Uncertain<T>: A First-Order Type for Uncertain Data

James Bornholt
Australian National University
u4842199@anu.edu.au

Todd Mytkowicz
Microsoft Research
toddm@microsoft.com

Kathryn S. McKinley
Microsoft Research
mckinley@microsoft.com

Abstract

Sampled data from sensors, the web, and people is inherently probabilistic. Because programming languages use discrete types (floats, integers, and booleans), applications, ranging from GPS navigation to web search to polling, express and reason about uncertainty in idiosyncratic ways. This mismatch causes three problems. (1) *Using an estimate as a fact* introduces errors (walking through walls). (2) *Computation on estimates compounds errors* (walking at 59 mph). (3) *Inference* asks questions incorrectly when the data can only answer probabilistic question (e.g., "are you speeding?" versus "are you speeding with high probability").

This paper introduces the *uncertain type* (*Uncertain*$\langle T \rangle$), an abstraction that expresses, propagates, and exposes uncertainty to solve these problems. We present its semantics and a recipe for (a) identifying distributions, (b) computing, (c) inferring, and (d) leveraging domain knowledge in uncertain data. Because *Uncertain*$\langle T \rangle$ computations express an algebra over probabilities, Bayesian statistics ease inference over disparate information (physics, calendars, and maps). *Uncertain*$\langle T \rangle$ leverages statistics, learning algorithms, and domain expertise for experts and abstracts them for non-expert developers. We demonstrate *Uncertain*$\langle T \rangle$ on two applications. The result is improved correctness, productivity, and expressiveness for probabilistic data.

1. Introduction

Applications that sense and reason about the complexity of the physical world use estimates. Examples include GPS sensors, which estimate location; search, which estimates information needs from search terms; software benchmarking, which estimates the performance of different software configurations; and political polling, which estimates election results. The difference between estimates and true values is *uncertainty*. Every estimated value has uncertainty and introduces random error.

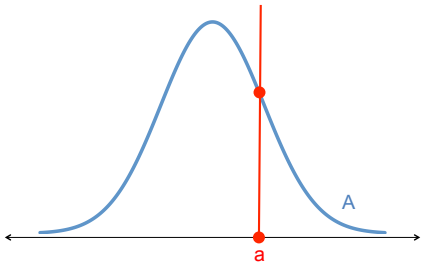


Figure 1: Sampling a probability distribution A produces a single point a, introducing uncertainty. The distribution quantifies potential errors. The sample a does not represent the mean of A, but many programmers treat it as if it does.

Random variables model estimation processes by expressing probability distributions of data. A distribution assigns a likelihood to each possible value of a random variable. For example, a flip of a biased coin may have a 90% chance of heads, and 10% chance of tails. The outcome of each flip is a random variable. Furthermore, the outcome of *one* flip is only a sample, not the expected value of a coin flip in the long run. A probability distribution represents the uncertainty in an observation, as shown in Figure 1. A probability distribution is fundamental to computing with estimates, since it expresses an estimate's uncertainty.

Most programming languages force developers to reason about estimated data with discrete types (floats, integers, and booleans), which do not capture uncertainty in their values. While work in probabilistic programming [10], and libraries for machine learning (e.g., Infer.NET [12]), specific domains [4, 8, 15, 18–20], and statistics (e.g., PaCAL [11], Matlab, R) address parts of this problem, they typically require domain, machine learning, and/or statistics expertise far beyond what many client applications require to consume uncertain data. Consequently, motivated developers reason about uncertainty in ad hoc ways, but because this task is complex, many more simply ignore uncertainty. For instance, we survey over 100 smartphone applications that use GPS APIs, which include estimated error, and find only one reasons about the error.

The mismatch between uncertain data and most programming languages leads three types of *uncertainty bugs*.

Using an estimate as a fact introduces errors because it ignores random noise in data.

Computation compounds errors because each computation on estimated data typically degrades accuracy further.

Inference on estimates creates errors when it asks concrete instead of probabilistic questions.

These uncertainty bugs cause programs to behave in unexpected and incorrect ways.

This paper introduces the *uncertain type*, *Uncertain*$\langle T \rangle$, a programming language abstraction that represents probability distributions of estimated values. The uncertain type defines an algebra on random variables which propagates uncertainty through calculations and inference. It exposes distributions to developers, making it easier to reason correctly with uncertain data. This abstraction gives developers the necessary tools to mitigate uncertainty bugs.

We introduce a four step recipe to create and modify programs that operate on estimated data. (1) Developers identify the probability distribution that underlies their estimated data. This distribution is domain-specific and may come from a library, or may be derived theoretically (e.g., from the central limit theorem), or estimated empirically. (2) Developers perform computations on random variables. The default algebra for computations involving multiple random variables assumes they are independent. Developers must therefore identify any dependent variables in computations and override their joint probability distribution. (3) Developers query distributions to make inferences. Rather than asking deterministic questions, such as "are you speeding?", developers ask probabilistic questions, such as "are you speeding with 99% confidence?" or "on average, are you speeding?". (4) Developers specify domain knowledge with prior distributions (e.g., physics, calendars, maps, facts), which they use to improve the quality of estimates.

The *Uncertain*$\langle T \rangle$ abstraction is useful to library experts, and to application developers, who do not need or want to become sensor, information retrieval, machine learning, or polling experts, but do want to use these results in their applications. Domain experts wrap existing libraries and create new libraries that expose *Uncertain*$\langle T \rangle$ distributions to client applications. The uncertain type benefits experts because its semantics allow for Bayesian inference, making adding models and improving estimates easier. We demonstrate these benefits and the recipe with two case studies (GPS-Walking and SensorLife), and show how *Uncertain*$\langle T \rangle$ helps improve expert and non-expert developer productivity and correctness.

In summary, the contributions of this paper are (1) identifying reasons for and types of uncertainty bugs; (2) a principled abstraction and semantics to addresses these problems; and (3) a demonstration that this abstraction improves productivity and correctness.

2. Background and related work

Probability theory This work defines the semantics of the uncertain type with probability theory and reviews it as needed throughout the paper. (Additional background sources are widely available [2].) The theory of probability rests on the concept of a *random variable*. For example, the outcome of a coin flip is a random variable with a domain Ω of two possible values (heads and tails), Each possible value has a probability associated with it. For a fair coin, both heads and tails have probability 0.5.

A *probability distribution* or *probability density function* $f : \Omega \to [0, \infty)$ represents these probabilities. For discrete variables, the value of $f(x)$ is the probability that the random variable is equal to $x \in \Omega$. Continuous variables require more care. The important point is that the probability distribution completely defines the random variable, because it encodes the probability that it takes on each possible value.

Probabilistic programming Current abstractions for programming with probabilities are too low-level, forcing developers to rephrase simple problems in complex ways. For example, the Church language offers stochastic primitive functions in which developers express probabilistic computations and generative models of data [10]. But it is complex, non-deterministic, and requires expertise in statistics to use correctly, making it inaccessible to many developers.

Infer.NET is a Machine Learning (ML) framework that democratizes ML by embedding a Bayesian inference engine *into* a general programming language [12]. Like Church, Infer.NET programmers write generative models of real world processes. Then, given a sequence of observations of a real world process, Infer.NET will run programs *backward* to infer parameters of the generative model. In contrast, *Uncertain*$\langle T \rangle$ has different goals: we focus on democratizing sound statistical techniques on estimated data for everyday programmers and on combining disparate models.

PaCAL is a software library for computing with probability distributions [11], which also expresses arithmetic computations on random variables. PaCAL, however, is restricted to closed form distributions, whereas *Uncertain*$\langle T \rangle$ includes closed forms and the more common arbitrary distributions. PaCAL requires developers to explicitly express each computation and the distributions that underlie them, and is therefore too low level for many developers. PaCAL lacks a semantics for inference over random variables, which is a common and critical function of programs that use uncertain data.

Domain-specific approaches to uncertainty Many domain-specific approaches offer methods for reasoning over uncertain data. For instance, input devices such as touch screens require software to determine the target of input gestures, which cover an area of the screen rather than a single point, making the intended target ambiguous. Schwarz et al. [18, 19] propose capturing input gestures as distributions to probabilistically determine the target. They represent input gestures as 2D Gaussian distributions, and for complex gestures like dragging, they defer selecting a target until more information is gathered. This approach generalizes to *multimodal inter-*

action, combining multiple input modes to reduce ambiguity from any one source [15], but is limited to input devices.

GPS sensors are a well-known source of uncertainty due to the inherent random error in their sensing process [20]. Mobile systems provide a GPS fix and an error estimate to developers, but current programming languages require developers to create their own idiosyncratic solutions to make use of the error estimate. For example, Newson and Krumm [14] match GPS samples to road maps using a hidden Markov model. Because there is no built-in support that exposes the distribution of GPS samples, they develop their own model. The authors improve on existing map-matching techniques by correctly addressing sampling error, but we cannot expect most developers to go to this extent to correctly address uncertainty, because it requires significant extra expertise and research.

Robustness of programs At a fundamental level, uncertainty is important because it violates the usual assumptions that programs are deterministic in their input. Programs using uncertain data can change their output due only to random error. Chaudhuri et al. [6] introduce a notion of robustness to formalize this loss of determinism. A robust program's output is Lipschitz continuous in its inputs. That is, a program is robust if a change in its input from x to $x+\varepsilon$ results in a change in its output of at most $K\varepsilon$, where K does not depend on x. Robust programs therefore handle uncertain data correctly, because input variation due to random error does not radically change the output. Programs that are not robust are exactly those that are susceptible to uncertainty bugs.

Approximate computing Not all programs require the guarantees provided by a programming language or underlying hardware. Approximate computations allow programmers to specify which parts of their computation are approximate which lets compilers and hardware trade off performance with quality. For example, *loop perforation* compiles an approximate loop into one that only executes a subset of loop iterations [5]. Likewise, EnerJ [17] and Energy Types [7] force programmers to encode approximate computations with static types which hardware can then exploit through energy efficient and approximate computations. These type systems only denote computations as being approximate and unlike $\mathit{Uncertain}\langle T\rangle$ do not combine computations via distributions nor force programmers to ask the right questions of their data.

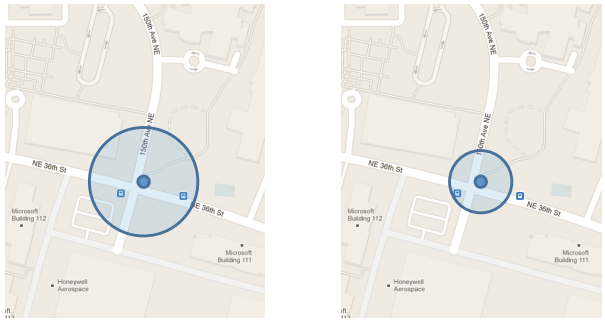
Summary Prior work is either too low-level, requiring programmers to have advanced degrees in statistics and ML, or too domain-specific requiring programmers to reason about uncertainty in ad-hoc ways. This paper introduces $\mathit{Uncertain}\langle T\rangle$: an abstraction targeted to developers who do not need or want to learn the formal background, yet still work with uncertain data.

3. Motivation

This section motivates the uncertain type abstraction by examining applications that compute on GPS sensor data.

```
public class GeoCoordinate {
    public double Latitude;
    public double Longitude;
    // 95% confidence interval for location
    public double HorizontalAccuracy;
}
public class GeoCoordinateWatcher {
    public GeoCoordinate GetPosition();
}
```

Figure 2: The geolocation API on Windows Phone (WP) returns Latitude, Longitude, and a radius of uncertainty.



(a) WP: 95% CI, $\sigma = 33$ m (b) Android: 68% CI, $\sigma = 39$ m

Figure 3: GPS samples at the same location. Although smaller circles appear more accurate, WP is more accurate after normalising the defintions.

Our cursory survey of smartphone applications shows that many use GPS location to compute the user's speed. This section then examines programming pitfalls and resulting errors of current discrete abstractions in more detail, finding discrete abstractions engender errors and errors are prevalent.

Interpreting estimates as facts Sensors are the interface between software and the physical world. Sensor observations estimate physical state, such as temperature, distance, pressure, mass, and location. On smartphones, Global Positioning System (GPS) sensors estimate location.

We surveyed the most popular Windows Phone (WP) and Android applications and found 22% of WP and 40% of Android applications use GPS for location. The GPS abstraction for providing error estimates is similar on both platforms. Figure 2 shows the Windows Phone API. On closer examination, we found only 5% of WP applications read the error estimate, and only one application (Pizza Hut) acts on it. The Pizza Hut application disregards GPS fixes if the error is increasing. Most programmers are ignoring GPS error, and treating estimates as facts. This error leads to absurd results, such as driving in oceans.

Figure 3 shows that even considering the error estimate is not enough–developers must also be interpret it correctly. Both WP and Android show circles to represent horizontal accuracy (smaller circles indicate less uncertainty). But on WP the circle is a 95% confidence interval, whereas on Android it is a 68% confidence interval. In Figure 3, even

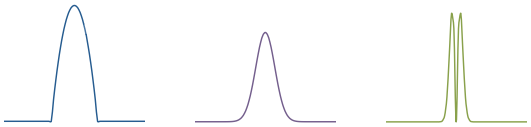


Figure 4: These probability distributions have the *same* mean and horizontal accuracy, but computing with them gives very different results.

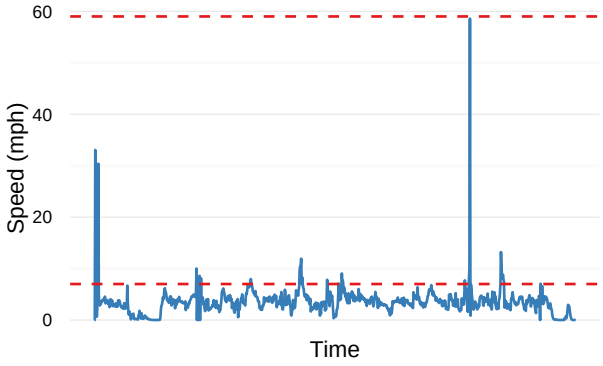


Figure 5: Speed computation on GPS data produces absurd walking speeds (59 mph, and 7 mph for 35 s, a running pace).

though the Android circle is smaller, it is actually less accurate.

Furthermore, even when interpreted correctly, error estimates are insufficient for computing on the underlying distributions. Figure 4 shows three distributions with the same error (i.e., mean and 95% confidence interval), but computing with them gives very different results. These errors all arise due to inappropriate abstractions for uncertain data.

Compounding error Computing on uncertain data compounds uncertainty, and current abstractions do not capture this important result. We performed an experiment that recorded GPS locations on WP while a user walked (ground truth) and use them to estimate speed. Figure 5 shows the calculated speed. The average human walks at 3 mph, and Usain Bolt runs the 100 m sprint at 24 mph. The experimental data shows an average walking speed of 3.5 mph, 35 s spent above 7 mph (a reasonable running speed), and at one point, a patently absurd walking speed of 59 mph. These errors, caused by compounding uncertainty, are significant in both magnitude and frequency.

Because the GPS location is an estimate, computations derived from that location are also estimates. Figure 6 shows how to *correctly* compute speed from GPS samples. The resulting speed is a distribution, and is wider (more uncertain) than the locations. Even if the GPS estimates are very accurate, the speed estimates are not. Current abstractions do not capture this compounding effect because they do not represent the distribution, and do not propagate uncertainty through calculations.

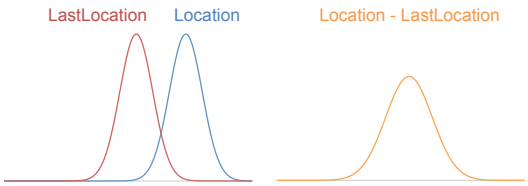


Figure 6: Calculating speed using GPS locations and time. Because location samples are estimates, substraction compounds the error in the estimated speed.

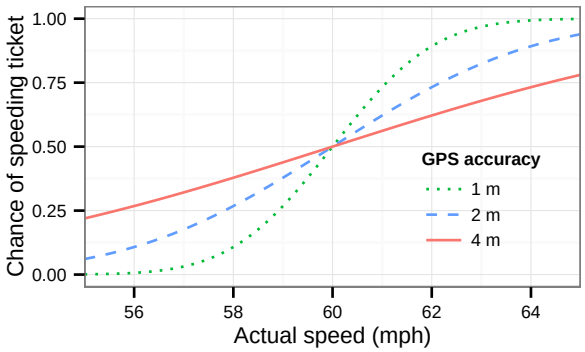


Figure 7: Probability of issuing a speeding ticket at a 60 mph speed limit. With a true speed of 57 mph and GPS accuracy of 4 m, there is a 32% chance of issuing a ticket.

Inference Developers use estimated data to infer answers to questions by using conditionals. Consider the case of using GPS-estimated speed to issue speeding tickets. If the estimated speed is 57 mph with a speed limit of 60 mph, there is a 32% probability of issuing a speeding ticket at 4 m GPS accuracy. Figure 7 graphs this probability across speeds and GPS accuracies. We have not observed GPS samples with accuracy below 4 m.

The problem is that the application is asking a deterministic question about probabilistic data. Instead, it should ask a probabilistic question, only issuing a speeding ticket if the probability is very high (say 95%) that the user is speeding. Without the right abstraction for uncertain data, correctly asking these questions is difficult and error-prone.

Uncertain data abstraction GPS applications are not unique in their treatment of uncertain data. A wide range of modern and emerging applications compute over uncertain data, including web search, benchmarking, medical trials, chemical simulations, and human surveys. Many domains require an expert to correctly characterize the uncertainty in their data as a distribution. But many more developers use this data and will benefit from capturing it in the appropriate abstraction. The next section describes how the uncertain type provides such an abstraction.

Arithmetic operators: $(+ - * /)$
$op :: Uncertain\langle T\rangle \to Uncertain\langle T\rangle \to Uncertain\langle T\rangle$
Equality operators $(< > \leq \geq \neq)$
$op :: Uncertain\langle T\rangle \to Uncertain\langle T\rangle \to Bernoulli$
Logical operators $(\wedge \vee \neg)$
$op :: Bernoulli \to Bernoulli \to Bernoulli$
Sampling methods
$ExpectedValue :: Uncertain\langle T\rangle \to SamplingDist\langle T^*\rangle$
(samples the expected value of the variable)
$Prob :: Bernoulli \to SamplingDist\langle Bernoulli\rangle$
(samples the probability of the *Bernoulli* variable)
$Project :: SamplingDist\langle T\rangle \to T$
(Project from $SD\langle T\rangle$ to summary statistic, T)
Hypothesis testing (for comparisons)
$HypTest :: SD\langle T\rangle \to SD\langle T\rangle \to [0,1] \to Boolean$
(hypothesis test at level α; null hypothesis $A = B$)

n.b. $SD\langle T\rangle$ is shorthand for $SamplingDist\langle T\rangle$.

Figure 8: $Uncertain\langle T\rangle$ operators and methods.

4. A first-order type for uncertain data

We propose a new generic data type, $Uncertain\langle T\rangle$, to capture and manipulate uncertainty as distributions. The $Uncertain\langle T\rangle$ abstraction correctly propagates error through computations over T and helps programmers to reason about uncertainty with proper statistical tests. This section describes the operations and semantics for $Uncertain\langle T\rangle$. Appendix B defines a concrete instantiation of these semantics in C#.

The *uncertain type* represents arbitrary distributions by approximating them with random sampling, which we implement concretely by storing a list of samples of type T. We can optimize this approach for distributions with a closed form, such as Gaussians, by subclassing the uncertain type and overloading operators. We note that many operations reduce from $O(N)$ to $O(1)$ with closed forms, but leave optimization for future work.

There are two sources of error that $Uncertain\langle T\rangle$ addresses: *observation* error and *sampling* error. Observation error is the error induced by the problem domain (e.g. a GPS sensor has inherent limitations which manifest in observational error). In other words, observation error is the difference between a measured value and its true value. Sampling error, on the other hand, is induced by the fact that $Uncertain\langle T\rangle$ cannot always use a closed-form representation of a distribution and has to approximate a distribution with a finite list of samples.

The following section describes the semantics of the uncertain type and how the semantics (i) correctly propagate both forms of error through calculations, (ii) help programmers to use proper statistical techniques, and (iii) correctly and efficiently evaluate inferences and conditionals on uncertain data.

4.1 Semantics of the uncertain type

The semantics of $Uncertain\langle T\rangle$ draw from both theoretical and sampling properties of probability distributions. Our semantics strike a balance between an expressive type, that helps programmers work correctly with uncertain data, and an efficient implementation of that type. Figure 8 provides an overview of the operators and methods for $Uncertain\langle T\rangle$.

Propagating error through computations To help developers write correct code on uncertain data, $Uncertain\langle T\rangle$ defines an algebra over random variables, which propagates uncertainty through calculations. In particular, $Uncertain\langle T\rangle$ lifts arithmetic operations on T to distributions over T via operator overloading. Likewise, to propagate error through comparison operators (i.e., anything with type: $T \to T \to Boolean$), $Uncertain\langle T\rangle$ lifts operations to return a *Bernoulli* distribution, which is an $Uncertain\langle Boolean\rangle$ that takes a value *True* with probability p and *False* otherwise.

Note that for any T, we can create an implicit conversion from T to a point-mass distribution $Uncertain\langle T\rangle$, which means programmers can write statements like $A + 10.0$ where A is an $Uncertain\langle Float\rangle$.

Sampling distributions To handle distributions that do not have a closed form, the $Uncertain\langle T\rangle$ implementation necessarily induces sampling error. To capture this error and incorporate it *into* the semantics, we introduce a type $SamplingDist\langle T\rangle$ which is itself a subclass of $Uncertain\langle T\rangle$. Only two methods extract data from $Uncertain\langle T\rangle$: *ExpectedValue* samples the expected value of an $Uncertain\langle T\rangle$ and *Prob* samples the probability of a *Bernoulli*. Because of sampling error, these methods return sampling distributions, of type $SamplingDist\langle T^*\rangle$ and $SamplingDist\langle Bernoulli\rangle$, respectively.

Notice that the expected value method returns a distribution of type $SamplingDist\langle T^*\rangle$. Because the definition of the sample mean involves division, we must expand the group T to a field T^* which possesses a multiplicative inverse. In practical terms, the sample mean of a distribution over integers of type $Uncertain\langle Integer\rangle$ is a rational number of type *Float*, instead of an integer.

One of the key insights of our abstraction is that these sampling distributions provide an efficient mechanism to evaluate their underlying sample statistics. For example, the Central Limit Theorem says that when sampling the expected value of a distribution A, the result has approximately Gaussian uncertainty, with variance inversely proportional to the square root of sample size, regardless of the distribution of A. Likewise, a *Binomial* distribution approximates successive *Bernoulli* trials, which efficiently infers the parameter p of the *Bernoulli*. These sampling distributions implement both *ExpectedValue* and *Prob* efficiently by dynamically taking only as many samples from the underlying distributions as necessary to evaluate the sample statistic accurately.

To extract a summary statistic from a *SamplingDist*$\langle T \rangle$, we define a *Project* function, which samples appropriately from the underlying distribution until the sample statistic satisfies the theoretical sampling distribution.

There are, of course, other sampling statistics programmers may want to extract from *Uncertain*$\langle T \rangle$, such as order statistics, including *min*, *max*, and *median*. They are simple to add and we leave them for future work.

Inference with hypothesis tests Developers often use variables to make inferences using conditionals. Inference over random variables is more complicated than with discrete types. Other areas of science use statistical tests as a principled mechanism to compare distributions, for example, the t-test. We believe programmers should too. For this reason, the only way a programmer can perform inference on an *Uncertain*$\langle T \rangle$ is through a hypothesis test. The semantics of *Uncertain*$\langle T \rangle$ can hide these details.

ExpectedValue and *Prob* return a *SamplingDist*$\langle T \rangle$ and *SamplingDist*$\langle$*Bernoulli*$\rangle$, respectively. *HypTest*, which compares two *SamplingDist*$\langle T \rangle$s, is the only way a developer can use an *Uncertain*$\langle T \rangle$ type in a conditional. Which underlying hypothesis test our type applies is dependent on the sampling distribution and hidden from the programmer.

These semantics are powerful because they make hypothesis tests opaque to a developer. Because hypothesis tests follow from known sampling distributions with advantageous theoretical properties, they are cheap to evaluate. For example, if we want to compare whether two expected values (represented as two *SamplingDist*$\langle T^* \rangle$s), we simply apply a standard t-test at the 95% confidence level. Likewise, to compare two *Bernoulli* distributions, we use the standard normal approximation to the binomial distribution.

Because the power of a statistical test grows with the sample size N, these approaches present a principled way to select the sample size required to answer a particular hypothesis test. For example, to decide if one expected value is smaller than another, we simply continue a hypothesis test while incrementing N, terminating the process when the test rejects the null hypothesis. Convergence is guaranteed *unless* the expected values are very close or equal, so we impose a maximum on N. In other words, we only draw as many samples as are required to successfully answer the conditional.

We should note a hypothesis test can only *disprove* a hypothesis (e.g., $A == B$ can only return *False* as failure to reject the null hypothesis is *not* the same as accepting it). But programmers are familiar with this behavior. For example, programmers rarely compare two floating point numbers for equality because rounding error makes it unlikely that two floating point numbers are ever equal. Programmer easily side steps this problem by asking the right question of the uncertain data, namely, $(A - \varepsilon < B) \wedge (A + \varepsilon > B)$ implies that A is ε-close to B.

4.2 Programming with the uncertain type

The next sections detail the following four step recipe for programming with *Uncertain*$\langle T \rangle$ to produce more correct, expressive, and accurate programs.

Identifying the distribution of uncertain data is necessarily domain-specific since it depends on the estimation process. In many cases, library writers will provide distributions and application writers will use and combine them, but in some cases applications writers will specify them.

Computing with distributions will include arithmetic operations (e.g., distance and speed), converting units, and combining with other distributions. The uncertain type must perform these calculations over random variables for the broad range of possible distributions.

Inference with distributions requires a new semantics for conditional expressions on probabilities, rather than the usual binary decisions on discrete types. To correctly make inferences using uncertain values, developers must ask probabilistic questions.

Improving estimates with domain knowledge combines and adds various pieces of probabilistic evidence. The uncertain type exploits Bayesian inference to concisely and efficiently combine evidence and explore hypotheses, which makes rich statistical techniques accessible to library developers and advanced application developers.

We expect experts will perform the first and last steps, where as all *Uncertain*$\langle T \rangle$ programmers will compute and make inferences on distributions.

5. Identifying distributions

The first step in programming with uncertain data (i.e., with estimated values) is to identify the underlying random variable and its probability distribution, which is necessarily domain-specific. We envision that libraries written by expert developers will define distributions for applications to use. Because the uncertain type encapsulates distributions, these expert developers will write their libraries to return instances of *Uncertain*$\langle T \rangle$, effectively hiding their details from client applications. The client developers do not need to know the details of how the library computed the distribution. They simply use it by following the recipe.

The expert developer has two broad approaches to selecting the right distribution for their particular problem.

Selecting a theoretical model Many estimation processes have theoretical error distributions. For example, the error in the mean of a dataset is approximately Gaussian distributed by the Central Limit Theorem. Furthermore, the literature abounds with sources of estimated data and their error models. Developers can adopt these models and implement them in their application.

Deriving an empirical model For less common data sources, empirical measurements may determine the right distribution.

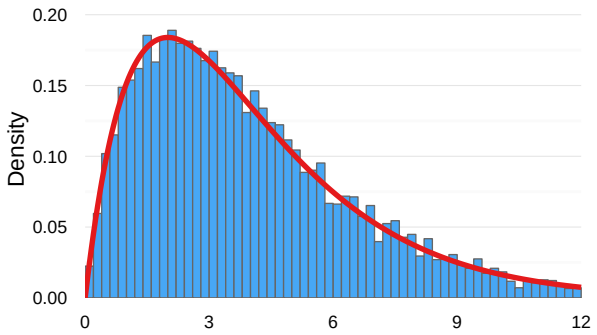


Figure 9: The empirical PDF approximates the exact PDF with a histogram of random samples.

A large enough sample from the underlying distribution approximates the distribution. This idea is similar to statistical bootstrapping, which estimates the distribution of a sample statistic by repeated random resampling.

Application developers may also create distributions by following the above approaches, but we expect many applications will use library implementations developed by experts.

5.1 Representing arbitrary distributions

A random variable is completely defined by its probability density function (or probability distribution). To represent an arbitrary random variable, we therefore need to encapsulate its probability density function.

In simple cases, we can encapsulate the density function exactly. For example, a Gaussian random variable with mean μ and variance σ^2 has density function

$$f(x;\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

Encapsulating this density function for a specific random variable uses this formula and the values of μ and σ^2. This equation represents any Gaussian random variable exactly (up to floating point error) in constant space.

But many important random variables do not have closed form density functions, such as road maps and calendars. Even if a distribution does have a closed form, the algebra for computing operations over it (such as adding two Gaussians) may become complex and unwieldy. To avoid these pitfalls, *Uncertain*$\langle T \rangle$ must represent arbitrarily complex density functions.

We approximate arbitrary density functions with large random samples of the underlying probability space. The intuition is that we sample the domain of the function proportionally to its value, so we are more likely to sample x when $f(x)$ is larger. Specifically, to represent a random variable X, we generate a sequence S_N of N independent random samples of X. The Glivenko-Cantelli theorem [21] tells us that this sequence of samples approximates the exact function f. Formally, it says that as $N \to \infty$, the empirical cumulative distribution function

$$F_N(x) = |\{s \in S_N \,|\, s < x\}| / N$$

converges in the uniform norm almost surely to the exact cumulative distribution function of f, namely $F(x) = \int f(x)\,\mathrm{d}x$.

Given any $\varepsilon > 0$, there is therefore a sample size N such that the error $|F_N(x) - F(x)| < \varepsilon$ for every x. We can thus create arbitrarily good approximations of any density function by taking a large enough random sample. Figure 9 shows this claim in practice. The histogram of 10,000 random samples is an approximation to the exact density function.

Of course, there are practical limits to the value of N, because we must store the value of each sample. The ideal value of N depends on the distribution we are approximating, and cannot be chosen exactly ahead of time. Empirically we have seen $N = 10,000$ to be a good sample size for approximating distributions with one-dimensional probability spaces. In Section 6 we discuss the trade-off that N represents, and in Section 7 we discuss some ways to choose N dynamically based on the questions the developer asks. We leave to future work further optimizing this representation.

6. Computing with distributions

The second step in programming with uncertain data is performing computations on random variables. For example, a developer might average a set of uncertain noise readings over time for use in a filter. Most computations use the four usual arithmetic operators, so we focus our attention on them.

Independent random variables Two random variables are independent if the value of one has no effect on the value of another. For example, two flips of an unbiased coin are independent of each other, because the result of the first flip has no effect on the result of the second flip. Formally, we say two random variables X and Y are independent if the combined random variable (X,Y) has probability density

$$f_{X,Y}(x,y) = f_X(x) f_Y(y). \tag{1}$$

Given two independent random variables, with a a sample of the random variable A, and b a sample of the random variable B, $a+b$ is a sample of the random variable $A+B$. Because we approximate probability distributions by large vectors of independent random samples, to compute $A+B$, we simply sum the vectors representing A and B. The new vector is an approximation of the probability density for $A+B$.

Formally, if $f_X(x)$ and $f_Y(y)$ are the probability densities for two independent random variables X and Y, then the probability density f_{X+Y} for the sum $X+Y$ is the convolution of f_X and f_Y,

$$f_{X+Y}(z) = \int_{-\infty}^{\infty} f_X(z-y) f_Y(y)\,\mathrm{d}y.$$

The intuition for the convolution is that, for a possible value z of the random variable $X+Y$, we consider each possible pair x,y such that $x+y=z$, and sum the probabilities of each of those pairs to find the probability of z.

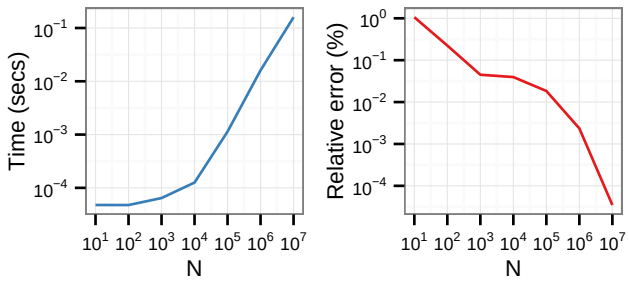


Figure 10: Approximating sum of two Gaussian distributions with the uncertain type. Vector size N controls speed (time) and accuracy (error in approximated mean) tradeoff.

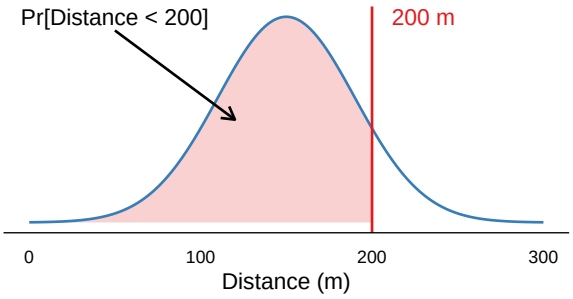


Figure 11: Developers expect simple conditional expressions to be deterministic, but there is a probability that the random variable is larger than 200 m, and that it is smaller.

Dependent random variables Arithmetic operations on dependent variables are not as straightforward because the value of a sample of one depends on the other. The probability of drawing a from A depends on the value of b from B, and requires information about the probability of drawing a from A given each particular possible value of B.

Formally, if two random variables are dependent, the developer must define their joint probability distribution function, written as the conditional probability

$$f_{A,B}(a,b) = f_A(a|B=b) \cdot f_B(b).$$

The probability that $A = a$ and $B = b$ is the probability that $B = b$ multiplied by the probability that $A = a$ given that $B = b$. Notice that if A and B are independent, then $f_A(a|B = b) = f_A(a)$, and this expression for $f_{A,B}(a,b)$ reduces to (1). Since the joint distribution is domain-specific, developers must override operations on dependent random variables to correctly compute with them.

Trading speed for accuracy We approximate the sum of two independent random variables X and Y by summing the two vectors that approximate the distributions for X and Y, which has complexity $O(N)$, where N is the size of the vector. Section 5 shows that the quality of the approximation improves as N becomes larger embodying the classic speed-accuracy trade-off. Larger values of N cause computation to take longer and improve accuracy.

Figure 10 shows an experiment that sums two Gaussian distributions using the uncertain type and records the time it takes. Because there is a closed form solution to this sum, we can measure the error between the approximated mean and the true mean. As N increases, the time to compute the sum increases, but the error in the approximated mean decreases. We choose $N = 10{,}000$ since it offers moderate execution time (10^{-3} s) for a low error rate (0.1%).

7. Inference with distributions

The third step in programming with uncertain data is using it to make decisions. With discrete types, a coffee shop application might trigger a notification if the user is within 200 m of a coffee shop by testing the conditional *DistanceToShop* < 200, and if true, alerting the user. But if we calculate *DistanceToShop* using the GPS, it is an estimate, because locations are estimates and distance is a function of location. Therefore *DistanceToShop* has a distribution. The problem is that the semantics of the expression *DistanceToShop* < 200 are not defined on a distribution. Figure 11 demonstrates this issue. The distribution defines a probability that *DistanceToShop* < 200 (the highlighted area under the curve), but developers expect that semantically, either a conditional is true or not. To introduce probabilities to these semantics creates confusion for developers.

7.1 Asking the right questions

The root of this conflict is at a high level — developers want to ask deterministic questions. In the coffee shop case, the deterministic question is "am I within 200 m of the coffee shop?" With perfect information, this question has a deterministic yes or no answer. But computers and sensors do not deliver perfect information about the physical world. This conflict is a type mismatch. Developers currently ask deterministic questions that the probabilistic cannot always answer correctly.

The questions *Uncertain*$\langle T \rangle$ answers correctly specify the *evidence* for a conclusion, for example, "how much evidence is there that I am within 200 m of the coffee shop?" These types of questions account for the uncertainty of the data in an intuitive style. If the data is very uncertain, the evidence is weaker. They also account for the magnitude of the data. Weaker evidence (more uncertain) is required if the distance is very far from 200 m, whereas stronger evidence (less uncertain) is required if the distance is very close to 200 m. Below, we describe the two mechanisms in *Uncertain*$\langle T \rangle$ for answering questions that leverage different areas of the underlying statistical theory.

7.2 Evidence thresholds

Our first approach is evidence thresholds. Consider the probability $p \in [0,1]$ that *DistanceToShop* < 200 (the area under the distribution to the left of $x = 200$ m) Evidence thresholds

choose a threshold $\alpha \in [0,1]$ and ask if $p > \alpha$. The threshold controls how strong the evidence must be. Intuitively, the system only says yes when *DistanceToShop* < 200 is highly likely. The evidence threshold α controls the trade-off between false positives and false negatives. Higher thresholds require stronger evidence and produce fewer false positives (extra reports when ground truth is false) but more false negatives (missed reports when ground truth is true).

Software benchmarking example One familiar use of evidence thresholds is software benchmarking [3, 9]. Suppose a researcher is developing a soft real-time garbage collector and wishes to evaluate its performance against a deadline [16]. She measures the time taken to perform a collection *GCTime* in milliseconds. The incorrect approach to this problem is trivial:

```
if (GCTime < 10)
  GetGrantMoney(); // Meets the deadline
else LoseTenure(); // Fails the deadline; lose my job!
```

This comparison ignores the effect of uncertainty. Random error in computer systems leads to variation in the runtime of each benchmark [3, 9, 13]. More uncertainty is introduced because it is not feasible to benchmark the entire population of programs, so we choose a sample of programs instead.

Comparing just one measurement, without considering uncertainty, is proven to lead to wrong conclusions [3, 9, 13] due to uncertainty bugs. The correct question must consider data over many benchmarks and executions, which we can capture in the form of a distribution.

There are at least two distinct ways a researcher can benchmark her new GC. Firstly, she can ask if, on average, the *GCTime* is less than 10 milliseconds. Under the hood, *Uncertain*$\langle T \rangle$ uses a t-test at the default 95% confidence interval to mitigate the effect of sampling error on this conditional.

```
if (GCTime.ExpectedValue() < 10)
  GetGrantMoney();    // Meet deadline on average
else if (GCTime.ExpectedValue() >= 10)
  LoseTenure();       // Fails deadline on average
else                  // Sampling error is too great
  HireGradStudent(); // Need more experiments
```

Here we have turned a deterministic question ("does the collector meet the deadline?") into a probabilistic one ("does the collector meet the deadline, on average?").

However, this may be too strict given that the collector is a soft-real time collector. Instead, the researcher may ask if the *GCTime* is less than 10 milliseconds 80% of the time, which as asked as follows:

```
if ((GCTime < 10).Prob() > 0.80)
  GetGrantMoney();    // Meets deadline with 80% prob
else if ((GCTime >= 10).Prob() > 0.80)
  LoseTenure();       // Fails deadline with 80% prob
else                  // The uncertainty is too great
  HireGradStudent(); // Need more experiments
```

Here we have again turned a deterministic question into a probabilistic one ("is there an 80% chance that the collector meets the deadline?"). Under the hood, *Uncertain*$\langle T \rangle$ again uses a hypothesis test at the 95% confidence level to mitigate sampling error. The 95% threshold is a commonly accepted practice for statistical significance. By reasoning about evidence, rather than just treating one sample as fact, we ask the right question and limit uncertainty bugs.

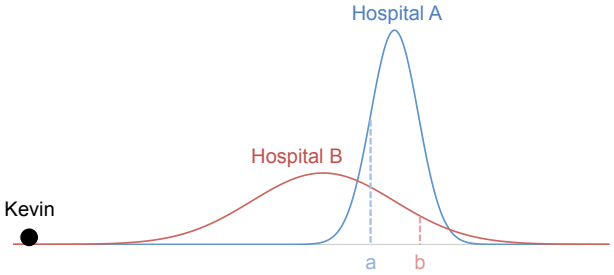


Figure 12: Kevin needs to select a hospital to walk to, but only has estimates of his distance from each hospital. Notice that single samples a and b are not sufficient to choose the right one.

Ternary logic The evidence threshold approach introduces ternary logic when the threshold α is not 0.5. Many developers expect and want a total order for comparisons, i.e., exactly one of $A < B$ or $A \geq B$ is true. With evidence thresholds, sometimes neither $A < B$ nor $A \geq B$ is true with α% confidence. This case corresponds to the "uncertainty is too great" answer to the question. Some problems however require a total order and for these problems, *Uncertain*$\langle T \rangle$ uses expected values.

7.3 Expected values

Uncertain$\langle T \rangle$ exposes the expected value (or mean) $E[A]$ for a distribution A with probability density function $f(x)$, which is defined as

$$E[A] = \int x \cdot f(x)\,\mathrm{d}x.$$

Other choices are however possible, including maximum likelihood and Bayes estimation, but the semantics of the expected value are interesting for two reasons. First, because X is real-valued, so too is $E[X]$. Therefore a total order exists over the expected values of a collection of random variables, resolving the ternary logic issue. Second, the expected value of a random variable is the long-run average value of that variable, so using it to compare two distributions is asking the order of the variables in the average case.

Hospital example Kevin has broken his leg. He *must* visit a hospital and he prefers the closest one. "Not confident enough to choose" is not an acceptable answer. There are two hospitals A and B in Kevin's home town, but he does not know exactly his distance from each of them. Instead, he has only two estimates (i.e., distributions) *DistanceToA* and *DistanceToB*, as shown in Figure 12. Because Kevin is excited about a recent paper he read, he realizes location is a distribution and the evidence threshold approach may not

choose a hospital, so he chooses expected values. Expected values are distinct from samples. Figure 12 shows two samples a and b, but using expected values in the comparison gives the opposite result to using the samples. Kevin writes the following code.

```
double a = DistanceToA.ExpectedValue().Project();
double b = DistanceToB.ExpectedValue().Project();
if (a < b)       GoTo(HospitalA);
else if (a >= b) GoTo(HospitalB);
```

This code eliminates the ternary logic problem, and Kevin's application guarantees he will visit one hospital. Moreover, this interpretation is optimal in the sense that, in the average case, the application will make the right choice of the closest hospital, but may make the wrong choice in individual cases. When Kevin's *iHospitalFinder* application is downloaded and used by millions of users, the average outcome will be correct.

8. Improving estimates with domain knowledge

The previous sections treat the estimation processes that produce uncertain data as immutable, but in many cases, adding domain knowledge to estimation will improve accuracy. *Uncertain*$\langle T \rangle$ unlocks this capability because it captures entire distributions, and may therefore leverage the rich statistical power of Bayesian inference.

8.1 Bayesian inference

Bayesian statistics is an interpretation of probability in which the true state of the world is represented by beliefs. Bayesian probability evaluates questions by first proposing a hypothesis, and then updating that hypothesis based on available data. Importantly, both the hypothesis and the updated hypothesis (known as the *prior* and *posterior*, respectively) are distributions, not discrete values. The distributions represent the belief that the variable takes a value.

This structure makes Bayesian probability well suited for use in working with estimation processes. For example, we can derive a prior hypothesis about a user's speed from physics and transportation mode, and then combine it with an observation of speed from a sensor. Bayesian inference is a principled way to combine evidence with prior knowledge to form a posterior belief about the user's speed. This inference is based on the sampled distribution, which is why the uncertain type makes such inference straightforward. The simplest approach assumes no prior knowledge, in which case the prior distribution is uniform.

Formally, Bayes' theorem takes two random variables, a target variable B and an estimation process E. Initially, the distribution $\Pr[B = b]$ for each b represents our prior assumptions about the target variable. Then we observe a sample e from the estimation process. Bayes' theorem tells us that the posterior distribution is

$$\Pr[B = b|E = e] = \frac{\Pr[E = e|B = b] \cdot \Pr[B = b]}{\Pr[E = e]}.$$

Notice the result is a function of b, that is, a posterior distribution for B. This equation tells us what our updated belief for the target variable B is, given that we observed a piece of evidence e. The term $\Pr[E = e|B = b]$ is a *likelihood model* for E. It is inherent to the estimation process, and represents the likelihood of observing the evidence e assuming that the true value of the target is b. The likelihood model is therefore related to the distribution for the estimation process.

8.2 Incorporating knowledge through priors

This Bayesian approach is powerful, because developers may encode domain knowledge as prior distributions and naturally incorporate that knowledge into estimation processes.

Developers may source prior distributions from other data sources or create new models. For example, a developer may turn a road map into a distribution and use it to support the hypothesis that the user is on a road when driving or that the user is next to the road or on a sidewalk when walking. Alternatively, developers may construct priors from theory. For example, human walking speeds could be Gaussian distributed with a given mean and standard deviation, and a walking application could use this model as a prior distribution for speed.

Because the joint distribution of independent events is found by multiplying their probability densities, library writers may easily combine prior distributions together. We expect that for many data sources, library writers will develop the models for priors using the uncertain type to improve their estimation processes transparently to client applications. These implementations will support application developers who simply wish to use estimated data without considering Bayesian inference or statistics. Some advanced developers may want direct access to these models. Furthermore, some applications may need to combine evidence in new and unexpected ways, and therefore libraries should expose each individual model implemented with *Uncertain*$\langle T \rangle$.

One middle ground is to expose some models through a *constraint abstraction* interface for priors, giving client applications the option to turn on or off different prior distributions on a data source. This middle ground does not require statistical sophistication from application developers. For example, a GPS library may provide prior distributions sourced from calendars (for meeting locations), road maps, location history, etc. The client application can toggle each of these distributions on or off independently, and the library will incorporate the enabled priors into the estimation process. For example, a GPS navigation application specifies domain knowledge over location — the user is driving on roads (a distribution over locations) and in a car (a distribution over speeds). The constraint abstraction therefore makes the rich statistical technique of Bayesian inference accessible to all developers, but implemented by library experts.

9. Case studies

This section demonstrates using the uncertain type and our programming recipe for uncertain data with two case studies. To demonstrate that *Uncertain*⟨*T*⟩ helps programmers easily write code with uncertain data, we apply our recipe to a GPS-based walking application. To demonstrate *Uncertain*⟨*T*⟩ correctly deals with errors in estimations, we induce noise into a game and show that with the uncertain type, a program can simply and correctly deal with error.

9.1 GPS-Walking

Our first example is a smartphone application that uses the GPS to measure the user's walking speed, a common feature of fitness applications. GPS-Walking estimates the user's walking speed by taking two samples from the GPS and computing the distance and time between them. Figure 13 shows the code for the main loop of GPS-Walking before and after being updated to use the uncertain type.

Defining the distributions The first step is to identify and define the distributions. We use the GPS sensor to estimate location. We can model GPS as a random variable with a probability distribution over location. Because most smartphones have GPS libraries provided by the operating system, we assume an expert developer updates the GPS library to use the uncertain type and derives the error distribution for a GPS observation. We outline this process for GPS theoretically in Appendix A, and present an implementation in Appendix C. The updated GPS library provides the function

```
Uncertain<GeoCoordinate> GPSLib.GetGPSLocation();
```

which returns a distribution over locations, representing a belief about the user's current location.

Computing with random variables Because the locations returned by `GetGPSLocation` are estimates, so too is the distance calculated by `Distance`. Therefore, speed is an estimate. Since `Distance` returns an *Uncertain*⟨*Double*⟩, speed is also an *Uncertain*⟨*Double*⟩.

This type change requires a change in the application code, because the speed is now of type *Uncertain*⟨*Double*⟩, but the call to `Display` does not understand distributions. We choose to throw away information, because we do not want to display the entire distribution. Instead we display just the mean of the speed distribution, writing:

```
Display(Speed.ExpectedValue().Project());
```

This method of calculating speed captures the compounding of error from each of the two GPS distributions. Figure 6 shows this effect in a 1D world. Even if two location fixes are very good, the resulting calculation of speed is still very uncertain. For example, two GPS fixes with 95% confidence intervals of 4 m result in a speed with a 95% confidence interval of 12.7 mph.

The uncertainty of the speed calculation explains the absurd values in Figure 5. Figure 14(a) shows the same data but with 95% confidence intervals, which we can calculate easily because we are now using the uncertain type. The size of these confidence intervals shows that the absurd results are due to random error. This effect is easily revealed because we use the uncertain type; the developer did not need to change the calculation code at all.

Inference over random variables Early work on human locomotion suggested that humans walking faster than approximately 4 mph used more energy than running. More recent work shows this hypothesis is likely incorrect, but regardless we will sound an alarm whenever the user walks faster than 4 mph, suggesting they slow down. We want to write the conditional `Speed > 4`. But speed is an estimate, so the uncertain type forces us to interpret this conditional. Our guiding principle is not to annoy the user with false positives, because speed is a very noisy estimate. We use the evidence threshold approach and write

```
if ((Speed > 4).Prob() > 0.75) {
    SoundSpeedAlarm();
}
```

This code only sounds the alarm if the evidence says there is a 75% chance that `Speed > 4`. The choice of 75% is arbitrary and reflects the developer's choice of the balance between false positives and false negatives.

Improving GPS estimates with priors The power of the uncertain type is that we can incorporate prior knowledge to improve the quality of estimated data. For GPS-Walking, we assume the user only uses the application when they are walking, and so can assume they are walking. (Alternatively, a walking detection algorithm could use accelerometer sensors to generate a probability distribution, which we leave for future work.) We incorporate this domain knowledge by constructing a prior distribution about the user's speed. It is incredibly unlikely that a human is walking at 60 mph or even 10 mph. This prior distribution does not have to be a perfect truth. Furthermore, strong evidence from the GPS may override it.

The code in Figure 13(b) implements this improvement using the constraint abstraction. The library developer adds support for incorporating speed priors into GPS locations, and the application developer provides their chosen speed prior to the GPS library. Figure 14(b) shows the improved results achieved by using this prior knowledge. The confidence interval is much smaller than in Figure 14(a), and the physics model removes the absurd results, such as walking at 59 mph.

Summary GPS-Walking uses the uncertain type to display the user's walking speed. The developer only made minimal changes to their application code, but is rewarded with improved correctness, by both reasoning correctly about absurd data (Figure 14(a)) and by eliminating it with domain knowledge (Figure 14(b)). This complex logic is difficult to implement without the uncertain type, as the application developer must know the error distribution for GPS, how to

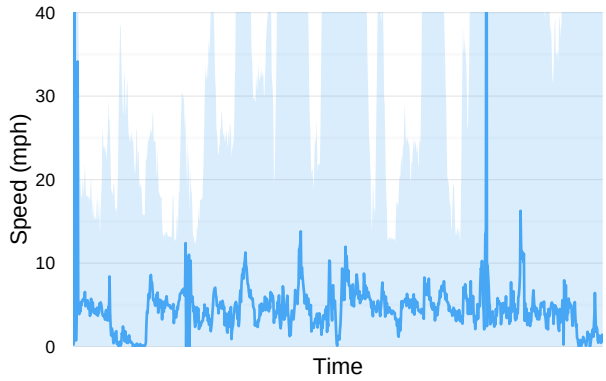
```
int dt = 1;


GeoCoordinate LastLocation
  = GPSLib.GetGPSPosition();

while (true) {
    Sleep(dt); // wait for dt seconds
    GeoCoordinate Location
      = GPSLib.GetGPSPosition();
    double Speed =
      GPSLib.Distance(Location, LastLocation) / dt;
    Display(Speed);
    if (Speed > 4) {
      SoundSpeedAlarm();
    }
    LastLocation = Location;
}
```

(a) Without the uncertain type.

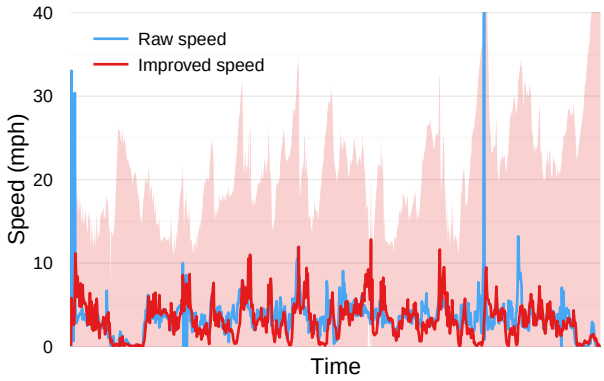
```
int dt = 1;
Uncertain<double> SpeedPrior
  = Uncertain<double>.Gaussian(0, 2);
Uncertain<GeoCoordinate> LastLocation
  = GPSLib.GetGPSPosition(GPSLib.WALKING, SpeedPrior);

while (true) {
    Sleep(dt); // wait for dt seconds
    Uncertain<GeoCoordinate> Location
      = GPSLib.GetGPSPosition(GPSLib.WALKING, SpeedPrior);
    Uncertain<double> Speed =
      GPSLib.Distance(Location, LastLocation) / dt;
    Display(Speed.ExpectedValue().Project());
    if ((Speed > 4).Prob() > 0.75) {
      SoundSpeedAlarm();
    }
    LastLocation = Location;
}
```

(b) With the uncertain type.

Figure 13: The main loop of GPS-Walking, before and after being updated to use the uncertain type.



(a) Confidence interval for speed.



(b) Incorporating prior knowledge from a physics model. Confidence interval is for improved speed.

Figure 14: Data from the GPS-Walking application. The uncertain type allows the developer to calculate confidence intervals for the estimated speed, and also to incorporate prior domain knowledge to improve the estimate.

propagate it through calculations, and how to incorporate extra domain knowledge to improve the results. The uncertain type abstraction hides this complexity from application developers, improving programming productivity and application correctness.

9.2 Conway's Uncertain Game of Life: SensorLife

This section demonstrates how *Uncertain*$\langle T \rangle$ simplifies error handling for an application that employs noisy digital sensors. In particular, we emulate a ubiquitous binary sensor with Gaussian noise. This case study serves two purposes. First, it demonstrates how *Uncertain*$\langle T \rangle$ enables non-expert programmers to work with noisy sensors and how expert programmers can simply and succinctly use domain knowledge (i.e., the fact that the sensor has Gaussian noise) to improve the sensor's estimates. Second, because we *induce* noise into sensors, we have ground truth for comparing the results of a noisy program to one without noise.

Conway's Game of Life (SensorLife) is a cellular-automata simulation that operates in a two-dimensional grid of cells. Each cell is in one of two states: dead or alive. The game is broken up into generations. During each generation, the program updates each cell in the 2D world by (i) sensing the state of the cell's 8 neighbors, (ii) summing the binary value (i.e., dead or alive) of each of those 8 neighbors, and (iii) using the sum in the following rules.

1. Any live cell with less than 2 live neighbors dies due to under-population.
2. Any live cell with 2 or 3 live neighbors lives on to the next generation.
3. Any live cell with more than 3 live neighbors dies due to overcrowding.
4. Any dead cell with exactly 3 live neighbors becomes alive due to reproduction.

Despite simple rules, mathematicians and computer scientists have found that the Game of Life provides complex and interesting dynamics (e.g. Game of Life is Turing complete [1]).

Defining the distributions The standard implementation of Game of Life does not have error. We induce error into SensorLife in order to compare the results of the error free implementation with the uncertain one. To induce error, we simulate sensors with Gaussian noise. In particular, every cell senses whether its 8 neighbors are alive or dead using 8 distinct sensors. We model each sensor as a binary sensor with added Gaussian noise (e.g., $s + N(0,\sigma)$ where s is the binary value of the sensor and σ defines the amount of noise in each sensor).

Computing with random variables In each generation, SensorLife (i) senses each of its 8 neighbors, (ii) sums all 8 sensors and (iii) applies the above rules to determine if the cell is alive or dead in the next generation. The original application uses 4 conditionals, whereas *Uncertain*$\langle T \rangle$ applies 4 t-tests. To get ground truth, we perform the same steps in concert, but without added noise.

Errors in each sensor are uncorrelated and as such, the only changes required to the SensorLife are to change the SenseNeighbors function. As the name suggests, this function senses whether the 8 neighbors of a cell are alive or dead, and returns an *Uncertain*$\langle Double \rangle$ instead of an integer, which is the sum of all 8 sensors.

Inference over random variables After calling SenseNeighbors, we implement each of the above rules using hypothesis tests at a user-defined confidence level. For example, to implement rule 1. listed above, which determines if a cell lives to the next generation, we write:

```
// @ 60% confidence level
if ((numNeighbours.ExpectedValue() < 2.0).HypTest(0.6))
  shouldLive = false;
```

Improving estimates This section demonstrates how to use domain knowledge, specifically the type of noise on each sensor, to improve the estimates in SensorLife.

Suppose we observe a value v from a sensor. By definition, v is drawn from either $N(0,\sigma)$ or $N(1,\sigma)$ (i.e., v is either a 0 with added noise, or 1 with added noise). The result is two *hypotheses* as to the origin of v: H_0, which implies v is drawn from $N(0,\sigma)$, and H_1, which implies v is drawn from $N(1,\sigma)$.

To improve estimates, we note that a sensor reading, v, is *evidence*, and Bayes' theorem provides a mechanism to calculate posteriors $\Pr[H_0|E=v]$ and $\Pr[H_1|E=v]$ given this evidence. In other words, Bayes' theorem provides us with a mechanism to calculate the most likely source of v, H_0 or H_1, respectively, *given* the evidence, v. To *improve* an estimate, we calculate the most likely source of v and then "fix" the sensor reading to be 0 or 1, depending on whether $\Pr[H_0|E=v] > \Pr[H_1|E=v]$.

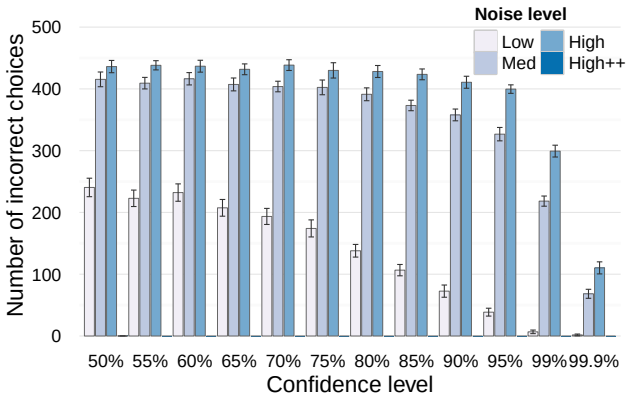


Figure 15: *Uncertain*$\langle T \rangle$ mitigates errors in uncertain programs

To use Bayes' theorem to calculate these posteriors given our evidence, v, requires we add prior knowledge about our domain. In particular, we need (i) the likelihood of H_0 and H_1, and (ii) a way to calculate the likelihoods $\Pr[E=v|H_0]$ that v is drawn from H_0, and $\Pr[E=v|H_1]$ that v is drawn from H_1.

For this example, we assume $\Pr[H_0] = \Pr[H_1] = 0.5$ (i.e., we have no prior knowledge whether a cell is more likely to be dead or alive). Because we know the error model is Gaussian, we use the Gaussian density function to calculate the probability of the evidence v being drawn from H_0 and H_1:

$$\Pr[E=v|H_0] = \frac{1}{\sigma\sqrt{2\pi}} \exp^{-\frac{(v-0)^2}{2\sigma^2}}$$
$$\Pr[E=v|H_1] = \frac{1}{\sigma\sqrt{2\pi}} \exp^{-\frac{(v-1)^2}{2\sigma^2}}$$

We just substitute v into the Gaussian density function and read out the value. Given these probabilities, we use Bayes' theorem to solve for the posteriors $\Pr[H_1|E=v]$ and $\Pr[H_0|E=v]$:

$$\Pr[H_0|E=v] = \frac{\Pr[E=v|H_0]\cdot\Pr[H_0]}{\Pr[E=v|H_1]\cdot\Pr[H_1] + \Pr[E=v|H_0]\cdot\Pr[H_0]}$$
$$\Pr[H_1|E=v] = \frac{\Pr[E=v|H_1]\cdot\Pr[H_1]}{\Pr[E=v|H_1]\cdot\Pr[H_1] + \Pr[E=v|H_0]\cdot\Pr[H_0]}$$

With these posteriors in hand, the output of the faulty sensor is 1.0 if H_1 is more likely to be the source of the sample, and 0.0 otherwise.

Evaluation We use SensorLife, with noise, to evaluate how *Uncertain*$\langle T \rangle$ allows non-expert programmers to succinctly and correctly deal with estimates in their program.

Figure 15 demonstrates that as we change the confidence level of the hypothesis test we affect the accuracy — the number of times our *Uncertain*$\langle T \rangle$ program sets a cell to alive when it should be dead. To collect these data, we ran SensorLife 30 times, each time for 25 generations. Each run of

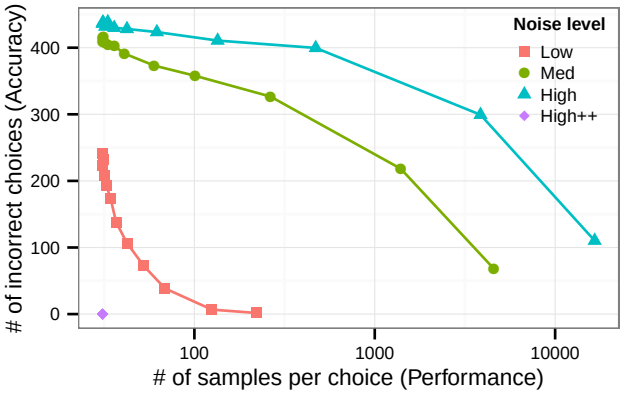


Figure 16: *Uncertain*$\langle T \rangle$ allows programmers to balance performance and accuracy

the program evaluates 80,000 cells in total and counts the total number of incorrect cell updates over those 25 generations. Each execution is parameterized by the confidence level (x axis) and executes in three configurations with different amounts of Gaussian noise added to each sensor: Low noise (i.e., $N(0, 0.01)$), Med noise (i.e., $N(0, 0.05)$), and High noise (i.e., $N(0, 0.1)$). The fourth configuration High++ is the High noise configuration with our improved estimates applied to the sensor output, as described in Section 9.2. A bar on this graph (x,y) demonstrates how, as the confidence level (x) increases, so too does the accuracy of SensorLife (y).

Even with very high noise, the *Uncertain*$\langle T \rangle$ type and the correct question mitigates the errors. At the 99.9% confidence level with high noise, on average, the errors drop to 100 out of 80,000 cell updates, versus 200 to 400 at lower confidence levels. As the confidence interval increases, errors reduce, showing that *Uncertain*$\langle T \rangle$ helps programmers mitigate errors with a single parameter. But for all confidence intervals above 50%, High++ makes no incorrect choices, mitigating all the noise and error. This result demonstrates that by making it easy to improve estimates, *Uncertain*$\langle T \rangle$ gives developers a powerful tool for delivering high accuracy even with large uncertainty.

The ability to mitigate noise has a cost. For example, the uncertain type has to do more work per decision (branch) of the program, as we evaluate a hypothesis test for each branch. Figure 16 demonstrates that if a program requires higher accuracy (by settings a higher confidence level), it necessarily has to do more work per branch (take more samples at each branch). A point on this graph (x,y) plots the number of samples required to come to a conclusion per hypothesis test in our program (x) against the accuracy of the result of that branch (y). As we increase the confidence level (each point on a line is a different confidence level), the overheads increase.

The High++ approach demonstrates that even with high noise, *Uncertain*$\langle T \rangle$ mitigates incorrect decisions due to noise by adding better models. High++ makes no incorrect choices on average, and only requires a handful of samples (30) per hypothesis test (i.e., in Figure 16, High++ is a point at $(30, 0)$ for all confidence intervals). By making it easy for programmers to combine models and evidence, *Uncertain*$\langle T \rangle$ achieves both better efficiency and accuracy.

10. Conclusion

Programmers need help as they seek to solve increasingly ambiguous and challenging problems with big and little data from sensors, biological and chemical processes, nondeterministic algorithms, and people. These applications need to operate over uncertain data. This paper identifies three fundamental problems programmers currently face when programming with uncertain data in imperative languages and how they lead to bugs. (1) Treating estimates as facts quickly leads to wrong conclusions. (2) Computation on estimates compounds errors, leading to more bugs. (3) Inference on estimates must consider probability to produce sensible answers.

We propose a new abstraction, called the uncertainty type, and show how it helps experts and application developers to correctly operate and reason with estimated data. We describe its syntax and semantics. We use two case studies to explore the job of writing libraries and application code with *Uncertain*$\langle T \rangle$. We show how the semantics and concision of *Uncertain*$\langle T \rangle$ improves programmer productivity, expressiveness, and correctness. We point to future work for improving efficiency and modeling common phenomena, such as physics, calendar, and history in *Uncertain*$\langle T \rangle$ libraries to better support application writers. Although we focus on mobile sensor examples, we believe that *Uncertain*$\langle T \rangle$ will help developers with large data sets as well.

References

[1] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways for Your Mathematical Plays, Volume 4*. A K Peters, Wellesley, MA, 2004.

[2] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, NY, 2006.

[3] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, OOPSLA 2006, Portland, OR, USA, October 22 - 26, 2006*. ACM, 2006.

[4] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[5] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Verified integrity properties for safe approximate program transformations. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation, PEPM 2013, Rome, Italy*. ACM, 2013.

[6] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving programs robust. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, FSE 2011, Szeged, Hungary, September 5 - 9, 2011*. ACM, 2011.

[7] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. In *Proceedings of the 27nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, OOPSLA 2007, Tucson, AZ, USA, October 21 - 25, 2012*. ACM, 2012.

[8] J. Droppo, A. Acero, and L. Deng. Uncertainty decoding with SPLICE for noise robust speech recognition. In *Proceedings of the 27th International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2002, Orlando, FL, USA, May 13 - 17, 2002*. IEEE, 2002.

[9] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, OOPSLA 2007, Montreal, QC, Canada, October 21 - 25, 2007*. ACM, 2007.

[10] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, UAI 2008, Helsinki, Finland, July 9 - 12, 2008*. AUAI Press, 2008.

[11] S. Jaroszewicz and M. Korzeń. Arithmetic operations on independent random variables: A numerical approach. *SIAM Journal on Scientific Computing*, 34: A1241–A1265, 2012.

[12] T. Minka, J. Winn, J. Guiver, and D. Knowles. Infer.NET 2.5, 2012. URL `http://research.microsoft.com/infernet`. Microsoft Research Cambridge.

[13] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems, ASPLOS 2009, Washington, DC, USA, March 7 - 11, 2009*. ACM, 2009.

[14] P. Newson and J. Krumm. Hidden Markov map matching through noise and sparseness. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2009, Seattle, WA, USA, November 4 - 6, 2009*. ACM, 2009.

[15] S. Oviatt. Ten myths of multimodal interaction. *Commun. ACM*, 42(11):74–81, Nov. 1999.

[16] T. Printezis. On measuring garbage collection responsiveness. *Science of Computer Programming*, 62(2):164 – 183, 2006.

[17] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI 2011, San Jose, CA, USA, June 4 - 8, 2011*. ACM, 2011.

[18] J. Schwarz, S. Hudson, J. Mankoff, and A. D. Wilson. A framework for robust and flexible handling of inputs with uncertainty. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology, UIST 2010, New York, NY, USA, October 3 - 6, 2010*. ACM, 2010.

[19] J. Schwarz, J. Mankoff, and S. Hudson. Monte carlo methods for managing interactive state, action and feedback under uncertainty. In *Proceedings of the 24th annual ACM symposium on User interface software and technology, UIST 2011, Santa Barbara, CA, USA, October 16 - 19, 2011*. ACM, 2011.

[20] R. Thompson. Global positioning system: the mathematics of GPS receivers. *Mathematics Magazine*, 71(4):260–269, 1998.

[21] F. Topsøe. On the Glivenko-Cantelli theorem. *Zeitschrift fr Wahrscheinlichkeitstheorie und Verwandte Gebiete*, 14:239–250, 1970.

[22] F. van Diggelen. GNSS Accuracy: Lies, Damn Lies, and Statistics. *GPS World*, 18(1):26–32, 2007.

Appendix A GPS error distribution

Expert developers must define error distributions for uncertain data. This appendix shows how to derive an error distribution for a GPS observation. The expert developer would perform this derivation and use it to update the GPS library, replacing the method

```
GeoCoordinate GetGPSLocation();
```

with the new method

```
Uncertain<GeoCoordinate> GetGPSLocation();
```

which expresses a distribution over possible locations.

We adopt the convention within equations that discrete values like $Actual_t$ are set in italics and random variables like $\mathbf{GPS}_t$ are set in bold.

Theoretical setup Formally, we can express the GPS process in the following way. Define

$$World := [-90, 90] \times (-180, 180]$$

and say that at time t our true location is the point

$$Actual_t := (TrueLat_t, TrueLong_t) \in World.$$

Then the GPS sensor's view of our location at time t is a random variable

$$\mathbf{GPS}_t = (TrueLat_t + \mathbf{LatErr}_t, TrueLong_t + \mathbf{LongErr}_t).$$

Here the random variables $\mathbf{LatErr}_t$ and $\mathbf{LongErr}_t$ represent the error in each direction, due to inherent flaws or biases in the sensor and due to environmental conditions at time t (such as atmospheric conditions, obstructions, etc.).

The act of taking a GPS sample at time t is the act of drawing a sample of the random variable $\mathbf{GPS}_t$, which yields a discrete point

$$Sample_t = (SampleLat_t, SampleLong_t).$$

It is this discrete point that most geolocation libraries provide today and this pointmass representation clearly ignores the distribution of $\mathbf{GPS}_t$.

The distribution of $\mathbf{GPS}_t$ clearly depends on the distributions of $\mathbf{LatErr}_t$ and $\mathbf{LongErr}_t$. Knowing these distributions exactly based on theory is a difficult problem, but the literature suggests a model which we will adopt [22]. This model says that $\mathbf{LatErr}_t$ and $\mathbf{LongErr}_t$ are independent and identically distributed (i.i.d.), and follow a normal distribution, with mean zero and an unknown variance. Formally, this says that

$$\mathbf{LatErr}_t \sim N(0, \sigma_t^2)$$
$$\mathbf{LongErr}_t \sim N(0, \sigma_t^2)$$

where the fact that the mean is zero reflects an unbiased sensor, and the fact that σ depends on t reflects the environmental conditions at time t.

Belief in location The first important problem with the theoretical derivation is that of course our software does not know $Actual_t$. What we are trying to do is estimate the value of $Actual_t$ based on observations from the GPS sensor. Bayesian statistics provides a framework to represent this approach. We introduce a random variable $\mathbf{Location}_t$, which represents our software's *belief* about our location. Initially, before taking a sample, we know nothing about our location; that is, for every point p in the world,

$$\Pr[\mathbf{Location}_t = p] = Uniform$$

If we had an oracle, it could tell us the "perfect" belief in $\mathbf{Location}_t$, namely

$$\Pr[\mathbf{Location}_t = Actual_t] = 1$$
$$\Pr[\mathbf{Location}_t = p] = 0, \quad p \neq Actual_t.$$

We use Bayes' theorem to incorporate the GPS sensor as evidence into our belief about $\mathbf{Location}_t$. Intuitively, Bayes' theorem tells us that, if we observe a GPS sample $Sample_t$, the most likely values of $\mathbf{Location}_t$ are exactly those locations most likely to cause the GPS to generate that sample. So, for example, it is unlikely that $\mathbf{Location}_t$ is a long distance from $Sample_t$, because it is unlikely according to our theoretical derivation that the GPS would generate a sample a long distance from the true location. Formally, Bayes' theorem says that

$$\Pr[\mathbf{Location}_t = p | \mathbf{GPS}_t = Sample_t]$$
$$\propto \Pr[\mathbf{GPS}_t = Sample_t | \mathbf{Location}_t = p] \cdot \Pr[\mathbf{Location}_t = p].$$

Notice that the left hand side is a function of p. For each point p in the world, this function gives the probability that the true location is p given that we observed a GPS sample $Sample_t$. This function is called the *posterior* distribution for $\mathbf{Location}_t$, because it represents our belief about the true location after observing a sample from the GPS. Because we have no prior knowledge about $\mathbf{Location}_t$, we assumed that $\Pr[\mathbf{Location}_t = p] = 1$, which simplifies this function to

$$\Pr[\mathbf{Location}_t = p | \mathbf{GPS}_t = Sample_t]$$
$$\propto \Pr[\mathbf{GPS}_t = Sample_t | \mathbf{Location}_t = p]. \quad (2)$$

It is this posterior distribution that we want to return from the GPS sensor. Rather than a discrete point, this distribution captures how likely the user is to be standing at each point in the world, given the evidence from the GPS.

Deriving a likelihood model In the expression for the posterior distribution, the term

$$\Pr[\mathbf{GPS}_t = Sample_t | \mathbf{Location}_t = p]$$

is a *likelihood model* for the GPS sensor. It captures the likelihood of the GPS generating the particular sample $Sample_t$

if the true location was p. Substituting p into the expression for **GPS** gives us

$$\mathbf{GPS}_t = (p_{Lat} + \mathbf{LatErr}_t, p_{Long} + \mathbf{LongErr}_t).$$

So the likelihood that $\mathbf{GPS}_t = Sample_t$ is exactly the likelihood that

$$(\mathbf{LatErr}_t, \mathbf{LongErr}_t) = Sample_t - p. \tag{3}$$

But in our model, we do not know the scale of the distributions for $\mathbf{LatErr}_t$ and $\mathbf{LongErr}_t$, so we cannot evaluate this likelihood directly.

One solution is simply to assume the scale as part of our model. However, this assumption does not take into account the fact that the scale varies according to the environment of the sensor. Most GPS sensors, however, also give an estimated confidence interval for the GPS error. This confidence interval ε_t is the value in metres such that there is a 95% probability that $Actual_t$ is within ε_t metres of the GPS sample. Formally, this says that

$$\Pr[\|\mathbf{GPS}_t - Actual_t\| < \varepsilon_t] = 0.95. \tag{4}$$

But observe that

$$\begin{aligned}
&\|\mathbf{GPS}_t - Actual_t\| \\
&= \|(TrueLat_t + \mathbf{LatErr}_t, TrueLong_t + \mathbf{LongErr}_t) \\
&\quad - (TrueLat_t, TrueLong_t)\| \\
&= \|(\mathbf{LatErr}_t, \mathbf{LongErr}_t)\|.
\end{aligned}$$

Furthermore, since our model assumes that $\mathbf{LatErr}_t$ and $\mathbf{LongErr}_t$ are i.i.d. with mean zero, it is a well-known identity that

$$\begin{aligned}
\|(\mathbf{LatErr}_t, \mathbf{LongErr}_t)\| &= \sqrt{\mathbf{LatErr}_t^2 + \mathbf{LongErr}_t^2} \\
&\sim \text{Rayleigh}(\rho_t).
\end{aligned}$$

for some unknown parameter ρ_t. The Rayleigh distribution is a continuous single-parameter non-negative probability distribution with density function

$$\text{Rayleigh}(x;\rho) = \frac{x}{\rho^2} e^{-\frac{x^2}{2\rho^2}}, \quad x \geq 0.$$

But (4) allows us to calculate ρ_t since $\|\mathbf{GPS}_t - Actual_t\| = \|(\mathbf{LatErr}_t, \mathbf{LongErr}_t)\|$ and we know the distribution of the right hand side. In particular, this means that

$$\begin{aligned}
0.95 &= \int_0^{\varepsilon_t} \frac{x}{\rho_t^2} e^{-\frac{x^2}{2\rho_t^2}} \, \mathrm{d}x \\
&= 1 - e^{-\frac{\varepsilon_t^2}{2\rho_t^2}} \\
\therefore \rho_t &= \varepsilon_t / \sqrt{\ln[(1-0.95)^{-2}]} \\
&= \varepsilon_t / \sqrt{\ln 400}.
\end{aligned}$$

So now we know that

$$\|(\mathbf{LatErr}_t, \mathbf{LongErr}_t)\| \sim \text{Rayleigh}(\varepsilon_t / \sqrt{\ln 400})$$

where ε_t is known from the GPS sensor, which trivially gives us a way to approximate (3). We can say that the likelihood that

$$(\mathbf{LatErr}_t, \mathbf{LongErr}_t) = Sample_t - p$$

is approximated by the likelihood that

$$\|(\mathbf{LatErr}_t, \mathbf{LongErr}_t)\| = \|Sample_t - p\|.$$

Since we know how to evaluate this likelihood, we have found way to evaluate

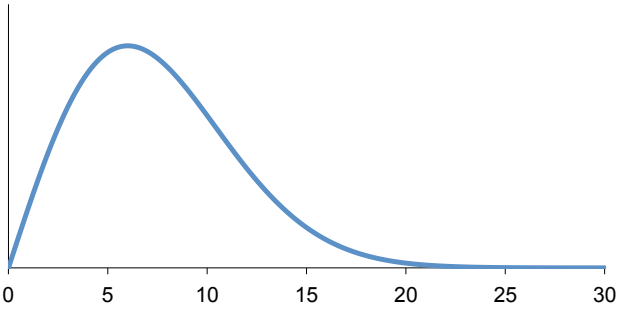
$$\Pr[\mathbf{GPS}_t = Sample_t | \mathbf{Location}_t = p]$$

which is the likelihood model needed to evaluate the posterior function in (2).

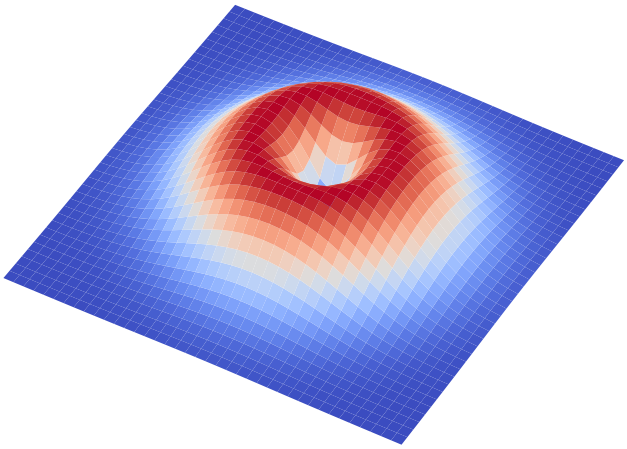
Summary We derived a posterior distribution for the user's location based on GPS evidence. The GPS sensor returns to us a location $Sample_t$ and a confidence interval ε_t. Then for any point p in the world, the likelihood of the user being at location p given this GPS sample is

$$\begin{aligned}
&\Pr[\mathbf{Location}_t = p | \mathbf{GPS}_t = Sample_t] \\
&\quad = \Pr[\mathbf{GPS}_t = Sample_t | \mathbf{Location}_t = p] \\
&\quad = \text{Rayleigh}(\|Sample_t - p\|; \varepsilon_t / \sqrt{\ln 400})
\end{aligned}$$

up to a normalising constant. We use the posterior to model estimation in our GPS version of *Uncertain*$\langle T \rangle$, described in Section 9.1. Figure 17 shows the Rayleigh distribution and the resulting posterior distribution. It is interesting to note that the distribution carries little mass in the *center* of the distribution, implying a user is *less* likly to be at the center of a GPS sample.



(a) The Rayleigh distribution Rayleigh($\rho = 6$).



(b) The posterior distribution for location is a distribution over the Earth's surface.

Figure 17: The derivation of a posterior distribution for location uses the Rayleigh distribution as a likelihood model for the GPS sensor. Notice that the centre of the posterior (which is the point $Sample_t$) has very low likelihood; a surprising consequence of the assumed model for GPS error.

Appendix B Implemented semantics

This appendix defines *Uncertain*⟨*T*⟩'s semantics in C# (following the description from Section 4) and demonstrates how the type system and operator overloading makes computing and inference with distributions opaque to developers.

```
// The uncertain type represents arbitrary distributions. It implements
// IEnumerable, because it encapsulates a list of Ts (a list of random samples)
public class Uncertain<T> : IEnumerable<T> {

    // Arithmetic operators lift the operation on type T to operate on
    // distributions over Ts
    public static Uncertain<T> operator +(Uncertain<T> lhs, Uncertain<T> rhs);

    // Comparing two distributions results in a Bernoulli distribution, which
    // represents the probability that A < B
    public static Bernoulli operator <(Uncertain<T> lhs, Uncertain<T> rhs);
    public static Bernoulli operator >(Uncertain<T> lhs, Uncertain<T> rhs);

    // The expected value is of type T*, but for implementation in languages
    // without type classes, we assume T has a multiplicative inverse (e.g. float)
    public SamplingDistribution<T> ExpectedValue();

    // Values of type T can be implicitly treated as point mass distributions
    public static implicit operator Uncertain<T>(T t);
}

// Sampling distributions represent the sampling error created by approximating
// a distribution
public class SamplingDistribution<T> : Uncertain<T> {

    // Compare two sampling distributions by performing a hypothesis test
    public static HypothesisTest operator <(SamplingDistribution<T> lhs, SamplingDistribution<T> rhs);
    public static HypothesisTest operator >(SamplingDistribution<T> lhs, SamplingDistribution<T> rhs);

    // Values of type T can be implicitly treated as point masses for use in
    // hypothesis tests
    public static HypothesisTest operator <(SamplingDistribution<T> lhs, T rhs);
    public static HypothesisTest operator >(SamplingDistribution<T> lhs, T rhs);

    // Project a sampling distribution down to an estimate of the statistic
    public T Project();
}

// A hypothesis test compares two sampling distributions at a given confidence
// threshold. This class is never instantiated by clients, only by the uncertain
// type.
private class HypothesisTest {
    // Perform the encapsulated hypothesis test at the given confidence
    public bool HypTest(double confidence);

    // Implicitly perform the hypothesis test at 95% confidence -- the statistical default
    public static implicit operator bool(HypothesisTest t);
}

// Bernoulli distributions are random variables with two possible values, true
// (with probability p) or false (probability 1-p). An instance of Bernoulli is
// bound to a specific comparison.
public class Bernoulli : Uncertain<bool> {
    // Sample the underlying comparison this Bernoulli is bound to, to produce a
    // distribution over Bernoullis.
    public SamplingDistribution<Bernoulli> Prob();

    // Floats in the range [0,1] can be implcitly treated as Bernoulli distributions
    public static implicit operator Bernoulli(double t);

    // Get the parameter p of this distribution
    public double P();
}
```

```
// A demonstration of how to use this implementation
class Program {
    static void Main(string[] args) {
        var X = new Uncertain<double>(); // Some arbitrary distribution
        var Y = new Uncertain<double>(); // Some arbitrary distribution

        // Perform some computation
        var Z = X + Y;

        // Compare two expected values using a hypothesis test
        if (Z.ExpectedValue() < X.ExpectedValue()) {
            Console.WriteLine("E[Z] < E[X]");
        }

        // Compare an expected value to a pointmass using a hypothesis test
        if (Z.ExpectedValue() < 10) {
            Console.WriteLine("E[Z] < 10");
        }

        // Perform an evidence threshold test, asking if there is an 85% chance
        // that Z < X. This also performs a hypothesis test.
        if ((Z < X).Prob() > 0.85) {
            // This is saying we are confident that the parameter of the Bernoulli
            // distribution Z < X is at least 0.85.
            Console.WriteLine("Pr[Z < X] > 0.85");
        }

        // Perform an evidence threshold test with a pointmass, asking if there
        // is an 85% chance that Z < 60.
        if ((Z < 60).Prob() > 0.85) {
            Console.WriteLine("Pr[Z < 60] > 0.85");
        }

        // Perform an evidence threshold test, but using a different confidence
        // level for the hypothesis test
        if (((Z < 60).Prob() > 0.85).HypTest(0.99)) {
            Console.WriteLine("Pr[Z < 60] > 0.85, at the 99% confidence level");
        }

        // Compare an expected value without doing a hypothesis test, to avoid
        // ternary logic.
        double ZMean = Z.ExpectedValue().Project();
        if (ZMean < 60) {
            Console.WriteLine("E[Z} < 60");
        } else if (ZMean >= 60) {
            Console.WriteLine("E[Z] >= 60");
        } else {
            assert(false);  // Unreachable because Project() ensures a total order
        }

        // These tests do not compile because they are uncertainty bugs, asking
        // the wrong questions of estimated data

        // if (Z < X) { }
        // if (Z < 10) { }
    }
}
```

Appendix C Instantiating the uncertain type for GPS

This appendix shows how to instantiate an *Uncertain⟨GeoCoordinate⟩* in code. It uses the theoretical model derived in Appendix A and the implementation of *Uncertain⟨T⟩* presented in Appendix B. This implementation would be written by a library programmer, and client programs would simply the `GetGPSLocation` method and receive a GPS distribution.

```
public class GPSLib {
    // How many points to use in approximating the distribution
    private int SAMPLE_SIZE = 10000;
    // We need to convert between metres and degrees when using the error estimate
    private double EARTH_RADIUS = 6371*1000;
    private double DEGREES_PER_METRE = Math.Degrees(1/EARTH_RADIUS);

    public Uncertain<GeoCoordinate> GetGPSLocation() {
        // Firstly, get the estimate from the hardware
        GeoCoordinate Point = GetSampleFromGPSHardware();
        double ErrorRadius  = GetErrorEstimateFromGPSHardware();

        // Compute the parameter rho of the Rayleigh distribution
        double rho = ErrorRadius / Math.Sqrt(Math.Log(400));

        // We generate samples from the surface (Figure 18(b)) in polar
        // coordinates: the radius is a Rayleigh sample, and the angle a uniform
        // random angle

        // Generate samples from the Rayleigh distribution
        double[] radii = RandomRayleigh(rho, SAMPLE_SIZE);
        // Generate random angles to rotate by from a uniform distribution
        double[] thetas = RandomUniform(0, 2*Math.PI, SAMPLE_SIZE);

        // Convert the polar coordinates to x,y coordinates in degrees. Assumes
        // +, *, Sin and Cos all do elementwise operations on vectors.
        double[] x = Point.Longitude + radii*Math.Cos(thetas)*DEGREES_PER_METRE;
        double[] y = Point.Latitude  + radii*Math.Sin(thetas)*DEGREES_PER_METRE;

        // Transpose the list to be 10,000 pairs (x_i, y_i)
        GeoCoordinate[] coords = Zip(x, y);

        return new Uncertain<GeoCoordinate>(coords);
    }

    // These methods expose the raw estimate from the hardware

    // Return the sampled point
    private GeoCoordinate GetSampleFromGPSHardware();
    // Return the estimated 95% confidence interval
    private double GetErrorEstimateFromGPSHardware();

    // These methods return random samples from distributions. Most statistics
    // libraries would provide implementations of RandomRayleigh, and all
    // math libraries provide implementations of RandomUniform.

    // Generate Rayleigh random samples
    private double[] RandomRayleigh(double rho, int size) {
        // If X ~ N(0, s^2) and Y ~ N(0, s^2) then R = sqrt(X^2 + Y^2) ~ Rayleigh(s)
        double[] x = RandomNormal(0, rho, size);
        double[] y = RandomNormal(0, rho, size);

        // x and y are vectors, we assume *, + and Sqrt all do elementwise operations
        return Math.Sqrt(x*x + y*y);
    }
    private double[] RandomNormal(double mean, double stdev, int size);
    private double[] RandomUniform(double low, double high, int size);
}
```