A Programming Language Extension for Probabilistic Robot Programming

Sebastian Thrun
School of Computer Science
Carnegie Mellon Univrsity
Pittsburgh, PA
http://thrun.org

Abstract

In recent years, probabilistic techniques have led to improved solutions for many robotics problems. However, no general tools are currently available to aid the development of probabilistic robotic software. This paper presents a programming language extension to C++ that integrates probabilistic computation and learning. Its two main ideas are to make probability distributions as usable as floating-point numbers, and to smoothly integrate function approximators into C++ code. These innovations facilitate the development of robust, probabilistic robot software, as illustrated by a proto-type program for a mail delivery robot.

1 Introduction

Robots, like other sensor-based systems, have to cope with uncertainty. Uncertainty arises from sensor limitations and unpredictable aspects of robot environments. In recent years, there has been an increased interest to apply probabilistic techniques and learning algorithms to problems in robotics. It is now an established fact that in many real-world domains, probabilistic algorithms are more robust than traditional non-probabilistic solutions (see [11] for a survey). However, virtually all published algorithms in this field are tuned to specific robot problems and little, if any, assistance is available for developing new probabilistic programs. This observation raises the question: Can we distill out the principles of probabilistic robotics, and package them in a domain-independent way? And if so, what are these principles?

This paper describes an attempt to develop a special-purpose programming language for developing probabilistic robot programs. This language, called CES (short for: C++ for embedded systems), is an extension of C++, which together with C is arguably the most popular programming language in robotics to date. CES offers two new basic features, currently not found in any programming environment: A mechanism for computing with probability distributions, and a mechanism for incorporating learning algorithms. In particular, CES makes programming with probability distributions as easy as programming with floating-point numbers. The learning mechanism allows for data-driven adaptation of software, so that robot software can be shaped through teaching. Several existing implementations illustrate that our approach reduces the programming effort for probabilistic software by an order of magnitude (or more); the one discussed here is a gesture-driven mail-delivery robot, whose programs is only 137 lines long but requires approximately two hours of training to be functional.

2 Probabilistic Computation

2.1 Probabilistic Data Types

A basic computational entity in CES is a probability distribution. Probability distribution are defined over sets of events of underlying random variables. If the event space of a random variable is of a type `base-type`, the template

```
prob<base-type>
```

defines a probability distribution over the type `base-type`. The base type may be any atomic numerical data type (`int`, `double`, etc.) or composite data type (`struct`, `vector`, etc.). For example, if the base type is `double`, the declaration

```
prob<double> x;
```

declares a variable `x` that is a probability distribution over the one-dimensional continuous space of all doubles.
Just as there are multiple ways to define constants in C++, there are multiple ways to define constants for probabilistic variables. CES offers four different approaches:

1. **Enumeration.** A probability distribution can be specified by a list of the type

   ```
   { {v1, p1}, {v2, p2}, ..., {vN, pN} }
   ```

   where each tuple `{v,p}` specifies a discrete event `v` and its associated probability `p`. The length of the list may vary. For example, the statement

   ```
   x = {{1, 0.5}, {2, 0.5}};
   ```

   assigns to the probabilistic variable `x` the distribution that assigns probability 0.5 to the two values 1 and 2, and probability 0 to all other values in the domain of the variable's base type.

2. **Type conversion.** Point-mass distributions, which assign all probability mass to a single value in the base-type, can be specified by this very value. For example, the assignment

   ```
   x = 3;
   ```

   assigns to `x` a point-mass distribution centered on the value 3. This statement is an example of an implicit type conversion from base types to their probabilistic generalizations.

3. **Library.** Common discrete and continuous distributions are offered through a library. For example, the assignments

   ```
   x = UNIFORM1D(0.0, 1.0);
   y = NORMAL1D(0.0, 1.0);
   ```

   assign to `x` and `y` a (one-dimensional) uniform distribution in $[0; 1]$, and a normal distribution with mean 0 and variance 1, respectively.

4. More complex probability distributions can also be constructed using the `probloop` command, which will be described further below.

Type conversions are also straightforwardly defined. Of particular interest is the conversion of a probabilistic variable to its base type, and vice versa. A statement of the type

```
x = y;
```

assigns to `x` a point-mass distribution if `x` is probabilistic, and `y` is its base type. Conversely, if `y` is the probabilistic variable and `x` is its base type, this statement assigns to `x` the expectation

$$x \;=\; \int \alpha \; P(y = \alpha) \; d\alpha$$

For example, the code segment

```
prob<double> x = {{1, 0.3}, {2, 0.7}};
double       y = x;
```

assigns to `y` the value 1.7. Notice that the expectation of certain base types (e.g,. `int`) might not be part of their domains, in which case the result will be truncated.

2.2 Arithmetic Operations

All arithmetic operators in C++ are applicable to probabilistic variables as well. However, probability distribution are more complex than numbers. Consequently, statements affecting more than one probabilistic variable can be interpreted in more than one way.
To see, consider the example

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

where x and y represent probability distributions. Two possible interpretations of this statement are: (1) it is a multiplication of two independent random variables defined over different (and independent) event spaces, or (2) it is a multiplication of two probability distributions defined over the same event space. To illustrate these different views, let us assume we have the two probabilistic variables

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

According to our first interpretation, x and y correspond to independent events. The set of possible outcomes of the multiplication is then 1, 2, and 4, since each of the two underlying random variables may independently take on the values 1 or 2. Because of our independence assumption, the resulting distribution is then:

```
z = {{1, 0.1}, {2, 0.5}, {4, 0.4}};
```

Under the second interpretation, both random variables take on the same value at all times. The product of x and y is then the product of the individual probability measures, which (after normalization) is the following:

```
z = {{1, 0.2}, {2, 0.8}};
```

To see, we note that the product probability that the underlying event is 1 is given by $x(1) \cdot y(1) = 0.2 \cdot 0.5 = 0.1$. Similarly, $x(2) \cdot y(2) = 0.8 \cdot 0.5 = 0.4$. After normalization, the resulting distribution is the one specified above.
While there are more ways to interpret statements like the ones above, these are the two most important ones. We will call operations that adhere to the first interpretation *type-I* operations, and those that correspond to the second *type-II* operations. They will now be defined in more general terms.

Type-I Operations: Let "$\circ$" be a binary (2-argument) operator (e.g., $+$, $-$, $\cdot$, $/$, etc.), and let x and y be probabilistic variables over the same base type. Then x$\circ$y is also a probabilistic variable that represents the following probability distribution:

$$P(x \circ y{=}\alpha) \quad = \quad \int \int I_{\alpha=\beta\circ\gamma}\ P(x{=}\beta)\ P(y{=}\gamma)\ d\beta\ d\gamma$$

Here I is an indicator variable that is 1 if its argument is true, and 0 otherwise. Put into words, the probability assigned to the event that x$\circ$y takes on a value α is the total probability of all values β and γ in the domains of x and y such that $\beta \circ \gamma$ equals γ. In probability variables with discrete base-type, the integrals are replaced by sums. As the reader should quickly verify, this general definition is compliant with the example in the beginning of this section. In our implementation, type-I operations are overloaded over the conventional function symbols.

Type-II Operations: Type-II operators manipulate probability measures directly, assuming that all operands are defined over the same event space (and hence fully dependent). As before, let "$\circ$" be a binary operator, and x, y probabilistic variables. Then x$\circ$y is the a probabilistic variable with the following distribution

$$P(x \circ y{=}\alpha) \quad = \quad \frac{P(x{=}\alpha)\ \circ\ P(y{=}\alpha)}{\int P(x{=}\beta)\ \circ\ P(y{=}\beta)\ d\beta}$$

Clearly, this expression makes only sense under three conditions: (1) $\circ$ must be defined on real numbers (which is the domain of probabilities), (2) the combination $\mathtt{x}(\alpha) \circ \mathtt{y}(\alpha)$ must be nonnegative, and (3) the denominator must be larger than zero.
A good example for a type-II operator is Bayes rule:

$$P(a|b) \quad \propto \quad P(b|a)\ P(a)$$

Suppose x represents $P(b|a)$ and y represents $P(a)$, Bayes rule is obtained as x$\cdot$y using a type-II multiplication.
In our implementation, type-II operators are realized using textual commands:

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

which distinguishes them from type-I operators. Type-II operators are similar to a language described in [3].

2.3 Probabilistic Statement Execution

A final, powerful tool enables programmers to extend probabilistic type-I calculations to more complex elements of C++, such as loops, recursions, and complex functions.
Suppose we would like to manipulate the elements of a probabilistic variable `x` via some complex function f, but f is defined in conventional, non-probabilistic variables. Following the idea of type-I operations, the resulting distribution is given by the following integral (or sum, in the discrete case):

$$P(f(x)=\alpha) \quad = \quad \int I_{\alpha=f(\beta)}\, P(x=\beta)\, d\beta \tag{1}$$

The *probloop* command generalizes this equation to multiple input and output variables. In particular, a `probloop` command is of the form

```
probloop (x1, ..., xN; y1, ..., yM){
  <sequence-of-statements>
}
```

where the `<sequence-of-statements>` implements the function f. The variables `x1` through `xN` are input variables, and `y1` through `yM` are output variables of f. All of these variables must be probabilistic, though not necessarily of the same type. Inside the loop, however, these variables are reduced to their non-probabilistic base type—just as the use of $\circ$ in the definition of type-I operators applied to the base-type elements of the participating probabilistic variables. This construct makes it possible to 'bootstrap' arbitrary C++ statements to probabilistic variables.
Let us illustrate the `probloop` via an example. Suppose `x` and `y` are given by

```
x = {{3, 0.4}, {4, 0.6}};
y = {{6, 0.2}, {7, 0.8}};
```

Furthermore, consider the statement

```
probloop (x, y; z){
  if (x > 3.5) then z = 0 else z = y;
}
```

with input variables `x` and `y`, and output variable `z`. A truly probabilistic interpretation of this statement will execute the then-clause with .4 probability, and the else-clause with .6 probability. The resulting variable `y` hence will represent the following distribution

```
{{0, 0.4}, {6, 0.12}, {7, 0.48}};
```

In fact, this is exactly what the `probloop` command generates. Conceptually, the `probloop` is executed over all values of the input variables, for which it calculates the corresponding values of `z`, while keeping track of the corresponding probabilities defined by the input distributions.
The probloop can be applied to any sequence of C++ commands, thereby generalizing the idea of probabilistic computation to loops, nested loops, recursion, and any other element of conventional C++ programming. In fact, the binary operators discussed above can all be viewed as special cases of the `probloop` command.
The `probloop` gives a programmer control over the scope of dependencies in his program. For example, the statements

```
x = {{10, 0.4}, {20, 0.6}};
probloop (x; y){
  y = x;
  y = y - x;
}
```

will return the point-mass distribution centered on 0. This is because no matter what value x takes, y will be 0. In contrast, the same statements without the probloop:

```
x = {{10, 0.4}, {20, 0.6}};
y = x;
y = y - x;
```

will treat x and y as independent variables in the second statement, and hence return

```
y = {{-10, 0.16}, {0, 0.48}, {10, 0.36}};
```

2.4 Implementation of Probabilistic Variables

CES separates the interface from the actual implementation of probabilistic variables. However, since CES allows programmers to define distributions over arbitrary data types, many operations cannot be implemented exactly. For example, probabilistic variables of the type prob<double>, if implemented exactly, would require as many parameters as there are different doubles (effectively 10^{11} GB of RAM).
Currently, two implementations exist, using two different representations for probabilistic variables:

1. **Histogram representations** were historically the first [10]. They represents probabilistic variables by histograms over their domain. For example, a variable of type prob<double> might be approximated by a histogram of N cells, covering the support of the distribution in equi-distance intervals. Unfortunately, the histogram representation is not applicable to arbitrary probabilistic variables (e.g., variables defined over complex data structures), for which reasons it was later abandoned.

2. **Sampling.** Here we represent distributions by a set of (weighted) samples, drawn from the base type (domain) of a distribution. For example, consider a probabilistic variable x of type prob<double>. This distribution is represented by N values of type double, which are drawn according to the probability distribution x. Sample representations are common-place in the Markov Chain Monte Carlo (MCMC) literature. They have two pleasing properties: First, they can accommodate any base type, and second, under very mild conditions [1] they converge uniformly to the true probability distribution as N increases.

Under the sampling representation, implementing constants, type-I operations including the probloop command is remarkably straightforward. For example, the following statements

```
prob<double> x = UNIFORM1D(0.0, 1.0);
prob<double> y = NORMAL1D(0.0, 1.0);
prob<double> z = x + y;
```

are translated into: Generate N samples according to a uniform distribution, and memorize all N samples as x. Subsequently, generate N samples according to a normal distribution for y. Finally, generate N pairs of samples $\langle i, j \rangle$ from the sets of samples representing x and y, and memorize their sum $i+j$ as z.

3 Training

The second, important aspect of CES is the ability to tune code using built-in learning mechanisms. Consider, for example, a robot programmed to perform mail delivery. Imagine that the robot occasionally gets lost, or collides with obstacles. One way to remedy this problem would be to analyze and modify the software until the problem is fixed. An alternative way would be to present the robot with target values of the desired behavior in specific situations. For example, if a robot collides, the programmer could specify that the desired behavior (in this situation) would have been a left-turn, whereas the robot executed a right-turn. Similarly, for a robot who lost track of its position, a programmer could tell the robot the correct location. Such situation-specific feedback can be interpreted as training examples for the desired output of the robot in specific situations.
The key idea of learning in CES is similar to Jordan and Rumelhart's notion of *learning with a distal teacher* [2]. Imagine a function approximator (such as a neural network) is used in the program code. One way of training the neural network would be to provide target outputs for the network, and use Back-propagation to tune the weights. However, this would require that target signals are available for the neural network.

An alternative view, put forward in [2] (and adopted here) is to provide target signals for variables that in some way depend in a function approximator. Such a "distal teacher" approach releases the programmer from the burden of having to provide target values for a function approximator directly. In our robot example, a specification that a right-turn was the better action is sufficient to train function approximator, as long as they influenced the actual control decision.

On the flip side, such an approach creates a difficult credit assignment problem: that of assigning credit or blame to the function approximator from a "distal" target signal. Our approach follows the rationale of Jordan and Rumelhart's distal teacher. In particular, CES performs error minimization by gradient descent. But whereas in Jordan's approach gradients were propagated through a neural network, CES propagates gradients through program code. The necessary gradients are all propagated "forward" through the code, so that no back-propagation phase is necessary.

3.1 Function Approximators

Function approximators map floating-point values or vectors of such values into other values or vectors of values. They are declared via the following class template:

```
fa<intype, outtype>;
```

Here `intype` must the type of the input variables, and `outtype` must be the type of the output variables of the function approximator. Both can be doubles, vectors of doubles, or their probabilistic counterparts:

```
double                vector<double>
prob<doube>           prob<vector<doube> >
```

Function approximator declarations require additional information: parameters specifying the type of function approximator, the dimension of the input, and the dimension of the output. In addition, certain function approximators require additional parameters that specify their internal structure. For example, the declaration

```
fa<prob<vector<double> >, prob<double> >
      net(twoLayerNeuronet, 10, 1, 5);
```

declares a neural network with 5 hidden units that maps a 10-dimensional probabilistic vector to a one-dimensional probabilistic variable. Several other options are available, as documented in CES, and users can program their own function approximators as long as they can be trained using gradient descent.

Once declared, function approximators are used just like mathematical functions via the method `fa::eval()`. For example, the following statement assigns to `x` the value of running the function approximator `net` on the variable `y`.

```
x = net.eval(y);
```

Type mismatches in assignments involving function approximators are caught at compile time.

The function `train()`, which is a member of probabilistic variables (and not of function approximators!), is used to for training. For example, the statement

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

specifies that the desired value for the probabilistic 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`. CES then changes each function approximator that took part in the calculation of `x` accordingly.

3.2 Example

To illustrate these concepts, let us consider an example: a program that processes a sensor scan (e.g., a sonar scan) and generates a motion command for the robot that aligns it parallel to the nearest wall. Let us assume that finding the orientation of the nearest wall is a difficult problem, which we much rather train by examples than code by hand. However, once the wall orientation is known, generating the corresponding turning command is trivial. How would one go about coding such a routine? Obviously, the wall orientation cannot be determined with certainty; hence, we use a probabilistic variable:

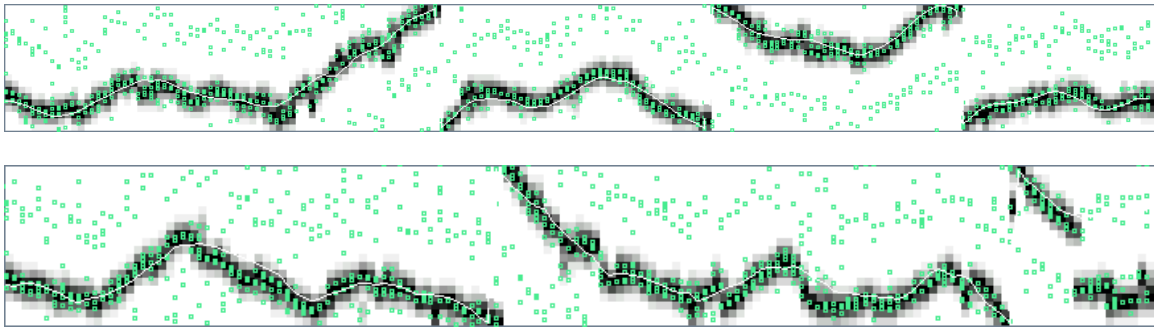

Figure 1: Training set result (top) and testing set result (bottom) of learning to estimate robot orientation (vertical axis) for a 62 second sequence (the horizontal axis is time). The white solid line indicates the true orientation. The small circles are samples, and the gray ares is the result of applying a Gaussian smoother to these samples. Notice that the orientation is tracked very well.

```
prob<double> orientation;
```

Next, we need a function approximator that maps sensor measurements (which are assumed to reside in a vector) to the orientation:

```
fa<vector<double>, prob<double> >
      net(twoLayerNeuronet, 24, 1, 6);
orientation = net.eval(sensors);
```

The motion command is then calculated using a conversion back to doubles. Here is a example:

```
double turn_cmd = double(orientation) + 3.1415;
```

And finally, we need to train the code. If the correct motion command resides in a variable `target_cmd`, training the function approximator is achieved by the following statement:

```
turn_cmd.train(target_cmd);
```

Figure 1 shows, for a code segment similar to the one described here, the result of training. The solid white line depicts the true wall angle as a function of time, for a randomly rotating robot. The small dots are the samples, and the grayly shaded area indicates the robot's probabilistic estimate which, after training, is quite accurate.

3.3 The Importance of Probabilities for Learning

The notion of probabilistic computation is essential for training function approximators via gradient descent. This is because conventional C++ code is usually non-differentiable. An example of a typical C++ statement is the following:

```
if (x > 15) then y = 4 else y = 5;
```

The gradient of the variable `y` with respect to the variable `x`, $\frac{\partial y}{\partial x}$, is zero almost everywhere, except for $x = 15$, where the statement is not differentiable. Zero gradients cause gradient descent to be stuck; hence, no adaptation will occur. This problem naturally arises in C or C++.
The picture is different if `x` and `y` are probabilistic variables. Let $P(x \leq 15)$, and $P(x > 15)$, be the total probability that the distribution `x` assigns to values less than or equal to 15, and to values greater than 15, respectively. Then the gradients

$$\frac{\partial P(y=4)}{\partial P(x \leq 15)} = +1 \qquad \frac{\partial P(y=5)}{\partial P(x \leq 15)} = -1$$
$$\frac{\partial P(y=4)}{\partial P(x > 15)} = -1 \qquad \frac{\partial P(y=5)}{\partial P(x > 15)} = +1$$

are never 0. This comes at no surprise, as probabilities are differentiable even if their base values are not. The differentiability of probabilistic statements encompasses arbitrary *probloop*s and probabilistic variables with complex base types. Thus, the notion of learning and probabilistic computation are highly intertwined in CES.

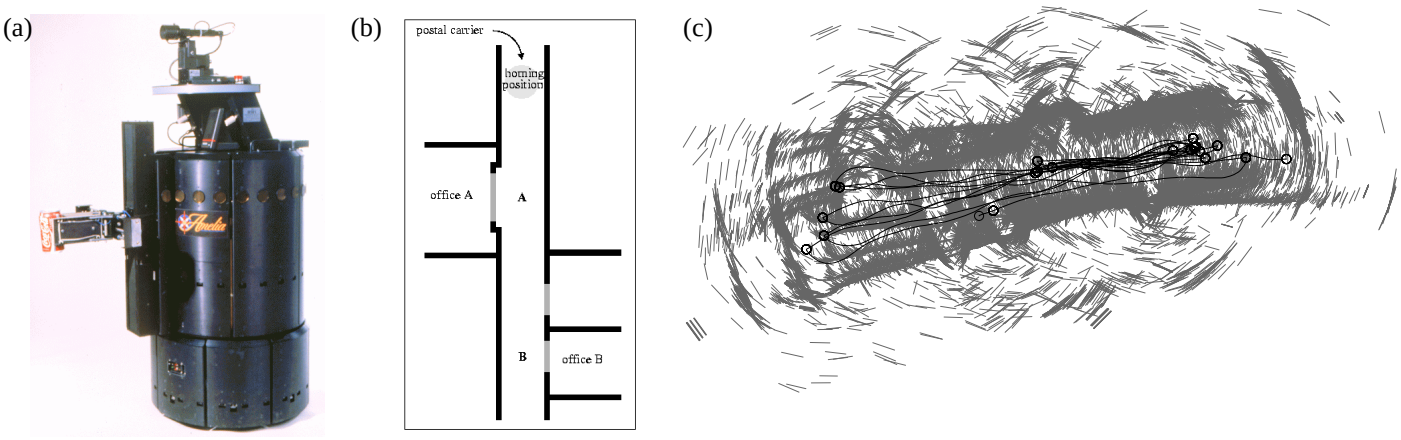

Figure 2: (a) The Real World Interface B21 robot used in our research. (b) Schematics of the robot's environment. (c) Plot of the robot trajectory (raw odometry) during 8 consecutive runs (11 pieces of mail delivered). Shown here are also the raw sonar measurements. The robot reaches the various destination points with high accuracy, despite the rather significant error in the robot's odometry.

3.4 Implementation

The mechanism for propagating gradients through CES code accounts for substantial overhead in the implementation, but it is conceptually relatively straightforward. Our implementation calculates gradients in a forward manner, along with all values (exploiting the chain rule of differentiation). In the histogram representation, gradients are calculated of individual histogram probabilities with respect to any contributing function approximator. The sample-based representation is similar: here gradients measure the posterior probability of a sample with respect to the parameters of a function approximator. When a training signal is available, no further gradients have to be calculated, and the parameters can be adjusted immediately. This effectively implements an online training scheme with a distal teacher.

When the same function approximator is used more than once in the calculation of a variable, CES detects this and exploits the additivity of gradients by integrating multiple gradients into one. The computational overhead of gradient computation can be substantial; however, programs like the one discussed in the next section are easily executed in real-time.

4 Mail Delivery Robot

Our central objective is that CES facilitates the development of robust robot software. This section describes a prototype implementation, which serves as a proof-of-concept. Clearly, a single program is insufficient to demonstrate the utility of a programming framework (see [10] for additional examples). However, it is remarkable that a 137-lines long program (plus less than 2 hours worth of training) suffices to control a mobile robot—from raw sensor data to motor velocity commands. We also remark that CES has *not* been developed with this specific application in mind. Instead, this application was put forward as a challenge after developing CES.

The target application is that of a mail-delivery robot. Figure 2 shows the robot, equipped with 24 sensors and a camera, and a schematic drawing of its environment. The robot is instructed through visual gestures. When delivering mail, it has to navigate to one or two pre-defined target locations, honk a horn, wait for a person to pick up the mail, and return to the home location. There are clearly restrictive assumptions here: For example, the robot knows only two locations for delivery, it operates in a single corridor, it does not monitor its battery, etc. Nevertheless, this task is challenging, since it involves localization, collision avoidance, way-point navigation, and gesture recognition.

Table 1 shows the full program. The final program is of lesser importance, since stripped of its comments it is nearly illegible (like most robot software!). More important is the process of getting there. Programming began with implementing an estimator for wall orientation (lines 32-34 and 53-58). This code uses Bayes rule to incorporate new sensor readings (line 34) and convolution with a normal distribution to accommodate robot motion (lines 54-57). The mapping from sonar sensors to angle estimates is realized using a neural network (line 33) and trained using hand-labeled data (lines 126/127), collected in a few minutes and labeled in less than an hour's time.

```
001: main(){
002:
003:   // ============= Declarations ==============
004:   fa<vector<double>,        prob<double> > netSonar(twoLayerNeuronet, 5);
005:   fa<prob<vector<double> >, prob<double> > netX(twoLayerNeuronet, 5),
006:                                           netY(twoLayerNeuronet, 5);
007:   fa<vector<double>,        prob<int> >    netLeft(twoLayerNeuronet, 5),
008:                                           netRight(twoLayerNeuronet, 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:
070:     // ------------ Gesture Interface & Scheduler ------------
071:     GETIMAGE(image);
072:     gestureLeft  = netLeft.eval(image);
073:     gestureRight = netRight.eval(image);
074:     if (numGoals == 0){                // wait for gesture
075:       if (double(gestureLeft) > 0.5){
076:         stack[numGoals  ].x   = XA;    // location A on stack
077:         stack[numGoals  ].y   = YA;
078:         stack[numGoals++].dir = 1.0;
079:       }
080:       if (double(gestureRight) > 0.5){
081:         stack[numGoals  ].x   = XB;    // location B on stack
082:         stack[numGoals  ].y   = YB;
083:         stack[numGoals++].dir = 1.0;
084:       }
085:       if (numGoals > 0){
086:         stack[numGoals  ].x   = XHOME; // HOME location on stack
087:         stack[numGoals  ].y   = YHOME;
088:         stack[numGoals++].dir = -1.0;
089:         activeGoal = 0;
090:       }
091:     }
092:     else if (stack[activeGoal].dir *   // reached a goal?
093:              (double(y) - stack[activeGoal].y) > 0.0){
094:       SETVEL(0, 0);                    // stop robot
095:       activeGoal = (activeGoal + 1) % depth;
096:       if (activeGoal)
097:         for (HORN(); !GETBUTTON(); );  // wait for button
098:       else
099:         numGoals = 0;
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.

The wall orientation estimator operates in the interval $[0; \pi]$; lines 35-41 and 59-62 generate an orientation estimate in $[0; 2\pi]$, assuming that the initial robot orientation is known (for breaking symmetry). Next, we implemented a localization algorithm in x-y-space, which utilizes the fact that a probabilistic orientation estimate is available by mapping sonar sensors (lines 42-51 and 63-68). This estimator is again trained using a distal teacher (lines 128-131).

Next, we implemented a gesture recognizer (lines 70-72) which uses two neural networks to map images to gestures (one for each hand that can be raised as a gesture). This network is trained with a few minutes worth of data (lines 132-135). The probabilistic output variable becomes input a stack of target locations (lines 74-99). Finally, a very rudimentary way-point navigation system is implemented that drives the robot to its predefined locations (lines 102-122) while stopping if the path is blocked (line 121). Here probabilistic variables are converted back to numerical values (lines 119/120) to determine the actual velocity commands for the robot's motors. The use of probabilistic data types in this program did not require any specific skill, and the entire program was implemented and trained in a matter of one day.

We found that our CES program is robust enough to control a mobile robot reliably in a crowded environment. In extensive tests, the robot navigated for long periods of time (like: 2 hours) in a populated corridor without ever loosing its position or colliding with an obstacle. Figure 2c 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 acceptable 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). Other examples are described in [10].

5 Related Work

Since CES computes with probability distribution, it is immanently related to Bayes networks (BNs) [6]. The relation between CES and BNs is best explained by an analogy: CES is to BNs as are procedural (or object-oriented) programming languages to declarative ones (like Prolog, see also [7]). In BNs, the inference and the knowledge representation are strictly separated, as is the case in Prolog; whereas in CES program statements are computational, like in C and C++. This characteristic has multiple ramifications. For example, Bayes networks (like knowledge bases) can be used for inference from any subset of variables to any other. However, the representation does not include such powerful concepts as recursion, iteration, and nested function calls. CES, like C or C++, allows such things, but CES programs are computational statements that cannot be inverted, or marginalized. We therefore consider CES a viable alternative for software design with probabilistic flavor.

The field of robotics has developed a multitude of special purpose languages for robot programming, such as [4, 9]. These languages address important topics like concurrency and real-time control. Consequently, the issues addressed here are therefore entirely orthogonal. To our knowledge robotics language design has not addressed these issues before.

The idea of learning with prior knowledge has been studied extensively in the machine learning community [8, 12]. Prior knowledge is usually represented in declarative form (e.g., Horn clauses), but to our knowledge the idea of integrating learning into a procedural programming language is novel. The most related approach are evolutionary algorithms (EAs) [5], which modify program code directly instead of built-in function approximators. We believe that a stricter separation of learning and programming increases the human comprehensibility of program code, which is typically rather low for EAs.

6 Conclusion

We have presented a special-purpose language extension of C++ for robot aimed at integrating probabilistic computation and learning into mainstream programming. The language condenses the essentials out of a large number of approaches in the field of probabilistic robotics, by providing programmers with mechanisms for handling uncertainty and learning. We conjecture that these mechanisms are essential for robust software development; however, the real strength of our approach comes from the integration of these mechanism with conventional programming in C++.

This research opens a range of interesting follow-up questions, such as: Can the principles advocated here be stipulated to other programming languages and paradigms? Is it worthwhile to embed other learning approaches, such as reinforcement learning, along with other credit assignment mechanisms into mainstream programming? Is there a place for parametric representations of probability distributions in programming? How can a CES program be debugged, if randomness play a crucial part in its execution?

Despite these open questions, we believe that our current results indicate that probabilistic data types and learning have a clear role to play at the core level of robotic programming language design. We also believe that the ideas presented here transcend beyond robotics, and are applicable to sensor-based embedded systems at large.

Acknowledgement

The author would like to express his gratitude to Frank Pfenning, Sungwoo Park, and Jonathan Moody, whose input has been invaluable.

This research is sponsored by the National Science Foundation (CAREER grant number IIS-9876136 and regular grant number IIS-9877033), which is gratefully acknowledged. 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 the United States Government or the National Science Foundation.

References

[1] W.R. Gilks, S. Richardson, and D.J. Spiegelhalter, editors. *Markov Chain Monte Carlo in Practice*. Chapman and Hall/CRC, 1996.

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

[3] 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.

[4] K. Konolige. Colbert: A language for reactive control in saphira. In *KI-97: Advances in Artificial Intelligence*, LNAI, pages 31–52. Springer verlag, 1997.

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

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

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

[8] S.J. Russell. Prior knowledge and autonomous learning. *Robotics and Autonomous Systems*, 8:145–159, 1991.

[9] R. Simmons and D. Apfelbaum. A task description language for robot control. In *Proceedings of the Conference on Intelligent Robots and Systems (IROS)*, Victoria, CA, 1998.

[10] S. Thrun. A framework for programming embedded systems: Initial design and results. Technical Report CMU-CS-98-142, Carnegie Mellon University, Computer Science Department, Pittsburgh, PA, 1998.

[11] S. Thrun. Probabilistic algorithms in robotics. *AI Magazine*, 21(4):93–109, 2000.

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