    /* estimate robot' pose */
          probloop(landmark, data[k].landmark; )
            if (data[n].episode_num != data[k].episode_num &&
                landmark == data[k].landmark){            /* ...by comparing it to */
              pose_prime.x     = data[k].target.x;        /* ...landmark vectors with */
              pose_prime.y     = data[k].target.y;        /* ...known poses */
              pose_prime.theta = data[k].target.theta;
            }
      pose = pose # pose_prime;                           /* integrate into estimate */
      pose = probtrunc(pose, 0.01);                       /* remove small probabilities */
      data[n].landmark = landmark;                        /* update data set (map) */
      prob_rotation   = (probfloat) data[n].rotation      /* incorporate control noise */
                  + NORMAL1D(0.0, 0.1 * fabs(data[n].rotation));
      prob_transl = (probfloat) data[n].transl
                  + NORMAL1D(0.0, 0.1 * fabs(data[n].transl));
      probloop(pose, prob_rotation, prob_transl; pose){   /* robot kinematics */
        pose.theta = (pose.theta + prob_rotation) % (2.0 * M_PI);
        pose.x     = pose.x + prob_transl * cos(pose.theta);
        pose.y     = pose.y + prob_transl * sin(pose.theta);
      }
      pose <- data[n].target;                             /* training, BaLL */
    }
}
```

Table 4: CES implementation of the BaLL mobile robot localization algorithm. This code is trained using sequences of sensor snapshots (camera, sonar) labeled with the position at which they were taken.

When a sensor datum is processed, a landmark vector is extracted and the momentary estimate is constructed, using the variable `pose_prime`:

```
34:   landmark = p(data[n].sensor_data);
35:   probloop(; pose_prime)
36:     for (k = 0; k < num_data; k++)
37:       probloop(landmark, data[k].landmark; )
38:         if (data[n].episode_num != data[k].episode_num &&
39:             landmark == data[k].landmark){
40:           pose_prime.x     = data[k].target.x;
41:           pose_prime.y     = data[k].target.y;
42:           pose_prime.theta = data[k].target.theta;
43:         }
```

Notice the nested `probloop` commands. The first is used to indicate that the variable `pose_prime` will be computed in the body of the `probloop` command. The second iterates over all landmark values for the actual observation (`landmark`) and the reference map (`data[k].landmark`).

The first condition in the `if`-clause (line 38) ensures that when constructing the momentary belief, the program does not use use data from the same episode. This is important for learning, as the program "simulates" a map built by independently collected data (see [104]). The second condition (line 39) checks the consistency of the landmark observations. To the extent that they are consistent, the momentary estimate `pose_prime` is updated accordingly. As a result, `pose_prime` contains a probability distribution for the robot's pose conditioned on the sensor data item.

After computing the momentary estimate `pose_prime`, it is integrated into the robot's belief using the Bayes operator:

```
44:   pose = pose # pose_prime;
```

The subsequent truncation command

```
45:   pose = probtrunc(pose, 0.01);
```

removes low-likelihood poses from future considerations. While it is not part of the basic Markov localization algorithm, it reduces the computational complexity by several orders of magnitude while altering the results only minimally [12].

2. **Motion.** To incorporate a motion command, it is first converted into a probabilistic variable that models the noise in robot motion. In the current program, this noise is described by a zero-centered Gaussian variable whose variance is proportional to the motion command:

```
47:   prob_rotation   = (probfloat) data[n].rotation
48:                   + NORMAL1D(0.0, 0.1 * fabs(data[n].rotation));
49:   prob_transl = (probfloat) data[n].transl
50:                   + NORMAL1D(0.0, 0.1 * fabs(data[n].transl));
```

Subsequently, the robot's pose is updated by convolving the previous belief with the motion command:

```
probloop(pose, prob_rotation, prob_transl; pose){
  pose.theta = (pose.theta + prob_rotation) % (2.0 * M_PI);
  pose.x     = pose.x + prob_transl * cos(pose.theta);
  pose.y     = pose.y + prob_transl * sin(pose.theta);
}
```

These equations are a probabilistic variant of the familiar kinematics of wheeled mobile robots like Amelia.

All code discussed thus far implements the basic Markov localization algorithm, using a neural network to extract landmark information from sensor readings, and a pre-recorded data set as reference map.

In CES, BaLL is a two-line extension of Markov localization: First, it uses CES's built-in learning mechanism to train the neural network so as to extract landmarks that minimize the localization error:

```
pose <- data[n].target;
```

Second, it continually updates the reference map:

```
data[n].landmark = landmark;
```

In [104], BaLL is evaluated by comparing it to other, popular localization algorithms. In particular, this paper compares the utility of learned landmarks with popular choices such as doors and ceiling lights. In all these comparisons BaLL performs favorably. It localizes the robot faster and maintains higher accuracy, due to the fact that it can learn its own, environment- and sensor-specific landmarks. The interesting aspect here is that in CES it can be implemented in 58 lines, and that in CES, that BaLL is a two-line modification of the basic Markov localization approach. For this example, the use of CES reduced several weeks of programming effort to just a few hours.

6 Related Work

Historically, the field of AI has largely adopted an inference-based problem solving perspective. Typical AI systems are programmed declaratively, and they rely on built-in inference mechanisms for computing the desired quantities. A typical example is the Prolog programming language [54], where programs are collection of Horn clauses, and a built-in logical inference mechanism (a theorem prover) is used to generate the program's output. Another popular example is Bayes networks [41, 81], where programmers specify probability distributions using a graphical language, and built-in probabilistic inference mechanisms are applied to marginalize them. To date, there exists a diverse variety of frameworks for knowledge representation (e.g., first order logic, influence diagrams, graphical models), along with a wide variety of "general-purpose" inference mechanisms, ranging from theorem provers and planners to probabilistic inference algorithms. CES differs from all this work in that it is a *procedural* programming language, not a declarative one. In CES, the program code specifies directly the computation involved in arriving at a result; thus, CES lacks a general-purpose inference mechanism of the type discussed above. In fields like robotics, procedural languages like C are by far the most popular programming tool. In comparison to declarative languages, procedural languages offer much tighter control over the program execution, they often enable programmers to arrive at more efficient solutions, and they also facilitate debugging during software development.

The issue of integrating learning into inference systems has been studied intensely before. For example, recent work on explanation-based learning [69, 42, 20], theory refinement [95, 108, 80, 78], and inductive logic programming [73, 86] has led to a variety of learning algorithms that modify programs written in first order logic based on examples. Several research teams have integrated such learning algorithms into problem solving architectures, such as SOAR [90, 26, 66, 57] PRODIGY [67, 39] and THEO [70]. These architectures all require declarative theories of the domain, using built-in theorem provers or special-purpose planners to generate control. Learning is applied to modify the domain theory in response to unexplained observations, or to speed up the reasoning process. In some systems, humans are observed to learn models of their problem solving strategies, in order to facilitate subsequent problem solving [109].

Despite several attempts (see e.g., [3, 48, 68]), such approaches have had little impact on robotics, for various reasons. First, inference mechanisms are often slow and their response characteristics are too unpredictable, making them inadequate for the control of real-time systems (as noted above). Second, these approaches are often inappropriate for perception—a major issue in robotics and embedded systems in general—since they lack the flexibility to robustly deal with noisy and high-dimensional sensor data. Third, the built-in learning mechanisms are often too brittle, restrictive, or data-intense to be useful in domains where data is noisy and costly to obtain. For example, explanation-based learning is often used to compile existing knowledge, not to add new knowledge [25]. Approaches that go beyond this limitation by including an inductive component [4, 80, 78, 89, 103] are often not robust to noise. Inductive logic programming increases the hypothesis space size with the amount of background knowledge, imposing intrinsic scaling limitations on the amount of background knowledge that may be provided. Logic-based learning algorithms are often brittle if data is noisy, the environment changes over time, and data spaces are high-dimensional.

As the results in this paper demonstrate, CES can successfully learn in the context of noise and high-dimensional sensor data while retaining the full advantages of procedural programming languages. It is common practice to program embedded systems using procedural programming languages, such as C or C++. From a machine learning point of view, program code in CES is analogous to domain theories in the AI architectures discussed above. CES's "domain theory" is procedural C code, which integrates the convenience of conventional programming with the advantages of adaptive mechanisms and mechanisms for handling uncertain information.

Probabilistic representations have proven to be useful across a variety of application domains. Recent work on Bayes networks [41, 81] and Markov chains [47, 87, 59, 46] has demonstrated, both on theoretical and practical ends, the usefulness of probabilistic representations in the real world. In robotics, integrating uncertain sensor information over time using Bayes rule is common practice. For example, most approaches to building occupancy grid maps, an approach to learning an environmental model which was originally proposed by Moravec and Elfes [29, 30, 72] and since applied in numerous successful robotic systems [6, 38, 113], employs update rules that are equivalent to the Bayes operator in CES. Markov localization, a probabilistic method for mobile robot localization that recently enjoyed enormous practical success [12, 13, 45, 49, 75, 97, 106], uses Bayes rule for integrating sensor information. Hidden Markov models [87], Kalman filters

[47, 34, 65, 111], and dynamic belief networks [18, 92] are other, successful approaches that employ Bayes rule in temporal domains. CES's Bayes operator supports these approaches. In fact, most of these algorithms can—at least in principle—be implemented much more efficiently in CES than in conventional programming languages.

In the tradition of AI, much of the work on probabilistic algorithms focuses on efficient inference and problem solving. For example, the Bayes network community has proposed convenient ways to compactly specify structured probability distributions, along with efficient inference mechanisms for marginalizing them [81]. Learning is typically applied to construct a probabilistic "domain theory," e.g., the Bayes network, from examples. Recognizing the analogy, some researchers proposed methods that bridge the gap between logic and probabilistic representations [40, 36, 52, 84].

CES differs from Bayes networks in about the same way as C differs from PROLOG. Bayes networks specify joint distributions of random variables in a way that facilitates computationally efficient marginalization. Thus, inference mechanisms for Bayes networks keep track of *all* dependencies between random variables. As a result, computing the marginal distributions can be computationally challenging (e.g., for a Bayes network with undirected cycles, see [81]). If in CES, assignments would be interpreted as constraints on the joint distribution of random variables, programs that contain cyclic dependencies, such as

```
y = NORMAL1D(x, 1.0);
z = UNIFORM1D(-x, x);
a = y + z;
```

would be similarly difficult to compute. Program statements in CES are computational rules for manipulating data, not mathematical constraints on probability distributions. Just as in C, statements such as `x = y;` and `y = x;` have fundamentally different effects. CES's built-in independence assumption ensures the efficiency of execution and therefore the scalability to very large programs. It provides loops, if-then-else statements and recursion, currently not available in Bayes networks. It also facilitates the integration of probabilistic reasoning into mainstream programming, as it smoothly blends probabilistic and conventional representations. However, these advantages come with limitations. Just as in C, one cannot present the output of a CES program and ask for a distribution over its inputs—an operation supported by Bayes networks under the name of "diagnostic inference."

The ability to generate distributions procedurally by sequences of assignments in CES is similar in spirit to a recent proposal by Koller [51], who proposed a language for defining complex probability distributions. Her language, however, is exclusively tailored towards approximate probabilistic inference, and is therefore not suited as general-purpose programming language.

In the field of robotics, researchers have proposed alternative languages and methodologies for programming robots. None of these approaches integrates learning at the architectural level, and none supports computation with uncertain information. For example, Brooks's popular subsumption architecture [9, 10] provides a modular way for programming robots, by coupling together finite state machines that map sensor readings more or less directly into motor commands. Unfortunately, this approach does not address the uncertainty typically arising in robotic domains, and as a consequence it fails to provide adequate mechanisms for dealing with sensor limitations and unobservable state. As a result, robots programmed using the subsumption architecture are typically reactive, that is, their behavior is a function of the most recent sensor readings. In environments such as the ones considered here, dealing with perceptual noise and maintaining internal state is essential. In addition, the subsumption architecture does not support adaptation—even though some researchers successfully implemented adaptive mechanisms on top of it [61, 60, 63]. Other researchers have proposed more flexible programming languages for task-level robot control, providing specialized mechanisms that support concurrency, exception handling, resource management, and synchronization [31, 33, 53, 96, 98]. These languages address certain aspects that arise when interacting with complex, dynamic environments—such as unexpected conditions that might force a robot to deviate from a previously generated plan—but they do not address the uncertainty in robotic perception. In fact, they typically assume that all important events can be detected with sufficient certainty. Programming in these languages does not include learning phases.

As argued in the introduction, the vast majority of the research in the field of robot learning focuses on tabula rasa learning methods. In particular, approaches like reinforcement learning [2, 46, 100, 110] and evolutionary computation/genetic programming [55, 56] currently lack the necessary flexibility to integrate prior knowledge, and therefore are subject to scaling limitations, especially when training data is scarce. In reinforcement learning, for example, common ways to insert knowledge include choice of input representations, the type of function approximator used for generalization [7, 99, 102], and ways to decompose the controllers hierarchically [17, 24, 58, 79]. Similarly, genetic programming gives users the choice of the data representations, the building blocks of the programs that evolve, and the genetic operators used in their search [55, 56, 101]. In robotics, programmers often posses knowledge that cannot be expressed easily in these terms, such as knowledge of the performance task, the environment, or generic knowledge such as the laws of physics or geometry. The inability to integrate such knowledge into learning makes it difficult for these approaches to learn complex controllers from limited amounts of data. CES critically departs from this line of thought, in that it adopts a powerful (and commonly accepted) method for programming robots with the benefits of learning.

Currently, CES's built-in learning mechanism is less powerful than reinforcement learning and genetic programming, in that CES programs cannot learn from delayed penalty and reward; instead, they require target signals, very much like supervised learning. CES's learning component differs from genetic programming in that it does not manipulate program code. In principal, genetic programming can easily be applied to CES programs. Practical experience shows, however, that humans find it difficult to understand machine-generated program code, even for very simple problems [56].

7 Discussion

This paper described CES, a new programming language designed for programming robots and other sensor-based systems. CES is an extension of C, retaining C's full functionality but providing additional features. To accommodate existing difficulties in developing robotic software, CES offers its programmers the option to *teach* their code. CES programmers can use function approximators in their program, and teach them by providing examples. CES's built-in credit assignment mechanism allows programmers to provide training signals for arbitrary variables (e.g., the program output). In addition, CES provides mechanisms to adequately deal with the uncertainty, which naturally arises in any system that interacts with the real world. The idea of probabilistic data types makes programming with uncertain information analogous to programming with conventional data types, with the added benefit of increased robustness and performance.

To demonstrate the usefulness of these concepts in practice, this paper described the programming of a gesture-driven mobile mail delivery robot. A short CES program (144 lines), along with less than an hour of training, was demonstrated to control a mobile robot highly reliably when delivering mail and interacting with a postal carrier in a populated corridor. Comparable programs in conventional programming languages are typically orders of magnitude larger, requiring much higher development costs. To demonstrate this point, this paper showed that a CES implementation of a state-of-the-art localization algorithm was two orders of magnitude more compact than a previous implementation in C.

Our current implementation of CES possesses several limitations that warrant future research:

- We currently lack a suitable interpreter or compiler for CES. In fact, all our experiments were carried out using a C library, functionally equivalent to CES but not syntactically. This limitation is purely a limitation of the current implementation, and not a conceptual difficulty, as the syntax of the language is well-defined.

- Our current implementation uses piecewise constant functions for the representation of probability distributions. Such representations suffer several limitations. Their size scales exponentially with the dimension of compounded variables, making it infeasible to compute in high-dimensional spaces. They are unable to represent finite distributions exactly, such as the outcomes of tossing a coin. They also suffer from an inflexible assignment of resources (memory and computation); mechanisms that place resources where needed (e.g., in regions with high likelihood) would be advantageous. The use of piecewise constant representations is not a limitation of the language per se; it is only a shortcoming of our current implementation. Several other options exist, such as such as mixtures of Gaussians [22], Monte-Carlo approximations [21, 44, 50], and variable-resolution methods such as trees [8, 71, 77]. Of particular interest are resource-adaptive algorithms which can adapt their resource consumptions in accordance with the available resources [19]. Probabilistic representations facilitate the design of resource-adaptive mechanisms by selectively focusing computation on high-likelihood cases.

- As noticed above, CES's learning mechanism is restricted to cases where labeled data is available. While in all examples given in this paper, these labels were generated manually, labels can also be generated automatically. For example, for learning to predict upcoming collisions, a robot might wander around randomly and use its tactile sensors to label the data. Not addressed by CES, however, is the issue of *delayed* reward, as typically addressed in the reinforcement learning literature. Augmenting CES with a learning component that can learn control from delayed reward is a subject for future work. Also not addressed is learning from unlabeled data. Recent research, carried out in domains such as information retrieval, has demonstrated the utility of unlabeled data when learning from labeled data [14, 15, 74, 76]. In principle, unlabeled data can be collected in everyday operation, and it could be used to further train CES's functions approximators. To what extent such an approach can improve the performance of a CES program remains to be found out.

Despite these opportunities for future research, CES in its current form is already well-suited for a wide range of robotic tasks, as demonstrated by the experimental results in this paper.

The true goal of this research, however, is to change the current practice of computer programming, for embedded systems and beyond. At present, instructing computers focuses narrowly on conventional programming, where keyboards are used to instruct robots. People, in comparison, are instructed through much richer means, involving teaching, demonstrating, explaining, answering questions, letting them learn through trial-and-error, and so on. All these methods of instruction possess unique strengths and weaknesses, and much can be gained by combing them. There is no reason why we should not teach our programs, instead of just programming them. CES goes a step in this direction, by providing mechanisms for writing adaptable software that can improve based by learning from examples. We hope that this paper stimulates further research in this direction, as the space of possible learning languages is huge and barely explored.

8 Acknowledgment

This paper benefited greatly from a continuing discussion with Frank Pfenning, which is gratefully acknowledged. The author also thanks Tom Dietterich and Tom Mitchell for insightful and detailed comments on earlier drafts of this paper. He gladly acknowledges various stimulating discussions with Craig Boutilier, Tom Dietterich, Tom Mitchell, Andrew W. Moore, Daphne Koller, Ray Reiter, Nicholas Roy, Stuart Russell, and various members of the Robot Learning Lab at CMU.

References

[1] M. Asada, S. Noda, S. Tawaratsumita, and K. Hosoda. Purposive bevahior acquisition for a real robot by vision-based reinforcement learning. *Machine Learning*, 23, 1996.

[2] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

[3] S. W. Bennett and G. F. DeJong. Real-world robotics: Learning to plan for robust execution. *Machine Learning*, 23(2/3):5–46, 1996.

[4] F. Bergadano and A. Giordana. Guiding induction with domain theories. In *Machine Learning Vol. 3*, pages 474–492. Morgan Kaufmann, San Mateo, CA, 1990.

[5] C. Bischof, L. Roh, and A. Mauer-Oats. ADIC: An extensible automatic differentiation tool for ansi-c. Technical Report ANL/MCS-P626-1196, Argonne National Laboratory, Argonne, IL, 1998.

[6] J. Borenstein and Y. Koren. The vector field histogram – fast obstacle avoidance for mobile robots. *IEEE Journal of Robotics and Automation*, 7(3):278–288, June 1991.

[7] J. A. Boyan. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, Cambridge, MA, 1995. MIT Press.

[8] L. Breiman, J. H. Friedman, R. A. Ohlsen, and C. J. Stone. Classification and regression trees. 1984.

[9] R. A. Brooks. A robot that walks; emergent behaviors from a carefully evolved network. *Neural Computation*, 1(2):253, 1989.

[10] R.A. Brooks. Intelligence without reason. In *Proceedings of IJCAI-91*, pages 569–595. IJCAI, Inc., July 1991.

[11] J. Buhmann, W. Burgard, A.B. Cremers, D. Fox, T. Hofmann, F. Schneider, J. Strikos, and S. Thrun. The mobile robot Rhino. *AI Magazine*, 16(1), 1995.

[12] W. Burgard, A.B., Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. The interactive museum tour-guide robot. In *Proceedings of the AAAI Fifteenth National Conference on Artificial Intelligence*, 1998.

[13] W. Burgard, D. Fox, D. Hennig, and T. Schmidt. Estimating the absolute position of a mobile robot using position probability grids. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Menlo Park, August 1996. AAAI, AAAI Press/MIT Press.

[14] V. Castelli and T. Cover. On the exponential value of labeled samples. *Pattern Recognition Letters*, 16:105–111, January 1995.

[15] V. Castelli and T. Cover. The relative value of labeled and unlabeled samples in pattern recognition with an unknown mixing parameter. *IEEE Transactions on Information Theory*, 42(6):2101–2117, November 1996.

[16] J. H. Connell and S. Mahadevan, editors. *Robot Learning*. Kluwer Academic Publishers, 1993.

[17] P. Dayan and G. E. Hinton. Feudal reinforcement learning. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems 5*, San Mateo, CA, 1993. Morgan Kaufmann.

[18] T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150, 1989.

[19] T. L. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceeding of Seventh National Conference on Artificial Intelligence AAAI-92*, pages 49–54, Menlo Park, CA, 1988. AAAI, AAAI Press/The MIT Press.

[20] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[21] F. Dellaert, D. Fox, W. Burgard, and S. Thrun. Monte-carlo localization. (submitted for publication), 1998.

[22] A.P. Dempster, A.N. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society, Series B*, 39(1):1–38, 1977.

[23] E.D. Dickmanns, R. Behringer, D. Dickmanns, T. Hildebrandt, M. Maurer, J. Schiehlen, and F. Thomanek. The seeing passenger car VaMoRs-P. In *Proceedings of the International Symposium on Intelligent Vehicles*, Paris, France, 1994.

[24] T. Dietterich. The maxq method for hierarchical reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, Madison, WI, 1998.

[25] T. G. Dietterich. Learning at the knowledge level. *Machine Learning*, 1:287–316, 1986.

[26] R. E. Doorenbos. Matching 100,000 learned rules. In *Proceeding of the Eleventh National Conference on Artificial Intelligence AAAI-93*, pages 290–296, Menlo Park, CA, 1993. AAAI, AAAI Press/The MIT Press.

[27] M. Dorigo and M. Colombetti. Robot shaping: Developing autonomous agents through learning. *Artificial Intelligence*, 71(2):321–370, 1994.

[28] D. Driankov, H. Hellendoorn, and M. Reinfrank. *An Introduction to Fuzzy Control*. Springer Verlag, Berlin, 1996. 2nd rev.

[29] A. Elfes. Sonar-based real-world mapping and navigation. *IEEE Journal of Robotics and Automation*, RA-3(3):249–265, June 1987.

[30] A. Elfes. *Occupancy Grids: A Probabilistic Framework for Robot Perception and Navigation*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1989.

[31] R.J. Firby. An investigation into reactive planning in complex domains. In *Proceedings of the National Conference on Artificial Intellgence (AAAI)*, pages 809–815. AAAI, 1987.

[32] H. Friedrich, S. Münch, R. Dillman, S. Bocionek, and M. Sassin. Robot programming by demonstration (rpd): Supporting the induction by human interaction. *Machine Learning*, 23(2/3):5–46, 1996.

[33] E. Gat. Esl: A language for supporting robust plan execution in embedded autonomous agents. In *Working notes of the AAAI Fall Symposium on Plan Execution*, Boston, MA, 1996. AAAI.

[34] A. Gelb. *Applied Optimal Estimation*. MIT Press, 1974.

[35] M. Gherrity. A learning algorithm for analog, fully recurrent neural networks. In *Proceedings of the First International Joint Conference on Neural Networks, Washington, DC*, San Diego, 1989. IEEE, IEEE TAB Neural Network Committee.

[36] Glesner.S. and D. Koller. Constructing flexible dynamic belief networks from first-order probabilistic knowledge bases. In Ch. Froidevaux and J. Kohlas, editors, *Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 217–226, Berlin, 1995. Springer Verlag.

[37] A. Griewank, D. Juedes, and J. Utke. ADOI-C, a package for automatic differentiation of algorithms written in c/c++. *ACM Transaction on Mathematical Software*, 22(2):131–167, 1996.

[38] D. Guzzoni, A. Cheyer, L. Julia, and K. Konolige. Many robots make short work. *AI Magazine*, 18(1):55–64, 1997.

[39] K.Z. Haigh and M.M. Veloso. High-level planning and low-level execution: Towards a complete robotic agent. In W. Lewis Johnson, editor, *Proceedings of First International Conference on Autonomous Agents*, pages 363–370. ACM Press, New York, NY, 1997.

[40] J. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46:311–350, 1990.

[41] D. Heckerman. A tutorial on learning with bayesian networks. Technical Report Technical Report MSR-TR-95-06, Microsoft Research, 1995. (revised November, 1996).

[42] H. Hirsh. Combining empirical and analytical learning with version spaces. In B. Spatz and J. Galbraith, editors, *Proceedings of the Sixth International Workshop on Machine Learning*, pages 29–33, San Mateo, CA, June 1989. Morgan Kaufmann Publishers, Inc.

[43] K. Ikeuchi, T. Suehiro, and S.B. Kang. Assembly plan from observation. In K. Ikeuchi and M. Veloso, editors, *Symbolic Visual Learning*, pages 193–224. Oxford University Press, 1996.

[44] M. Isard and A. Blake. Condensation: conditional density propagation for visual tracking. *International Journal of Computer Vision*, in press 1998.

[45] L.P. Kaelbling, A.R. Cassandra, and J.A. Kurien. Acting under uncertainty: Discrete bayesian models for mobile-robot navigation. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1996.

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

[47] R. E. Kalman. A new approach to linear filtering and prediction problems. *Trans. ASME, Journal of Basic Engineering*, 82:35–45, 1960.

[48] V. Klingspor, K.J. Morik, and A.D. Rieger. Learning concepts from sensor data of a mobile robot. *Machine Learning*, 23(2/3):189–216, 1996.

[49] S. Koenig and R. Simmons. Passive distance learning for robot navigation. In L. Saitta, editor, *Proceedings of the Thirteenth International Conference on Machine Learning*, 1996.

[50] D. Koller and R. Fratkina. Using learning for approximation in stochastic processes. In *Proceedings of the International Conference on Machine Learning (ICML)*, 1998.

[51] D. Koller, D. McAllester, and A. Pfeffer. Effective bayesian inference for stochastic programs. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI)*, Providence, Rhode Island, 1997.

[52] D. Koller and A. Pfeffer. Object-oriented bayesian networks. In *Proceedings of the 13th Annual Conference on Uncertainty in AI (UAI)*, Providence, Rhode Island, 1997.

[53] K. Konolige. Colbert: A language for reactive control in saphira. In *German Conference on Artificial Intellgence*, Berlin, 1997. Springer Verlag.

[54] R.A. Kowalski. *Logic for Problem Solving*. North Holland, 1979.

[55] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[56] J. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, 1994.

[57] J. Laird, P. Rosenbloom, and A. Newell. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46, 1986.

[58] L.-J. Lin. *Self-supervised Learning by Reinforcement and Artificial Neural Networks*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, 1992.

[59] M.L. Littman, A.R. Cassandra, and L.P. Kaelbling. Learning policies for partially observable environments: Scaling up. In A. Prieditis and S. Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, 1995.

[60] P. Maes and R. A. Brooks. Learning to coordinate behaviors. In *Proceedings Eighth National Conference on Artificial Intelligence*, pages 796–802, Cambridge, MA, 1990. AAAI, The MIT Press.

[61] S. Mahadevan and J. Connell. Scaling reinforcement learning to robotics by exploiting the subsumption architecture. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 328–332, 1991.

[62] M. Matarić and D. Cliff. Challenges in evolving controllers for physical robots. Technical Report CS-95-184, Brandeis University, Computer Science Department, Waltham, MA, 1995.

[63] M. J. Matarić. A distributed model for mobile robot environment-learning and navigation. Master's thesis, MIT, Cambridge, MA, January 1990. also available as MIT AI Lab Tech Report AITR-1228.

[64] M. J Matarić. Reinforcement learning in the multi-robot domain. *Autonomous Robots*, 4(1):73–83, January 1997.

[65] P. Maybeck. *Stochastic Models, Estimation, and Control, Volume 1*. Academic Press, Inc, 1979.

[66] C. S. Miller and J. E. Laird. A constraint-motivated lexical acquisition model. In *Proceedings of the Thirteenth Annual Meeting of the Cognitive science Society*, pages 827–831, Hillsdale, NJ, 1991. Erlbaum.

[67] S. Minton, J. Carbonell, C. A. Knoblock, D. R. Kuokka, O. Etzioni, and Y. Gil. Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40:63–118, 1989.

[68] T. M. Mitchell. Becoming increasingly reactive. In *Proceedings of 1990 AAAI Conference*, Menlo Park, CA, August 1990. AAAI, AAAI Press / The MIT Press.

[69] T. M. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[70] T.M. Mitchell, J. Allen, P. Chalsani, J. Cheng, O. Etzioni, M. Ringuette, and J.C. Schlimmer. Theo: A franework for self-improving systems. In K. VanLehn, editor, *Architectures for Intelligence*. Lawrence Erlbaum, 1989.

[71] A.W. Moore, J. Schneider, and K. Deng. Efficient locally weighted polynomial regression predictions. In *Proceedings of the International Conference on Machine Learning*. Morgan Kaufmann Publishers, 1997.

[72] H. P. Moravec. Sensor fusion in certainty grids for mobile robots. *AI Magazine*, pages 61–74, Summer 1988.

[73] S. Muggelton. *Inductive Logic Programming*. Academic Press, New York, 1992.

[74] K. Nigam, A. McCallum, S. Thrun, and T. Mitchell. Learning to classify text from labeled and unlabeled documents. In *Proceedings of the AAAI Fifteenth National Conference on Artificial Intelligence*, 1998.

[75] I. Nourbakhsh, R. Powers, and S. Birchfield. DERVISH an office-navigating robot. *AI Magazine*, 16(2):53–60, Summer 1995.

[76] J.J. Oliver and D.L. Dowe. Using unsupervised learning to assist supervised learning. In *Proceedings of the Eighth Australian Joint Conference on AI*, 1995.

[77] S. M. Omohundro. Efficient Algorithms with Neural Network Behaviour. *Journal of Complex Systems*, 1(2):273–347, 1987.

[78] D. Ourston and R. J. Mooney. Theory refinement with noisy data. Technical Report AI 91-153, Artificial Intelligence Lab, University of Texas at Austin, March 1991.

[79] R. Parr and S. Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 10*, Cambridge, MA, 1998. MIT Press.

[80] M. J. Pazzani, C. A. Brunk, and G. Silverstein. A knowledge-intensive approach to learning relational concepts. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 432–436, Evanston, IL, June 1991.

[81] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers, San Mateo, CA, 1988.

[82] D. Pomerleau. *Neural Network Perception for Mobile Robot Guidance*. Kluwer Academic Publishers, Boston, MA, 1993.

[83] D. A. Pomerleau. Rapidly adapting neural networks for autonomous navigation. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 429–435, San Mateo, 1991. Morgan Kaufmann.

[84] D. Poole. Probabilistic horn abduction and bayesian networks. *Artificial Intelligence*, 64:81–129, 1993.

[85] M.L. Puterman. *Markov Decision Processes*. John Wiley & Sons, New York, 1994.

[86] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[87] L.R. Rabiner and B.H. Juang. An introduction to hidden markov models. In *IEEE ASSP Magazine*, 1986.

[88] L.B. Rall. *Automatic Differentiation: Techniques and Applications*. Springer Verlag, Berlin, 1981.

[89] P. S. Rosenbloom and J. Aasman. Knowledge level and inductive uses of chunking (EBL). In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 821–827, Boston, 1990. AAAI, MIT Press.

[90] P. S. Rosenbloom and J. E. Laird. Mapping explanation-based generalization onto soar. Technical Report 1111, Stanford University, Dept. of Computer Science, Stanford, CA, 1986.

[91] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing. Vol. I + II.* MIT Press, 1986.

[92] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[93] S. Schaal. Learning from demonstration. In *Advances in Neural Information Processing Systems 8*, Cambridge, MA, 1997. MIT Press.

[94] A. Schultz and J. Grefenstette. Evolving robot behaviors. Technical Report AIC-94-017, Navy Center for Applied Research in Artificial Intelligence, Washington DC, October 1994.

[95] J. W. Shavlik. Combining symbolic and neural learning. *Machine Learning*, 14(3):321–331, 1994.

[96] R. Simmons. Concurrent planning and execution for autonomous robots. *IEEE Control Systems*, 12(1):46–50, February 1992.

[97] R. Simmons and S. Koenig. Probabilistic robot navigation in partially observable environments. In *Proceedings of IJCAI-95*, pages 1080–1087, Montreal, Canada, August 1995. IJCAI, Inc.

[98] R. Simmons and S. Thrun. Languages and tools for task-level robotics integration. In *Proceedings of the 1998 AAAI Spring Symposium*, 1998.

[99] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In D. Touretzky, M. Mozer, and M.E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, Cambridge, MA, 1996. MIT Press.

[100] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[101] A. Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming II*, Cambridge, MA, 1996. MIT Press.

[102] G. J. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8, 1992.

[103] S. Thrun. *Explanation-Based Neural Network Learning: A Lifelong Learning Approach*. Kluwer Academic Publishers, Boston, MA, 1996.

[104] S. Thrun. Bayesian landmark learning for mobile robot localization. *Machine Learning*, 33(1), 1998.

[105] S. Thrun. Learning metric-topological maps for indoor mobile robot navigation. *Artificial Intelligence*, 99(1):21–71, 1998.

[106] S. Thrun, A. Bücken, W. Burgard, D. Fox, T. Fröhlinghaus, D. Henning, T. Hofmann, M. Krell, and T. Schmidt. Map learning and high-speed navigation in RHINO. In D. Kortenkamp, R.P. Bonasso, and R Murphy, editors, *AI-based Mobile Robots: Case Studies of Successful Robot Systems*. MIT Press, 1998.

[107] S. Thrun, D. Fox, and W. Burgard. A probabilistic approach to concurrent mapping and localization for mobile robots. *Machine Learning*, 31:29–53, 1998. also appeared in Autonomous Robots 5, 253–271.

[108] G. G. Towell and J. W. Shavlik. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(1/2):119–165, 1994.

[109] X. Wang. Planning while learning operators. In *Proceddings of the International Conference on Artificial Intelligence Planning Systems (AIPS)*, 1996.

[110] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, England, 1989.

[111] G. Welch and G. Bishop. An introduction to the kalman filter. Technical Report TR 95-041, University of North Carolina, Department of Computer Science, 1995.

[112] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989. Also appeared as: Technical Report ICS Report 8805, Institute for Cognitive Science, University of California, San Diego, CA, 1988.

[113] B. Yamauchi and P. Langley. Place recognition in dynamic environments. *Journal of Robotic Systems*, Special Issue on Mobile Robots, (to appear). also located at http://www.aic.nrl.navy.mil/∼yamauchi/.