Uncertain⟨T⟩

Written by Mattt — July 25th, 2025

You know what's wrong with people? They're too sure of themselves.

Better to be wrong and own it than be right with caveats. Hard to build a personal brand out of nuance these days. People are attracted to confidence — however misplaced.

But can you blame them? (People, that is) Working in software, the most annoying part of reaching Senior level is having to say *"it depends"* all the time. Much more fun getting to say *"let's ship it and iterate"* as Staff or *"that won't scale"* as a Principal.

Yet, for all of our intellectual humility, why do we ~~write~~ vibe code like this?

```swift
if currentLocation.distance(to: target) < 100 {
    print("You've arrived!") // But have you, really? 🤨
}
```

GPS coordinates aren't exact. They're noisy. They're approximate. They're probabilistic. That `horizontalAccuracy` property tucked away in your `CLLocation` object is trying to tell you something important: you're *probably* within that radius. *Probably.*

A `Bool`, meanwhile, can be only `true` or `false`. That `if` statement needs to make a choice one way or another, but code like this doesn't capture the uncertainty of the situation. If truth is light, then current programming models collapse the wavefunction too early.

Picking the Right Abstraction

In 2014, researchers at the University of Washington and Microsoft Research proposed a radical idea: What if uncertainty were encoded directly into the type system? Their paper, *Uncertain<T>: A First-Order Type for Uncertain Data* () introduced a probabilistic programming approach that's both mathematically rigorous and surprisingly practical.

(I found a copy of this paper while cleaning out my `~/Downloads` folder over the weekend. I remember seeing it right around when Swift was announced, thinking it would be perfect for testing the new language's generics. But I never got around to it, until now 🥲)

As you'd expect for something from Microsoft in the 2010s, the paper is implemented in C#. But the concepts translate beautifully to Swift.

You can find my port on GitHub () :

```
import Uncertain
import CoreLocation

let uncertainLocation = Uncertain<CLLocation>.from(currentLocation)
let nearbyEvidence = uncertainLocation.distance(to: target) < 100
if nearbyEvidence.probability(exceeds: 0.95) {
    print("You've arrived!") // With 2σ confidence 🤓
}
```

When you compare two `Uncertain` values, you don’t get a definitive `true` or `false`. You get an `Uncertain<Bool>` that represents the *probability* of the comparison being `true`.

> Under the hood, `Uncertain<T>` models GPS uncertainty using a Rayleigh distribution (). GPS errors are typically circular around the true position, with error magnitude following this distribution.

The same is true for other operators, too:

```
// How fast did we run around the track?
let distance: Double = 400 // meters
let time: Uncertain<Double> = .normal(mean: 60, standard
Deviation: 5.0) // seconds
let runningSpeed = distance / time // Uncertain<Double>

// How much air resistance?
let airDensity: Uncertain<Double> = .normal(mean: 1.225, standard
Deviation: 0.1) // kg/m³
let drag
Coefficient: Uncertain<Double> = .kumaraswamy(alpha: 9, beta: 3) // slightly rig
let frontalArea: Uncertain<Double> = .normal(mean: 0.45, standard
Deviation: 0.05) // m²
let airResistance = 0.5 * airDensity * frontalArea * drag
Coefficient * (runningSpeed * runningSpeed)
```

This code builds a computation graph, sampling only when you ask for concrete results. The library uses Sequential Probability Ratio Testing (SPRT) () to efficiently determine how many samples are needed — maybe a few dozen times for simple comparisons, scaling up automatically for complex calculations.

```
// Sampling happens only when we need to evaluate
if ~(runningSpeed > 6.0) {
    print("Great pace for a 400m sprint!")
}
```

```
// SPRT might only need a dozen samples for this simple comparison

let sustainableFor5K = (runningSpeed < 6.0) && (airResistance < 50.0)
print("Can sustain for 5K: \(sustainableFor5K.probability(exceeds: 0.9))")
// Might use 100+ samples for this compound condition
```

Using an abstraction like `Uncertain<T>` forces you to deal with uncertainty as a first-class concept rather than pretending it doesn't exist. And in doing so, you end up with much smarter code.

To quote Alan Kay ():

> Point of view is worth 80 IQ points

Before we dive deeper into probability distributions, let's take a detour to Monaco and talk about Monte Carlo sampling ().

The Monte Carlo Method

Behold, a classic slot machine (or "fruit machine" for our UK readers 🇬🇧):

```
enum SlotMachine {
    static func spin() -> Int {
        let symbols = [
            "⬜", "⬜", "⬜",  // blanks
            "🍒", "🍋", "🍊", "🍇", "💎"
        ]

        // Spin three reels independently
        let reel1 = symbols.randomElement()!
        let reel2 = symbols.randomElement()!
```

```swift
        let reel3 = symbols.randomElement()!

        switch (reel1, reel2, reel3) {
        case ("💎", "💎", "💎"): return 100  // Jackpot!
        case ("🍒", "🍒", "🍒"): return 10
        case ("🍇", "🍇", "🍇"): return 5
        case ("🍊", "🍊", "🍊"): return 3
        case ("🍋", "🍋", "🍋"): return 2
        case ("🍒", _, _), // Any cherry
             (_, "🍒", _),
             (_, _, "🍒"):
            return 1
        default:
            return 0  // Better luck next time
        }
    }
}
```

Should we play it?

(Are you feeling lucky?)

Now, we *could* work out these probabilities analytically — counting combinations, calculating conditional probabilities, maybe even busting out some combinatorics.

Or we could just let the computer pull the lever a bunch and see what happens.

(Are you feeling... lazy?)

```swift
let expectedPayout = Uncertain<Int> {
    SlotMachine.spin()
}.expectedValue(sampleCount: 10_000)
```

```
print("Expected value per spin: $\(expectedPayout)")
// Expected value per spin: ≈ $0.56
```

At least we know one thing for certain: *The house always wins.*

Beyond Simple Distributions

While one-armed bandits demonstrate pure randomness, real-world applications often deal with more predictable uncertainty.

`Uncertain<T>` provides a rich set of probability distributions ( ) :

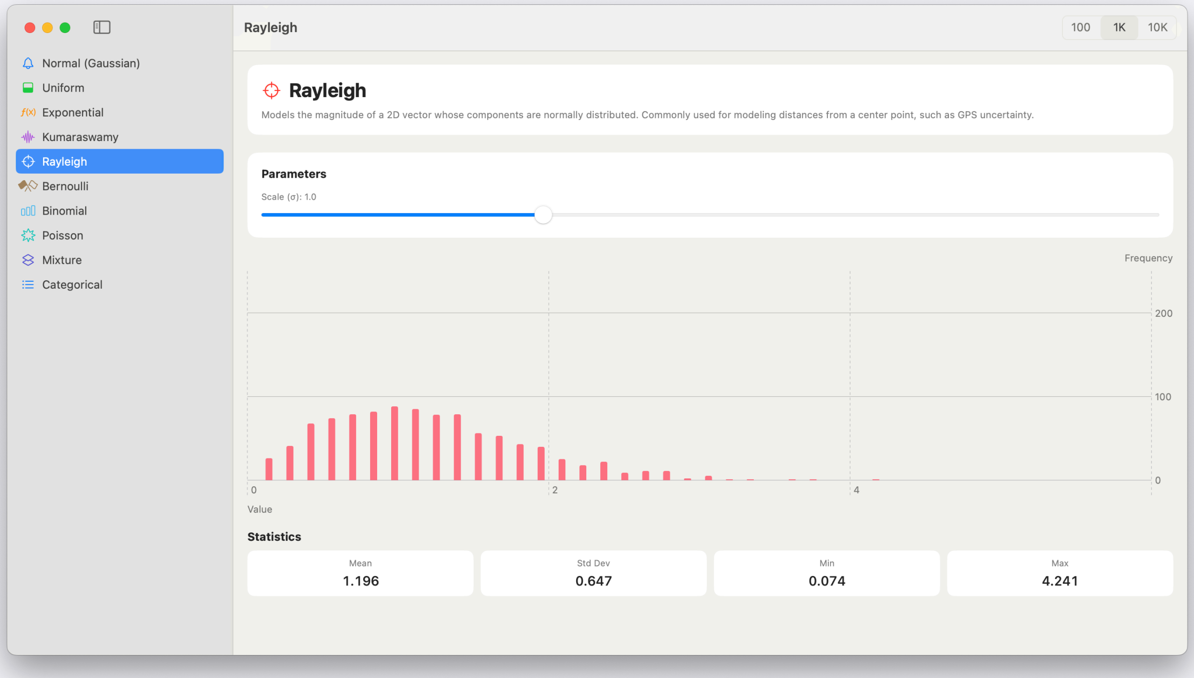
```
// Modeling sensor noise
let rawGyroData = 0.85  // rad/s
let gyroReading = Uncertain.normal(
    mean: rawGyroData,
    standardDeviation: 0.05  // Typical gyroscope noise in rad/s
)

// User behavior modeling
let userWillTapButton = Uncertain.bernoulli(probability: 0.3)

// Network latency with long tail
let apiResponseTime = Uncertain.exponential(rate: 0.1)

// Coffee shop visit times (bimodal: morning rush + afternoon break)
let morningRush = Uncertain.normal(mean: 8.5, standard
Deviation: 0.5)  // 8:30 AM
let afternoonBreak = Uncertain.normal(mean: 15.0, standard
Deviation: 0.8)  // 3:00 PM
let visitTime = Uncertain.mixture(
    of: [morningRush, afternoonBreak],
    weights: [0.6, 0.4]  // Slightly prefer morning coffee
)
```

I wanted to develop an intuitive sense of how these probability distributions work, so I built this companion project () with interactive visualizations for each one. It also serves as a nifty showcase for Swift Charts (). So definitely check that out if you're uninitiated.



Rayleigh

Models the magnitude of a 2D vector whose components are normally distributed. Commonly used for modeling distances from a center point, such as GPS uncertainty.

Scale (σ): 1.0

Mean	Std Dev	Min	Max
1.196	0.647	0.074	4.241

Uncertain<T> also provides comprehensive statistical operations ():

```
// Basic statistics
let temperature = Uncertain.normal(mean: 23.0, standardDeviation: 1.0)
let avgTemp = temperature.expectedValue() // about 23°C
let tempSpread = temperature.standardDeviation() // about 1°C

// Confidence intervals
let (lower, upper) = temperature.confidenceInterval(0.95)
print("95% of temperatures between \(lower)°C and \(upper)°C")

// Distribution shape analysis
let networkDelay = Uncertain.exponential(rate: 0.1)
```

```swift
let skew = networkDelay.skewness() // right skew
let kurt = networkDelay.kurtosis() // heavy tail

// Working with discrete distributions
let diceRoll = Uncertain.categorical([1: 1, 2: 1, 3: 1, 4: 1, 5: 1, 6: 1])!
diceRoll.entropy()  // Randomness measure (~2.57)
(diceRoll + diceRoll).mode() // Most frequent outcome (7, perhaps?)

// Cumulative probability
if temperature.cdf(at: 25.0) < 0.2 {  // P(temp ≤ 25°C) < 20%
    print("Unlikely to be 25°C or cooler")
}
```

The statistics are computed through sampling. The number of samples is configurable, letting you trade computation time for accuracy.

Putting Theory to Practice

Users don't notice when things work correctly, but they definitely notice impossible behavior. When your running app claims they just sprinted at 45 mph, or your IRL meetup app shows someone 500 feet away when GPS accuracy is ±1000 meters, that's a bad look 🤡

So where do we go from here? Let's channel our Senior+ memes from before for guidance.

That Staff engineer saying *"let's ship it and iterate"* is right about the incremental approach. You can migrate uncertain calculations piecemeal rather than rewriting everything at once:

```swift
extension CLLocation {
    var uncertain: Uncertain<CLLocation> {
        Uncertain<CLLocation>.from(self)
    }
}

// Gradually migrate critical paths
let isNearby = (
    currentLocation.uncertain.distance(to: destination) < threshold
).probability(exceeds: 0.68)
```

And we should consider the Principal engineer’s warning of “*that won’t scale*”. Sampling has a cost, and you should understand the computational overhead for probabilistic accuracy:

```
// Fast approximation for UI updates
let quickEstimate = speed.probability(
    exceeds: walkingSpeed,
    maxSamples: 100
)

// High precision for critical decisions
let preciseResult = speed.probability(
    exceeds: walkingSpeed,
    confidenceLevel: 0.99,
    maxSamples: 10_000
)
```

> OTOH (ON THE OTHER HAND 💁‍♀️), modern devices are pretty amazing.
>
> Remember kids, `Instruments.app` is your friend. Use profiling to guide your optimizations.
>
> Senior 🤝 Staff 🤝 Principal

Start small. Pick one feature where GPS glitches cause user complaints. Replace your distance calculations with uncertain versions. Measure the impact.

Remember: the goal isn't to eliminate uncertainty — it's to acknowledge that it exists and handle it gracefully. Because in the real world, nothing is certain except uncertainty itself.

And perhaps, with better tools, we can finally stop pretending otherwise.