Towards Programming Tools for Robots That Integrate Probabilistic Computation and Learning

Sebastian Thrun
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Abstract

This paper describes a programming language extension of C++, called CES, specifically targeted towards mobile robot control. CES's design is motivated by a recent series of successful probabilistic *methods for mobile robot control, with the goal of facilitating the development of such probabilistic software in future robot applications. CES extends C++ by two ideas: Computing with probability distributions, and built-in mechanisms for learning from examples as a new means of programming. An example program, used to control a mail-delivering robot with gesture commend interface, illustrates that CES may reduce the code development by two orders of magnitude. CES differs from other special-purpose programming languages in the field, which typically emphasize concurrency and real-time/event-driven processing.*

1 Introduction

In recent years, *probabilistic algorithms* have become popular in mobile robotics. Probabilistic algorithms explicitly represent a robot's uncertainty explicitly, enabling it to cope with ambiguities and noise in a mathematically sound and elegant way. Probabilistic approaches are typically more robust than traditional methods, as demonstrated by a series of fielded systems such as (in our own work) the RHINO and MINERVA tour-guide robots [3, 25].

Equally popular has become the idea of *learning* in robotics. Learning has successfully been applied for the *design* of mobile robot software—such as in Pomerleau's autonomous ALVINN vehicle [20] which was trained by watching a human drive—as well as *during robot operation*—such as in systems that learn maps on-the-fly [5, 3].

The underlying mechanisms—probabilistic computation and learning—are generic and transcend beyond a specific application. Moreover, the development of probabilistic software or learning software is often tedious and time-consuming. Thus, there is a clear need for programming tools that support for these ideas and facilitate their use in future robotic applications.

This paper presents an extension to C++, called CES (short for: C++ for embedded systems) that integrates learning and probabilistic reasoning into C++:

1. **Probabilistic computation.** To handle uncertain information, data types are provided that represent probability distributions. Computing with probability distributions is very similar to computing with conventional data types, with the added benefit of increased robustness.

2. **Learning.** CES provides function approximators and the necessary infrastructure (credit assignment mechanisms) to *train* them based on experience. In particular, CES uses gradient descent to learn from "distal" target information.

C++ has been chosen as base language because of the popularity of C and C++ in robotics. The remainder of this paper describes the major concepts and demonstrates their benefits in the context of an implemented mobile robot system.

2 Probabilistic Computation in CES

2.1 Probabilistic Data Types and Assignments

The template `prob` constructs *probabilistic* variables from conventional data types. Examples of probabilistic variables are

```
prob<char>     prob<int>     prob<double>     ...
```

which parallel the standard data types `char`, `int`, `double`. Probabilistic variables represent *probability distributions* over their respective base data types.

In CES, such distributions are represented using lists of values, annotated by a numerical probability. If a distribution is finite, such list may contain any value of the variable's domain annotated by their probability. In continuous variables, such as `prob<double>`, the list is a sample-based approximation of a (hypothetical) true density. If constructed appropriately, such sample approximations converge at a speed of $\frac{1}{\sqrt{N}}$, with N denoting the number of samples [18, 22]. The reader may notice that such sample-based approximations have been applied with great success in robotics [4, 7] and computer vision **??**.

Just as conventional variables, probabilistic variables can be initialized by constants. For example, the assignment

```
prob<double> x = 2.4;
```

assigns a Dirac distribution to `x` whose probability is centered on 2.4. Finite distributions are specified trough lists. The assignment

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

assigns to `x` a distribution with three elements: 1, 2, and 10, whose probabilities are 0.5, 0.3, and 0.2, respectively:

$$\begin{aligned} \Pr(x = 1) &= 0.5 \\ \Pr(x = 2) &= 0.3 \\ \Pr(x = 10) &= 0.2 \end{aligned}$$

CES offers a range of frequently used distributions, such as:

```
x = normal1d(0.0, 1.0);
x = uniform1d(-1.0, 1.0);
```

Arithmetics with probabilistic data types is analogous to arithmetic with conventional data types. For example, consider the following code segment:

```
prob<int> x, y, z;
x = { {2, 0.5}, {3, 0.5} };
y = { {10, 0.1}, {11, 0.9} };
z = x + y;
```

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

```
{ {12, 0.05}, {13, 0.5}, {14, 0.45} };
```

Just as in C++, CES offers type conversions between conventional and probabilistic variables. Suppose `x` is declared as a `double`, and `y` is declared as a `prob<double>`. The statement

```
y = prob<double>(x);
```

assigns to `y` a Dirac distribution whose probability mass is centered on the value of `x`. The inverse statement,

```
x = double(y);
```

assigns to `x` the *mean* of the distribution `y`, computed in the obvious way. For additional means of converting distributions to values, CES offers functions such as `mean()`, `ml()`, `median()`, and `variance()`.

2.2 Independence

To keep probabilistic computation tractable, CES makes an implicit and very important *independence assumptions* between probabilistic variables. The variables on the right hand-side of statements like

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

are assumed to be *independent*, that is, their joint distribution is the product of the marginal distributions:

$$\Pr(x, y) \quad = \quad \Pr(x)\ \Pr(y) \tag{1}$$

The independence assumption is essential to maintain tractability in large CES programs.

To see, consider the situation in Bayes networks [9, 19], a popular, alternative framework for reasoning probabilistically. Bayes networks interpret statements like `z=x-y` as *constraints on a high-dimensional joint distribution*; hence, they must keep track of implicit dependencies arising from such statements (e.g., z depends on z and y). As a result, evaluating a Bayes network (marginalizing a high-dimensional joint distribution) can be difficult, specifically if the network possesses loops [19]. CES interprets program code as computational transformations, instead of constraints on a joint distribution. This is semantically much closer to conventional procedural programming languages. It ensures tractability of large CES programs with loops.

2.3 The `probloop` Command

Sometimes, however, CES's independence assumption is too strong. CES offers a mechanisms for explicitly maintaining dependencies arising in the computation: the `probloop` command. This command provides a sound mechanism to maintain arbitrary dependencies within a limited code segment of the programmer's choice.

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 code. Variables may appear in both lists, and either list may be empty.

The `probloop` command interprets the variables in *var-list-in* as probability density functions. It executes the *program-code* with all combinations of values for the variables in the *var-list-in*, with the exception of those whose probability are zero. Inside the *program-code*, all variables in both lists (*var-list-in*, *var-list-out*) are 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 execution, two things are cached: 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 densities for all probabilistic variables in the *var-list-out*.

For an example, consider the following code segment:

```
prob<int> x, y, z;
x = {{1, 0.2},  {2, 0.8}};
y = {{10, 0.5}, {20, 0.5}};
probloop(x, y; z)
  if (10 * x - 1 > y) z = 1; else z = 0;
```

The `probloop` instruction loops through all combinations of values of `x` and `y`, which are: $\langle 1, 10\rangle$, $\langle 1, 20\rangle$, $\langle 2, 10\rangle$, and $\langle 2, 20\rangle$. For all of those, 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`. The result is then converted into a probability distribution for `z`, the only variable in *var-list-out*:

```
{ {0, 0.6}, {1, 0.4} }
```

The `probloop` command provides a sound way to use probabilistic variables in commands such as `for` loops, `while` loops, and `if-then-else` (see also [24]).

2.4 Direct Manipulation of Densities

Statements like

```
x = y * z;
```

assign to `x` the distribution

$$\Pr(\mathtt{x}) \quad = \quad \int_{\mathtt{yz}=\mathtt{x}} \Pr(\mathtt{y}) \Pr(\mathtt{z}) \; d\langle \mathtt{y}, \mathtt{z}\rangle \qquad (2)$$

Such a manipulation is called *indirect*. It combines `y` and `z` by iterating through their domains. Sometimes, it is useful to manipulate or combine densities more directly, e.g., to obtain:

$$\Pr(\mathtt{x}) \quad = \quad \Pr(\mathtt{y}) \Pr(\mathtt{z}) \qquad (3)$$

This is achieved using built-in functions such as

```
x = multiply(y, z);
```

Direct manipulation of densities is useful for a range of operations, including *averaging* of multiple probabilistic variables, and implementing *Bayes rule*. For example, Bayes rule

$$\Pr(a|b) \quad = \quad \frac{\Pr(b|a) \Pr(a)}{\Pr(b)} \qquad (4)$$

is implemented in CES as

```
x = divide(multiply(y, xprior), yprior);
```

assuming that `y` holds $\Pr(b|a)$, `xprior` holds $\Pr(a)$, and `yprior` holds $\Pr(b)$.

To perform direct manipulations using our sample-based approximation of probability densities (e.g., for the data type `prob<double>`, an approximation of the (complete) density is temporarily constructed from the samples using density trees [11, 17]. Following the idea of importance factors [], two densities are combined by manipulating the sample probabilities of one sample set using the tree generated by the other sample set. Unfortunately, space limits prevent us from describing this mechanism in greater detail.

3 Learning

To support learning, CES provides built-in function approximators and a mechanism for credit assignment in program code. For example, the programmer might specify that an artificial neural network be used to map raw camera images into the steering direction of a vehicle (see [20] for a famous example of this approach). To "train" the program, the programmer may then provide a target signal (e.g., a target steering direction for a vehicle). The basic idea behind our approach is that programmers can leave "gaps" in their programs which are "filled" by learning/teaching.

3.1 Leaving Gaps

Function approximators are created through a template `fa` which requires the data type of the input and that of the output. Currently, input and output can be `double`, `prob<double>`, or vectors thereof (e.g., `vector<prob<double> >`). Additionally, the programmer must specify the type of the function approximator. For example

```
fa<vector<double>, prob<vector<double> > >
          mynet(oneLayerNeuronet, 10);
```

creates a Backpropagation neural network with one hidden layer and ten hidden units that maps a vector of doubles to a probability distribution over such vectors. The dimension of the vector is obtained automatically when using the function approximator.

Once created, function approximators can be accessed through the special method `eval()`:

```
vector<double> x(4);  // dimension 4
prob<vector<double> > y(3); // dimension 3
y = mynet.eval(x);
```

The output of a function approximator is restricted to lie between 0 and 1. Our current implementation offers a collection of function approximators (Backpropagation networks, linear regression, radial-basis functions).

3.2 Training

CES programs are trained using the method `train()`, which is defined for probabilistic data types. The assignment

```
x.train(y);
```

specifies that the desired value for the variable `x` is `y` (at the current point of program execution). Here both variables, `x` and `y`, are either of the same type, or `x` is of type `prob<foo>` if `y`'s type is `foo`.

In our current implementation, the training operator induces a quadratic error norm. When a training operator is encountered, CES adjusts the parameters of all contributing function approximators accordingly. To do so, CES possess a built-in built-in credit assignment mechanism that uses gradient descent. More specifically, let `y` be a probabilistic variable. An operation such as

```
y = f.eval(y);
```

attaches to each sample of `x`, denoted $\langle x, p_x \rangle$, a *gradient* $\frac{\partial p_x}{\partial w_i}$ where w_i is a *parameter* of the function approximator `f`. The gradients are then passed on to subsequent (dependent) probabilistic variables. For example if further down in the program execution CES encounters a statement such as

```
z = y + a;
```

the gradients $\frac{\partial p_z}{\partial w_i}$ are computed automatically using the chain rule of differentiation. When a training function is executed, the desired parameter updates can be computed immediately using these gradients and the chain rule.

As a pleasing consequence, 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. This is convenient, as it enables programmers to specify (through examples) the input-output behavior of code segments that might, internally contain one or more function approximator—without having to generate examples of the input-output behavior of the function approximator(s) itself.

3.3 The Importance of Probabilities for Learning

Probabilistic computation is a key enabling factor for CES's learning mechanism. Conventional (i.e., non-probabilistic) C++ code is usually not differentiable. Consider, for example, the statement

(a) (b)

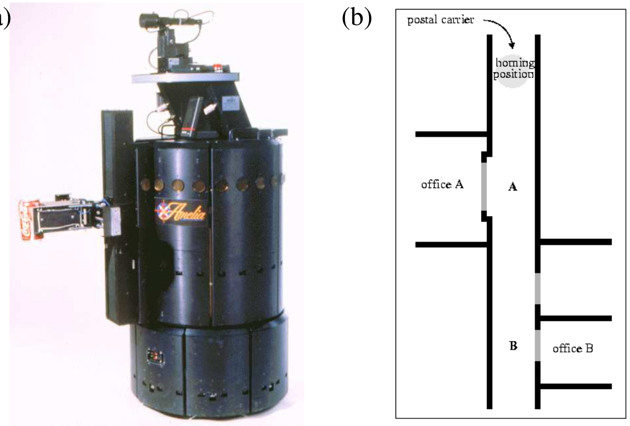

Figure 1: (a) The Real World Interface B21 robot used in our research. (b) Schematics of the robot's environment.

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

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

$$\frac{\partial \mathtt{y}}{\partial \mathtt{x}} = 0 \qquad \text{with probability 1.} \tag{5}$$

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 adaptation will occur.

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

$$\frac{\partial \Pr(\mathtt{y}=1)}{\partial \Pr(\mathtt{x}=a)} = \begin{cases} 1 & \text{if } a > 0 \\ -1 & \text{if } a \le 0 \end{cases} \tag{6}$$

$$\frac{\partial \Pr(\mathtt{y}=2)}{\partial \Pr(\mathtt{x}=a)} = \begin{cases} 1 & \text{if } a \le 0 \\ -1 & \text{if } a > 0 \end{cases} \tag{7}$$

Notice that none of the gradients are zero. The differentiability of probabilistic variables is essential for CES's gradient descent function approximator.

4 Proof-of-Concept: A Mail Delivery Robot

To provide a first proof-of-concept, we successfully implemented a program for a mail-delivery robot instructed through gestures. Obviously, a single example is insufficient to demonstrate the generality of the language; and only time can tell how useful the concepts are. However, in our example CES reduced the length of the program by more than two orders of magnitude, when compared

```
001: main(){
002:
003:   // ============= Declarations ==============
004:   fa<vector<double>,         prob<double> > netSonar(oneLayerNeuronet, 5);
005:   fa<prob<vector<double> >, prob<double> > netX(oneLayerNeuronet, 5),
006:                                              netY(oneLayerNeuronet, 5);
007:   fa<vector<double>,         prob<int> >    netLeft(oneLayerNeuronet, 5),
008:                                              netRight(oneLayerNeuronet, 5);
009:   prob<double>           alpha, alphaLocal, probRotation;
010:   prob<double>           thetaLocal, theta, transVel, rotVel;
011:   prob<double>           x, xLocal, y, yLocal, probTransl;
012:   prob<int>              coin = {{0, 0.5}, {1. 0.5}};
013:   prob<int>              gestureLeft, gestureRight;
014:   prob<vector<double >> newSonar(2);
015:   double                 alphaTarget, scan[24], image[300];
016:   double                 xTarget, yTarget, xGoal, yGoal, t, v;
017:   struct                 { double rotation, transl; } odometryData;
018:   struct                 { double x, y, dir; } stack[3];
019:   int                    targetLeft, targetRight;
020:   int                    numGoals = 0, activeGoal;
021:
022:   // ============= Initialization ==============
023:   alpha = UNIFORM1D(0.0, M_PI);
024:   theta = UNIFORM1D(0.0, M_PI);
025:   x = XHOME; y = YHOME;
026:
027:   // ============= Main Loop ==============
028:
029:   for (;;){
030:
031:     // ---------------- Localization ----------------
032:     GETSONAR(scan);                     // get sonar scan
033:     alphaLocal = netSonar.eval(scan) * M_PI;
034:     alpha = multiply(alpha, alphaLocal);
035:     probloop(alphaLocal, coin; thetaLocal){
036:       if (coin)
037:         thetaLocal = alphaLocal;
038:       else
039:         thetaLocal = alphaLocal + M_PI;
040:     }
041:     theta = multiply(theta, thetaLocal);// robot’s orientation
042:     probloop(theta; newSonar){
043:       int i = int(theta / M_PI * 12.0);
044:       int j = (i + 12) % 24;
045:       if (scan[i] < 300.0) newSonar[0] = scan[i];
046:       if (scan[j] < 300.0) newSonar[1] = scan[j];
047:     }
048:     xLocal = netX.eval(newSonar);
049:     yLocal = netY.eval(newSonar);
050:     x = multiply(x, xLocal);             // robot’s x coordinate
051:     y = multiply(y, yLocal);             // robot’s y coordinate
052:
053:     GETODOM(&odometryData);              // get odometry data
054:     probRotation = prob<double>(odometryData.rotation)
055:        + normal1d(0.0, 0.1 * fabs(odometryData.rotation));
056:     alpha += probRotation;
057:     if (alpha <   0.0) alpha += M_PI;
058:     if (alpha >= M_PI) alpha -= M_PI;
059:     theta += probRotation;               // new orientation
060:     if (theta <  0.0)        theta += 2.0 * M_PI;
061:     if (theta >= 2.0 * M_PI) theta -= 2.0 * M_PI;
062:     theta = probtrunc(theta, 0.01);
063:     probTransl = (prob<double>) odometryData.transl
064:       + NORMAL1D(0.0, 0.1 * fabs(odometryData.transl));
065:     x = x + probTransl * cos(theta);
066:     y = y + probTransl * sin(theta);
067:     x.truncate(0.01);                    // new x coordinate
068:     y.truncate(0.01);                    // new y coordinate
069:     // ------------ Gesture Interface & Scheduler ------------
070:     GETIMAGE(image);
071:     gestureLeft  = netLeft.eval(image);
072:     gestureRight = netRight.eval(image);
073:     if (numGoals == 0){                   // wait for gesture
074:       if (double(gestureLeft) > 0.5){
075:         stack[numGoals  ].x   = XA;    // location A on stack
076:         stack[numGoals  ].y   = YA;
077:         stack[numGoals++].dir = 1.0;
078:       }
079:       if (double(gestureRight) > 0.5){
080:         stack[numGoals  ].x   = XB;    // location B on stack
081:         stack[numGoals  ].y   = YB;
082:         stack[numGoals++].dir = 1.0;
083:       }
084:       if (numGoals > 0){
085:         stack[numGoals  ].x   = XHOME; // HOME location on stack
086:         stack[numGoals  ].y   = YHOME;
087:         stack[numGoals++].dir = -1.0;
088:         activeGoal = 0;
089:       }
090:     }
091:     else if (stack[activeGoal].dir *   // reached a goal?
092:              (double(y) - stack[activeGoal].y) > 0.0){
093:       SETVEL(0, 0);                     // stop robot
094:       activeGoal = (activeGoal + 1) % depth;
095:       if (activeGoal)
096:         for (HORN(); !GETBUTTON(); );  // wait for button
097:       else
098:         numGoals = 0;
099:     }
100:
101:     else{ // ---------------- Navigation ----------------
102:       xGoal = stack[activeGoal].x;
103:       yGoal = stack[activeGoal].y;
104:       probloop(theta, x, y, xGoal, yGoal;
105:                transVel, rotVel){
106:         double thetaGoal = atan2(y - yGoal, x - xGoal);
107:         double thetaDiff = thetaGoal - theta; // location of goal
108:         if (thetaDiff < -M_PI) thetaDiff += 2.0 * M_PI;
109:         if (thetaDiff >  M_PI) thetaDiff -= 2.0 * M_PI;
110:         if (thetaDiff < 0.0)
111:           rotVel =  MAXROTVEL;          // rotate left
112:         else
113:           rotVel = -MAXROTVEL;          // rotate right
114:         if (fabs(thetaDiff) > 0.25 * M_PI)
115:           transVel = 0;                 // no translation
116:         else
117:           transVel = MAXTRANSVEL;       // go ahead
118:       }
119:       v = double(rotVel);               // convert to double
120:       t = double(transVel);             // convert to double
121:       if (sonar[0] < 15.0 || sonar[23] < 15.0) t = 0.0;
122:       SETVEL(t, v);                     // set velocity
123:     }
124:
125:     // ---------------- Training ----------------
126:     GETTARGET(&alphaTarget);            // these command are
127:     alpha.train(alphaTarget);           // only enabled during
128:     GETTARGET(&xTarget);                // training. They are
129:     x.train(xTarget);                   // removed afterwards.
130:     GETTARGET(&yTarget);
131:     y.train(yTarget);
132:     GETTARGET(&targetLeft);
133:     gestureLeft.train(targetLeft);
134:     GETTARGET(&targetRight);
135:     gestureRight.train(targetRight);
136:   }
137: }
```

Table 1: The complete implementation of the mail delivery program. Line numbers have been added for the reader's convenience. Functions in capital letters (`GET_...` and `SET_...`) are part of the interface to the robot.

to a previous implementation in C. A second example, described in [], also led to code that was 2 orders of magnitude shorter than the original implementation in C.

Table 1 shows the full program, whose development is described in detail in [24]. This program, along with the appropriate training is sufficient to make the robot shown in Figure 1a deliver mail an office environment. The robot, which is instructed through visual gestures such as those shown in Figure 2, delivers mail to up to two designated locations in a populated corridor environment, labeled "A" and "B" in Figure 1b. The CES program, which is only 137 lines long and has been trained with less than two hours worth of data. The program operates on raw sensor data and directly controls the robot's motors (velocities). It successfully implements a highly reliably robotic controller involving collision avoidance, localization, point-to-point navigation, gesture recognition, and high-level scheduling of deliveries, in an ambiguous and dynamic environment.

Our program uses neural networks to recognize gestures from camera images (lines 70-72). Depending on the gesture, the robot schedules one or two target locations (lines 73-90) and then navigates there (lines 101-123). At the target locations, the robot honks a horn and waits for a person to pick up his mail (lines 91-99). While doing so, it maintains an accurate, probabilistic estimate of its location (in x-y-θ world coordinates), using sonar readings, neural networks, and Bayes rule to update its belief (lines 32-68). The robot also avoids collisions with unexpected obstacles

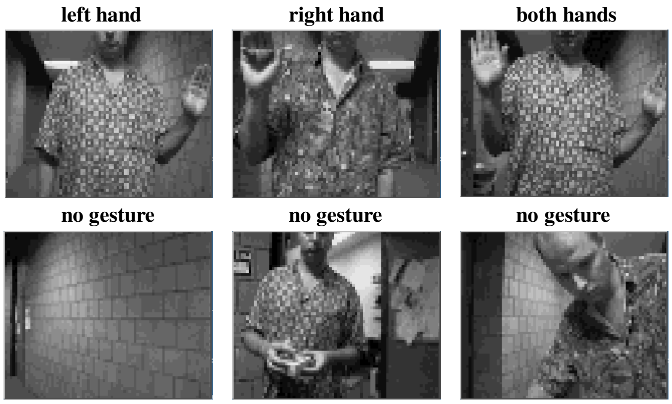

Figure 2: Positive (top row) and negative (bottom row) examples of gestures.

such as humans, by decelerating the proximity of such obstacles (line 121). As described in [24], the code was developed in stages, interleaving conventional programming and teaching. To train the various function approximators (lines 4 to 8), two data sets were collected and labeled manually (one for localization, lines 126-131, and one for the gesture-based interface, lines 132-135). Collecting and labeling the data took less than 2 hours, using our mobile robot and a graphical interface for data labeling.

We found that our CES program is indeed sufficient to control a mobile robot reliably in a crowded environment, despite its shortness. In extensive tests, the robot navigated for extended periods of time in a populated corridor without ever loosing its position, or colliding with an obstacle. Figure 3 shows an example trace (raw data), recorded during 20 minutes of continuous deliveries (58,992 sonar measurements, 2,458 odometry measurements). Notice that raw odometry is insufficient to track the robot's position. Our program reliably navigated the robot to the correct location with high accuracy (<1 meter) and delivered all pieces of mail correctly. In tests with independently collected data, the error rate of the gesture interface was consistently below 2.5% despite its simplicity (measured on 138 independently collected testing examples).

Mail delivery [13] and gesture-based human robot interaction [10, 14] have often been used as reference problems in the AI mobile robotics community. Even though our short program is obviously restricted in many ways (e.g., the robot must operate in a single corridor), the key result here is that with our new language, we were able to provide a solution with a 137-line program and less than 2 hours of training. Comparable systems usually require at least two orders of magnitude more code and are considerably more difficult to implement (see, for example, the various chapters in [13] and [2]). A second example in [24] demonstrates how a 5,000 lines state-of-the-art mobile robot localization algorithm [21, 23] has been implemented in 52 lines of CES code.

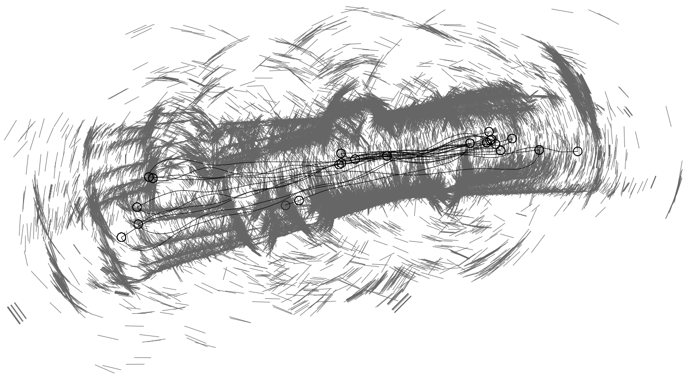

Figure 3: Plot of the robot trajectory (raw odometry) during 8 consecutive runs (11 pieces of mail delivered). Shown here are also the raw sonar measurements. The robot reaches the various destination points with high accuracy, despite the rather significant error in the robot's odometry.

5 Related Work

A large number of recent robot application illustrates the great power of probabilistic methods and learning for robotics (e.g., [3, 5, 20, 25]). These and similar applications should benefit greatly from CES, as the mechanisms described here are sufficiently generic to support a broad range of useful mechanisms in probabilistic robotics.

Probabilistic Reasoning. The vast majority of work in UAI, such as Bayes networks [9, 19], represent probabilistic knowledge *declaratively*, not *procedurally* as in CES. Bayes networks can be seen as a version of programming language which, for example, lacks loop statements like `for` and `while`. Contrary to CES, they employ probabilistic *separate inference mechanisms* to compute the desired quantities (marginal distributions). Some of the implications were discussed above, in Section 2.2. When compared to CES, such approaches actually have the advantage that they support reasoning in multiple directions (e.g., from the output of a network to its input). However, CES is much richer, as it includes conventional means of programming, and it has a different semantics. Since it is procedural, computation and knowledge representation are one and the same thing, thereby avoiding serious computational problems when scaling to large programs—which currently exist for large Bayes networks.

Machine Learning. From a machine learning point of view, CES provides a means for integrating prior knowledge (program code) and learning. Previous approaches to knowledge-based induction, such as the work originated in [16], require *declarative* representations for prior knowledge. CES represents such knowledge *procedurally*. Since declarative representations require inference methods whose timing is often unpredictable, they are rarely used in robotics.

CES differs from other procedural approaches, such as Generic Programming [15], in that its learning components keep the human-provided program code intact, making it easier for programmers to understand their code after learning.

Robotics. Various AI researchers have proposed special-purpose programming languages for robots and similar embedded systems. Existing languages typically support concurrency and event-driven execution [6, 8, 12], sometimes restricting the design of program modules [1]. These issues are entirely orthogonal to those pursued in CES.

6 Discussion

Recent trends in mobile robotics suggest that probabilistic robotics and robot learning are viable means of designing robot software; however, existing programming tools do not provide support for probabilistic computation or learning.

This paper described a language extension to C++, specifically developed for robots (and other sensor-based, embedded systems). It introduces two new ideas, previously not found in general purpose programming languages: probabilistic computation and exemplar-based learning/teaching. Software development in CES interleaves phases of conventional programming with phases where the program is taught using examples. As a result, functions that are difficult to program by hand (but easily trained) can be learned. CES's probabilistic data types facilitate computation with uncertain information (as generated by most sensors); hence provide additional robustness and aid the learning.

The primary benefit of the CES extension is that is facilitates rapid development of robust software. To validate this claim, we presented an example program for a gesture-instructed mail delivery robot. This example demonstrated an improvement (both in size of code and program development time) by more than an order of magnitude, when compared with today's best practice. Of course, a single example (and even a second one in [24]) are insufficient to fully assess the advantages of this new programming framework; however, since the ideas underlying CES are extremely generic, we are hopeful that similar benefits can be obtained in many other applications.

By integrating probabilistic computation and learning into a popular procedural programming language, we hope to facilitate the dissemination of these important ideas into mainstream robotics. Both ideas—probabilistic computation and learning—have shown great promise in a range of state-of-the-art robot applications; we hope that they will become an integral part of any future robot application faced with inaccurate sensing and unpredictable, dynamic environments.

Acknowledgement

The author thanks Frank Pfenning and the members of CMU's Robot Learning Lab for many inspiring discussions.

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), and also by the EC (contract number ERBFMRX-CT96-0049) under the TMR programme. 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, the United States Government, or the EC.

References

[1] R.A. Brooks. Intelligence without reason. *IJCAI-91*.

[2] W. Burgard, A.B., Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. The interactive museum tour-guide robot. *AAAI-98*.

[3] W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, to appear.

[4] F. Dellaert, W. Burgard, D. Fox, and S. Thrun. Using the condensation algorithm for robust, vision-based mobile robot localization. *CVPR-99*.

[5] A. Elfes. *Occupancy Grids: A Probabilistic Framework for Robot Perception and Navigation*. PhD thesis, ECE, CMU, 1989.

[6] R.J. Firby. An investigation into reactive planning in complex domains. *AAAI-87*.

[7] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. Monte Carlo localization: Efficient position estimation for mobile robots. *AAAI-99*.

[8] E. Gat. ESL: A language for supporting robust plan execution in embedded autonomous agents. *AAAI FSS, 1996*.

[9] D. Heckerman. A tutorial on learning with bayesian networks. TR MSR-TR-95-06, Microsoft Research, 1995.

[10] R.E. Kahn, M.J. Swain, P.N. Prokopowicz, and R.J. Firby. Gesture recognition using the perseus architecture. *CVPR-96*.

[11] D. Koller and R. Fratkina. Using learning for approximation in stochastic processes. *ICML-98*.

[12] K. Konolige. COLBERT: A language for reactive control in saphira. *KI-97*.

[13] D. Kortenkamp, R.P. Bonasso, and R. Murphy, editors. *AI-based Mobile Robots: Case studies of successful robot systems*, MIT Press, 1998.

[14] D. Kortenkamp, E. Huber, and P. Bonasso. Recognizing and interpreting gestures on a mobile robot. *AAAI-96*.

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

[16] T.M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[17] A.W. Moore, J. Schneider, and K. Deng. Efficient locally weighted polynomial regression predictions. *ICML-97*.

[18] R.M. Neal. Probabilistic inference using Markov chain Monte Carlo methods. TR CRG-TR-93-1, University of Toronto, 1993.

[19] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[20] D. A. Pomerleau. Rapidly adapting neural networks for autonomous navigation. *NIPS-91*.

[21] R. Simmons and S. Koenig. Probabilistic robot navigation in partially observable environments. *IJCAI-95*.

[22] M.A. Tanner. *Tools for Statistical Inference*. Springer, 1993.

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

[24] S. Thrun. A framework for programming embedded systems: Initial design and results. TR CMU-CS-98-142, CMU, 1998.

[25] S. Thrun, M. Bennewitz, W. Burgard, A.B. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. MINERVA: A second generation mobile tour-guide robot. *ICRA-99*.