Institut für Verteilte Systeme
Institute of Distributed Systems

universität ulm

Efficient and Accurate Encodings for Subjective Logic Opinions

Master Thesis

2025-10-29

Sophie Hirn
sophie.hirn@uni-ulm.de

Institute of Distributed Systems, Ulm University

Supervised by
Prof. Dr. rer.nat. Frank Kargl
Dennis Eisermann, M. Sc.

Examined by
Prof. Dr. rer.nat. Frank Kargl
Prof. Dr. Matthias Tichy

ABSTRACT

This work examines the effects of machine arithmetic rounding errors on subjective logic trust networks, and the potential for data compression when transmitting opinions wirelessly. For this, specific scenarios are defined, and then simulated using interval arithmetic to find an upper bound for the accumulated error.

The simulations show that the semantics of IEEE 754 binary64 floating point numbers are well suited for most calculations, but some implementation safeguards are necessary to ensure well-defined results.

Encoding opinions as triplets of fixed point numbers of the form UQ0.k is theoretically optimal for systems with uniform statistical distribution of transmitted opinion values. The examined formats achieved compression ratios of 4.0 to 6.0 over binary64 triplets with significant growth of error intervals, but without compromising system integrity in the simulated scenarios.

The latest version of this document is available at:
https://artifacts.sowophie.io/master/thesis.pdf

The sources for this document and the simulations are available at:
https://cgit.sowophie.io/master/

I would like to thank:

My mother Brigitte, who was co-working with me for most of this thesis,

Frank Kargl, Dennis Eisermann, and Nataša Trkulja, who offered
excellent mentoring, sometimes from across the globe,

Peggy Leupold, the best bureaucrat one could wish for.

Declaration of Autonomy

I hereby declare that this thesis titled:

Efficient and Accurate Encodings for Subjective Logic Opinions

is the product of my own independent work and that I have used no sources or materials other than those specified. The passages taken from other works, either verbatim or paraphrased in the spirit of the original quote, are identified in each individual case by indicating the source. I further declare that all my academic work was written in line with the principles of proper academic research according to the official "Satzung der Universität Ulm zur Sicherung guter wissenschaftlicher Praxis" (University Statute for the Safeguarding of Proper Academic Practice).

All work for this thesis was done without the assistance of large language model (LLM) technology.

Neu-Ulm, 2025-10-29, Sophie Hirn

Table of Contents

1 Introduction

The vision of the "smart" car, that drives itself and is connected to its environment, has inspired a lot of research activity in the past decade. In scientific literature, this is often referred to as **Connected, Cooperative and Autonomous Mobility (CCAM)**. This broad term entails many different technologies for different scenarios, such as:

- **Cooperative Intersection Management (CIM)** [1], the managing of an intersection using data sent from nearby vehicles to improve throughput and safety,
- **Cooperative Adaptive Cruise Control (CACC)**, vehicles forming so-called platoons to efficiently manage their road speed together.

The underlying networking technology for this is called **Vehicular Ad-Hoc Networks (VANETs)**, i.e. networks that form and dissolve organically around vehicles. They naturally deal with data from mostly untrusted sources, on which nodes must base their decisions, which in turn can affect the physical wellbeing of humans in and around the vehicles significantly. While it generally can be assumed that most nodes inside a VANET are not malicious [2], the consequences of system failures include threat to human lives, and therefore an abundance of caution is indicated. Dangerous data does not have to originate from a malicious attacker, but could also be emitted by faulty components.

Some technologies to tackle this problem are grouped under the umbrella term **Misbehavior Detection (MBD)**. Many systems are based on establishing and maintaining representations of *trust* in a node and/or its sensors. While susceptible to sudden, unannounced attacks, they can reliably deal with clients occasionally or consistently sending wrong data [3].

Subjective logic [4] is a framework for reasoning under uncertainty. It introduces the concept of **trust networks** to model potentially unreliable observations and observers. This approach has gained popularity for modeling trust in CCAM scenarios [1] [3] [5] [6]. The basis of subjective logic calculations is the **trust opinion**, an extension of a traditional probability distribution.

Trust opinions can stem from a variety of sources, e.g. misbehavior detection sensors. They are then processed using operators for transitive chains, aggregation of independent information from different sources, maintaining long term reputations, etc. Finally, an opinion can be collapsed to a single probability, which can then be the basis to decide whether to trust a given information.

1.1 Problem Statement

Probabilities, as used in subjective logic opinions, are real numbers by nature. Representing them on a computer is a trade-off between precision and computing resource requirements, such as the amount of memory needed for storage and the processing time required for computations.

One extreme would be boolean values. They only require a single bit to store, and a single CPU instruction can process multiple values at once. However, they are unsuitable for representing nuances of trust beyond "none" and "ultimate".

Another extreme would be symbolic computation as used in computer algebra systems. They can represent the result of any computation exactly. But the space required for a single number is unbounded, and even primitive operations require expensive algorithms like term rewriting.

Computers involved in CCAM naturally are embedded systems, making it especially important to be mindful of computing resource requirements. More efficient implementations can allow more kinds of devices to participate. They can also allow systems with a fixed amount of computing resources to process information from more participants, or more information on the same amount of participants.

The same consideration also applies to the transfer of opinions between nodes, which needs to be wireless in many cases. With wireless communication, each transmission interferes with all other exchange of information in the area for its duration. This effectively places an upper limit on the amount of data that can be exchanged in a local network. Reaching this limit could lead to service interruptions, or even an increased risk of accidents. The more an opinion can be compressed for transmission, the shorter each transmission can be, the higher the maximum throughput of the network.

However, scaling down precision too much can make trust models lose meaningfulness, and sometimes even diverge violently. Considering the potential gravity of mistakes, implementation decisions should be based on a solid theoretical foundation. Of particular interest are worst-case lower bounds on the possible error value. By basing their choices on the results of such an approach, implementations can be reasonably certain that their arithmetic error will be sufficiently small.

1.2 Research Questions

This thesis seeks to examine and answer the following research questions:

RQ1: How can the accuracy of a subjective logic network be modeled so that justified and meaningful recommendations can be derived?

RQ2: Given the currently mostly theoretical nature of subjective logic networks, how can the predictions of such a model be validated?

RQ3: Which concrete recommendations regarding the processing and transmission of opinions can be derived from this model?

1.3 Approach

For accurate estimation of error propagation, **interval arithmetic** is a proven mathematical tool [7]. Where calculations would normally be performed using a single value, interval arithmetic instead uses a lower and upper bound. Calculations in interval arithmetic map input intervals to a result interval. The guarantee is that if input values inside the interval are given to the real calculation, the result will be inside the output interval. By chaining interval arithmetic operations, precision can be accurately traced across complex calculations. The width of the resulting interval is a single number that bounds the expected error.

While arbitrary functions can have complex result intervals that are not necessarily contiguous or finite, the primitive arithmetic operators generally map closed finite input intervals to closed finite output intervals with simple bounds.[1] As the examined subjective logic operators are composed from the latter, they are well suited for examination using interval arithmetic.

This thesis uses computer simulation of different scenarios based on interval arithmetic. For this, it is important to isolate the error of the examined calculations from any error introduced by the finite precision of the simulation environment. Representing the operands as rational numbers, i.e. fractions of arbitrary-precision integers, is chosen to fulfill this requirement.

1.4 Results

For the simulated test cases, the following can be concluded:

1. With the precautions outlined below, performing calculations using the IEEE 754 binary64 format with the default "*round to nearest, ties to even*" rounding mode is **generally safe** for trust networks of all relevant sizes. The use of the IEEE 754 binary32 format however is strongly discouraged, as some concerning instances of paradoxical behavior have been observed. Most notably, forming consensus from generally favorable opinions resulted in complete distrust after the number of opinions crossed a threshold.
2. While subjective logic is correct under real numbers, some invariants do not hold when calculations are performed using finite precision. **Normalization** of operands at specific points into the

1. An exception to this is the division operator, given a divisor interval containing zero.

expected interval is therefore **vital** to prevent issues such as divisions by zero and out-of-bounds results. For the same reason, **disbelief should not be tracked** alongside the other values, but instead calculated from belief and uncertainty as needed.

3. Cumulative fusion of opinions with high uncertainty values resulted in an unexpectedly quick growth of error intervals. It is unclear whether this has an impact in the real world, or is just a property of the model implementation of this thesis. Pending further investigation, caution with this type of operation is advised.
4. By transmitting opinions as triplets of fixed point numbers, substantial bandwidth savings are possible. The examined formats are triplets of $(16, 16, 16)$, and $(11, 11, 10)$ bit numbers respectively, with a compression ratio of 4.0 and 6.0 respectively over uncompressed transmission. In tests spanning up to 100 nodes, the former had error intervals grow to ca. 6% in the worst case, the latter reached 36%. While this is a significant increase, the tested systems did not become unstable as a result of this.

It has to be stressed that these recommendations are derived from a number of scenarios that have been deemed reasonable and thoroughly tested. While they can be expected to generalize across many similar scenarios, they lack formal verification. The reader is encouraged to use the framework developed for this thesis to simulate their own scenarios.

1.5 Thesis Overview

This thesis is organized as follows. Chapter 1 has introduced the context of the topic, and given an overview over the problem, the approach, and the findings. Chapter 2 covers the theoretical background for this thesis. Chapter 3 shows that fixed point numbers are a substantially better choice of transport encoding than floating point numbers, and that their precision is theoretically optimal for opinions with uniformly distributed values. Chapter 4 explains the approach and implementation of the central thesis model. Chapter 5 describes the test cases that are built on this framework. Chapter 6 highlights important simulation results. Chapter 7 discusses the results and derives concrete implementation recommendations. Chapter 8 summarizes the results and limitations. Chapter 9 outlines possible future work. Finally, the appendix contains detailed results for all simulated test cases.

2 Background

The concepts in this chapter are the foundation that this thesis is built on. Chapter 2.1 introduces fixed point numbers, which are the basis for the transport encoding this thesis recommends. Chapter 2.2 explains the theory behind floating point numbers, the representation used for non-integer computations on almost all modern hardware. Chapter 2.3 covers rounding modes, which are an essential component of non-integer calculations. Chapter 2.4 introduces the concept of the *ulp* ("unit in the last place") as a measure for machine precision. Chapter 2.5 covers IEEE 754, the predominant standard for floating point implementations. Chapter 2.6 explains interval arithmetic, which is used in this thesis to measure the worst-case error of calculations. Finally, chapter 2.7 introduces subjective logic, the framework of which the accuracy is examined in this thesis.

2.1 Fixed Point Numbers

A simple solution for encoding non-integer numbers are so-called **fixed point** numbers. They consist of a single value n which is interpreted as

$$f = n \cdot 2^{-e}$$

for a pre-determined exponent $e \geq 0$, with the trivial case $e = 0$ representing traditional integers. Focusing more on the bitwise representation, if a fixed point number has k bits and exponent e, $k - e$ bits encode the integer part of the number, and e bits encode the fractional part. Signed fixed point numbers are most commonly represented using two's complement notation, so the first bit of the integer part becomes the sign bit. A common way to describe this characteristic of a fixed point number format is the **Q notation**.

Using this notation, a signed number type is written as **Q**$x.y$, an unsigned type as **UQ**$x.y$. The numbers x and y denote the amount of integer and fractional bits respectively. There is no consensus on whether the sign bit should be included in the count for x or not. The integer part and dot can be left out if the value is clear from the context. For example, an unsigned 16 bit number with exponent 4 would be written as **UQ12.4** or simply **UQ4**.

While explicit hardware support is uncommon, fixed point arithmetic can be efficiently implemented using integer arithmetic. Addition and subtraction do not need to take the exponent into account at all, as

$$a \cdot 2^{-e} \pm b \cdot 2^{-e} = (a \pm b) \cdot 2^{-e}$$

Multiplication and division can be implemented using an additional bit shift, as

$$a \cdot 2^{-e} \times b \cdot 2^{-e} = (a \times b \times 2^{-e}) \cdot 2^{-e}$$

and

$$\frac{a \cdot 2^{-e}}{b \cdot 2^{-e}} = \frac{a}{b} = \left(\frac{a}{b} \times 2^{e}\right) \cdot 2^{-e}$$

The latter two should be calculated in a wider integer type, as the significant result bits are formed outside the range of the result type, and then shifted into place. This may make the maximum width integer type unsuitable for representing fixed point numbers.

2.2 Floating Point Numbers

The floating point number representation is how most computers represent numbers with both very small and very large magnitude in a single type. It is based on the scientific number notation, as demonstrated here using the gravitational constant of our universe:

$$+6.6743015 \cdot 10^{-11} \frac{Nm^2}{kg^{-2}}$$

The components are, from left to right:

- The **sign**, which can be either positive or negative, for zero both are valid.
- The **mantissa**, a sequence of digits with a decimal point. It defines the digits of the number.
- The **base**, which is the same as the numeric base the mantissa was written in.
- The **exponent**, a superscripted integer that shifts the comma of the mantissa.
- A **unit**, which is not part of the number.

Floating point numbers are stored as a packed triplet of values, (s, e, m). s encodes the sign of the result, and is either 0 or 1. e is the exponent of a fixed base b (= 2 for binary numbers), which is not stored. m is the mantissa. The encoded value f is defined as

$$f \;=\; (-1)^s \cdot m \cdot b^e$$

The numbers are usually *normalized*, i.e. mantissa and exponent are chosen such that $1 \leq m < b$, and the decimal point is positioned between the first and the second digit.

Base 2 is most natural for binary machines. But it is only partially compatible with base 10, the base predominantly used by humans. 10 has the factorization $2 \cdot 5$, but 2 does not have 5 as a factor, so while multiples of 5^{-1} have a finite decimal representation, their exact binary representation has infinite digits. For example, $0.1_{10} = 2^{-1} \cdot 5^{-1} = 0.0\overline{0011}_2$. As a result, conversions from decimal numbers to binary floating point numbers are not always exact. On the other hand, conversion from binary to decimal are always exact, so if numbers originate from a base 2 computer system, conversion loss is not a problem.

An alternative are base 10 floating point numbers using binary-coded decimal notation. Each decimal digit is represented by four bits, e.g. $0.15_{10} = (0001_2) \cdot 10^{-1} + (0101_2) \cdot 10^{-2}$. This allows for exact conversions, but calculations are generally slower and more complicated to implement. Base 10 floating point numbers are used and supported a lot less commonly than base 2, and will not be used for this thesis.

Base 16 is sometimes used as an interchange format. Each base 16 digit corresponds exactly to four base 2 digits, so conversions are always exact in both directions. Hexadecimal floating point notation is particularly useful as a human-readable string representation of binary floating point numbers. A number of languages, such as C since C99 [8], have support for hexadecimal floating point literals of the form

$$\mathbf{0x}\, m\, \mathbf{p}\, e$$

where m is a mantissa in base 16, and e is the exponent in decimal notation. While base 16 is used for the mantissa, the literal still represents a binary number, and so the base for the exponent is 2, not 16. A normalized number always has a 1 as the first digit. For example, $0.2_{10} = 0.\overline{0011}_2$, normalized and evaluated to IEEE 754 binary64 precision (see "2.5.1 Representations", pg. 8) is

$$1\,.\,1001\;1001\;1001\;1001\;1001\;1001\;1001\;1001\;1001\;1001\;1001\;1001\;1001_2 \cdot 2^{-3}$$

, which can be expressed as `0x1.999999999999p-3`.

2.3 Rounding

When performing calculations with floating point numbers, the exact result will often not be representable in the same floating point format. A nearby number has to be chosen, which is then pronounced "close enough" to the exact result to be used for further operations.

Mapping a precise number to a nearby one that is more easily representable has been done long before humanity developed electronic computing. The process is commonly referred to as **rounding**. Across history, a few different schemes have been established. Most of them have a common structure:

1. If the number to be rounded is exactly representable, it can be used as-is.
2. Otherwise, it falls between two representable numbers. One of them is picked according to a rounding rule.

A simple rule for positive numbers (e.g. "always pick the smaller one") can usually be extended in two ways to also cover negative numbers. The rule can be applied to the magnitude of the value ("always pick the one closer to zero"), effectively mirroring the rounding pattern at zero. Alternatively, the rule can be organically continued past zero ("always pick the one closer to $-\infty$"). As a result of this, a lot of rounding modes differ only in the handling of negative numbers, and are redundant when examining positive numbers.

The absolute difference between input and rounded result is called **rounding error**. When a calculation involves a large number of rounding operations, a phenomenon called **rounding bias** can occur. If the rounding errors mainly occur in one direction, they can quickly add up to a sizable difference to the exact result. The effect depends on the nature of both the rounding mode and the set of rounded numbers. For example, "round to next integer toward 0" would introduce an error of 0.9 to the average of $round(0.9)$ and $round(1.9)$, but an error of 0 to the average of $round(-0.9)$ and $round(0.9)$.

A few other properties of rounding operations are also worth highlighting. The rounding schemes mentioned in this thesis are **pure** and **deterministic**, i.e. rounding the same input number will always have the same result. Rounding is an **idempotent** operation, i.e. rounding a value once is equivalent to rounding it any number of times using the same scheme. And finally, rounding is a **weak order preserving** operation, i.e. if $a \leq b$, the same will also hold for the rounded values. Note that rounding does not preserve strict order, as two different values might be rounded to the same result. The **maximum rounding error** of any rounding scheme cannot be less than half of the distance between two adjacent rounding results.

2.4 Machine Precision

Computers represent numbers in discrete steps. The size of each such step is equal to the value of the least significant bit. This distance is referred to as **unit in the last place**, or **ulp** [9]. For integers, ulp $= 1$. For fixed point numbers of the form Qn and UQn, ulp $= 2^{-n}$. For both of them, the ulp depends on the type alone, and is equal for any two numbers of the same type.

For floating point numbers on the other hand, the ulp depends on the width of the mantissa and the value of the exponent, so different numbers of the same type can have different ulps. If the exponent is zero, the most significant mantissa bit has a value of $2^0 = 1$. Consequently, for a number f with $1 + k$ mantissa bits[2], the value of the least significant bit will be 2^{-k}. A non-zero exponent e introduces a bit shift, equivalent to a multiplication of 2^e, resulting in a final ulp$(f) = 2^{e-k}$.

2. This choice of k is particularly convenient for working with IEEE 754 binary floating point numbers, as explained in "2.5.1 Representations", pg. 8.

2.5 IEEE 754

Floating point numbers are standardized in **IEEE 754** [10], and ubiquitous on most modern hardware. Even the rare, usually embedded, platforms that do not have hardware floating point support usually provide software implementations.

IEEE 754 is also part of most modern programming languages. A quick survey of programming languages shows the universality of this standard. Of the top 20 most used programming, scripting and markup languages in the StackOverflow Developer Survey 2024[3] (JavaScript[4], SQL[5] [6], HTML/CSS[7], Python[8], TypeScript[9], Bash[10] / Shell[11], Java[12], C#[13], C++[14], PHP[15], C [8], Go[16], PowerShell[17], Rust[18], Kotlin[19], Dart[20], Ruby[21], Lua[22], Swift[23], Visual Basic[24]):

- All except two (Bash/Shell, HTML/CSS) have a primitive number type based on IEEE 754 binary formats.
- Five (SQL, HTML/CSS, C#, PowerShell, Visual Basic) have a primitive type that can represent non-integer real numbers that is **not** based on the IEEE 754 binary formats (excluding complex numbers). Of those languages, only HTML/CSS does not also offer the latter. The provided formats fall in the category of fixed and floating point decimal numbers.

The outliers both use string based representations:

- Bash and other POSIX shells are untyped. All values are strings, calculations are performed using strings that resemble decimal numbers. POSIX mandates integer arithmetic, implementations can also provide floating point arithmetic with C semantics. Bash and other common shells only implement the former. The traditional POSIX tool for non-integer calculations is `bc`, which processes strings using a decimal fixed point representation with configurable precision.
- HTML and CSS are more akin to exchange formats than programming languages. They can contain numbers and expressions, but the representation used for evaluation is an unstandardized implementation detail.

3. https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language-prof
4. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Data_structures
5. https://mariadb.com/docs/server/reference/data-types/numeric-data-types
6. https://www.postgresql.org/docs/current/datatype-numeric.html
7. https://www.w3.org/TR/css-values-4/#numeric-types
8. https://docs.python.org/3/library/stdtypes.html#typesnumeric
9. https://www.typescriptlang.org/docs/handbook/2/everyday-types.html
10. https://www.gnu.org/software/bash/manual/bash.html#Shell-Arithmetic
11. https://pubs.opengroup.org/onlinepubs/9799919799/utilities/V3_chap02.html#tag_19_06_04
12. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html
13. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/built-in-types
14. https://en.cppreference.com/w/cpp/language/types.html
15. https://www.php.net/manual/en/language.types.float.php
16. https://kuree.gitbooks.io/the-go-programming-language-report/content/3/text.html
17. https://learn.microsoft.com/en-us/powershell/scripting/lang-spec/chapter-04?view=powershell-7.5
18. https://doc.rust-lang.org/book/ch03-02-data-types.html
19. https://kotlinlang.org/docs/numbers.html#floating-point-types
20. https://dart.dev/language/built-in-types#numbers
21. https://docs.ruby-lang.org/en/3.4/ (In Ruby everything is an object, so the core classes are considered as primitive types)
22. https://www.lua.org/manual/5.4/manual.html#2.1
23. https://docs.swift.org/swift-book/documentation/the-swift-programming-language/thebasics/
24. https://learn.microsoft.com/en-us/dotnet/visual-basic/language-reference/data-types/

2.5.1 Representations

The most common floating point representation is IEEE 754 binary64, commonly known as *double precision*. Alternatively, binary32, commonly known as *single precision* is sometimes used as well. It has less precision, but can yield faster calculations in some scenarios.

This thesis will mainly focus on binary64. It packs sign, mantissa, and exponent into 64 bits, which is a convenient size for most modern processor architectures.

Use	Sign	Exponent	Mantissa
Bit	63 (MSB)	62 ... 52	51 ... 0 (LSB)

The base is 2, implicitly. The sign is represented using a single bit, and the exponent is 11 bits wide. The remaining 52 bits are used for the mantissa, with one noteworthy quirk: as number are normalized ($1 \le m < b = 2$), the leading bit has to be a 1, so it is omitted. As a consequence, the mantissa has an effective width of 53 bits.

The effective value of the exponent is calculated from the stored bits by adding a fixed *bias*. For binary64, the bias is -1023, i.e. a stored 1023 corresponds to an effective exponent of 0. The smallest and largest possible exponent values have special meanings: the exponent bits 00000000000 denote a *subnormal* number, and the exponent bits 11111111111 denote the special values *infinity* (mantissa $= 0$) and *not a number* (mantissa $\neq 0$). Of those, only subnormal numbers are relevant to this thesis.

Subnormal numbers address the relatively large gap between the smallest representable normal number $n_{\min} = 1.0 \cdot 2^{e_{\min}}$ and zero. While the distance to the next bigger representable number $1.000...1 \cdot 2^{e_{\min}}$ is the value of the *least* significant bit of $n_{\min}$, the distance to zero is the value of the *most* significant bit of $n_{\min}$. By allowing the implicit leading bit to be 0, the gap can be filled with numbers in the range from $0.000...1 \cdot 2^{e_{\min}}$ to $0.111...1 \cdot 2^{e_{\min}}$, evenly spaced out by $\mathrm{ulp}(n_{\min})$.

IEEE 754 reserves the exponent bit pattern 00000000000 for subnormal numbers. A normal number with exponent $e_{\min}$ has the exponent bit pattern 00000000001. Notably, both correspond to the same effective exponent $e_{\min}$, but with different leading mantissa bits.

2.5.2 Rounding Modes

For floating point numbers, IEEE 754 defines five distinct rounding modes, corresponding to common real-world practice. For each of them, the example shows how they would perform close to 0, given ulp $= 1$.

- **Round toward nearest, ties to even**: If one value is closer to the exact result than the other, it is picked. Otherwise, the value with a 0 as the least significant bit is picked.

  $[-2.5, -1.5], (-1.5, -0.5), [-0.5, 0.5], (0.5, 1.5), [1.5, 2.5] \quad \longrightarrow \quad -2, -1, 0, 1, 2$

  This mode has an optimal maximum rounding error of 0.5ulp, and avoids rounding bias in most cases. IEEE 754 specifies it as the default mode if no other mode is set explicitly.

- **Round toward nearest, ties to away**: If one value is closer to the exact result than the other, it is picked. Otherwise, the value with the greater magnitude is picked.

  $(-2.5, -1.5], (-1.5, -0.5], (-0.5, 0.5), [0.5, 1.5), [1.5, 2.5) \quad \longrightarrow \quad -2, 1, 0, 1, 2$

  This corresponds to a very common traditional rounding mode. It also has an optimal maximum rounding error of 0.5ulp, but has a rounding bias away from zero if the signs of the data are not evenly distributed. Implementation of this mode is optional for IEEE 754 binary numbers.

- **Round toward $-\infty$**: The smaller value is picked, even if the other might be closer.

  $[-2, -1), [-1, 0), [0, 1), [1, 2), [2, 3) \quad \longrightarrow \quad -2, 1, 0, 1, 2$

  This corresponds to traditional "rounding down", extended naturally for negative numbers. It has a maximum rounding error of 1ulp, and a bias toward $-\infty$ in most cases.

- **Round toward** 0: The value with the smaller magnitude is picked, even if the other might be closer.

  $(-3, -2], (-2, -1], (-1, 1), [1, 2), [2, 3) \quad \longrightarrow \quad -2, -1, 0, 1, 2$

  This corresponds to traditional "rounding down" applied to the magnitude of the number. It is also the natural result of dropping trailing digits in a sign-magnitude representation, such as the decimal Arabic numerals used in large parts of the modern world, and floating point numbers. The maximum rounding error is 1ulp, and there is a bias toward zero if the signs of the data are not evenly distributed. Finally, the interval of numbers rounded to 0 is 2ulp wide, which is double the amount of any other number.
- **Round toward** $+\infty$: The greater value is picked, even if the other might be closer.

  $(-3, -2], (-2, -1], (-1, 0], (0, 1], (1, 2] \quad \longrightarrow \quad -2, -1, 0, 1, 2$

  This corresponds to traditional "rounding up", extended naturally for negative numbers. It has a maximum rounding error of 1ulp, and a bias toward $+\infty$ in most cases.

The rounding mode is initialized with the default at program startup, and can be changed at any point for all subsequent operations. The operating system is expected to preserve the setting for each scheduled process across context switches.

IEEE 754 requires that the result of the primitive arithmetic operations – and some other operations[25] – must be **correctly rounded**, i.e. equal to the result of the selected rounding function applied to the the exact result. To achieve this, the standard recommends the use of an extended-precision internal format to perform the requested operation, then round as the result is retrieved. On x86 machines for example, the so-called x87 instruction set for floating point operations uses an 80 bit internal representation.

2.5.3 Implementation by Programming Languages

IEEE 754 is very rigorous in most of its semantics. However, the original version (1985) does not define how programming languages should expose the machine instructions to the programmer. As a result, many programming languages have developed and standardized their own rules for implementing expressions, in some cases with large degrees of freedom.

One such inconsistency is the choice of precision for intermediate results. If an expression consisting of multiple floating point operations is evaluated, the intermediate results can be kept in the extended precision format, and only the final result needs to be rounded. This is allowed, as IEEE 754 only specifies a minimum precision for evaluation. However, otherwise deterministic evaluation results can differ based on this choice. A survey of some commonly used programming languages shows how differently this can be implemented.

- **C** has traditionally been a language designed to make very few assumptions about the underlying hardware. The C99 standard [8] is the first iteration to specify requirements for floating point behavior.

  If a floating point implementation is to be provided, some baseline requirements are outlined in section 5.2.4.2.2. Definitions from the header file `<float.h>` can be used to determine the rounding mode and evaluation precision, both of which can be "indeterminable". Additional definitions in `<fenv.h>` may be provided as supported to control settings, including rounding mode. As evaluation precision is a compile-time decision, it cannot be manipulated at runtime.

  If the platform supports IEEE 754 functionality (referred to as IEC 60599 by the C standard), it should be exposed as outlined in Annex F. The types `float` and `double` should map to IEEE 754 binary32 and binary64 respectively. The type `long double` can be an extended-precision format if available, or an alias for `double`. Operators are mapped to their IEEE 754 definitions, with default rounding by default. The intermediate precision of operations is still implementation specific, and might even be

25. Especially transcendental functions might require a disproportionate amount of precision for reliably determining correctly-rounded results. This problem is known as the **table-maker's dilemma**.

subject to compiler optimizations, e.g. depending on whether a store instruction can be optimized out. Naturally, implementations are free to provide more rigorous semantics.

- **Go** specifies the types `float32` and `float64` as binary32 and binary64 respectively[26]. The intermediate precision of operations is implementation specific[27].
- **ECMAScript**, the standard for a family of languages commonly called **JavaScript**, references IEEE754 heavily. It requires the default IEEE 754 rounding mode and a binary64 precision for all results, including intermediate values[28].
- **Python 3** uses a reference implementation model, where the language is defined by the behavior of a canonical implementation, instead of a specification document. According to its documentation[29], the type `float` is "usually implemented using `double` in C" . `sys.float_info` holds details about the underlying type. Other behavior does not seem to be documented.
- **Rust** does not have a formal specification at the time of writing. The language reference[30] specifies the types `f32` and `f64` as IEEE 754 binary32 and binary64 respectively. The documentation of the `std` crate[31] further specifies the primitive arithmetic operators to use the default IEEE 754 rounding mode, limited to the precision of the underlying type. An experimental interface for further floating point optimization exists as well.

2.6 Interval Arithmetic

Interval arithmetic is a well-known algebra that works with intervals of numbers. It is commonly used to examine the influence of errors and uncertainty on calculations. In general, a number x with a given error $\pm e$ can be written as a **error interval** of possible values $[x - e, x + e]$. In this form, the width of the interval is proportional to the error. Interval arithmetic can then be applied to find the relationship between input and output intervals of a calculation.

Using this technique, the final error of a calculation can be expressed as a function of the input errors. This technique is called **forward propagation**, and is central to this thesis. If the calculation is reversible, an output error can also be traced back to input errors. This is called **reverse propagation**, and can answer questions like "*given the maximum allowable tolerance for this result, how precise do the inputs have to be?*"

The fundamental theorem of interval arithmetic states that for any interval function F based on a real number function f

$$F(X_1, ..., X_n) = Y \implies \forall\, x_1 \in X_1, ..., x_n \in X_n \;:\; f(x_1, ..., x_n) = y \in Y$$

This property is referred to as **correctness**. A trivial solution for total functions would be $F(...) = (-\infty, \infty)$. This solution would not generally have the property of **optimality**, i.e. no function F' exists that yields correct intervals of less width than F for some inputs [7].

While deriving a correct interval function for any given real function may be difficult, the primitive arithmetic operators have simple interval forms that are also optimal [7]:

$$\begin{aligned}[a_{\min}, a_{\max}] + [b_{\min}, b_{\max}] &= [a_{\min} + b_{\min}, a_{\max} + b_{\max}] \\ [a_{\min}, a_{\max}] - [b_{\min}, b_{\max}] &= [a_{\min} - b_{\max}, a_{\max} - b_{\min}]\end{aligned}$$

26. https://go.dev/ref/spec#Numeric_types
27. https://go.dev/ref/spec#Arithmetic_operators
28. https://262.ecma-international.org/#sec-ecmascript-language-types-number-type
29. https://docs.python.org/3/library/stdtypes.html#typesnumeric
30. https://doc.rust-lang.org/stable/reference/types/numeric.html
31. https://doc.rust-lang.org/stable/std/primitive.f32.html

$$[a_{\min}, a_{\max}] \cdot [b_{\min}, b_{\max}] = [\min(S), \max(S)] : S = \{a_{\min}b_{\min}, a_{\max}b_{\min}, a_{\min}b_{\max}, a_{\max}b_{\max}\}$$

Division with $0 \in [b_{\min}, b_{\max}]$ has infinite interval endpoints, but otherwise it is simply

$$\frac{[a_{\min}, a_{\max}]}{[b_{\min}, b_{\max}]} = [\min(S), \max(S)] : S = \left\{\frac{a_{\min}}{b_{\min}}, \frac{a_{\max}}{b_{\min}}, \frac{a_{\min}}{b_{\max}}, \frac{a_{\max}}{b_{\max}}\right\}$$

The correct application of rounding to interval arithmetic is a question of almost philosophical nature. When only considering the rounded numbers, it is correct and optimal to apply the default rounding function to both the lower and the upper bound. But when considering the rounded numbers as an approximation of exact arithmetic this may not be correct, the lower bound should be calculated by rounding towards $-\infty$, and the upper bound should be calculated by rounding towards $+\infty$.

2.7 Subjective Logic

According to its inventor Audun Jøsang, subjective logic is "a formalism for reasoning under uncertainty" [4]. It extends traditional probability based logic. This thesis will focus on **binomial** logic, i.e. experiments with two possible outcomes, but subjective logic is also defined for **multinomial** logic, i.e. experiments with many possible outcomes, and **hypernomial** logic, where the outcomes of an experiment are the power set of a number of results.

Without uncertainty, a binomial distribution with the outcomes X and $\bar{X}$ is defined through a single probability. $P(X)$ is the probability that X is true, and $P(\bar{X}) = 1 - P(X)$ is the probability that X is false. The probabilities of all possible outcomes add up to 1. A **subjective logic opinion** models a binomial distribution with uncertainty. Instead of two probabilities with one degree of freedom, it is a tuple of four probabilities with three degrees of freedom.

$$\omega_X = (b_X, d_X, u_X, a_X)$$

The **belief mass** b_X quantifies the evidence for X, the **disbelief mass** d_X quantifies the evidence for $\bar{X}$. The sum of belief and disbelief is called **confidence**, and may be less than 1. The difference $u_X = 1 - b_X - d_X$ is called **uncertainty**.

Jøsang uses medical conditions as an example to explain subjective logic opinions. Let X be *"the person has the condition"*. When picking a random person on the street, there is no evidence for X or $\bar{X}$, so $b_X = d_X = 0$ and $u_X = 1$. If that person is tested using a method with perfect accuracy, the test may come back positive ($b_X = 1$, $d_X = u_X = 0$) or negative ($b_X = u_X = 0$, $d_X = 1$).

In the real world, no test is perfect, but the accuracy can usually be quantified. By considering the true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) found by a study, the following opinions can be derived from test results:

	b_X	d_X	u_X	a_X
untested	0	0	1	
positive result	$\frac{TP}{TP+FP}$	0	$\frac{FP}{TP+FP}$	$\frac{TP+FN}{TP+FP+TN+FN}$
negative result	0	$\frac{TN}{TN+FN}$	$\frac{FN}{TN+FN}$	

The last value in an opinion is the **base rate** a_X. It can range from 0 to 1 and quantifies the base probability of X being true in absence of any evidence. For a medical condition, this would be the fraction of the population that is affected. Through the base rate, an estimated probability can be derived from an opinion, even in case of remaining uncertainty. This value is called the **projected probability** and is defined as

$$P(X) = b_X + u_X a_X$$

There are a few named special cases for opinions. An opinion with $u_X = 1$ is called **vacuous**, an opinion with $0 < u_X < 1$ is called **uncertain**, an opinion with $u_X = 0$ is called **dogmatic**, and an opinion with $b_X = 1$ is called **absolute** [4, p. 20].

A key property of subjective opinions is that multiple parties can have different opinions on the same statement. The opinion of observer A on statement X is written as ω_X^A. So-called **trust networks** are graphs with opinions as edges, and observers and statements as nodes. Edges between two observers describe the trust one has in the statements of the other. If an observer A and a statement X are connected, directly or indirectly, all non-cyclic paths from observer to statement can be combined into a single opinion ω_X^A using **fusion** and **discounting** operators.

It should be noted that both fusion operators described below have special cases for combining two dogmatic opinions. These cases will not be covered in this thesis, as they introduce a discontinuity that complicates interval arithmetic operations significantly. On the topic of dogmatic opinions, Jøsang writes [4, p. 20]:

> *The intuition behind using the term 'dogmatic' is that a totally certain opinion (i.e. where $u = 0$) about a real-world proposition must be seen as an extreme opinion. From a philosophical viewpoint, no one can ever be totally certain about anything in this world. So when the formalism allows explicit expression of uncertainty, as opinions do, it is extreme, and even unrealistic, to express a dogmatic opinion.* [...] *It would require an infinite amount of evidence* [...], *which in practice is impossible, and therefore can only be considered in case of idealistic assumptions.*

2.7.1 Trust Operators

The **averaging fusion** operator $\underline{\oplus}$ is used to combine two opinions ω_X^A and ω_X^B that are not statistically independent from one another. Returning to the earlier example, two tests performed on the same person for the same condition at the same time are probably not independent. $\omega_X^{A\underline{\diamond}B} = \omega_X^A \underline{\oplus} \omega_X^A$ is defined as

$$b_X^{A\underline{\diamond}B} = \frac{b_X^A u_X^B + b_X^B u_X^A}{u_X^A + u_X^B}$$

$$d_X^{A\underline{\diamond}B} = \frac{d_X^A u_X^B + d_X^B u_X^A}{u_X^A + u_X^B}$$

$$u_X^{A\underline{\diamond}B} = \frac{2u_X^A u_X^B}{u_X^A + u_X^B}$$

$$a_X^{A\underline{\diamond}B} = \frac{a_X^A + a_X^B}{2}$$

In contrast, the **cumulative fusion** operator $\oplus$ is used to combine two opinions ω_X^A and ω_X^B that are statistically independent from one another. This operator can be used to accumulate a long term "reputation" of an actor from multiple observations of behavior at different points in time. $\omega_X^{A\diamond B} = \omega_X^A \oplus \omega_X^B$ is defined as

$$b_X^{A\diamond B} = \frac{b_X^A u_X^B + b_X^B u_X^A}{u_X^A + u_X^B - u_X^A u_X^B}$$

$$d_X^{A\diamond B} = \frac{d_X^A u_X^B + d_X^B u_X^A}{u_X^A + u_X^B - u_X^A u_X^B}$$

$$u_X^{A\diamond B} = \frac{u_X^A u_X^B}{u_X^A + u_X^B - u_X^A u_X^B}$$

$$a_X^{A\diamond B} = \begin{cases} \dfrac{a_X^A u_X^B + a_X^B u_X^A - (a_X^A + a_X^B) u_X^A u_X^B}{u_X^A + u_X^B - 2u_X^A u_X^B} & \forall\, u_X^A \neq 1, u_X^B \neq 1 \\ \dfrac{a_X^A + a_X^B}{2} & \forall\, u_X^A = u_X^B = 1 \end{cases}$$

Finally, the **trust discounting** operator $\otimes$ is used to establish opinions through transitivity. The notation $\omega_X^{[A;B]}$ is used to describe the **transitive trust path**, in this case the opinion A has on X through subjective trust in the direct observer B. $\omega_X^{[A;B]} = \omega_X^B \otimes \omega_B^A$ is defined as

$$b_X^{[A;B]} = P_B^A b_X^B$$

$$d_X^{[A;B]} = P_B^A d_X^B$$

$$u_X^{[A;B]} = 1 - P_B^A bx - P_B^A d$$

$$a_X^{[A;B]} = a_X^B$$

using the projected trustworthiness $P_B^A = b_B^A + u_B^A a_B^A$ [3]. There are many more combining operators for different kinds of situations, but they are not relevant for the evaluated scenarios.

3 Efficient Opinion Transfer Encoding

The in-memory representation of opinions on a node does not have to be the same as the format for transferring them between nodes. While the former is dictated by the arithmetic capabilities of the platform, the latter should primarily be as compact as possible without losing too much accuracy. This chapter explains the theory behind the choices made in the following chapters.

Compression can be achieved by considering the expected values to be encoded. If their distribution is uneven, they can be losslessly transformed into a more compact form with higher entropy, using techniques such as **run length encoding** [11] or **Huffman coding** [12]. Systems would generally be assumed to operate correctly, creating opinions with mostly high belief mass and low uncertainty. However, this is hard to quantify, and compressing based on wrong assumptions can lead to an increase in message size, the opposite of the desired effect.

A different approach is lossy compression, transforming the data in a way that loses some information deemed acceptable to discard. With less information to convey, a more compact representation is possible. For the purposes of this thesis, lossy compression amounts to rounding values to a representation with less precision. This can lead to dramatic improvements in data size, but the effects of the precision loss have to be evaluated carefully.

The rest of this chapter evaluates the suitability of floating and fixed point representations on a theoretical basis. The following definitions are foundational for this purpose:

- An **encoding** e is defined as a mapping of real numbers to a set of **canonical** numbers. For the evaluated number formats, it is advantageous to define the encoding on the basis of truncation, i.e. rounding toward 0.
- The **precision** $\delta_e(x)$ **of a single value** x using encoding e is defined as the width of the interval of real numbers that share a canonical number with x. For the examined encodings and truncation rounding, $\delta_e(x) = \mathrm{ulp}(x)$.
- The **precision** Δ_e **of an encoding** e is defined as the average precision of real numbers uniformly distributed over the examined interval $[a, b]$, i.e.

$$\Delta_e = \int_{x=a}^{b} \delta_e(x)\ \mathrm{d}x$$

For this thesis, encodings will only be examined in the interval $[0, 1)$. As mentioned above, opinion values are not necessarily uniformly distributed, so optimal precision as defined here does not necessarily imply optimality in the real world, but it provides at least some guidance.

3.1 Encoding as Floating Point Numbers

While floating point numbers are a reasonable representation for calculations, they are particularly ill-suited for the compact transmission of opinions. The value range for a probability is $[0, 1]$, which is only a small fraction of the range that a floating point number can represent.

Special values like -0, not-a-number, and the infinities are not needed for well-defined probability semantics. Barring negative zeroes, the sign of a probability is always positive, so the sign bit is redundant, as are all exponent values greater than 1. And while small exponent values increase the precision close to 0 dramatically, this effect is barely relevant considering the entire value range. This can be shown using the binary64 encoding as an example.

$$\Delta_e = \int_{x=0}^{1} \delta_e(x)\ \mathrm{d}x = \int_{x=0}^{1} \mathrm{ulp}(x)\ \mathrm{d}x$$

By splitting the interval based on exponent values n, after excluding subnormal numbers it can be written as

$$\Delta_e = \sum_{n=-1}^{-1022} \mathrm{ulp}(2^n) \cdot 2^n = \sum_{n=-1}^{-1022} 2^{n-52} \cdot 2^n$$

This is equal to the partial geometric sum

$$2^{52} \cdot \left(\left(\sum_{k=0}^{1022} 4^{-k} \right) - 1 \right) = 2^{-52} \cdot \left(\frac{1 - 4^{-1023}}{1 - 4^{-1}} - 1 \right)$$

The $1 - 4^{-1023}$ in the numerator can be rounded to 1, which is equivalent to performing the same calculations again for an infinite width exponent. After some simplifications, the final result can be compared to $\mathrm{ulp}_{\max} = 2^{-53}$, the worst case in the same range, also excluding 1.0:

$$\Delta_e = \frac{1}{3} \cdot 2^{-52} = \frac{2}{3} \cdot \mathrm{ulp}_{\max}$$

Even an exponent with infinite width adds less than a single bit of precision to the average case.

3.2 Encoding as Fixed Point Numbers

The inefficiencies of floating point numbers can be addressed by choosing a fixed sign and exponent, and removing the associated bits from the representation. However, the leading mantissa bit is no longer guaranteed to be 1, so it needs to be made explicit. For binary64 numbers, this frees up 11 bits. After adding one of them to the mantissa, the new format has a better precision. The result is the fixed point number encoding UQ0.54 .

	$\mathrm{ulp}_{\min}$	Δ_e	$\mathrm{ulp}_{\max}$
binary64	$2^{-1021} \cdot 2^{-53}$	$\frac{2}{3} \cdot 2^{-53}$	2^{-53}
UQ0.64	$2^{-11} \cdot 2^{-53}$	$2^{-11} \cdot 2^{-53}$	$2^{-11} \cdot 2^{-53}$

Fig. 1: Comparison of 64 bit encoding precision

UQ0.k has the optimal precision for representations with k bits, which can be shown by proving that:

- Uniform spacing of canonical numbers ($\delta_e(x) = \delta_e(y)\ \forall\ x, y$) leads to optimal precision,
- The Δ_e of any encoding with k bits and uniform spacing is at most as good as that of UQ0.k.

Uniform spacing is optimal

Let $x_1, ..., x_n$ be the canonical values of an encoding e covering $[0, 1)$. It is obvious that

1. $\sum_{i=1}^{n} \delta_e(x_i) = 1$
2. $\Delta_e = \int_{x=0}^{1} \delta_e(x)\ \mathrm{d}x = \sum_{i=1}^{n} \delta_e(x_i)^2$

Picking any two x_i and x_j, the respective precision can be expressed as

$$\delta_e(x_i) = a + b$$

$$\delta_e(x_j) = a - b$$

for some fixed a. Note that changing b does not violate point 1. The contribution of x_i and x_j in point 2 can now be expressed as

$$\Delta_e = (a+b)^2 + (a-b)^2 + \sum_{\alpha \neq i,j} \delta_e(x_\alpha)^2$$

This value is minimal for $b = 0$, meaning $\delta_e(x_i) = \delta_e(x_j) = a$. Therefore, for any encoding e with n distinct canonical values, the optimal precision is achieved with

$$\forall\, 1 \le i \le n : \delta_e(x_i) = \Delta_e = \frac{1}{n}$$

□

For uniform spacing, Δ_e of UQ0.k is optimal

A number with k bits can distinguish at most 2^k values. Taken as canonical values $x_1, ..., x_n$ of an encoding e covering $[0, 1)$, the interval can be split into a maximum of $n = 2^k$ sub-intervals. With uniform spacing, the width of each interval is

$$\delta_e(x) = \frac{1-0}{2^k} = 2^{-k}$$

On the other hand, for UQ0.k,

$$\delta_e(x) = \Delta_e = \text{ulp}(x) = 2^{-k}$$

Therefore, no encoding with uniform spacing and better precision than UQ0.k can exist.

□

One significant drawback of the UQ0.k encoding for probabilities is the **fencepost problem**, variations of which have been known to humanity since ancient times. After evenly dividing the range between 0 and 1 into 2^k intervals, $2^k + 1$ canonical values are needed to represent all the numbers from 0 to 1. As UQ0.k can only encode 2^k different values, there is no representation for 1, the greatest representable number is $1 - 2^{-k}$.

The encoding UQ1.k can be used if representation of 1 is absolutely necessary, but with a uniform spacing over $[0, 2)$, almost half the representable values fall outside the target range. A different approach could be to encode $2^k - 1$ intervals with even spacing instead, but this would either make all conversions lossy, or effectively double the interval of one specific canonical number based on rounding mode, or both. Since this thesis already employs value range normalization, the decision to round 1.0 down for transfer was made instead.

4 Evaluation Approach

The precision metric used in the last chapter is not a good choice for describing the accumulation of rounding errors over a long series of calculations. To this end, a different model is needed, which is described in this chapter. After discussing the merits of mathematical analysis versus computer simulation, a definition of the building blocks of the model follows, and the chapter concludes with a basic validation of the implementation. The scenarios and test cases built using this model are described in the next chapter, "5 Test Cases", pg. 23.

4.1 Central Methodology: Interval Arithmetic Assessment

The research subject of this thesis are trust networks, which combine trust opinions using fusion and discounting operators. The goal is to model the accumulated error, which is introduced by the limited precision of machine arithmetic. Th is done by examining operands and operations in the domain of **interval arithmetic**. The **width** of an interval, i.e. the difference between its upper and lower bound, gives a measure for the error of the subject.

For **floating point numbers**, which are the main representation of interest for calculations, the representation and evaluation semantics are defined in **IEEE 754** [10]. Due to the fact that operations are required to be **correctly rounded** (see "2.5.2 Rounding Modes", pg. 8), they can be modeled by simply rounding the bounds of a result interval to machine-representable numbers, the lower bound towards $-\infty$, the upper bound towards $+\infty$. If the unmodified interval operation is **optimal** (see "2.6 Interval Arithmetic", pg. 10), the rounded result is optimal regarding the exact result as well.

For intervals obtained through this method, the following statements will hold:

- For any combination of numbers inside the respective intervals chosen as inputs, the result obtained via the modeled finite-precision evaluation semantics is inside this interval.
- For any combination of numbers inside the respective intervals chosen as inputs, the exact result is inside the result interval.
- This is the smallest interval with the former two properties that can be derived with the chosen finite precision.

The width of the interval therefore is an **upper bound** on the absolute distance between exact and machine result. The actual distance can be significantly smaller in many cases. Given a maximum error tolerance, the result of this method is either "*guaranteed safe*" or "*not guaranteed safe*". In particular, "*guaranteed not safe*" is not a possible result.

4.2 Dead End: Error Function Analysis

An obvious way to arrive at results with this approach is symbolic analysis of the error function. The interval variants of the primitive arithmetic operators (see "2.6 Interval Arithmetic", pg. 10) can be applied to each of the examined subjective logic operators (introduced in "2.7 Subjective Logic", pg. 11) to obtain a term for the bounds of the result interval. The lower bound can then be subtracted from the upper bound to obtain the error of the operator, as a function of its inputs and their respective errors. This error function can be inspected for interesting properties like global maxima using traditional analysis, possibly assisted by a computer algebra system.

Unfortunately, the terms resulting from this method are exceedingly complex, to the point that deriving meaningful recommendations from them was not feasible in the scope of this thesis.

A major problem is the high dimensionality of both inputs and results of the function. An opinion consists of four values, each value is an interval defined by two numbers, and a binary operator combines two of them into one. This yields four distinct error functions with 16 inputs each per operator. The nature of interval arithmetic operators leads to a combinatorial explosion of sub-terms. Not all inputs are used by all operators, and the distinct error inputs can be substituted by one maximum error e. Nevertheless, obtaining useful results this way was still outside the scope of this thesis.

To illustrate the point, the following is an error term for the belief of the cumulative fusion operator. As this approach was abandoned early, the result may or may not be completely correct. The term was derived using sweeping simplifications. It assumes the use of FMA instructions to improve the precision of a multiplication followed by an addition. A number of interval arithmetic simplifications for positive numbers between 0 and 1 are also applied. The ulp of all numbers is rounded up to ulp(1), and the input errors are simplified into a single e. Of course, the more simplifications are applied, the less useful the results become. Still, the term is too complex to be useful for general recommendations.

$$ {b_X^{A\diamond B}}_{max} - {b_X^{A\diamond B}}_{min} = \frac{b_X^A u_X^B + b_X^B u_X^A + e(b_X^A + b_X^B + u_X^A + u_X^B) + 2e^2}{u_X^A + u_X^B - u_X^A u_X^B - 2\mathrm{ulp}(1) - e(u_X^A + u_X^B) - e^2} - \frac{b_X^A u_X^B + b_X^B u_X^A - 2\mathrm{ulp}(1)}{u_X^A + u_X^B - u_X^A u_X^B + 2e} + \mathrm{ulp}(1) $$

After further unsuccessful examination using the computer algebra system GNU Octave [13], this approach was ultimately abandoned due to the difficulties encountered.

4.3 Scenario Simulation

The alternative, and the approach ultimately chosen for this thesis, is **simulation**. This entails building a computer model that can be used to assemble and examine scenarios of interest. The behavior of the model should mirror how each scenario would unfold in the real world. But the goal is not simply to observe outcomes, but to gather additional information through the model that is not immediately present in the real world process. In case of this thesis, this additional information is the error interval of the trust calculations.

Simulation on a computer presents a unique challenge: the naive implementation suffers from the same rounding errors that are to be examined in the first place. A valid implementation must guarantee valid results independent of the influence of its own limitations. One approach would be to employ rounding in a pessimistic way to keep the safety guarantees absolute, even if the resulting thresholds are worse. But there is a more precise solution.

Any number in any base with a finite digit expansion is by definition a multiple of some small unit, and therefore a **rational number**. It can be represented exactly as a fraction of two integers. Without limitations on the magnitude of the latter, the results of primitive arithmetic operations are also exact. Existing implementations for arbitrary precision integers include the GNU Multiple Precision Arithmetic Library (GMP)[32], and the `Integer` type in Haskell[33]. Both also provide rational numbers based on them.

For this thesis, Haskell was chosen. Its arithmetic expressions are type-generic by default, allowing code to be written once and then evaluated using different underlying types. The arbitrary precision fraction type `Rational` is part of the base libraries, and has first class support for arithmetic expressions and many conversions.

The model developed as part of this thesis is built around `Rational`. On its basis, several variants of simulated floating and fixed point types have been implemented. They perform exact arithmetic, then round the result to a representable value of the simulated type. This can be done with single values using the default IEEE 754 rounding mode (type `Actual`), and with intervals using outward rounding bounds (type `ErrorInterval`).

The subjective logic implementation consists of the type `Opinion`, which is polymorphic in its number representation type. For the simulations, it is parameterized with the number types described above. On this type, the examined operators are implemented.

32. https://gmplib.org/

33. https://hackage.haskell.org/package/ghc-internal/docs/GHC-Internal-Integer.html#t:Integer

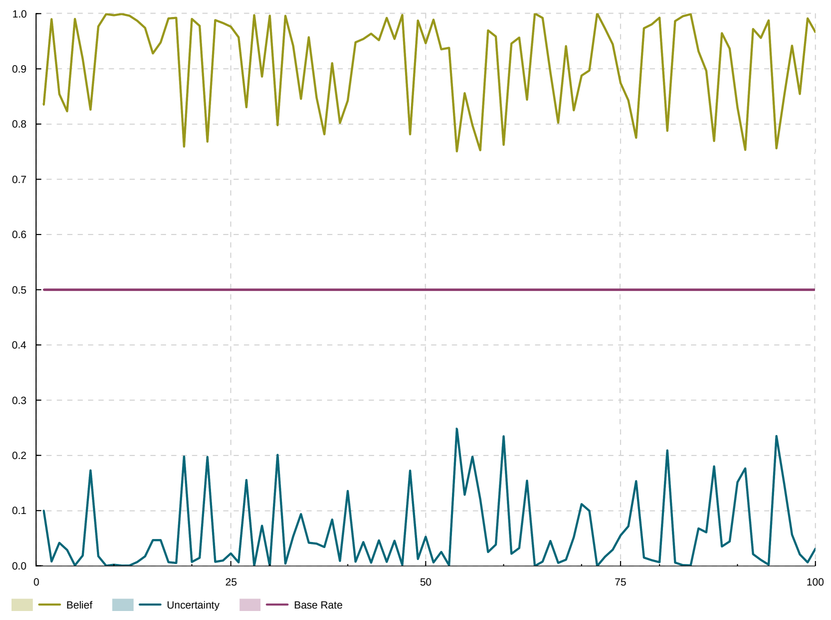


Fig. 2: 100 opinions generated by MostlyYes

Finally, a set of pseudorandom generators provide test opinions. The main generator `MostlyYes` generates opinions with generally high belief and more uncertainty than disbelief. It is based on pseudorandom integers $n_1, n_2, \ldots$ uniformly distributed in the range $[0, 2^{128}]$. Opinions are derived as follows:

$$\begin{aligned}
b &= 1 - \left(\frac{1}{2} \cdot \frac{n_a}{2^{128}}\right)^2 \\
u &= (1-b) \cdot \left(1 - \left(\frac{n_{a+1}}{2^{128}}\right)^2\right) \\
d &= 1 - b - u \\
a &= \tfrac{1}{2}
\end{aligned}$$

The `MostlyNo` and `MostlyUnknown` generators use the opinions generated by `MostlyYes`, but swap the belief with the disbelief and uncertainty values respectively. Finally, the `SpanishInquisition` generator uses the values from `MostlyYes`, but substitutes single opinions with those from `MostlyNo` with a probability of 10%. The base rate of all opinions is adjusted to 0.9 to reflect this.

Each generator takes a seed to produce values in a repeatable fashion. To reduce uniformity, the seed for the underlying integer PRNG is derived differently for each sequence.

4.4 Accommodations for Encountered Simulation Difficulties

One consequence of the rounding errors examined by this thesis is that model invariants may sometimes be violated. Especially the following properties do not always hold with finite precision calculations:

1. $0 \leq b, d, u, a \leq 1$
2. $b + d + u = 1$
3. $d \neq 0$ for all evaluated fractions $\frac{n}{d}$

Numbers that are close to their ulp in magnitude experience more significant relative adjustment by a rounding operation, so calculations on them will deviate farther from the exact result. The rounding mode is also significant. The outwardly rounded interval calculations in this thesis maximize the rounding error by design, so they are affected worse than a regular real-world round-to-nearest calculation would be.

For example, one possible cause for the first issue is the term $u_X^A + u_X^B - u_X^A u_X^B$, which appears in the denominator of both fusion operators. Using real number arithmetic with $u_X^A = u_X^B = n$ and an unspecified numerator m:

$$\lim_{n \to 0} \left(\frac{m}{u_X^A + u_X^B - u_X^A u_X^B} \right) = \lim_{n \to 0} \left(\frac{m}{2n - n^2} \right) = \lim_{n \to 0} \left(\frac{m}{2n} \right)$$

However, if n is equal to the smallest representable number ε_0 and rounding is performed towards $+\infty$, the result is instead:

$$\frac{m}{u_X^A + u_X^B - u_X^A u_X^B} = \frac{m}{2\varepsilon_0 - \varepsilon_0^2} \approx \frac{m}{2\varepsilon_0 - \varepsilon_0} = \frac{m}{\varepsilon_0} = \frac{m}{n}$$

The result of the fusion operator is off by a factor of 2. Note that the problem does not occur with default rounding in this particular case:

$$\frac{m}{u_X^A + u_X^B - u_X^A u_X^B} = \frac{m}{2\varepsilon_0 - \varepsilon_0^2} \approx \frac{m}{2\varepsilon_0 - 0} = \frac{m}{2\varepsilon_0} = \frac{m}{2n}$$

To address these issues, corrections are applied to all model calculations. They should also be considered for other finite precision implementations in general.

1. **The results of every fusion operator are normalized**, i.e. replaced with the closest number in the interval $[0, 1]$. Without this correction, probabilities outside this interval are possible, and the semantics of results may no longer be sound.

   The model further normalizes the uncertainty into the interval $(0, 1)$. This avoids discontinuities caused by special cases in the trust operators, which would otherwise complicate interval arithmetic operations significantly.
2. **The disbelief of an opinion is not tracked**. Only two values of b, d, u are needed explicitly, the other can be reconstructed as needed using $b + d + u = 1$. If all three are directly calculated as part of every operation, their sum quickly drifts away from 1. As d mostly occurs in the terms for the resulting d for the examined operators (see "2.7 Subjective Logic", pg. 11), it is the obvious candidate for elimination.
3. **Any denominator equal to zero is adjusted 1 ulp towards** $+\infty$, i.e. the division is performed with the smallest subnormal number instead.

These adjustments are designed to allow the evaluation to continue with minimal changes. They are not strictly "correct" in the mathematical sense, but rather a reasonably close approximation of reality. The same is true for floating point arithmetic in general.

4.5 Model Validation

To have any claim to correctness, the model must be able to pass tests that differentiate its output from random noise. Otherwise, the results have no scientific value as they could as well be completely arbitrary. The gold standard approach for such testing is the scientific method, used for centuries at the center of most credible branches of science. It typically entails devising experiments, predicting their outcomes with the model, running them in the real world, and comparing the results.

For this process, trust networks are not as tangible as an apple falling from a tree, which complicates the construction of suitable experiments. Validation should cover as much of the model as possible, and as the premise of this thesis is very broad, no suitable existing systems were available. Instead, an existing library implementation of subjective logic, `go-subjectivelogic`[34], was chosen to compare results against. It uses tuples of IEEE 754 binary64 numbers to represent opinions, and implements a large number of operators.

While the library cannot validate all aspects of the model, it can be compared against the model implementation of binary64 and the trust operators, which covers a significant fraction of the code base. Validation was done by generating the same opinion sequence twice, once using `go-subjectivelogic`, and once using the model. The results were then compared for consistency.

The test sequence is defined as follows. Given an input opinion sequence $\omega_0, ..., \omega_n$ and a binary trust operator $\bigcirc \in \{\oplus, \underline{\oplus}, \otimes\}$, let

$$\omega_{acc}(i) = \begin{cases} \omega_0 & \forall\, i = 0 \\ \omega_i \bigcirc \omega_{acc}(i-1) & \forall\, 1 \le i \le n \end{cases}$$

With this sequence, the following steps are performed:

1. A Go program generates $\omega_0, ..., \omega_{100}$ from MostlyYes. It then calculates $\omega_{acc}(1), ..., \omega_{acc}(100)$ using the `go-subjectivelogic` library operators. The results are written to a file, beginning with $\omega_{acc}(0)$, followed by the tuples $(\bigcirc, \omega_i, \omega_{acc}(i))$ in order. Each opinion is written as the tuple (b, u, a), using hexadecimal floating point literals to guarantee lossless (de-)serialization (see "2.2 Floating Point Numbers", pg. 5).
2. A Haskell program reads the generated file. It takes the contained input sequence, and calculates intervals for $\omega_{acc}(1), ..., \omega_{acc}(n)$ using the model. The results contained in the file are then compared against the results of the model. If a value falls outside the predicted interval, an error is raised.

Validation passed for all operators. Despite the simplifying adjustments described in the last section, the model was able to predict the opinion sequence generated using a reference implementation, with precision ranging between the magnitudes 10^{-6} and 10^{-17} (for reference, $\mathrm{ulp}(0.5) = 2^{-53} \approx 10^{-16}$). The exact end results were:

34. https://github.com/vs-uulm/go-subjectivelogic/

Avg. Fusion	belief	uncertainty	base rate
interval width	3.501407781048016e-6	8.463579934461785e-8	8.881784197001252e-16
lower bound	`0x1.d45a5adfefcc9p-1`	`0x1.6a459b0ceee01p-6`	`0x0.fffffffffffffp-1`
exact value (Go)	`0x1.d45a959e5b6aap-1`	`0x1.6a45c87d2e0c8p-6`	`0x1p-1`
upper bound	`0x1.d45ad05cccefdp-1`	`0x1.6a45f5ed71c9ap-6`	`0x1.0000000000007p-1`
Cum. Fusion	belief	uncertainty	base rate
interval width	8.985034938291392e-13	1.130788984584491e-18	4.57300863843102e-13
lower bound	`0x1.ffcec2768f97fp-1`	`0x1.52897c3bbfc0ap-20`	`0x0.ffffffffff802p-1`
exact value (Go)	`0x1.ffcec2769095ep-1`	`0x1.52897c3bc0682p-20`	`0x1.0p-1`
upper bound	`0x1.ffcec2769191cp-1`	`0x1.52897c3bc10e6p-20`	`0x1.0000000000819p-1`
Discounting	belief	uncertainty	base rate
interval width	3.176712365382528e-17	1.354472090042691e-14	1.1213252548714081e-14
lower bound	`0x1.255956976b35fp-11`	`0x1.ffb4a566239c6p-1`	`0x1.0p-1`
exact value (Go)	`0x1.255956976b3a1p-11`	`0x1.ffb4a566239fcp-1`	`0x1.0p-1`
upper bound	`0x1.255956976b484p-11`	`0x1.ffb4a56623a4p-1`	`0x1.0000000000065p-1`

5 Test Cases

The model can be used to simulate a large variety of test cases, being implemented as a set of building blocks. They include the aforementioned trust operators and opinion generators, as well as compression passes. For this thesis, they have been arranged to a number of scenarios, which have a number of parameters that can be adjusted. The challenge of picking a set of scenarios and parameters for simulation is to cover as many diverse configurations as possible, without creating an unfeasibly large number of test cases from the combinatorial explosion of variables. The general framework has been described in "4 Evaluation Approach", pg. 17, this chapter details how the test cases are assembled and analyzed.

5.1 Scenarios

The fundamental requirement for simulation scenarios is theoretically infinite scalability, as limits are found by increasing the network size until the error intervals become unacceptably large. An informal literature review for suitable scenarios had suboptimal results. One scenario archetype that was examined by multiple works is **platooning**, i.e. ad-hoc decentralized coordination between vehicle cruise control systems [3] [5] [14]. The other examined trust networks were generally simple in nature and involved only a small number of nodes [1] [6] [14] [15] [16] [17]. The scenarios implemented for this thesis are therefore very synthetic in nature, and should be considered more of a general exploration of the problem space than a recreation of concrete situations.

The **Jury** and **Reputation** scenarios benchmark the averaging and cumulative fusion operators respectively. They only involve local processing in a single representation, and no transmissions or conversions. The **Telephone** scenario, named after the children's game, evaluates trust discounting in transitive trust chains with transmissions between each node. Finally, the **Platoon** scenario is a stress test involving all three operators and transmissions.

5.1.1 Jury

This scenario tests the **averaging fusion** operator, which is used to combine opinions on statistically interdependent events. A common case are multiple observers of the same event, each with their own opinion. The namesake is a jury, a group of people deciding on a verdict in a court after hearing the evidence. A more technical application would be a group of sensors observing the same event from different perspectives, or using different principles of operation.

More formally, n different tests each produce an opinion $\omega_X^1, ..., \omega_X^n$ on the same statement X. They are combined into a single opinion $\omega_X(n)$ using averaging fusion. As repeated application of the (binary) averaging fusion operator $\underline{\oplus}$ produces a weighted result, a variadic generalization of the binary operator is used instead [18].

$$\omega(n) \;=\; \omega_X(n) \;=\; \underline{\bigoplus_{i=1}^{n}}\, \omega_X^i \;=\; (\; b_X(n) \;,\; 1 - b_X(n) - u_X(n) \;,\; u_X(n) \;,\; a_X(n) \;)$$

with

$$b_X(n) \;=\; \frac{\sum_{i=1}^{n}\left(b_X^i \cdot \prod_{j \neq i} u_X^j\right)}{\sum_{i=1}^{n}\left(\prod_{j \neq i} u_X^j\right)} \quad , \quad u_X(n) \;=\; \frac{n \cdot \prod_{i=1}^{n} u_X^i}{\sum_{i=1}^{n}\left(\prod_{j \neq i} u_X^i\right)} \quad , \quad a_X(n) \;=\; \frac{1}{n} \cdot \sum_{i=1}^{n} a_X^i$$

5.1.2 Reputation

A reputation is a long-term assessment of character, starting from complete uncertainty, and accumulating a large number of small observations over time. These observations are assumed to be generally statistically independent instances, apart from a bias by the examined characteristic. Subjective logic models this through the **cumulative fusion** operator, which is the subject of this scenario.

This principle is what the reputation scenario attempts to simulate. The reputation opinion is initially vacuous, i.e. $\omega_X(0) = (0, 0, 1, a)$. As opinions $\omega_X^1, ..., \omega_X^n$ on the trustworthiness of X arise over time, from n independent events, the reputation is updated using $\omega_X(i) = \omega_X^i \oplus \omega_X(i-1)$. Through the repeated use of the operator, the weight of each opinion decays exponentially, and more recent opinions carry more weight.

5.1.3 Telephone

In the children's game "Telephone", the participants form a line. The first one decides on a message, then whispers it into the ear of the next one. The message is thus relayed until the end of the line, where it is publicly compared to the original. In many cases, the message will have been completely mangled by the process, resulting in an entirely different phrase, or even just incomprehensible sounds. A successful world record attempt in 2008 with 1330 participants started with the message "together we will make a world of difference", after 200 steps it was mutated to "we're breaking a record", and the final result was "haaaaa".[35]

With transitive trust chains, a similar decay occurs. The popular theory "six degrees of separation" states that by following relations of the form "X personally knows Y", every person on earth is reachable from every other person with at most six steps. It stands to reason that, by blindly following similar relations of the form "X trusts Y (somewhat)", one could quickly reach people that should definitely not be trusted.

With subjective logic, the latter type of decay is expressed through the **trust discounting** operator $\otimes$. It can be used when messages are relayed from one unreliable observer to the other. An example of this could be a platoon of vehicles, where the first vehicle monitors the road ahead, and the information is propagated to the end of the platoon. If the opinions are transmitted through a lossy channel, the former type of decay becomes relevant as well.

Both effects are simulated in the Telephone scenario. With pairwise opinions $\omega_{X_0}^{X_1}, ..., \omega_{X_{n-1}}^{X_n}$, the value of $\omega(n)$ is recursively defined as

$$\omega(1) = \omega_{X_0}^{X_1} \quad , \quad \omega(n) = \omega_{X_{n-1}}^{X_n} \otimes f(\omega(n-1)) \quad \forall\, n > 1$$

with a transmission function f converting the opinion from the arithmetic representation to the transfer representation and back again.

5.1.4 Platoon

This scenario is a combination of the former three, and inspired by platooning concepts from existing research. Conceptually, each car in the platoon observes the car in front of it using different sensors, the values of which are combined using multinomial averaging fusion, as in the Jury scenario. Over time, each car builds a reputation from the averaged opinions using cumulative fusion, as in the Reputation scenario. The last car needs to estimate the trustworthiness of the first car, so the opinion has to be transmitted from car to car. As in the Telephone scenario, each hop passes the opinion to a transmission function, and each receiving car then discounts the result using its reputation of the transmitting car. The opinion calculated by the last car is the final result of the scenario.

For simplicity, only a single scaling factor n is used. Unlike with the previous scenarios, each increment in n adds three operator evaluation steps, one for each of the three operators involved. Consequently, this

35. https://www.thirdsector.co.uk/whisper-loud-record-broken/fundraising/article/859333

scenario is expected to scale worse than the others.

Without further modifications, the complexity of calculating step n is $O(n^3)$, and evaluation from 1 to n is $O(n^4)$ as the value $\omega(n)$ is not a function of $\omega(n-1)$. As a practically necessary optimization, each car receives the same input opinions, reducing the complexity for each step to $O(3n)$, and the sequence total to $O(3n^2)$.

More formally, using an opinion sequence $\omega_1, ..., \omega_n$, as well as a transfer function f, the reputation $\omega(n) = R_{X_0}^{X_n}(n)$ is derived using:

$$
\begin{aligned}
\omega_{X_{i-1}}^{X_i}(x) &= \bigoplus_{j=1}^{x} \omega_j \quad \forall\, 1 \le i \le x \le n \\
R_{X_{i-1}}^{X_i}(0) &= (0, 0, 1, a) \quad \forall\, 1 \le i \le n \\
R_{X_{i-1}}^{X_i}(x) &= \omega_{X_{i-1}}^{X_i}(x) \oplus f(\, R_{X_{i-1}}^{X_i}(x-1)\,) \quad \forall\, 1 \le i \le x \le n \\
R_{X_0}^{X_i}(x) &= R_{X_{i-1}}^{X_i}(x) \otimes R_{X_0}^{X_{i-1}}(x) \quad \forall\, 2 \le i \le x \le n
\end{aligned}
$$

5.2 Parameters

Each scenario maps the generated sequence $\omega_X^1, ..., \omega_X^n$ to the result sequence $\omega(1), ..., \omega(n)$. The opinions originate from pseudo-random **opinion generators**, the implementation of which is described in "4.3 Scenario Simulation", pg. 18. The *MostlyYes* generator produces a sequence of opinions with expected $d < u \ll b$. *MostlyNo* is similar, but with expected $b < u \ll d$, and *MostlyUnknown* produces opinions with expected $d < b \ll u$. They each use a fixed base rate, which is set to 0.5. To represent situations with an adversary among a majority of cooperating nodes, the generator *SpanishInquisition* uses the values from *MostlyYes*, but substitutes single opinions with those from *MostlyNo* with a probability of 0.1. To account for this, and to introduce more variety in the test cases, the base rate of all its opinions is adjusted to 0.9.

The **arithmetic representation** is relevant for any scenario. It determines the characteristics of arithmetic operations. Of primary interest are IEEE 754 binary32 and binary64 floating point numbers.

For scenarios involving data transmission, there is also a choice of **transfer representation**. The simplest case is the identity transformation, i.e. the transfer representation being equal to the arithmetic representation. This is referred to as *Local* encoding in the model, and is primarily intended as a baseline to compare other representations to. Further choices include triplets of fixed point numbers of the form UQ0.k. Powers of two are obvious candidates for k, preliminary experiments have shown that values $k \le 8$ compromise precision too quickly, while values $k > 16$ have relatively little additional benefit. This mainly leaves UQ0.16. Additionally, the format *Packed* encodes belief and uncertainty as UQ0.11 and base rate as UQ0.10, for a convenient total size of 32 bits per opinion.

With the scenarios detailed in the last section, the final set of test cases is:

$$\textit{local-scenario} \times \textit{generator} \times \textit{arith-rep} \;\cup\; \textit{remote-scenario} \times \textit{transfer-rep} \times \textit{generator} \times \textit{arith-rep}$$

$$
\begin{aligned}
\textit{local-scenario} &= \{\text{Jury}, \text{Reputation}\} \quad , \quad & \textit{remote-scenario} &= \{\text{Telephone}, \text{Platoon}\} \\
\textit{arith-rep} &= \{\text{binary32}, \text{binary64}\} \quad , \quad & \textit{transfer-rep} &= \{\text{Local}, \text{UQ0.16}, \text{Packed}\}
\end{aligned}
$$

$$\textit{generator} = \{\text{MostlyYes}, \text{MostlyNo}, \text{MostlyUnknown}, \text{SpanishInquisition}\}$$

5.3 Test Case Processing

For each test case, the data points $\{ (i, \omega(i)) \mid i \in [1, n] \}$ are calculated. This is done in two different variants:

- With singleton values, using default IEEE 754 "round to nearest, ties to even" rounding. This is how the scenario could be evaluated by a real processor.
- With intervals, rounding the upper bound towards $+\infty$ and the lower bound towards $-\infty$. This establishes the error interval, which is guaranteed to contain both the singleton value as well as the exact value calculated with infinite precision.

Each test is run with ten different seeds to the opinion generator, and the results are averaged. The result is plotted as a line graph, with the interval shaded and the singleton value as a line inside. The belief is plotted in green, the uncertainty in blue, and the base rate in magenta.

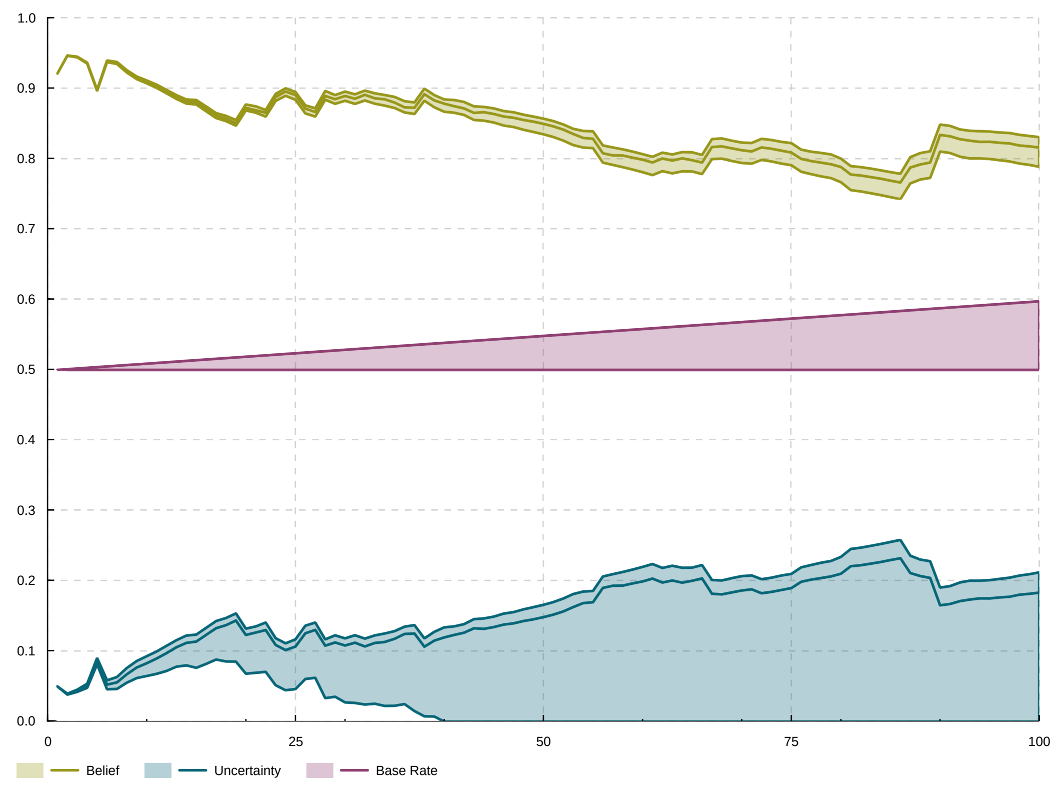


Fig. 3: Platoon-Packed-MostlyYes-binary64

An alternative plot type shows just the width of the intervals, which is useful if the intervals are too small to be visible to the naked eye in the other graph.

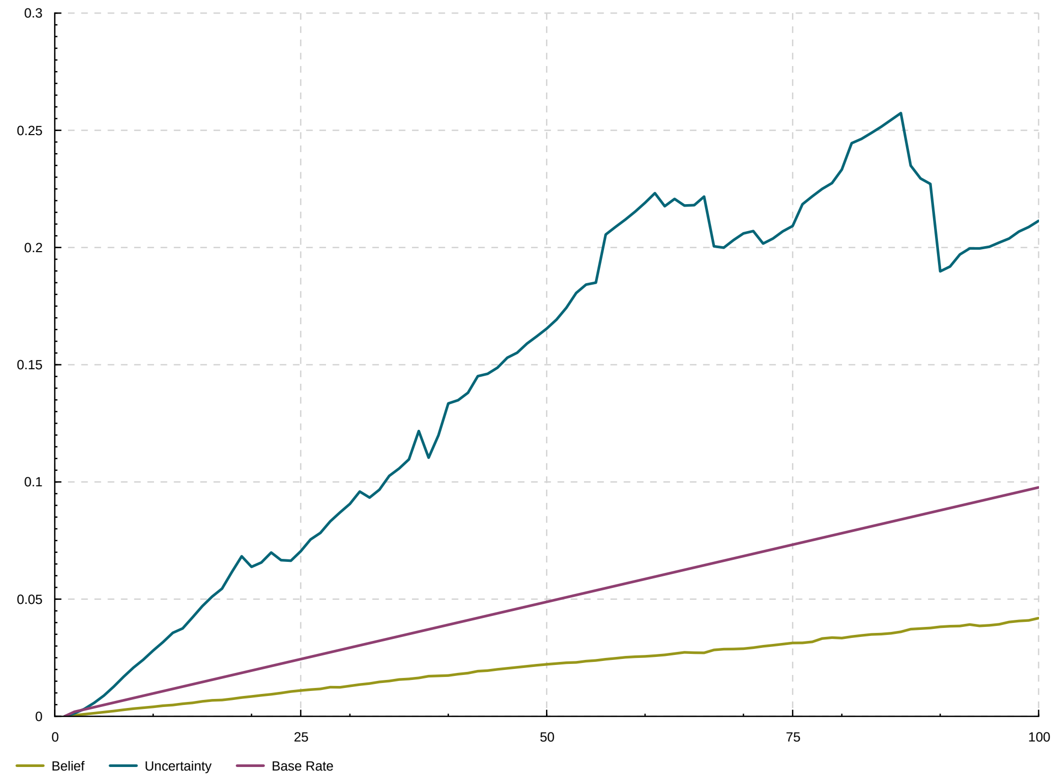


Fig. 4: Platoon-Packed-MostlyYes-binary64 - width plot

5.4 Simulation Size

The Reputation and Telephone scenarios can be efficiently evaluated sequentially, with a constant-time function mapping $\omega(n)$ to $\omega(n+1)$. Fully evaluating steps 1 to n of a test case has the complexity $O(n)$. With the Jury and Platoon scenarios however, each step i needs to be evaluated in isolation, with a complexity of $O(i)$. Fully evaluating steps 1 to n has the complexity $O(n^2)$. This puts a practical limit on the number of steps that can be simulated in a reasonable amount of time. For this thesis, $n = 100$ was chosen as the cutoff.

To put this number into perspective, examining the area covered by 100 cars in different situations can be helpful. Note that his image does not directly map to any of the scenarios. The topic of traffic capacity calculation is far outside the scope of this thesis, but simple calculations can provide a general scale of things. Models estimating the capacity of a lane of car traffic depend on a large number of factors, a somewhat arbitrary value of 2000 vehicles per hour, towards the upper end of the envelope, will be assumed.

- A typical German highway has two lanes per direction, with a target speed of 130 $\frac{km}{h}$. A $130km$ long lane segment contains 2000 vehicles – those that will pass through the end of the section within one hour. With each vehicle taking up a slot of $\frac{130km}{2000} = 65m$ length[36], a network of 100 cars on two lanes covers $\frac{100 \cdot 65m}{2} = 3.25km$ of highway.
- Under the same conditions, a platoon of 100 vehicles in one lane has a length of $100 \cdot 65m = 6.5km$.
- For a four-way intersection with two approaching lanes from each direction, a network of 100 cars spans the next $\frac{100}{8} \approx 12$ cars per lane.

36. Conveniently, this is exactly the recommended following distance of "half the speedometer in meters".

- The intersection is passed by 4000 vehicles per hour per direction, so with network size of 100 cars, each car can be part of the network for $\frac{100}{16000\frac{1}{h}} \cdot 3600\ \frac{s}{h} = 22.5s$.
- Assuming a constant approach speed of 50 $\frac{km}{h}$, which is the inner city speed limit in Germany, the network spans out 50 $\frac{km}{h} \cdot 3.6\ \frac{h \cdot m}{s \cdot km} \cdot 22.5s = 312.5m$ from each traffic light.
- On the other hand, if all of the vehicles at the intersection are completely stationary, assuming a slot length of $5m$ per car, the network spans out $\frac{100}{8} \cdot 5m = 62.5m$ instead.

6 Findings

The Cartesian product of the parameters described in the last chapter is a large number of test cases, the results for all of which can be found in the appendix. A lot of them behave as expected, with error intervals far below 1%. This chapter will focus on combinations of parameters that produce unfavorable results, as well as other aspects that are relevant for answering the research questions.

An important term to describe the behavior of the model is **divergence**. In this case, it is used for results moving away from the expected outcome. For intervals, this means a width approaching 1. For singleton values, this means a significant, growing distance from the exact result.

Technical note: this section, containing small paragraphs of text intermixed with large, similar plots, has been particularly hard to typeset coherently. As the goal is to illustrate points that are visually quite obvious, the decision has been made to shrink the plots down aggressively. The plot layout is described in "5.3 Test Case Processing", pg. 26, larger versions of all plots can be found in the appendix. Also in the appendix is the link to the raw data, as well as a description of its format.

6.1 Binary32

When compared to binary64, binary32 performs either similar or significantly worse, depending on the scenario. Concerning effects arise with the combinations Jury-MostlyYes-binary32 and Jury-SpanishInquisition-binary32. Apart from the intervals suddenly opening up after roughly 25 steps, the singleton belief value also jumps from almost 1 to almost 0. Transferred back to the courtroom analogy, an increase in jurors agreeing on the innocence of a defendant results in a "guilty" verdict. In contrast, the behavior with binary64 is completely innocuous.

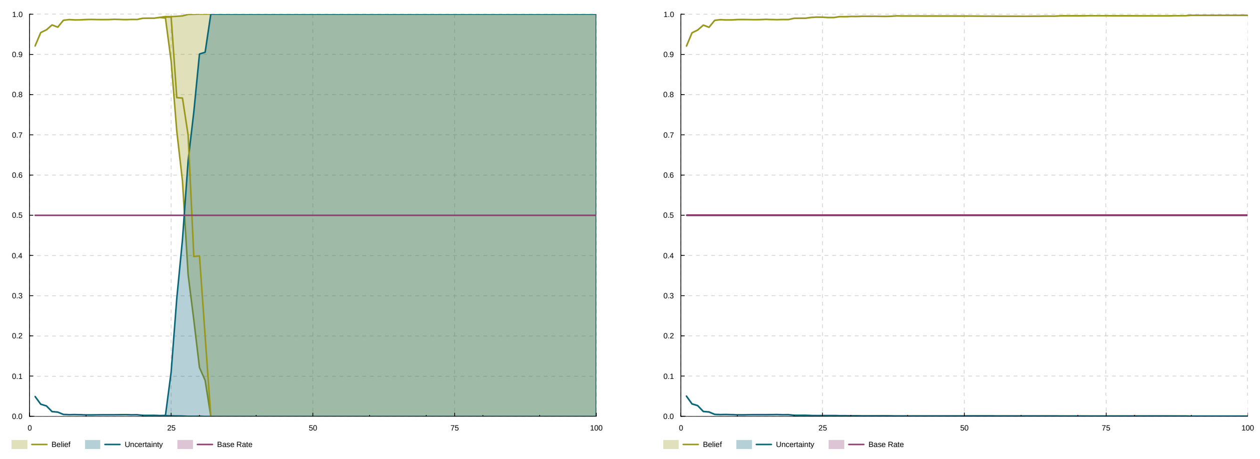


Fig. 5: Jury-MostlyYes-binary32 and Jury-MostlyYes-binary64

In the Platoon scenario, even with Local transmission, a sudden singleton value divergence can be observed in other variables as well. With the MostlyYes and SpanishInquisition generators, all variables quickly reach paradoxical values. Again, no such behavior occurs with binary64.

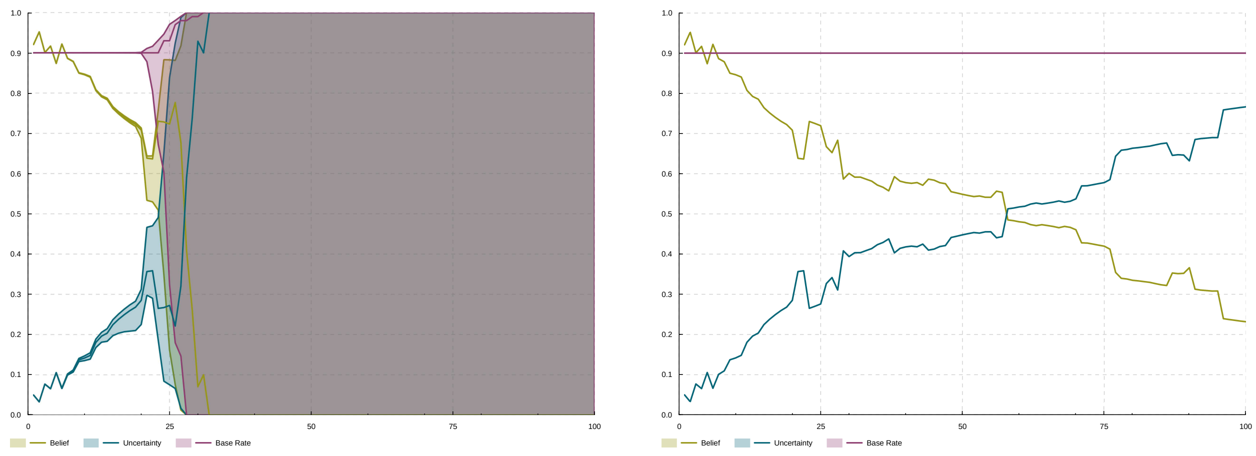


Fig. 6: Platoon-Local-SpanishInquisition-binary32 and Platoon-Local-SpanishInquisition-binary64

6.2 Cumulative Fusion - MostlyUnknown

For reasons that are not entirely clear at this point, repeated cumulative fusion of opinions with high uncertainty results in a significant growth of error intervals. This affects the Reputation and Platoon scenarios, occurring earlier with binary32, but also affecting binary64. The effect starts with the base rate, the other variables follow suit after a number of steps. While the intervals diverge, the singleton values stay stable in all test cases.

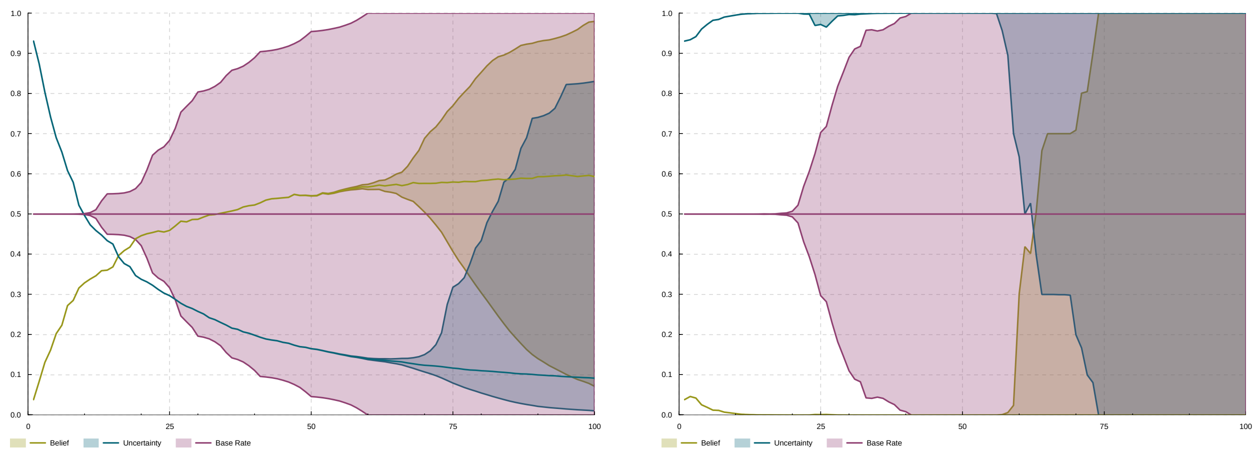


Fig. 7: Reputation-MostlyUnknown-binary64 and Platoon-Local-MostlyUnknown-binary64

6.3 Transfer as UQ0.16

Compared to Local, UQ0.16 doubtlessly introduces additional error interval growth. However, when combined with binary64, it does not cause additional test cases to diverge, and the error stays well below 1% in most of them. The exceptions are Platoon-UQ0.16-MostlyYes-binary64 and Platoon-UQ0.16-SpanishInquisition-binary64, reaching a maximum interval width of roughly 6% and 3% respectively.

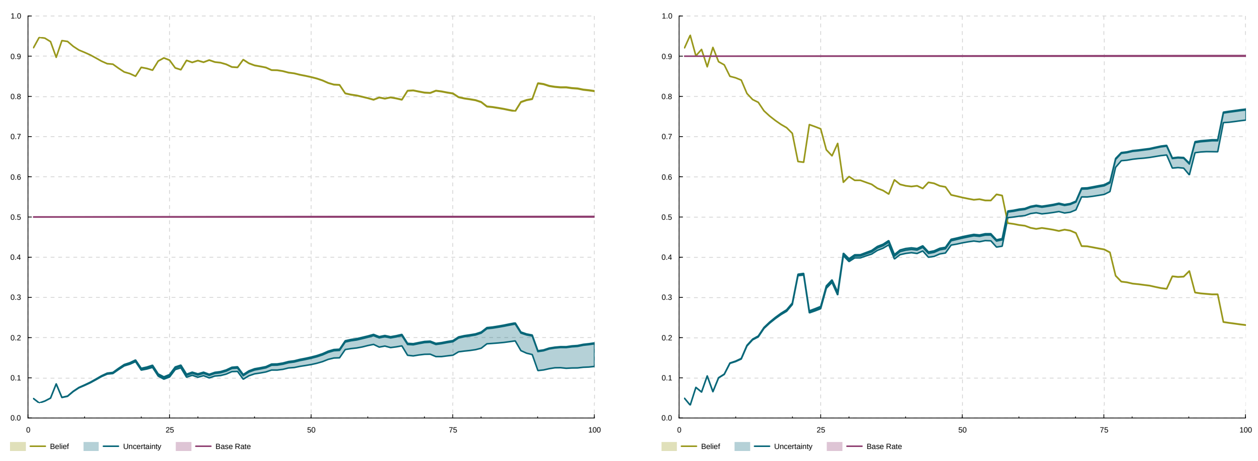


Fig. 8: Platoon-UQ0.16-MostlyYes-binary64 and Platoon-UQ0.16-SpanishInquisition-binary64

6.4 Transfer as Packed

As expected, Packed transmission performs strictly worse than UQ0.16. However, while error intervals reach significant sizes, no additional configurations diverge as a result. In the Platoon scenario, the uncertainty interval grows as large as roughly 27% using MostlyYes, and 36% with SpanishInquisition. In the Telephone scenario, an uncertainty interval width of ca. 18% is reached with MostlyYes.

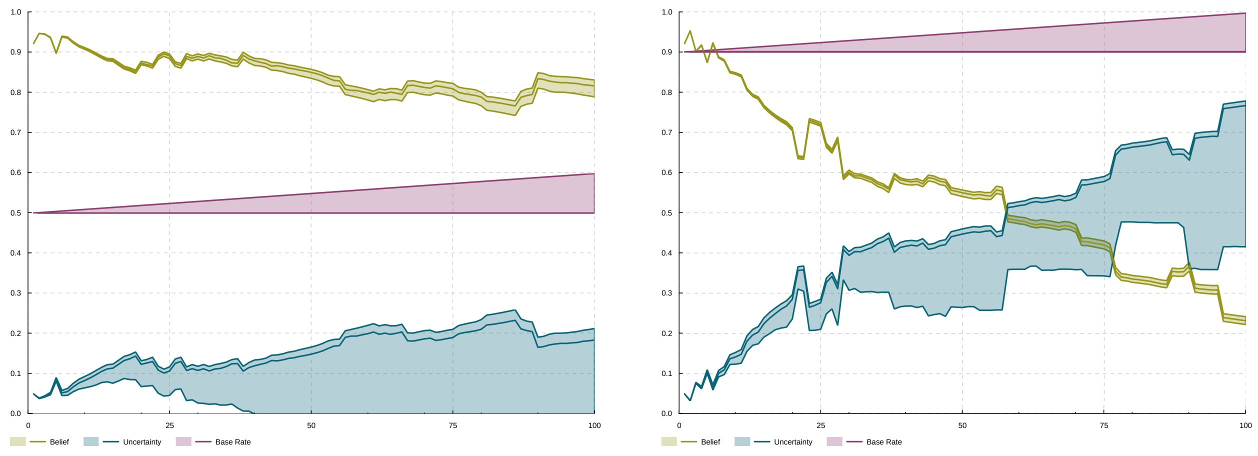


Fig. 9: Platoon-Packed-MostlyYes-binary64 and Platoon-Packed-SpanishInquisition-binary64

6.5 Simulation Errors

In the combinations Platoon-Packed-SpanishInquisition-binary32 and Platoon-UQ0.16-SpanishInquisition-binary32, singleton values outside the error interval occurred. This happened after 31 steps, where the model had already diverged completely. In both cases, the uncertainty interval spanned from 0 to almost 1, while the singleton value was exactly 1. The root cause of this issue was ultimately not found in time, but is presumably an interaction between model arithmetic, normalization at the interval bounds, and the averaging of multiple runs of the scenario with different seeds.

As this is a small difference at a point where the model has already lost all predictive meaningfulness, this was ultimately not considered severe enough to compromise the rest of the evaluation.

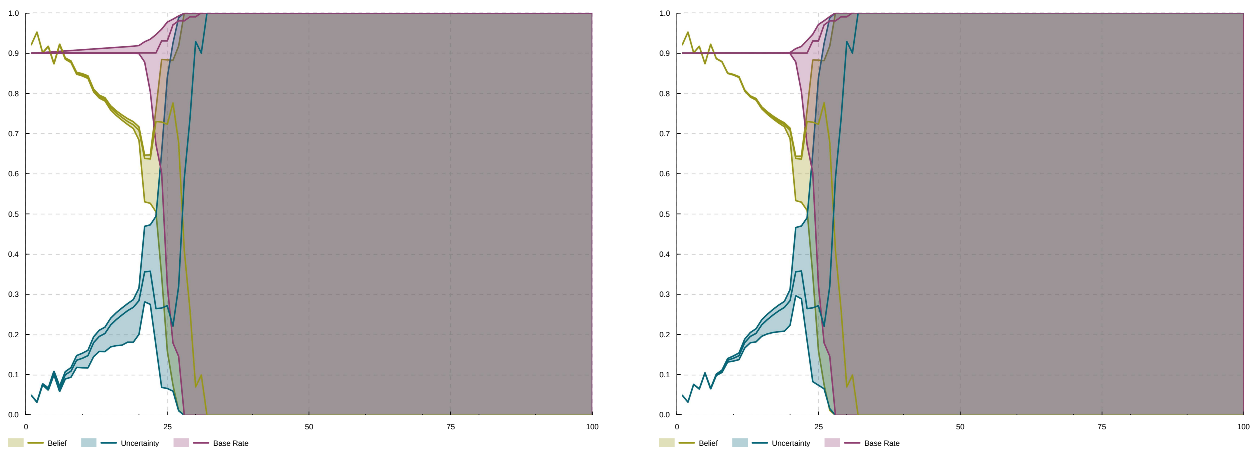


Fig. 10: Platoon-Packed-SpanishInquisition-binary32 and Platoon-UQ0.16-SpanishInquisition-binary32

7 Discussion

Strict criteria for what constitutes an "acceptable" level of precision are difficult to define; the exact requirements will vary depending on the application. Nevertheless, the simulation results can be grouped into broad categories, based on the behavior exhibited by one or more values, and the expected behavior of a real system with similar parameters. They are ordered from most to least undesirable:

1. **Both singleton value and interval diverge.** Paradoxical behavior has been observed in the model, and should be expected to occur in any system with a similar configuration. This should be avoided at all costs.
2. **The error interval diverges, but the singleton value stays stable.** Simulations did not find a concrete example for problematic behavior, but correctness cannot be guaranteed.
3. **The error interval covers a clearly visible fraction of the value range $[0, 1]$, but does not diverge completely.** Incorrect trust decisions may occur, but possible value divergence is bounded.
4. **The error interval is at most barely visible in the diagram.** The system will remain stable, and incorrect trust decisions are statistically unlikely.

Category 1 behavior was only observed with binary32 calculations, it did not occur with binary64. Category 2 behavior was observed with any representation as a result of repeated cumulative fusion of opinions with high uncertainty. Excluding that, occurrences of category 3 behavior were observed as a result of using the Packed transmission format, and could be pushed toward category 4 by using UQ0.16 for transmission instead.

Based on these findings, the following recommendations can be made:

1. Calculations should be carried out using binary64 precision. Downsizing to binary32 is likely to lead to system stability issues.
2. Pending further investigation, opinions with high uncertainty should not be used in cumulative fusion. Otherwise, system stability cannot be guaranteed at this point.
3. Significant transfer size reduction can be achieved by encoding opinions as fixed point numbers of the form UQ0.k. Both UQ0.16 triplets and Packed triplets may be good candidates depending on implementation requirements, with a total size of 48 and 32 bits per opinion respectively.

8 Conclusion

This concludes this examination of arithmetic precision in subjective logic networks. Based on the findings, the research questions can be answered as follows:

RQ1: How can the accuracy of a subjective logic network be modeled so that justified and meaningful recommendations can be derived?

Using interval arithmetic to simulate a set of given scenarios with different parameters produced useful results, although with limited potential for generalization. Symbolic error function analysis did not lead to useful results with reasonable effort.

RQ2: Given the currently mostly theoretical nature of subjective logic networks, how can the predictions of such a model be validated?

The primitive operators of the model have been validated against an existing implementation of subjective logic. Investigating predictive value further would require more concrete reference implementations and scenarios.

RQ3: Which concrete recommendations regarding the processing and transmission of opinions can be derived from this model?

The simulation results suggest that for the relatively small scenarios covered in existing research, loss of arithmetic precision due to floating point inaccuracies is not a significant factor. However, as models grow in size beyond a handful of opinions, the effect becomes more and more important to consider. Simulations of up to 100 nodes revealed the following pitfalls:

- Implementations should use IEEE 754 binary64 or more precise formats for calculations, use of binary32 can produce paradoxical results under specific circumstances.
- The disbelief should not be explicitly tracked as part of the opinion state tuple, but instead calculated using $d = 1 - b - u$ as needed. Otherwise, the sum $b + d + u$ will likely drift away from 1 after only a few trust operations. Violation of this core invariant compromises the overall correctness of subjective logic calculations.
- When implementing operators, the inputs, intermediate values, and results should be bounds-checked and adjusted accordingly. Otherwise, arithmetic errors such as division by zero, and illegal states such as probability values outside the range $[0, 1]$ are a likely consequence.
- Opinions with high uncertainty value might not be safe for repeated cumulative fusion. While no paradoxical behavior was observed, the total absence of if cannot be guaranteed at this point. However, as the vacuous opinion with equal base rate is the neutral element of cumulative fusion, the utility of such an operation is limited even with perfect arithmetic precision. Instead of fusing such an opinion to a long-term reputation, simply discarding it might be prudent until further investigation.
- Fixed point formats should be strongly considered for space-efficient opinion transmission. UQ0.k numbers are theoretically optimal for opinions with uniform value distribution, and especially superior to floating point formats in most cases. However, the inability to represent 1.0 should be kept in mind. Should this capability be essential, UQ1.k can be used, but this is much less efficient as almost half of the representable values fall outside $[0, 1]$. The examined variants – UQ0.16 triplets and (b: UQ0.11, u: UQ0.11, a: UQ0.10) – showed promising results, giving compression ratios of 4.0 and 6.0 respectively over the uncompressed binary64 without compromising system integrity in the simulated scenarios.

9 Future Work

The encountered problem with cumulative fusion of opinions with high uncertainty should be examined more closely. It is possible that this behavior is exclusively a result of the particularly pessimistic modeling approach of this thesis, and has no impact on calculations performed by a real machine. Further experiments were not possible in the time frame of the thesis.

The premise of this thesis should be re-investigated once the parameters for concrete implementations are better understood, and more specific and realistic scenarios are available for simulation. Important variables that are unclear at this point include the general nature of generated opinions, the topology of trust networks, the number of participating nodes, the structure of communication protocols, the bandwidth and other characteristics of communication channels, and the capabilities of the hardware. The current evaluation is only a broad exploration with coarse results, and important details might have been missed between the data points.

More efficient transfer formats can be developed once the distribution of transmitted opinions for typical use cases is more thoroughly understood.

The analytical approach, i.e. derivation of meaningful operator error functions, turned out to be infeasible within the scope of this thesis. Nevertheless, the prospect of generalizable guarantees might make another investigation – maybe from a formal mathematical background and/or with more resources – worth considering.

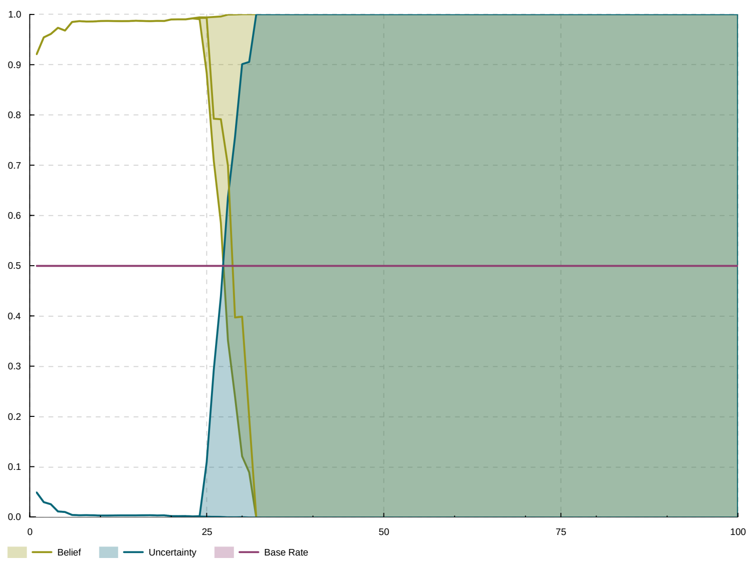
A significant number of lower-powered embedded processors do not have support for floating point arithmetic. Fixed point arithmetic is frequently used as a substitute, and could generally be well suited for the problem domain. A major issue is that precise evaluation of multiplication and division requires integer operations with up to double the width of the fixed point type. As solutions to this will most likely be hardware specific, the possibilities have not been examined as part of this thesis.

Appendix: Detailed Results

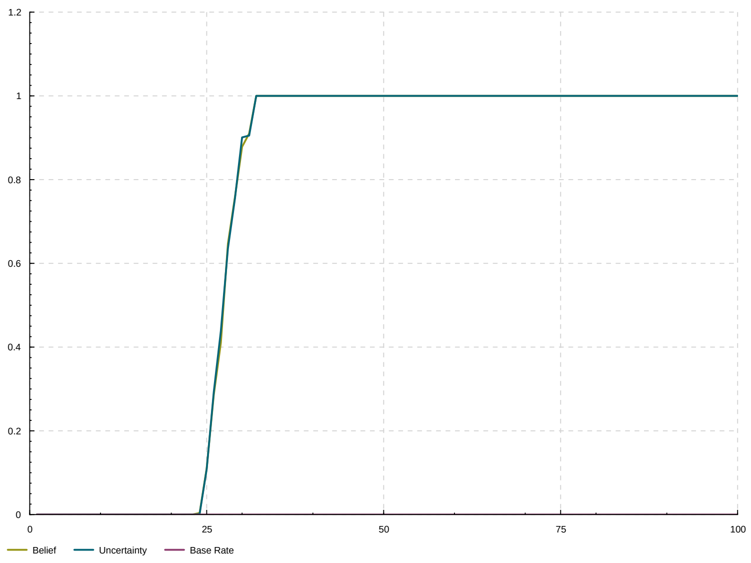
This section holds reports of all simulation results in detail. Each page contains a chart with the simulation results, a chart with the width of each interval, and the exact final width of each interval.

The raw values can be found in the TSV files at https://cgit.sowophie.io/master/tree/simulations/out. The fields for belief, uncertainty, and base rate are separated by tab characters. Each field contains the lower interval bound, singleton value, upper interval bound, and interval width, separated by spaces. Fields with singleton values outside the interval are surrounded by `!>` and `<!`.

Jury-MostlyYes-binary32



Values

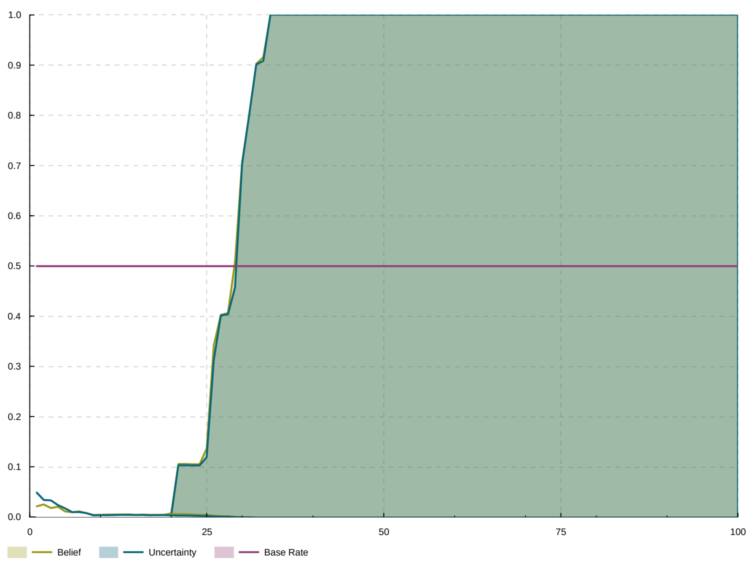


Interval Width

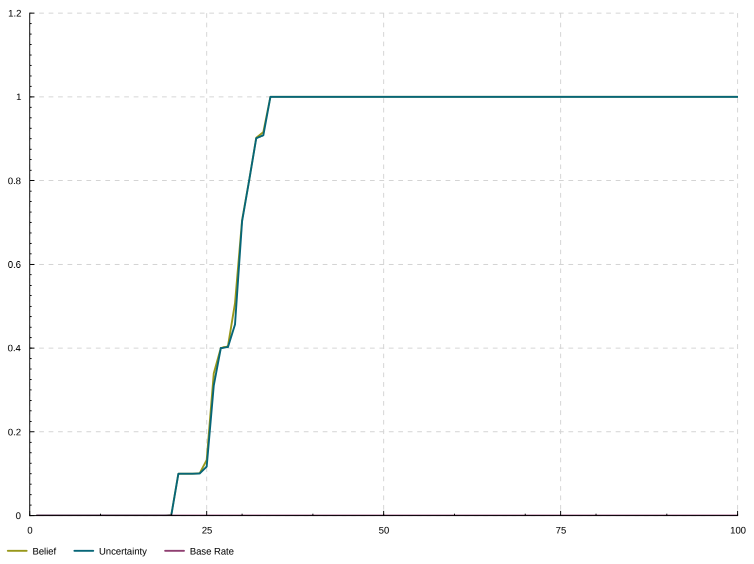
Final interval width:

```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.3ap-18
```

Jury-MostlyNo-binary32



Values

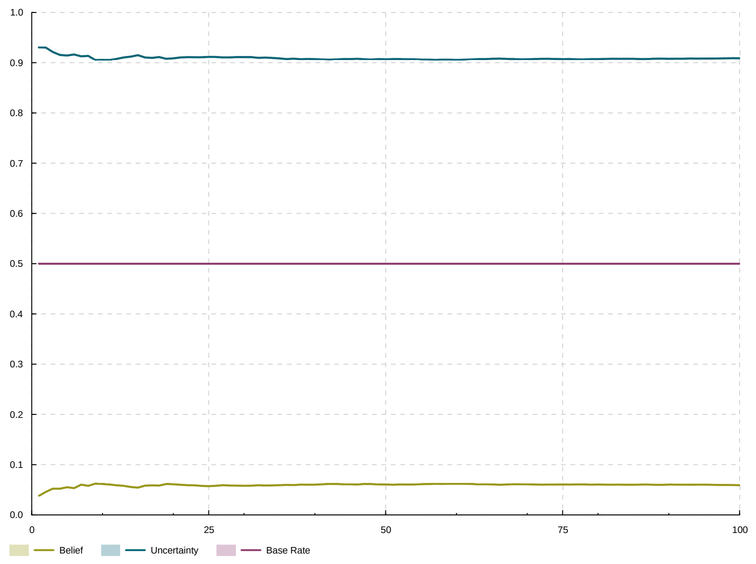


Interval Width

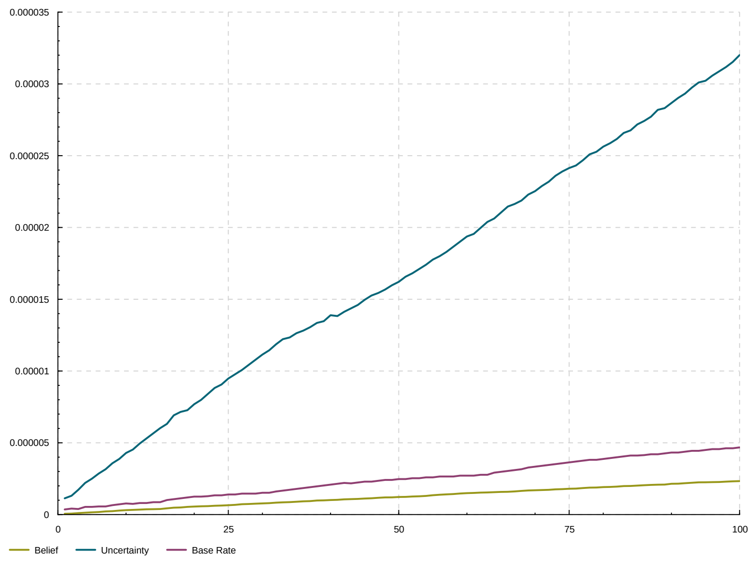
Final interval width:

```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.3ap-18
```

Jury-MostlyUnknown-binary32



Values

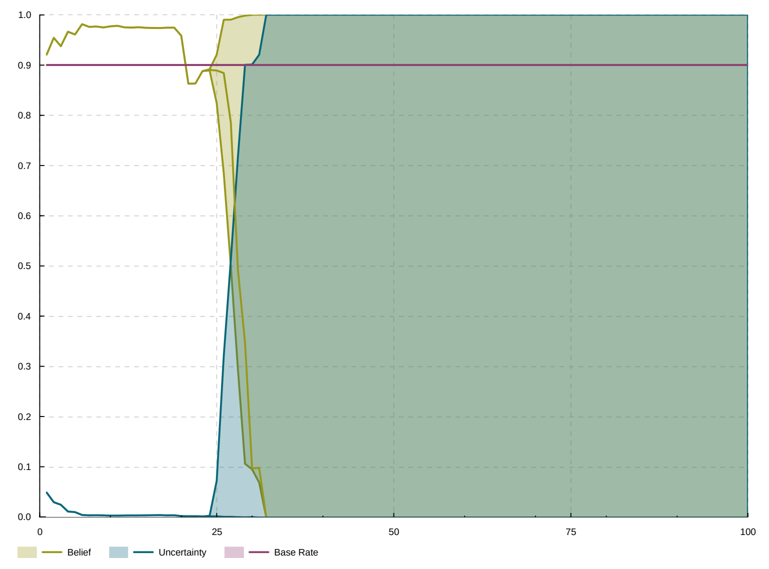


Interval Width

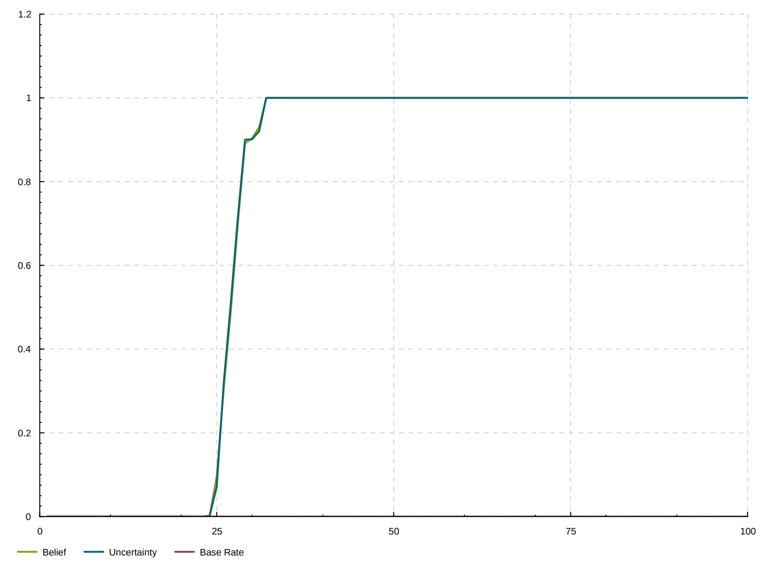
Final interval width:

```
b = 0x1.39p-19
u = 0x1.0c8p-15
a = 0x1.3ap-18
```

Jury-SpanishInquisition-binary32



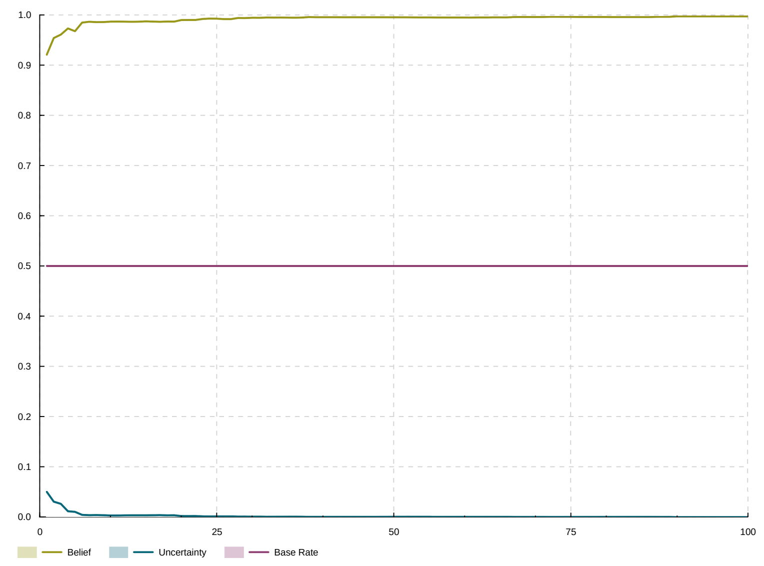
Values



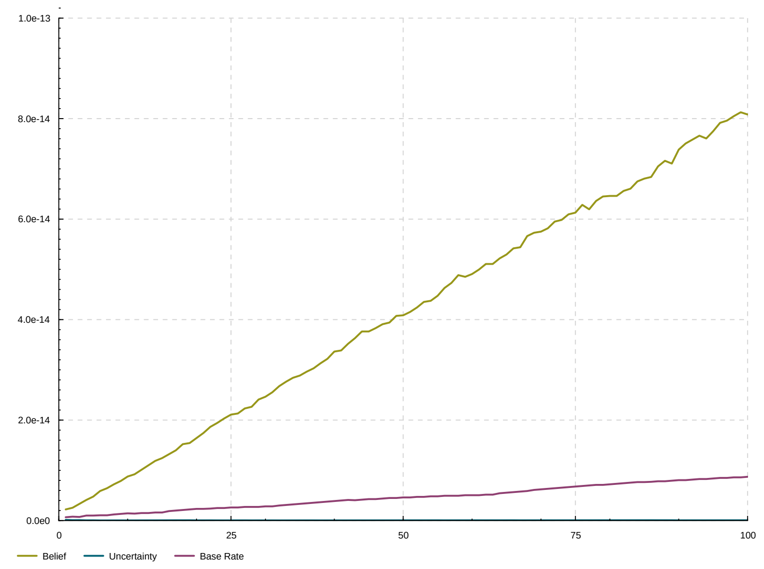
Interval Width

Final interval width:
```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.1ap-17
```

Jury-MostlyYes-binary64



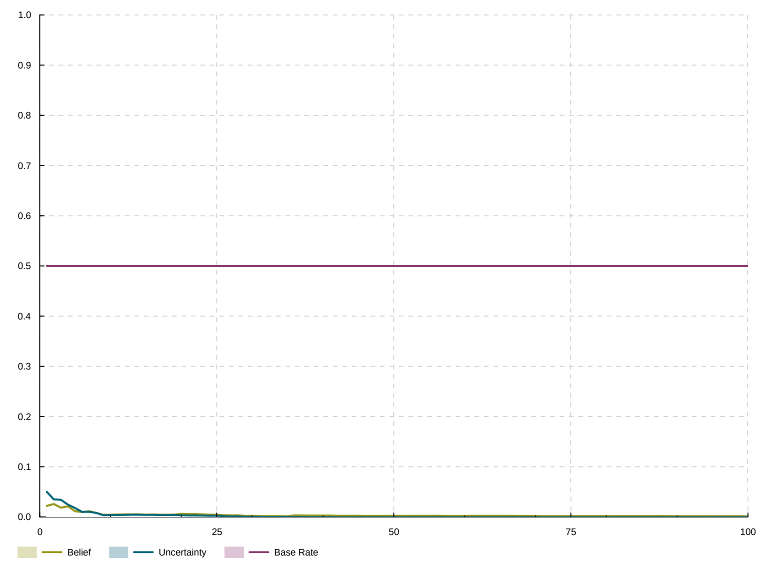
Values



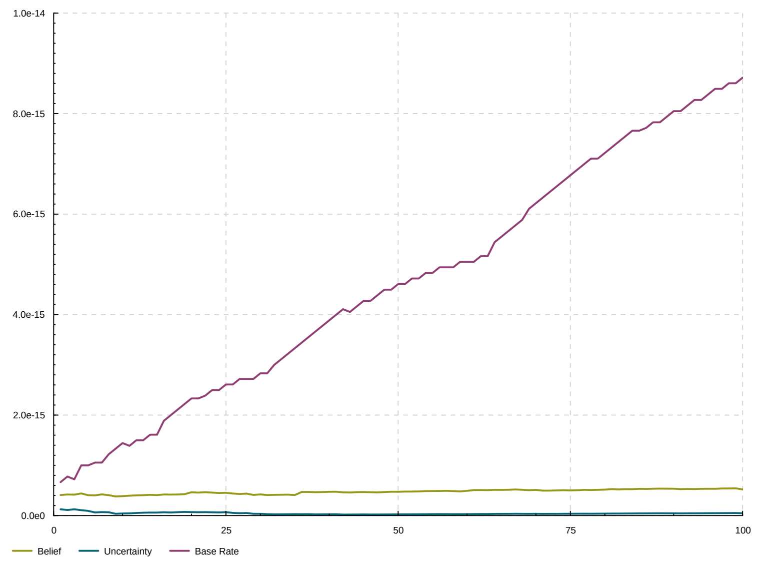
Interval Width

Final interval width:
```
b = 0x1.6cp-44
u = 0x1.298p-54
a = 0x1.3ap-47
```

Jury-MostlyNo-binary64



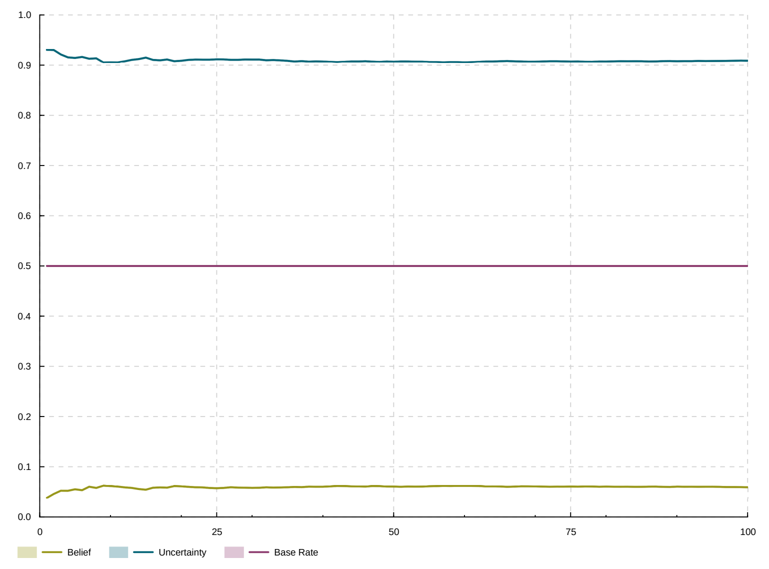
Values



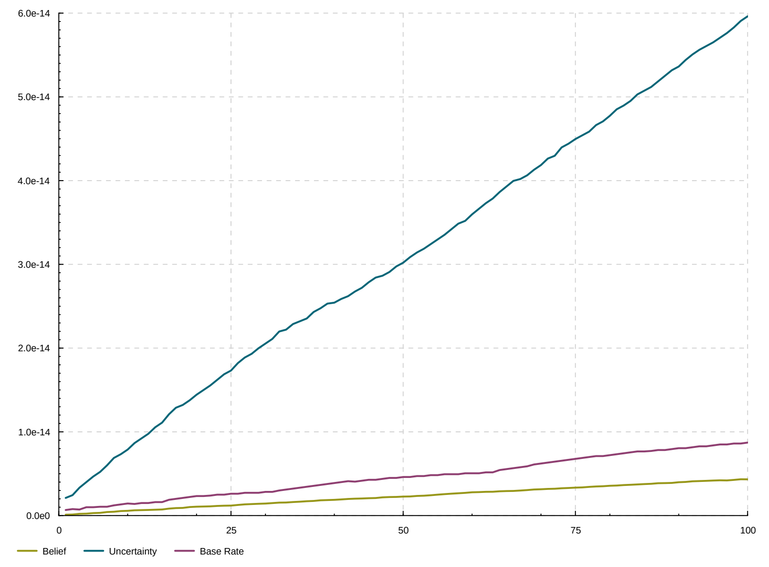
Interval Width

Final interval width:
```
b = 0x1.2ccp-51
u = 0x1.aep-55
a = 0x1.3ap-47
```

Jury-MostlyUnknown-binary64



Values

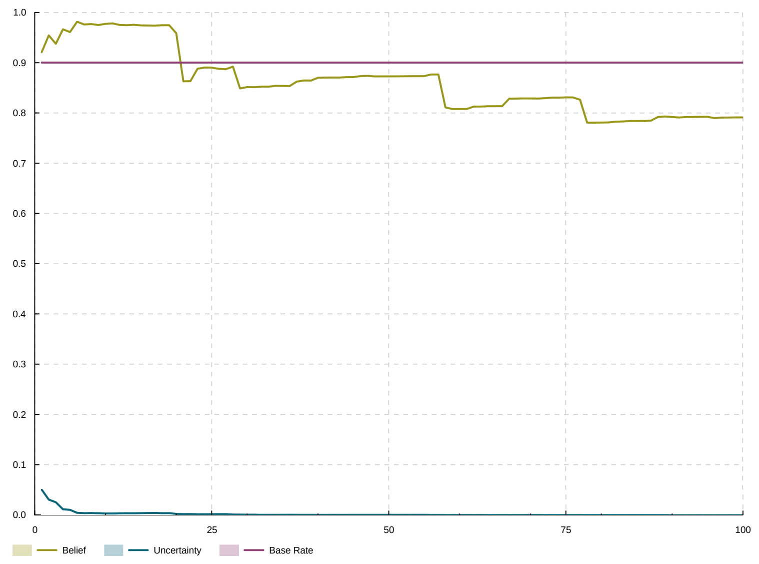


Interval Width

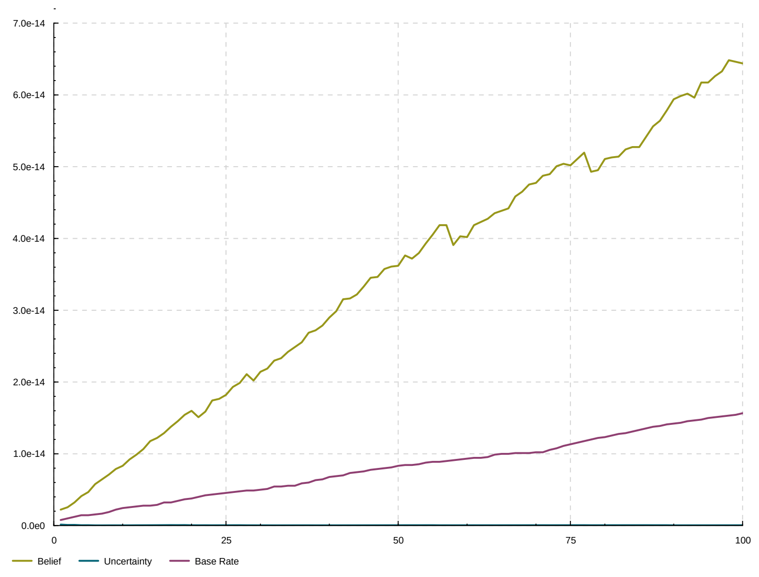
Final interval width:

```
b = 0x1.38p-48
u = 0x1.0c8p-44
a = 0x1.3ap-47
```

Jury-SpanishInquisition-binary64



Values

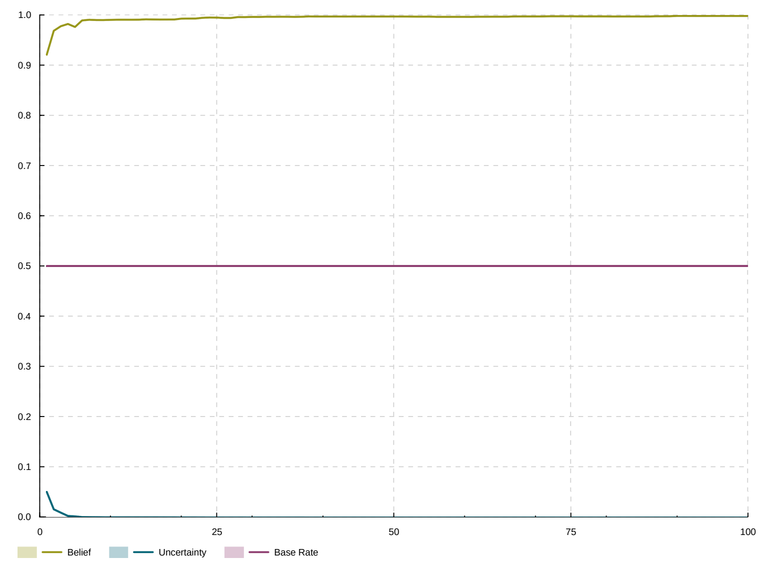


Interval Width

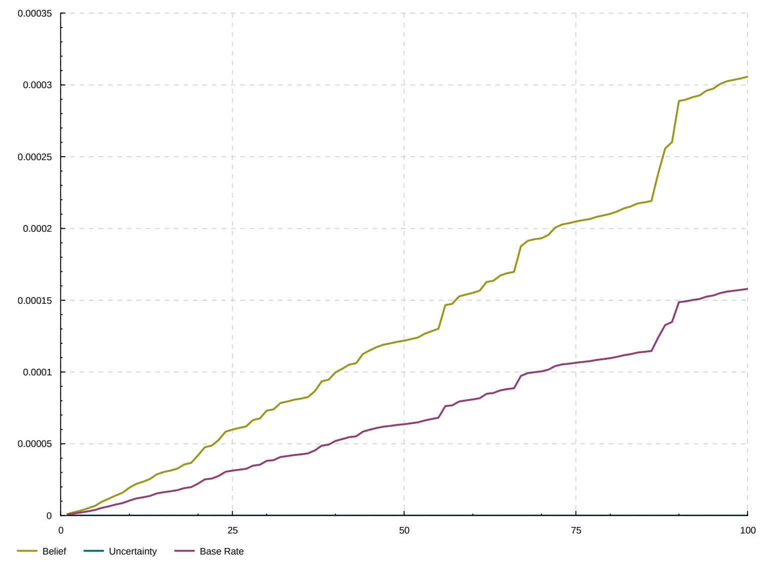
Final interval width:

```
b = 0x1.22p-44
u = 0x1.a2p-55
a = 0x1.1ap-46
```

Reputation-MostlyYes-binary32



Values

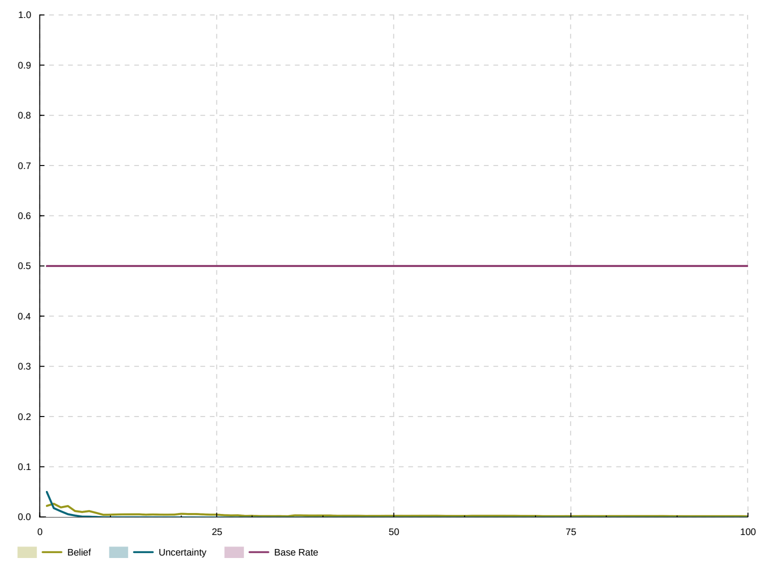


Interval Width

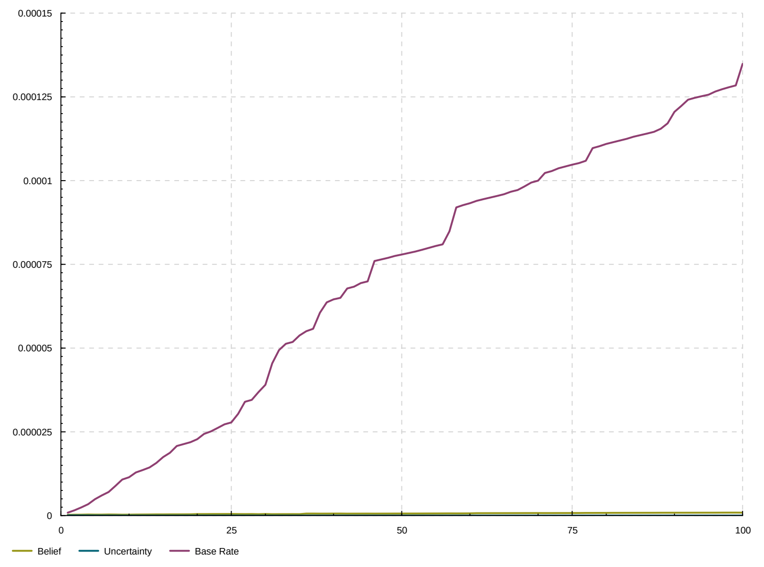
Final interval width:

```
b = 0x1.409p-12
u = 0x1.582p-29
a = 0x1.4b3p-13
```

Reputation-MostlyNo-binary32



Values

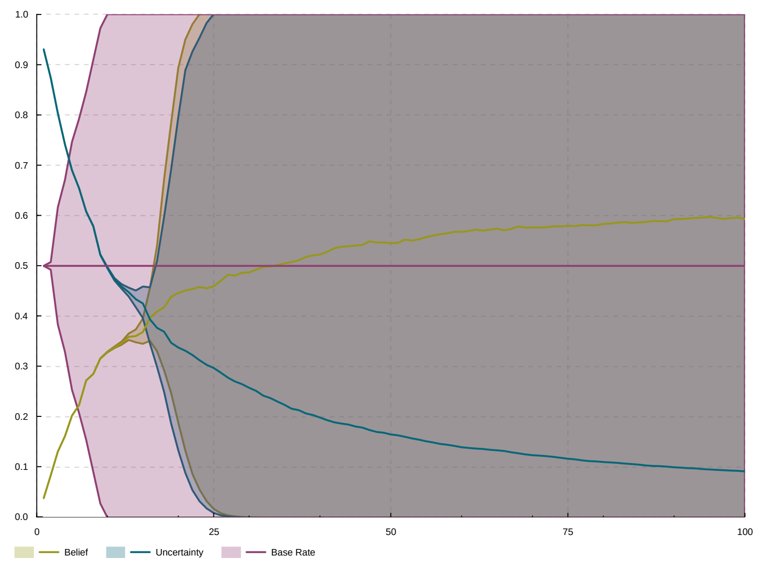


Interval Width

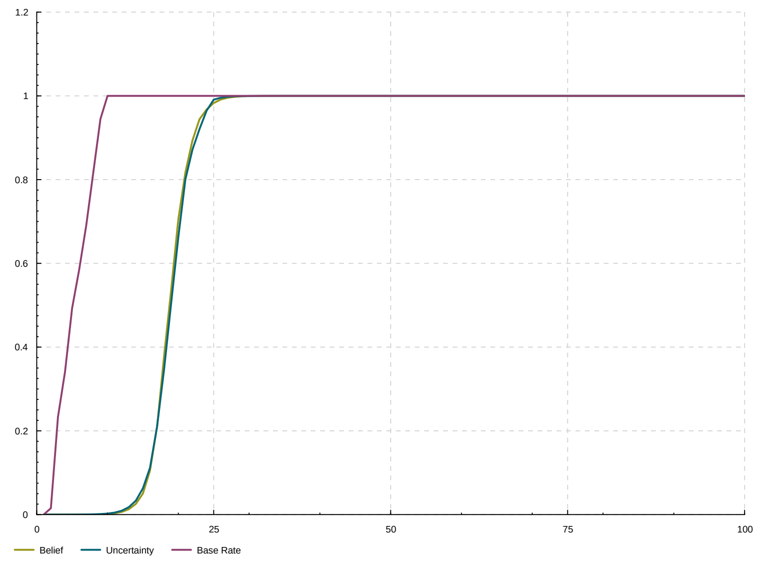
Final interval width:

```
b = 0x1.d4ep-21
u = 0x1.f98p-30
a = 0x1.1adp-13
```

Reputation-MostlyUnknown-binary32



Values

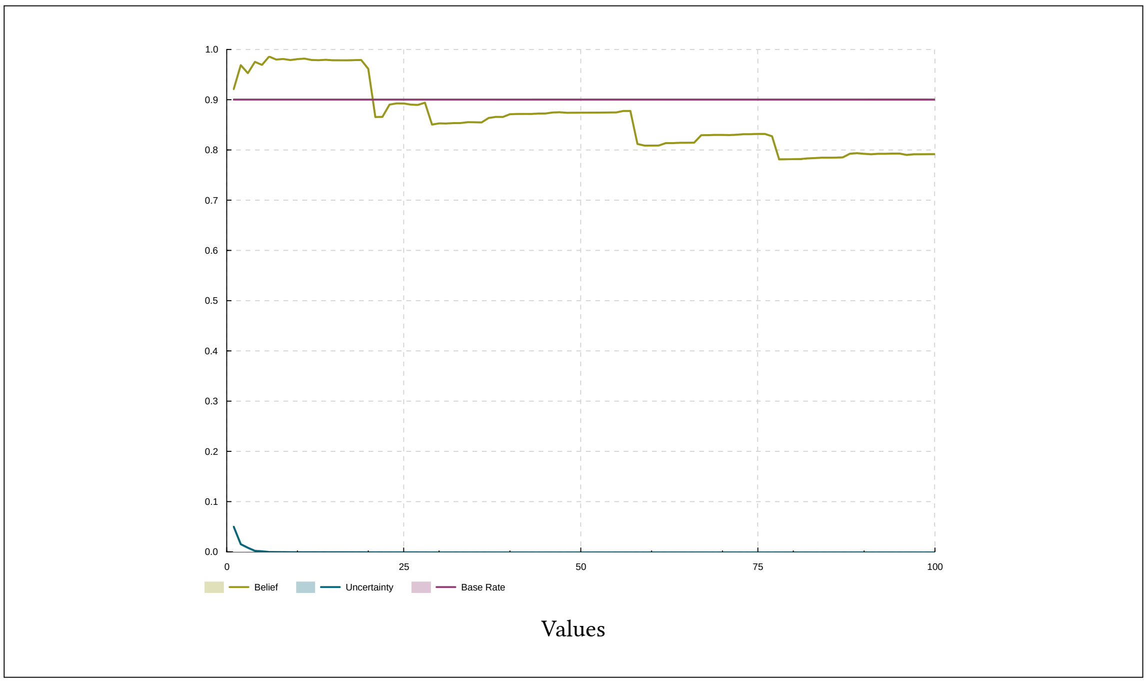


Interval Width

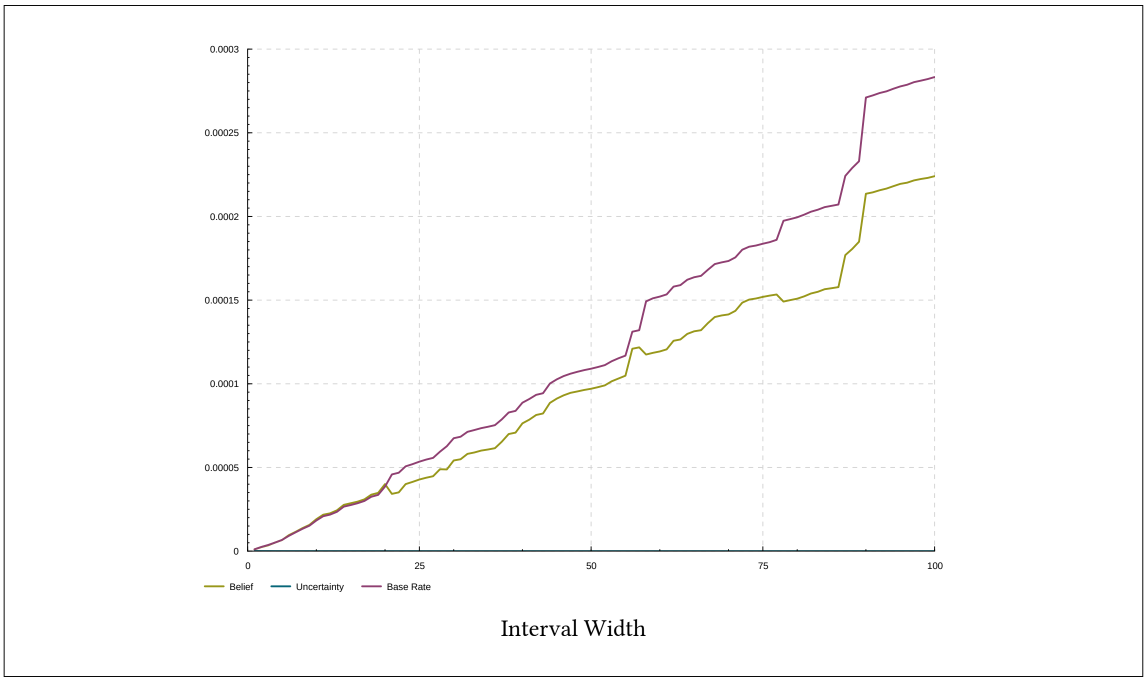
Final interval width:

```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.000004p0
```

Reputation-SpanishInquisition-binary32



Values

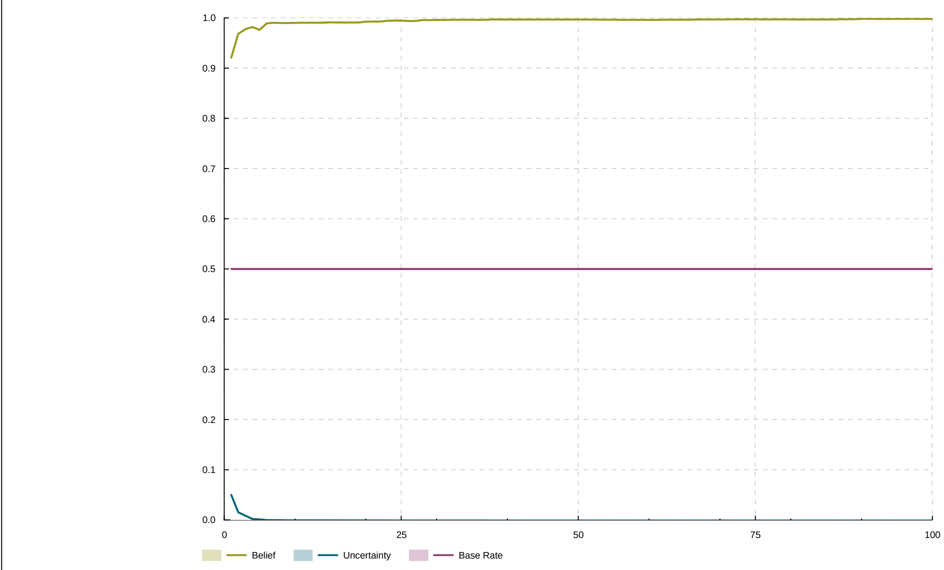


Interval Width

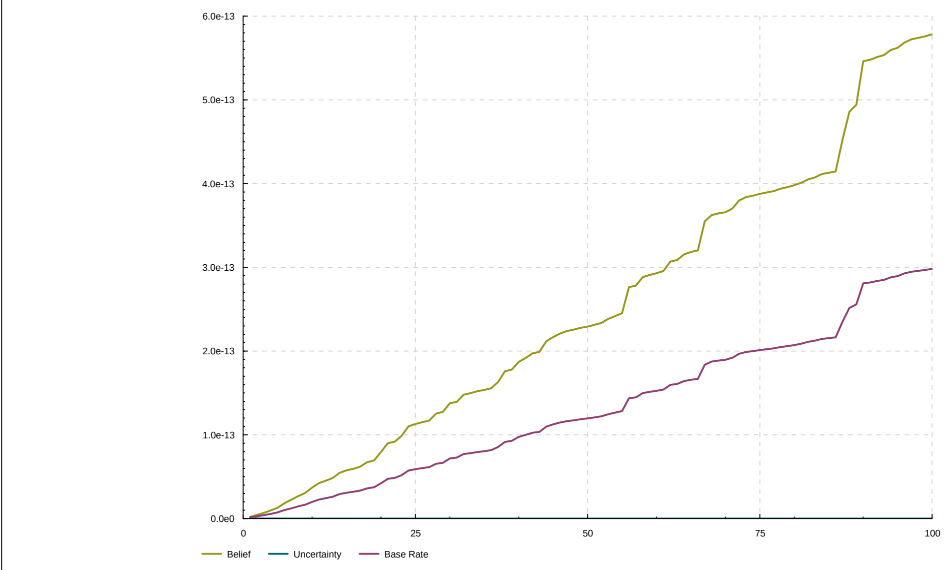
Final interval width:

```
b = 0x1.d5ep-13
u = 0x1.f4ap-30
a = 0x1.291p-12
```

Reputation-MostlyYes-binary64



Values

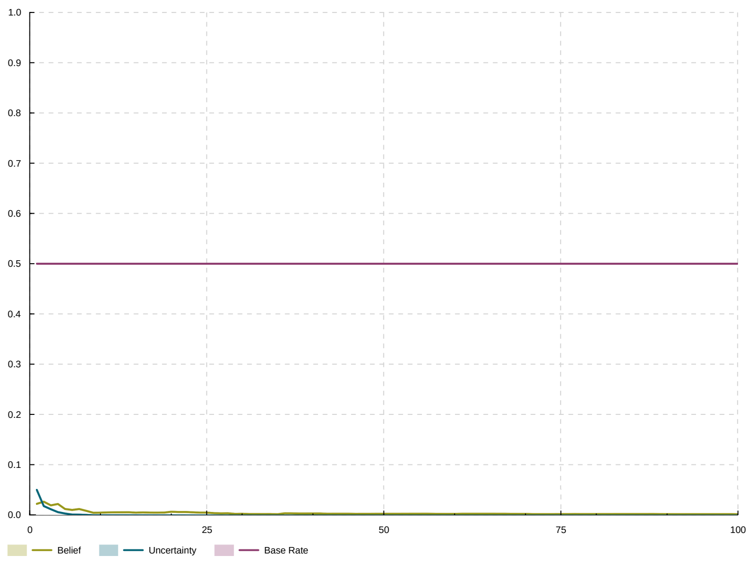


Interval Width

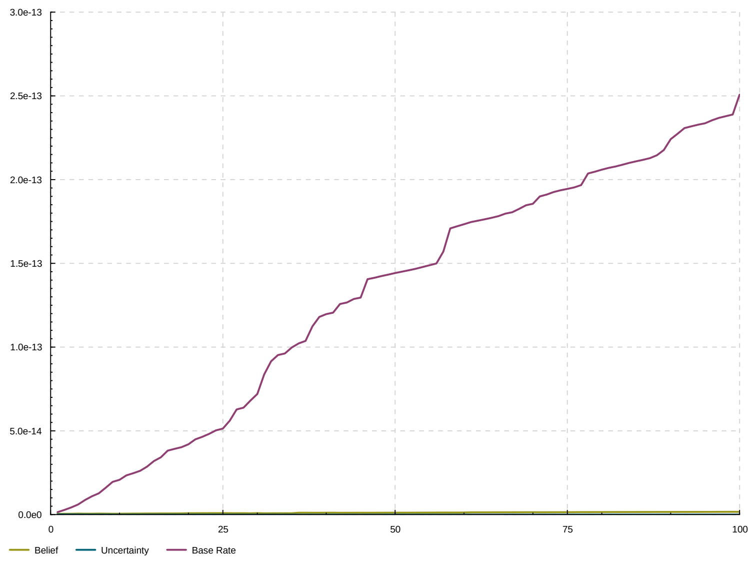
Final interval width:

```
b = 0x1.459p-41
u = 0x1.5d8p-58
a = 0x1.4fcp-42
```

Reputation-MostlyNo-binary64



Values

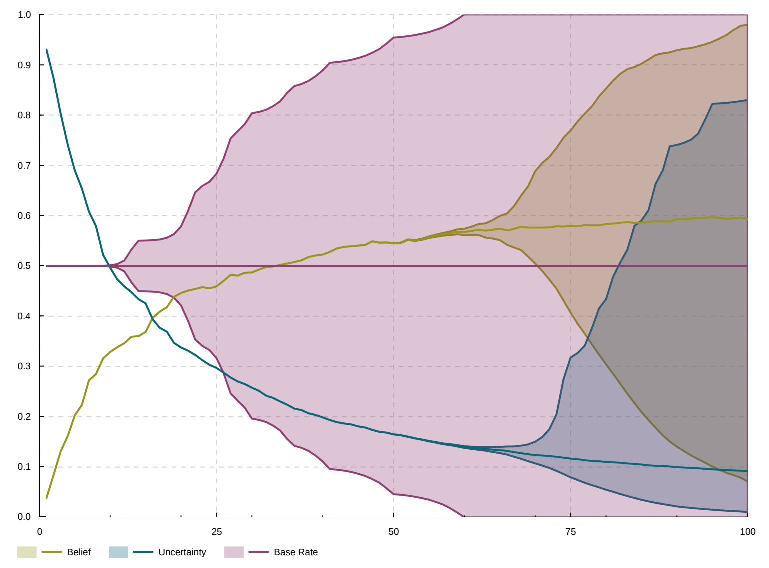


Interval Width

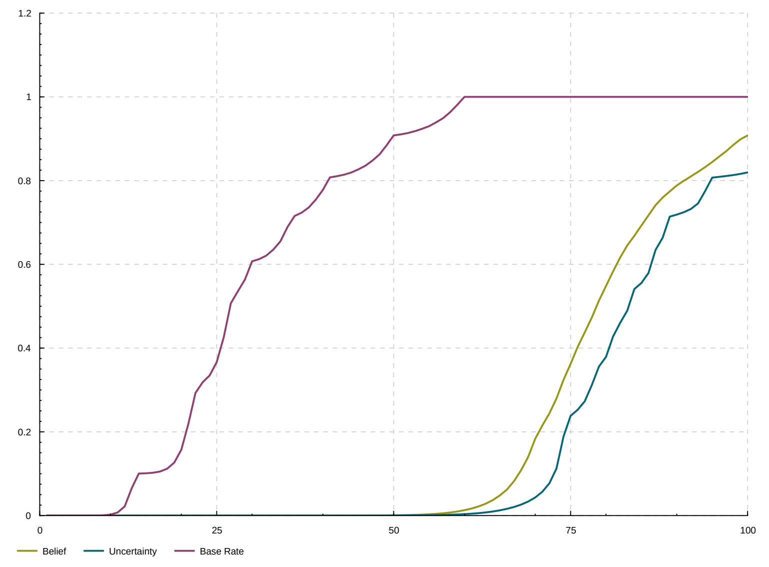
Final interval width:

```
b = 0x1.d4ap-50
u = 0x1.fap-59
a = 0x1.1a5p-42
```

Reputation-MostlyUnknown-binary64



Values

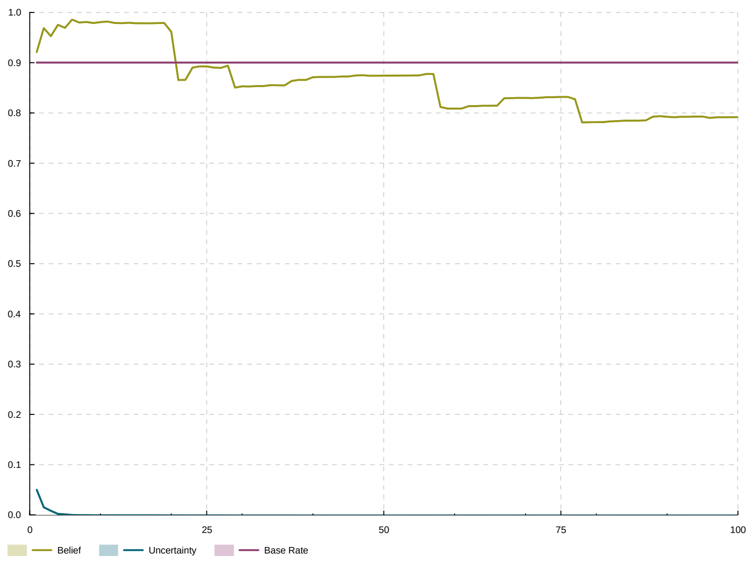


Interval Width

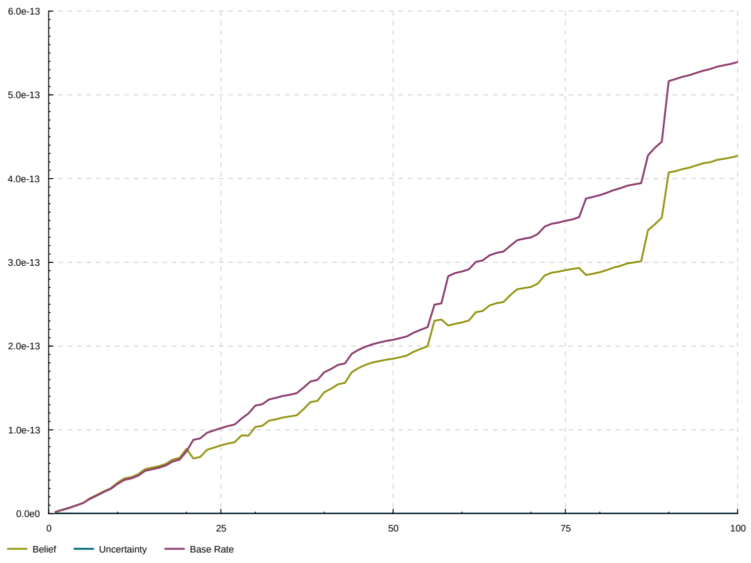
Final interval width:

```
b = 0x1.d0d02d4e56bd6p-1
u = 0x1.a39d1c525787ap-1
a = 0x1.0000000000002p0
```

Reputation-SpanishInquisition-binary64



Values

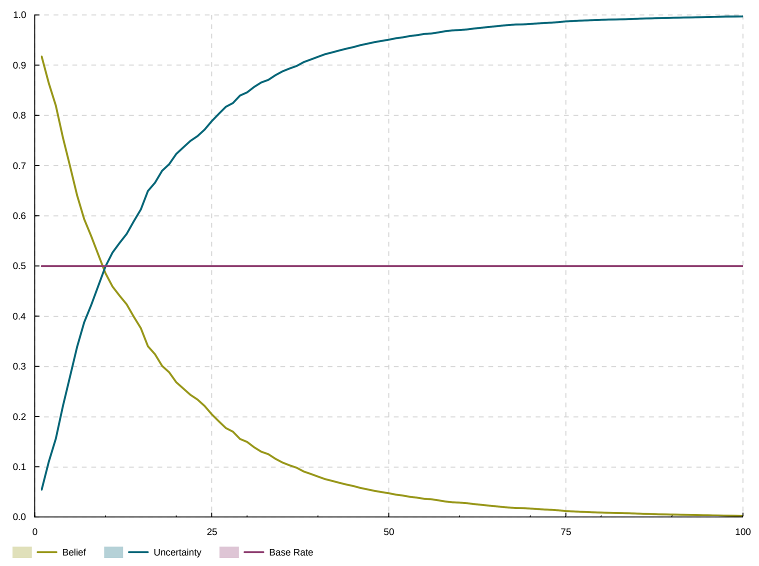


Interval Width

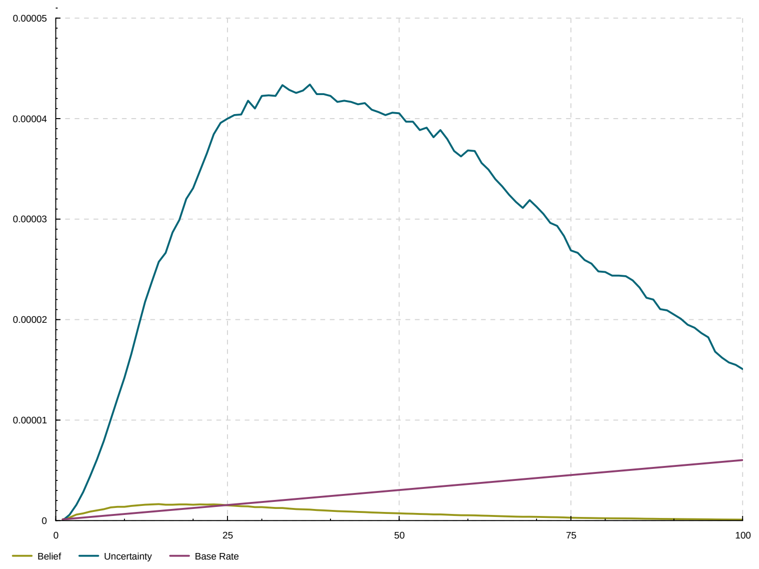
Final interval width:

```
b = 0x1.e0ep-42
u = 0x1.012p-58
a = 0x1.2f9p-41
```

Telephone-Local-MostlyYes-binary32



Values

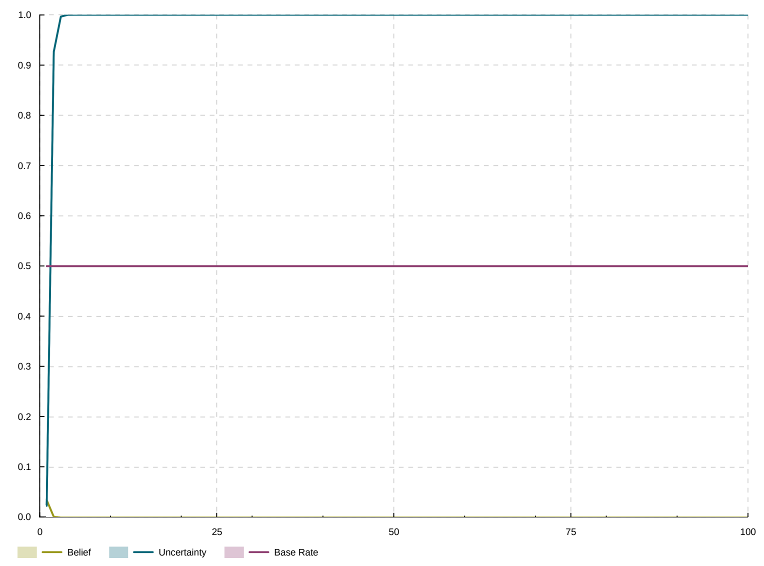


Interval Width

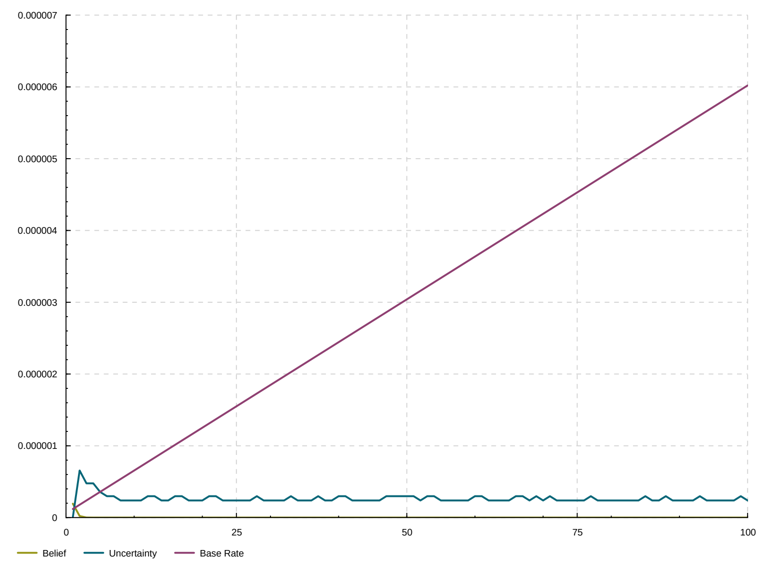
Final interval width:

```
b = 0x1.8cp-24
u = 0x1.fap-17
a = 0x1.94p-18
```

Telephone-Local-MostlyNo-binary32



Values

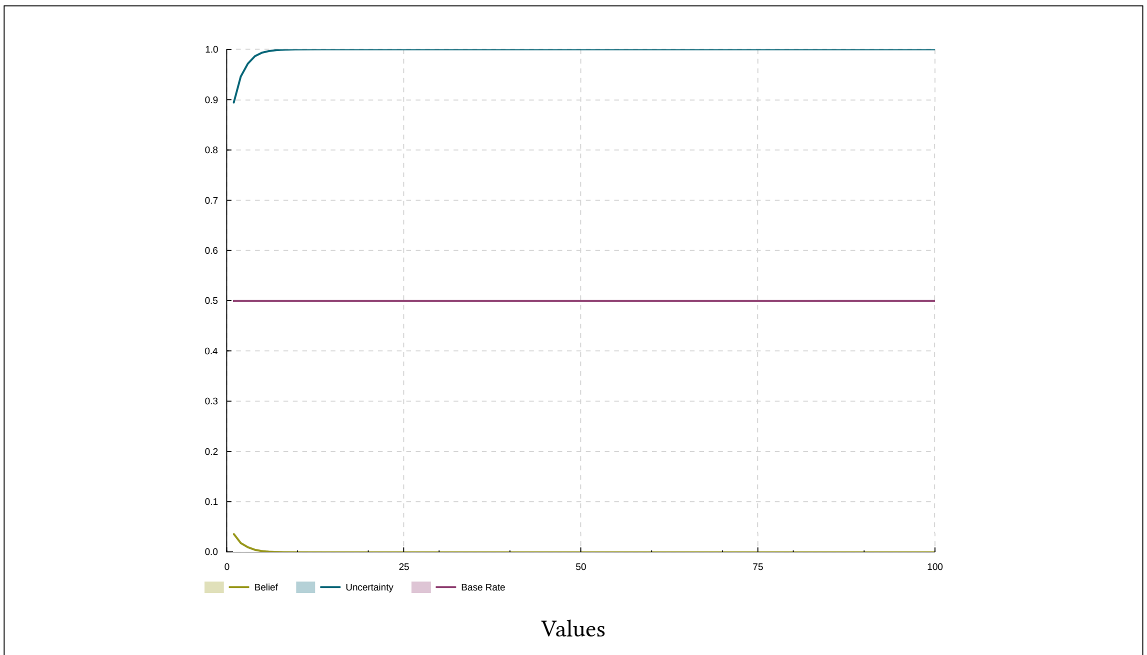


Interval Width

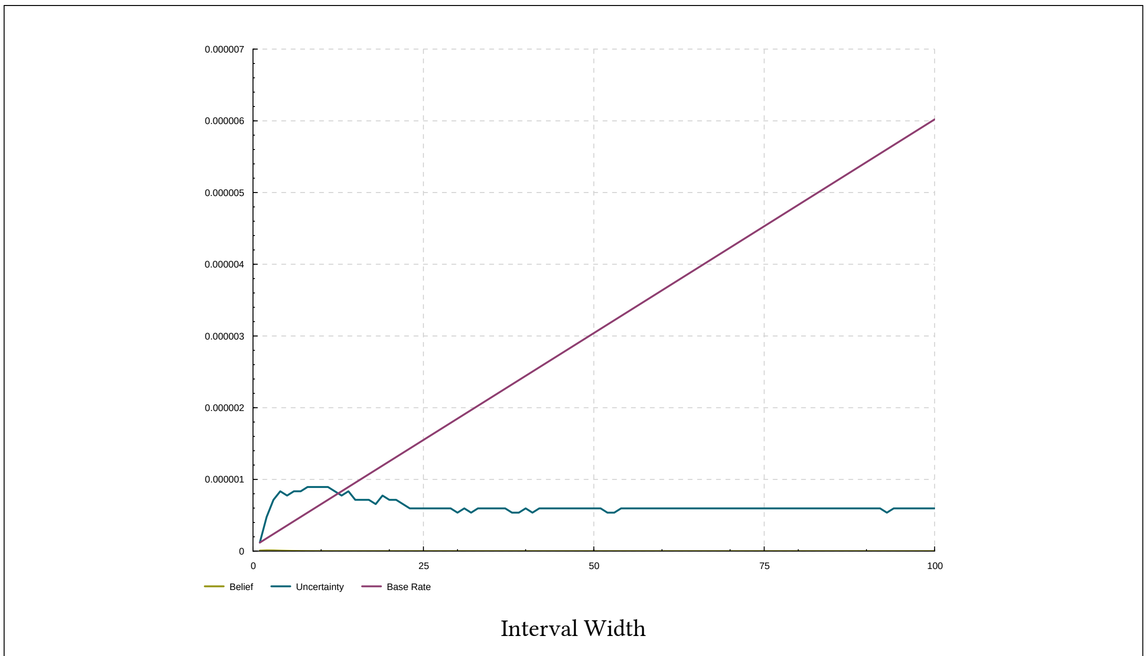
Final interval width:

```
b = 0x1.8p-148
u = 0x1p-22
a = 0x1.94p-18
```

Telephone-Local-MostlyUnknown-binary32



Values

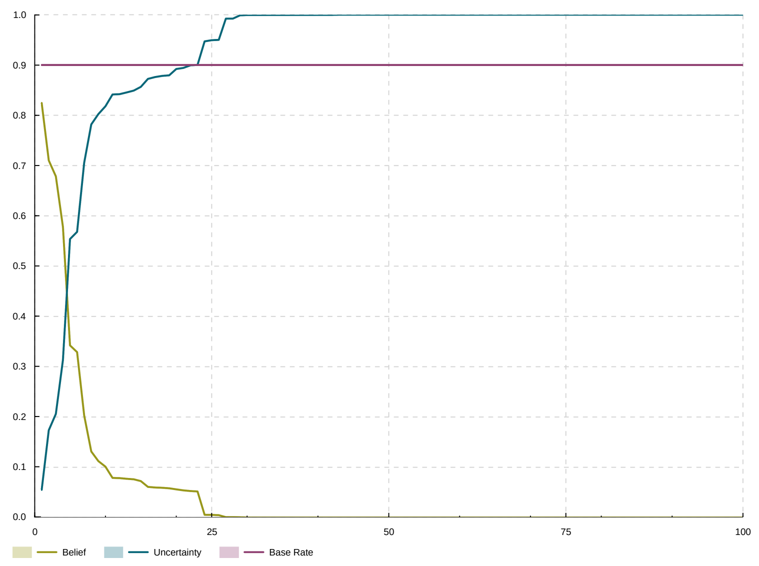


Interval Width

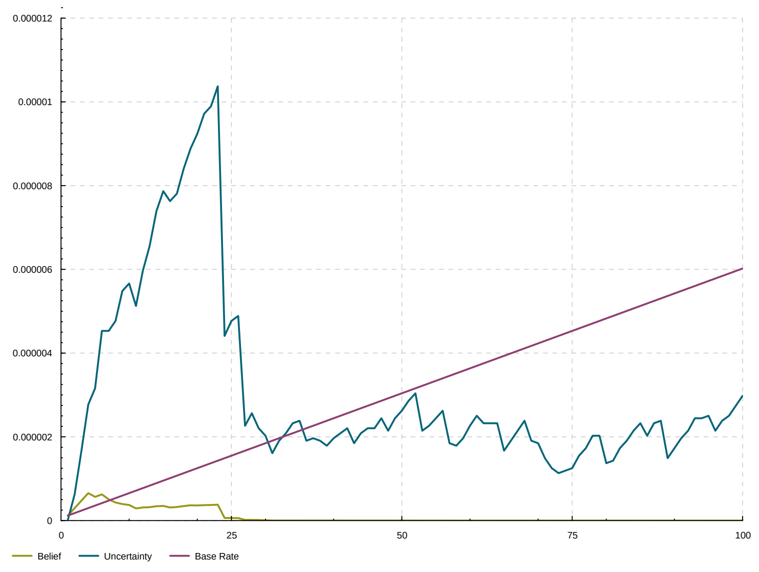
Final interval width:

```
b = 0x1.6b8p-115
u = 0x1.4p-21
a = 0x1.94p-18
```

Telephone-Local-SpanishInquisition-binary32



Values

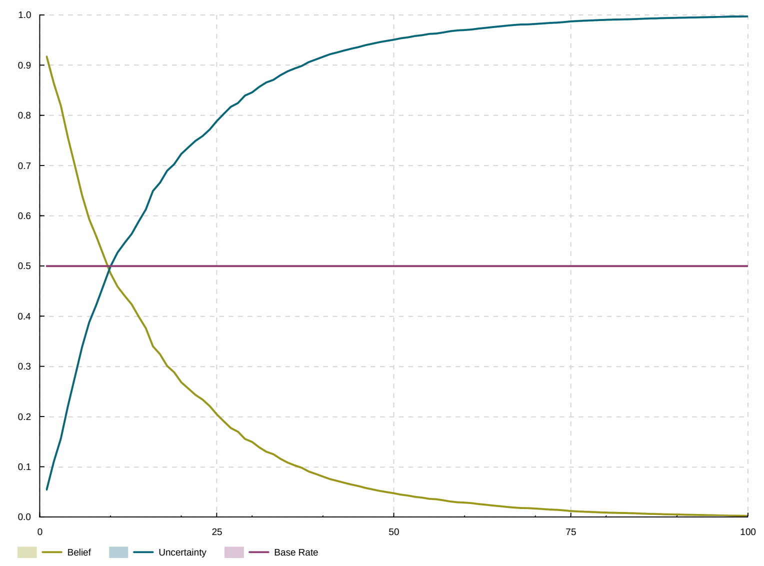


Interval Width

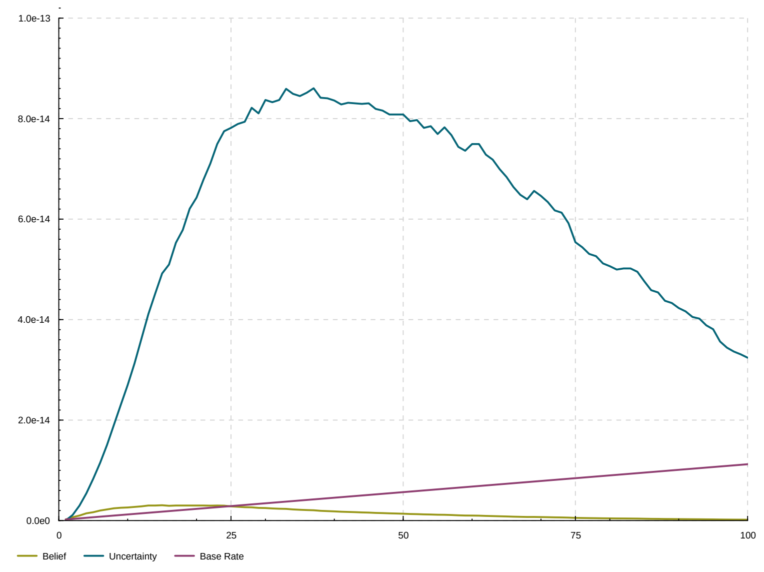
Final interval width:

```
b = 0x1.b5ap-48
u = 0x1.9p-19
a = 0x1.94p-18
```

Telephone-Local-MostlyYes-binary64



Values

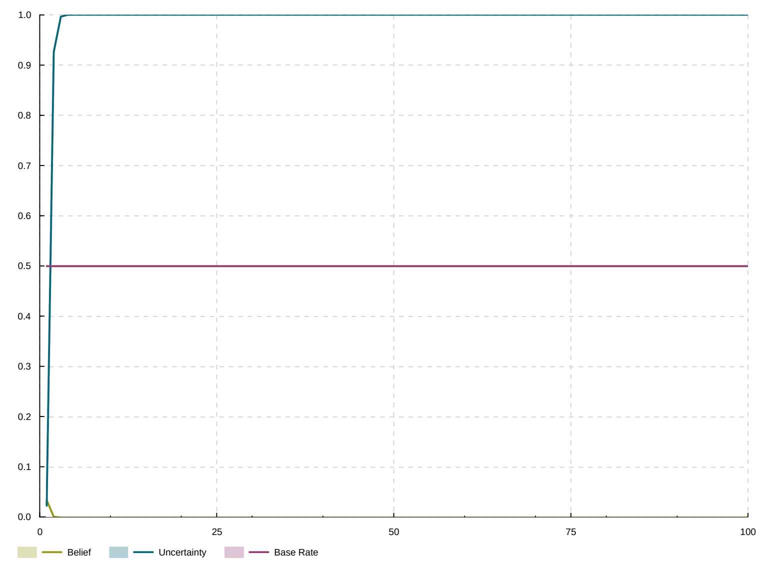


Interval Width

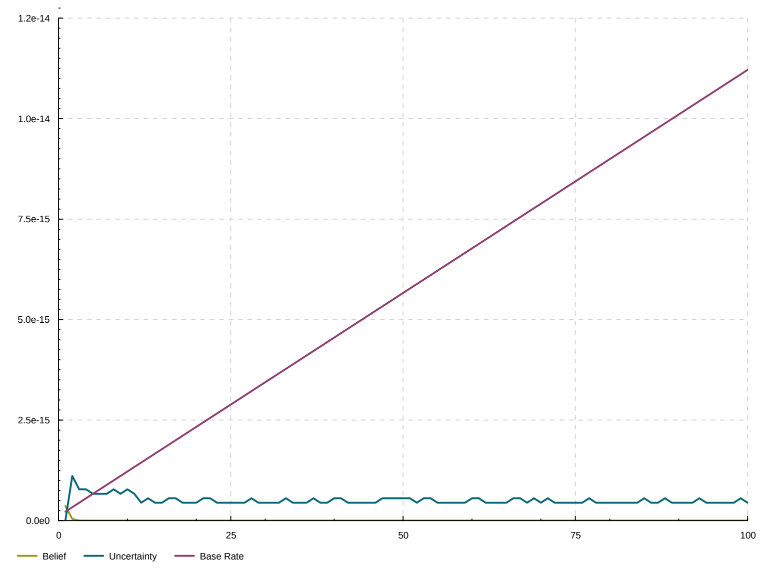
Final interval width:

```
b = 0x1.8fp-53
u = 0x1.24p-45
a = 0x1.94p-47
```

Telephone-Local-MostlyNo-binary64



Values

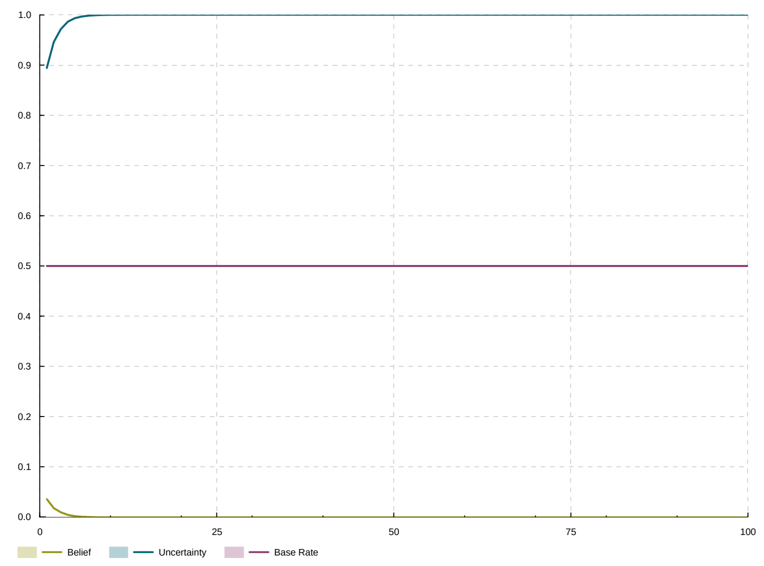


Interval Width

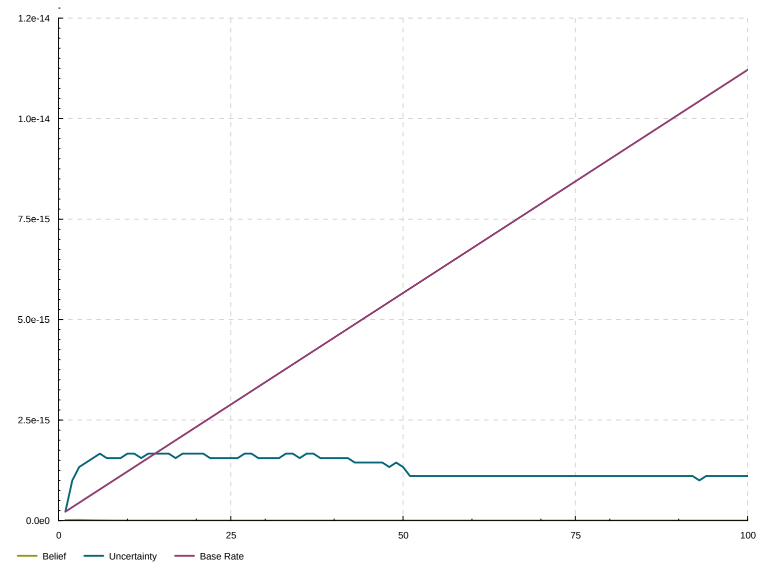
Final interval width:

```
b = 0x1.9a658p-533
u = 0x1p-51
a = 0x1.94p-47
```

Telephone-Local-MostlyUnknown-binary64



Values

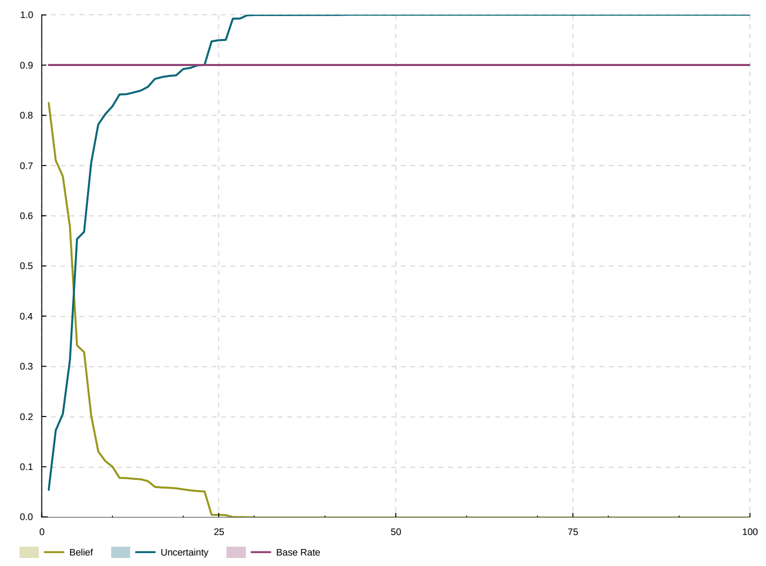


Interval Width

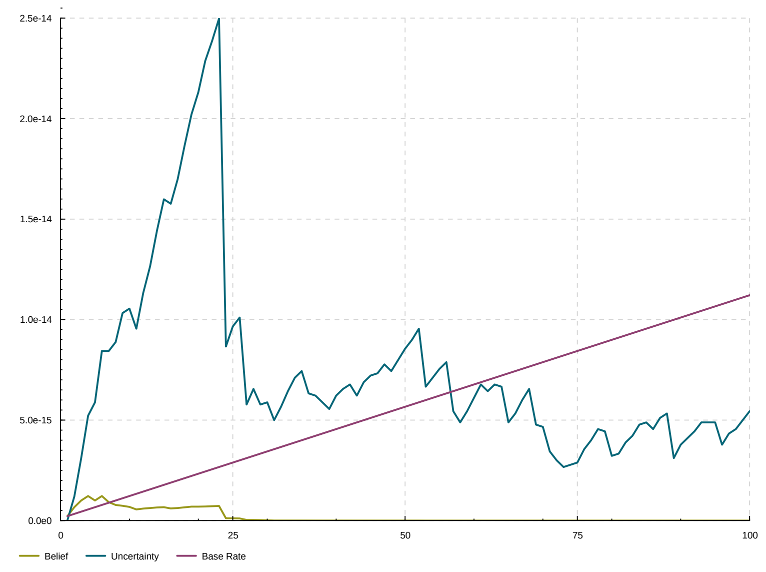
Final interval width:

```
b = 0x1.6dp-144
u = 0x1.4p-50
a = 0x1.94p-47
```

Telephone-Local-SpanishInquisition-binary64



Values

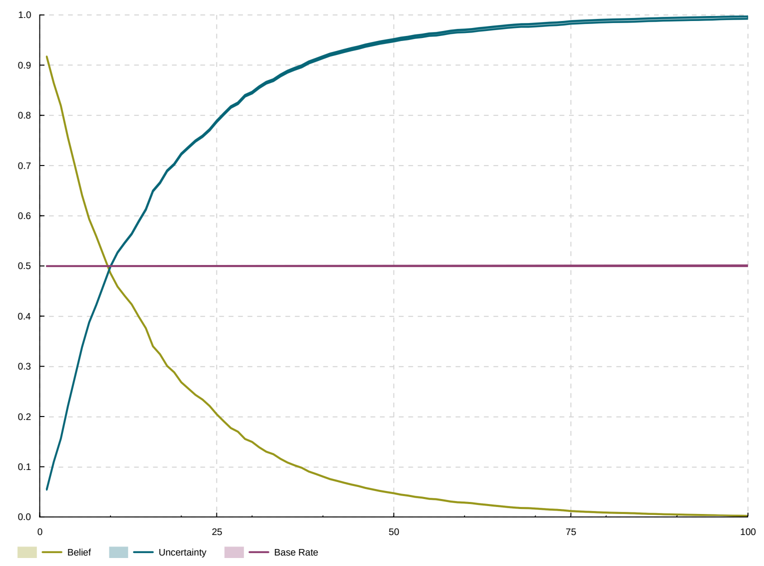


Interval Width

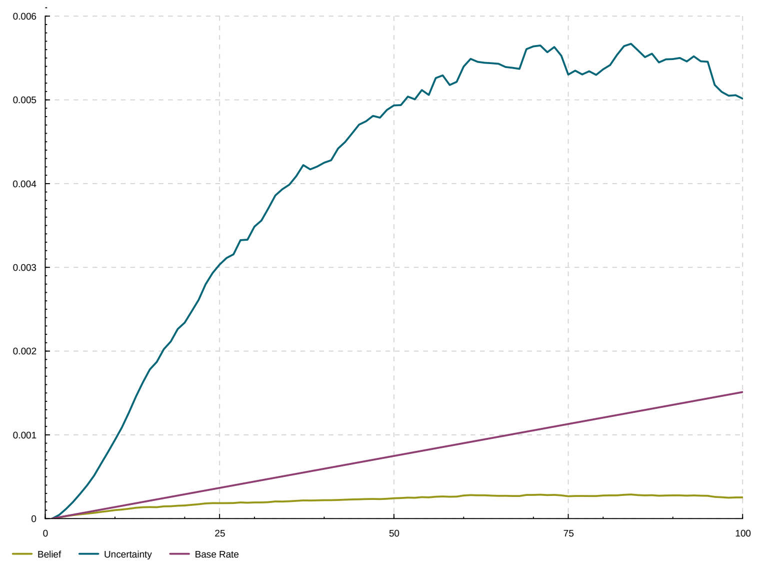
Final interval width:

```
b = 0x1.b56p-77
u = 0x1.88p-48
a = 0x1.94p-47
```

Telephone-UQ0.16-MostlyYes-binary32



Values

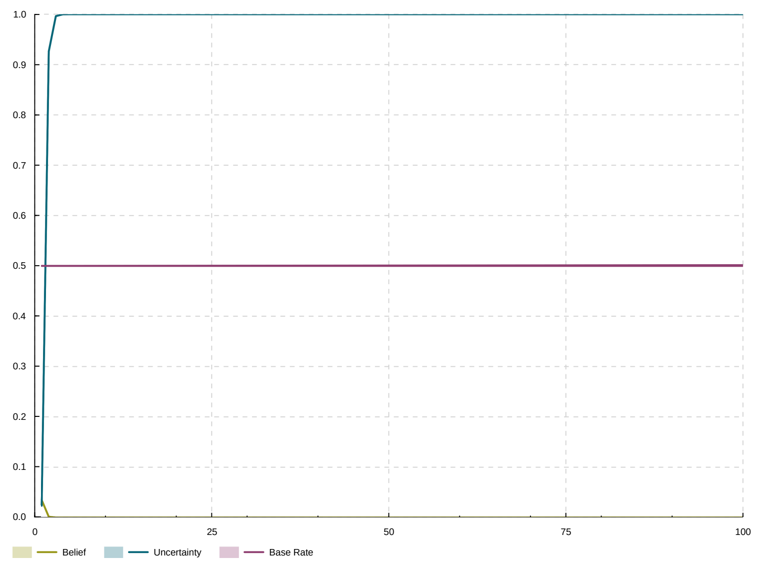


Interval Width

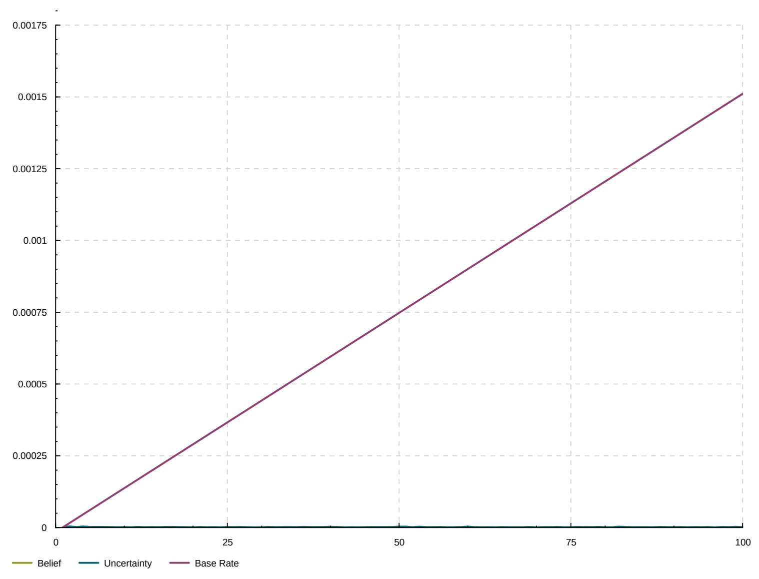
Final interval width:

```
b = 0x1.09591p-12
u = 0x1.48adp-8
a = 0x1.8c0cp-10
```

Telephone-UQ0.16-MostlyNo-binary32



Values

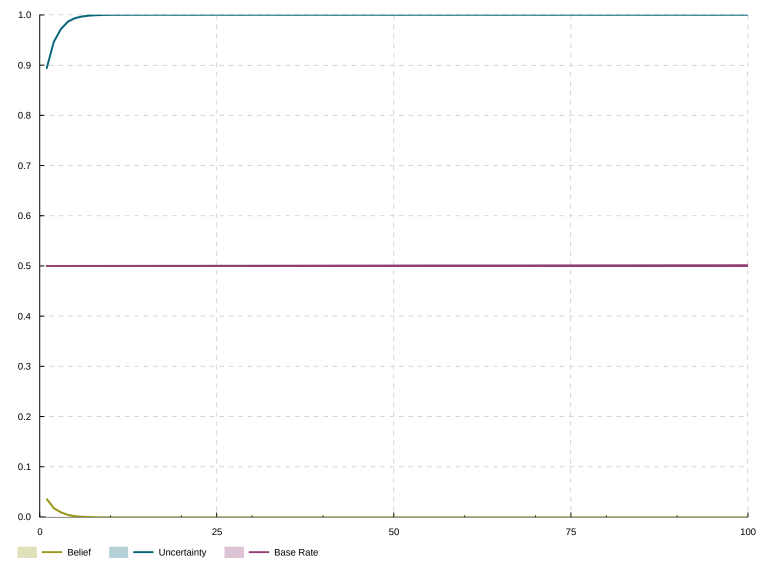


Interval Width

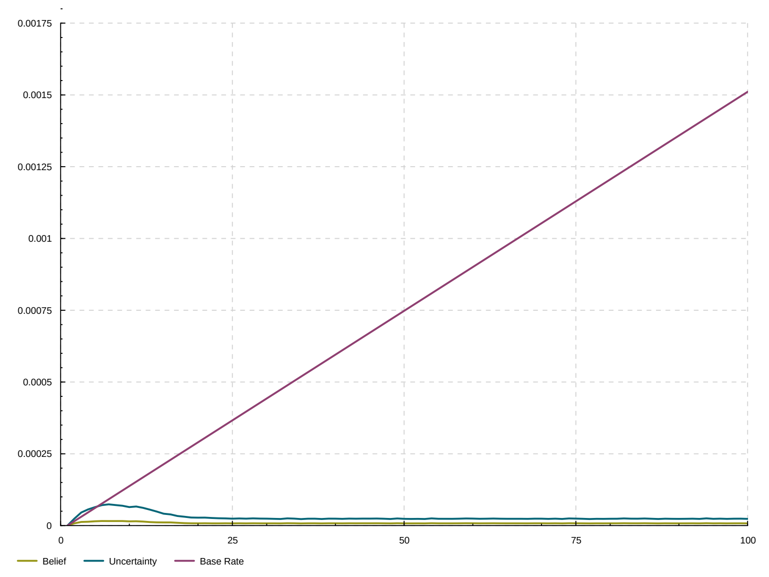
Final interval width:

```
b = 0x1.83677cp-21
u = 0x1.cp-20
a = 0x1.8c0cp-10
```

Telephone-UQ0.16-MostlyUnknown-binary32



Values

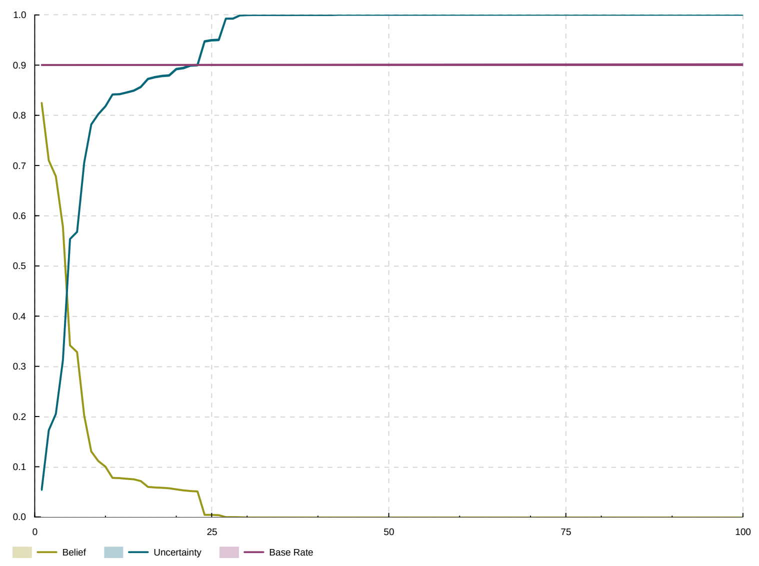


Interval Width

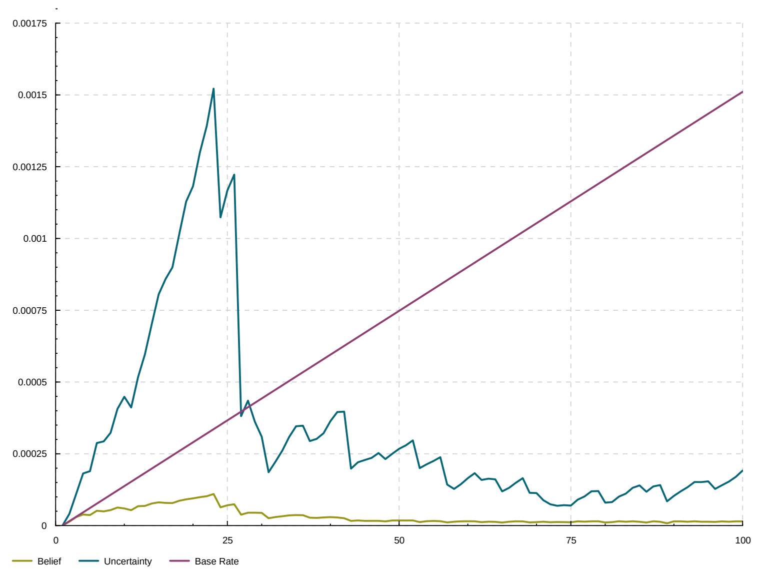
Final interval width:

```
b = 0x1.01d2bcp-17
u = 0x1.88p-16
a = 0x1.8c0cp-10
```

Telephone-UQ0.16-SpanishInquisition-binary32



Values

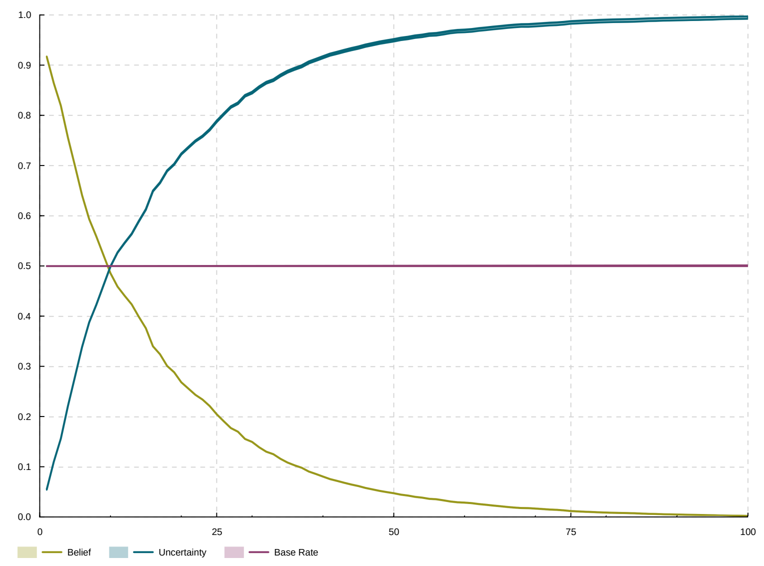


Interval Width

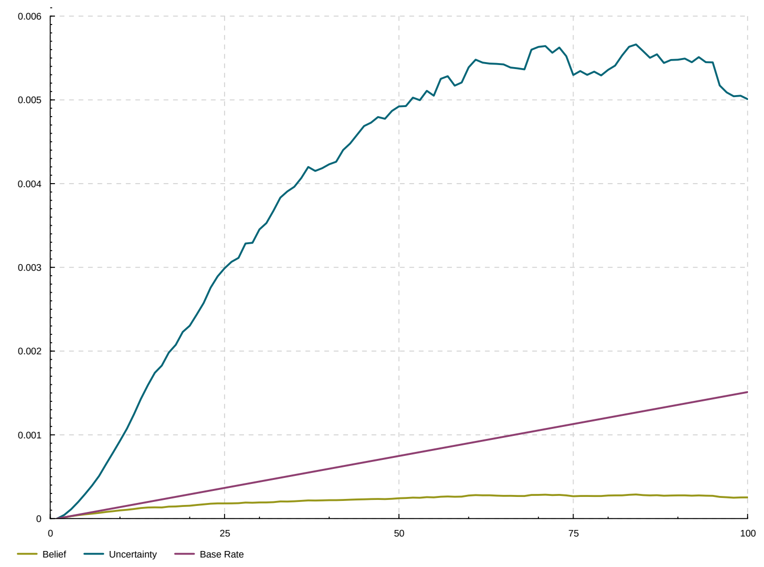
Final interval width:

```
b = 0x1.ee54dcp-17
u = 0x1.91ep-13
a = 0x1.8c0cp-10
```

Telephone-UQ0.16-MostlyYes-binary64



Values

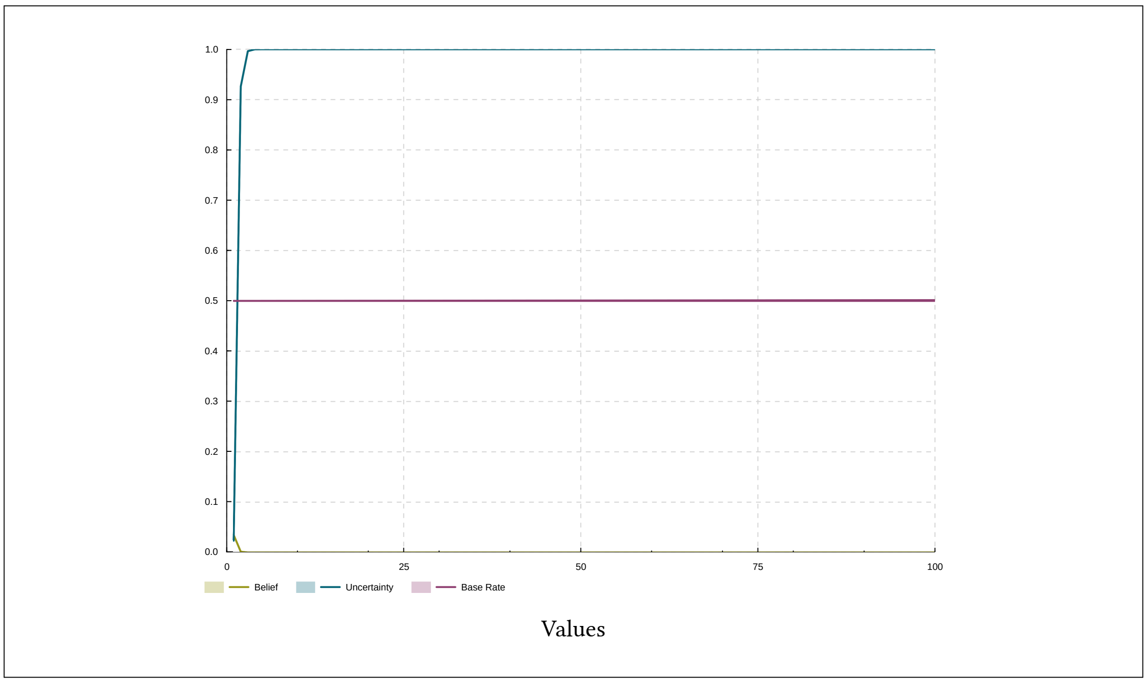


Interval Width

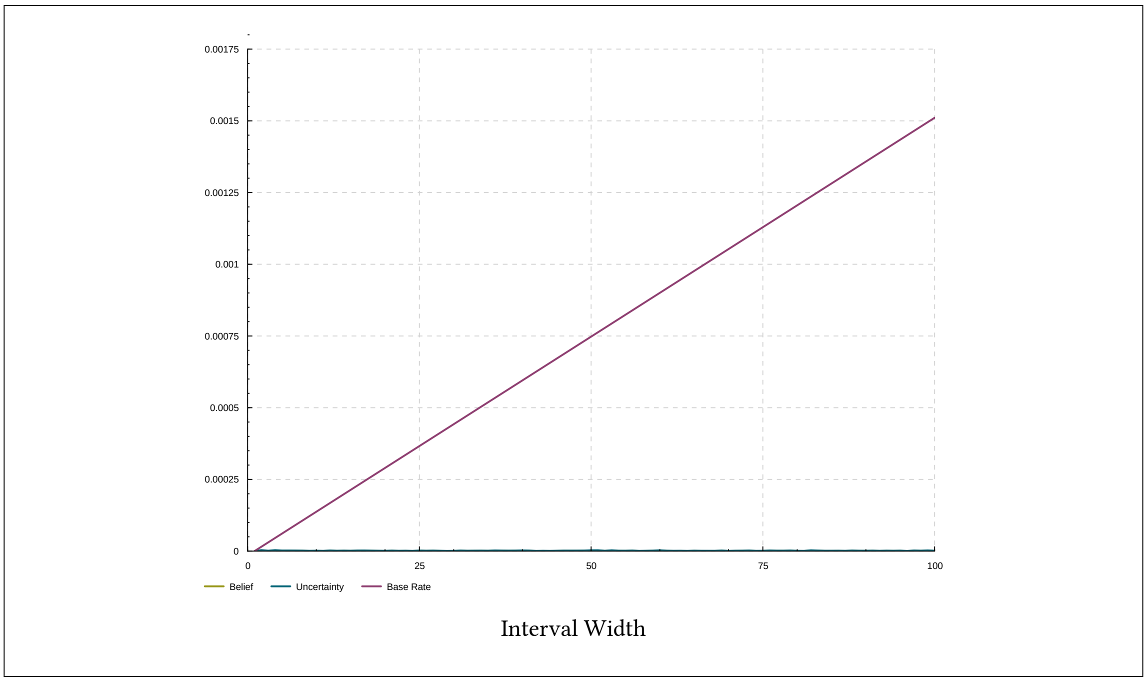
Final interval width:

```
b = 0x1.0958b3a189618p-12
u = 0x1.483fd6ea833p-8
a = 0x1.8c000000006p-10
```

Telephone-UQ0.16-MostlyNo-binary64



Values

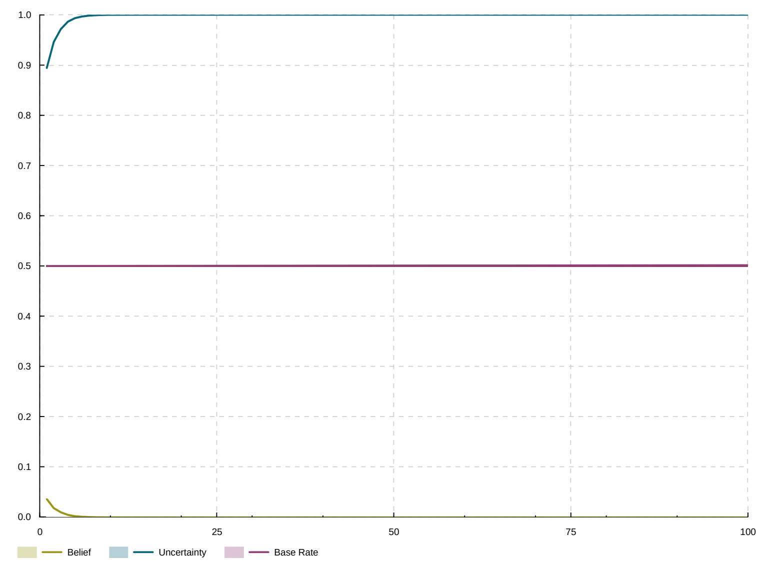


Interval Width

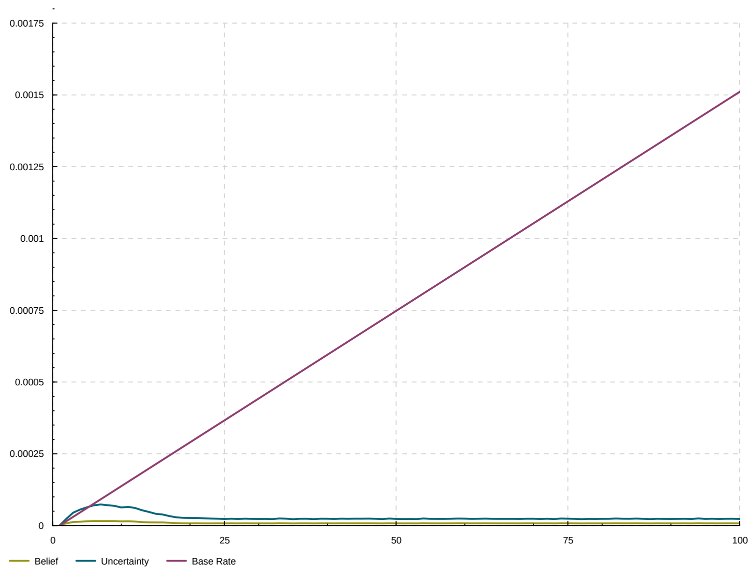
Final interval width:

```
b = 0x1.83671a91dc015p-21
u = 0x1.83671a938p-20
a = 0x1.8c000000006p-10
```

Telephone-UQ0.16-MostlyUnknown-binary64



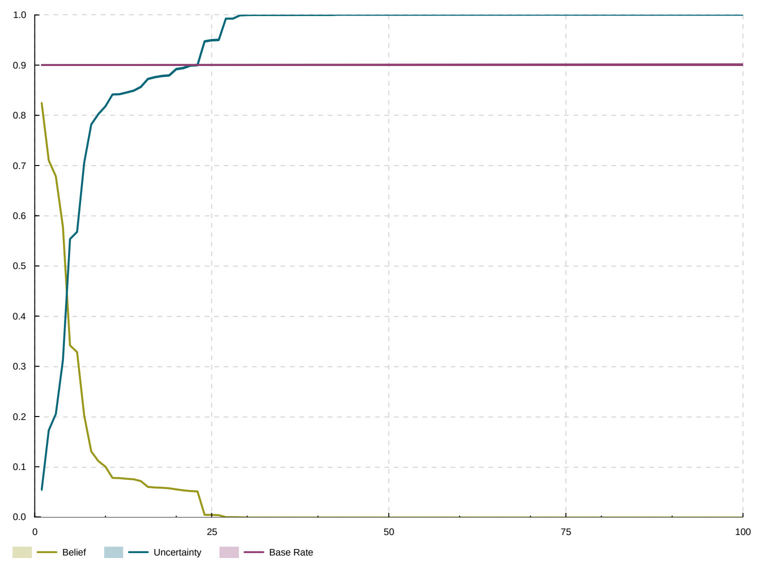
Values



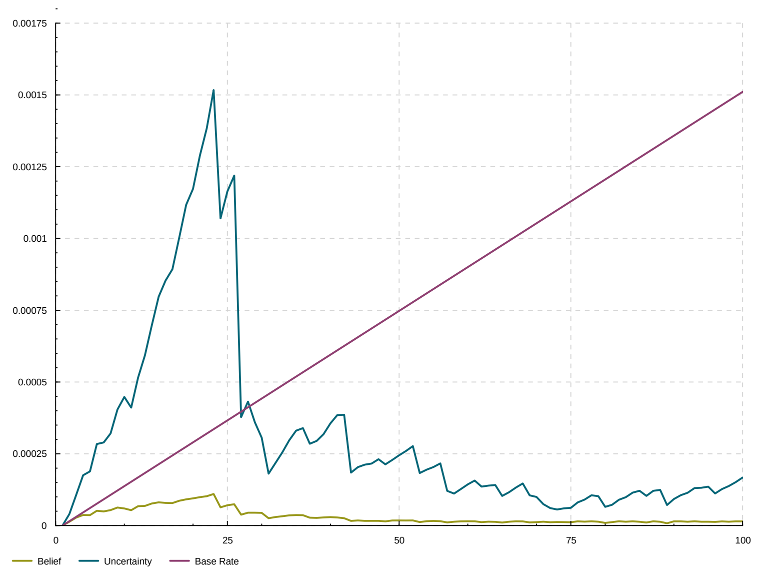
Interval Width

Final interval width:
```
b = 0x1.01d2b39d816d4p-17
u = 0x1.82bc0d6c7p-16
a = 0x1.8c000000006p-10
```

Telephone-UQ0.16-SpanishInquisition-binary64



Values

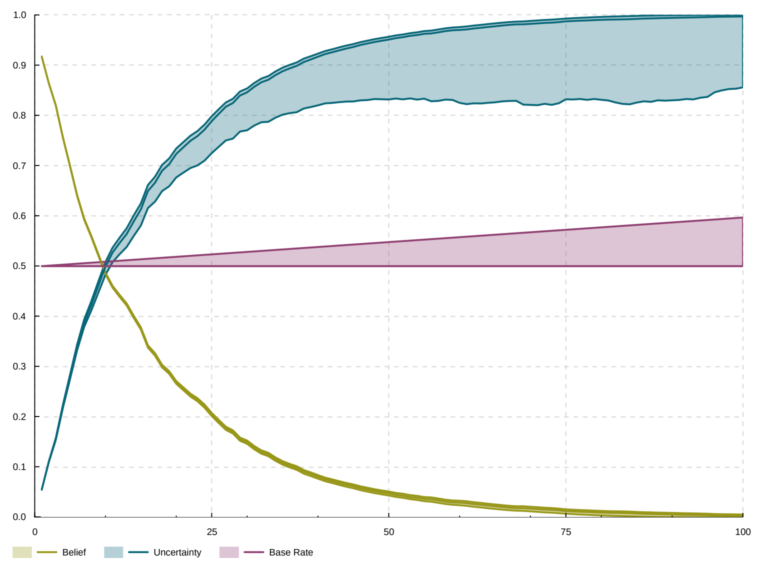


Interval Width

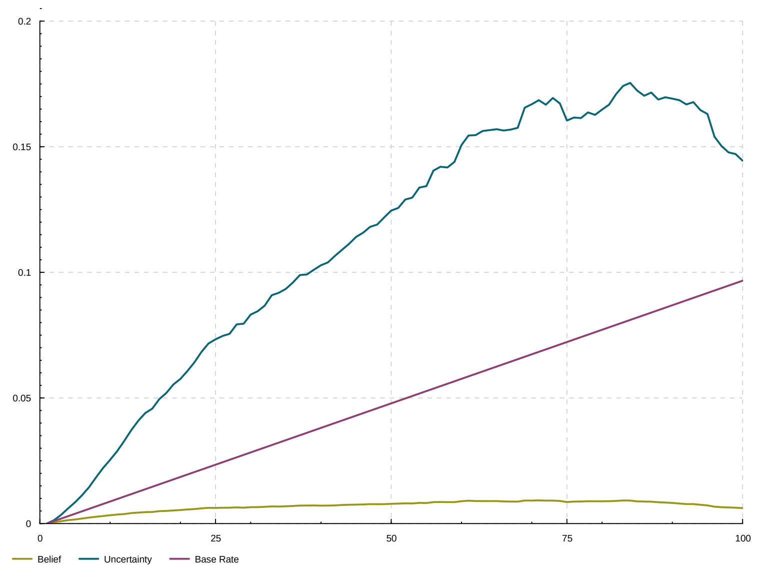
Final interval width:

```
b = 0x1.ee54d2c7d7046p-17
u = 0x1.5ece25c5ecp-13
a = 0x1.8c000000006p-10
```

Telephone-Packed-MostlyYes-binary32



Values

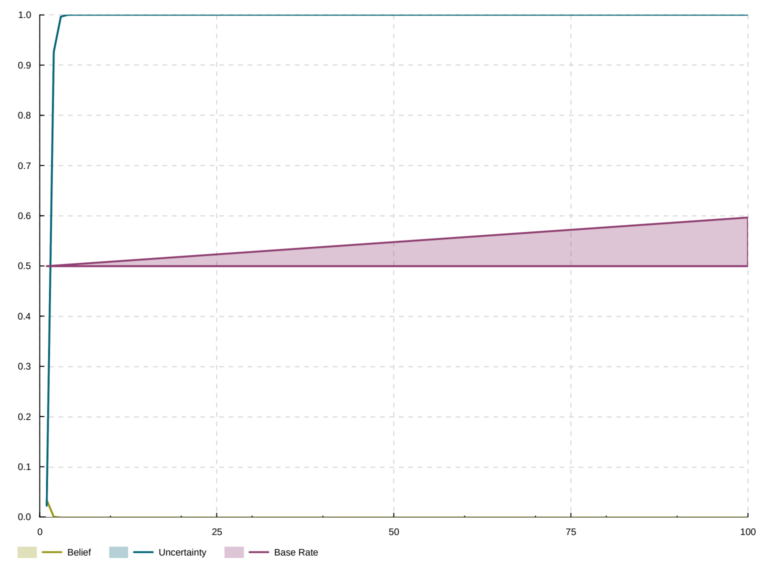


Interval Width

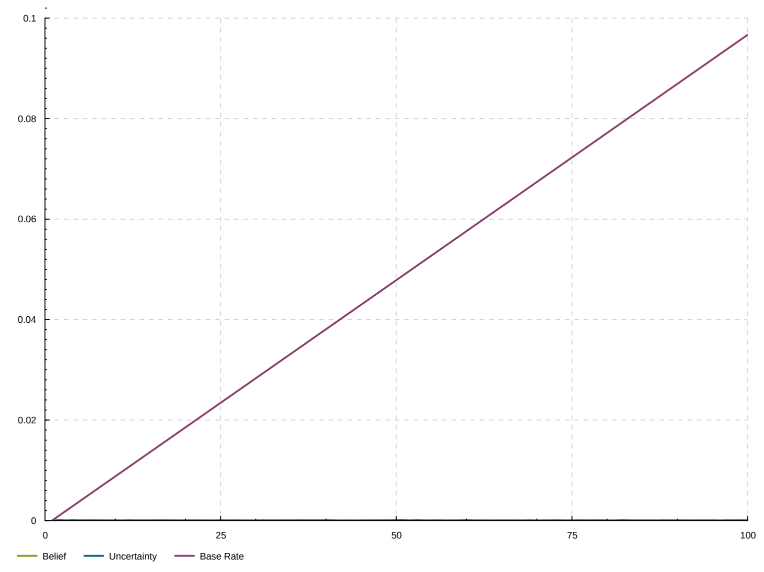
Final interval width:

```
b = 0x1.9389ccp-8
u = 0x1.27d28p-3
a = 0x1.8c003p-4
```

Telephone-Packed-MostlyNo-binary32



Values

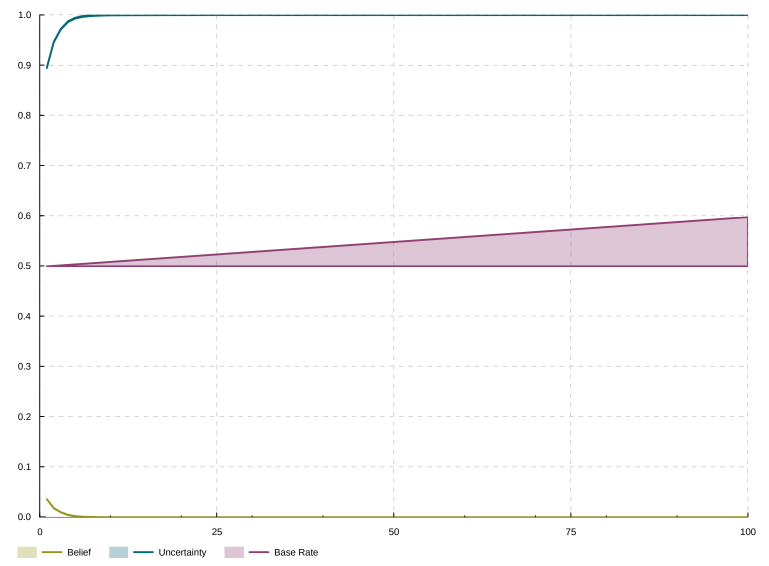


Interval Width

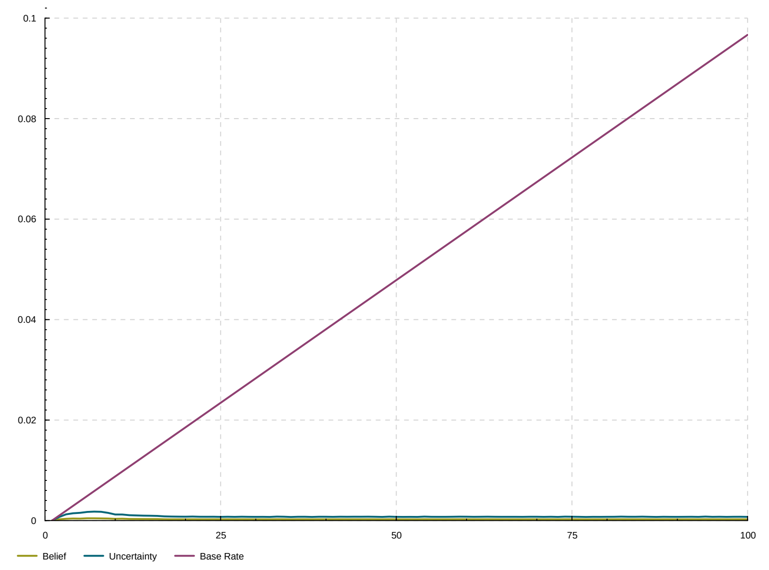
Final interval width:

```
b = 0x1.83677cp-16
u = 0x1.85p-15
a = 0x1.8c003p-4
```

Telephone-Packed-MostlyUnknown-binary32



Values

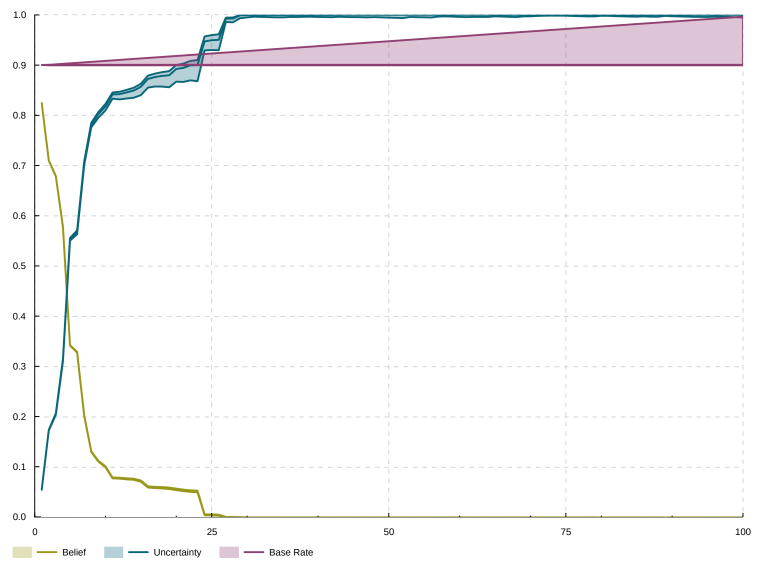


Interval Width

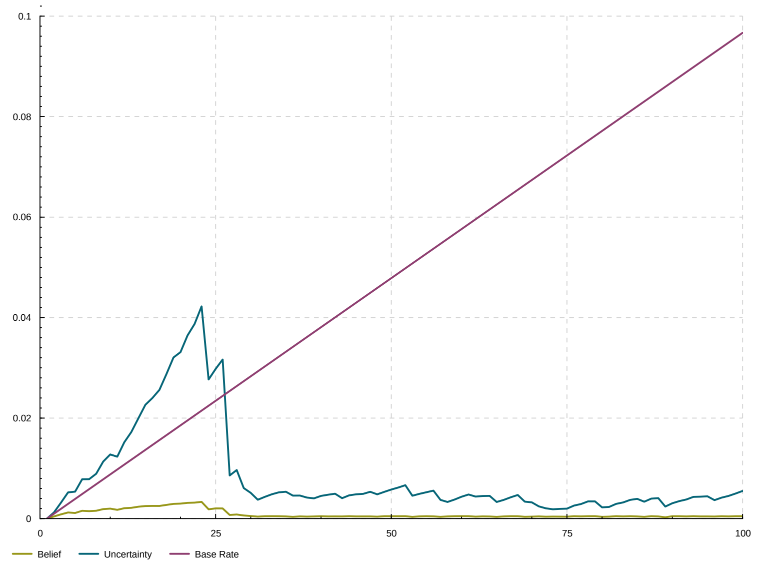
Final interval width:

```
b = 0x1.01d2bcp-12
u = 0x1.82e8p-11
a = 0x1.8c003p-4
```

Telephone-Packed-SpanishInquisition-binary32



Values

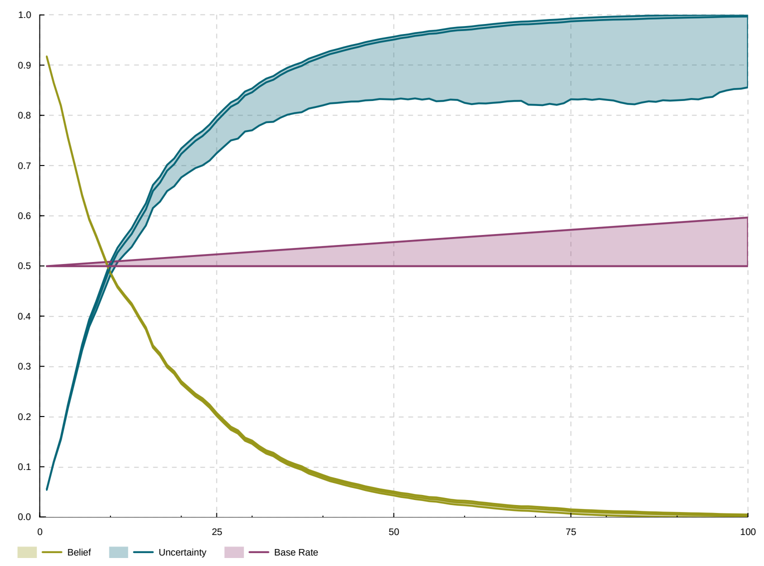


Interval Width

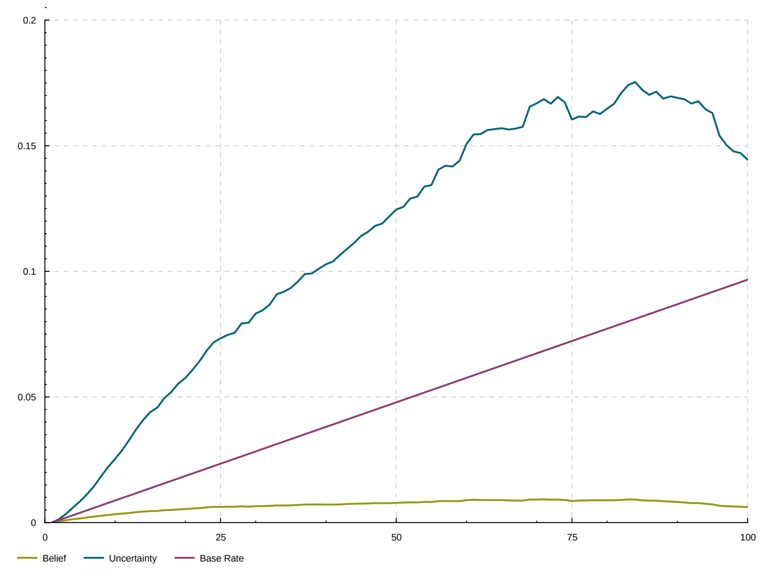
Final interval width:

```
b = 0x1.ee54dcp-12
u = 0x1.686p-8
a = 0x1.8c003p-4
```

Telephone-Packed-MostlyYes-binary64



Values

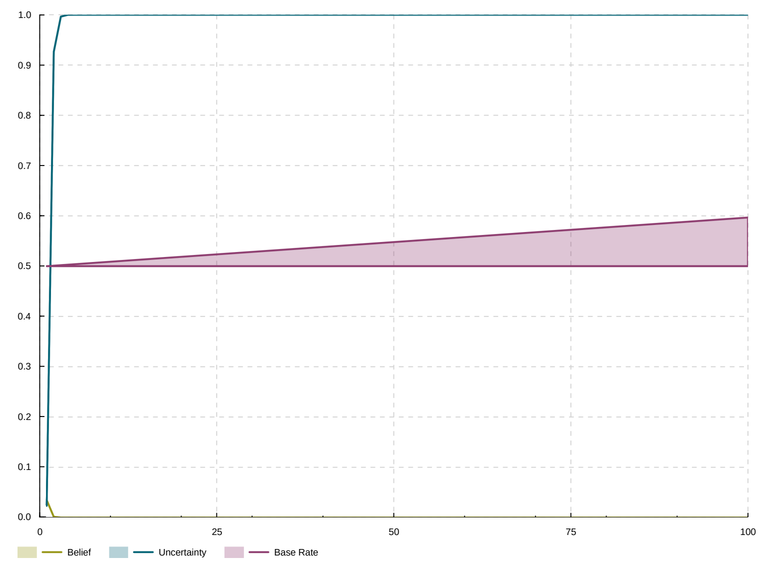


Interval Width

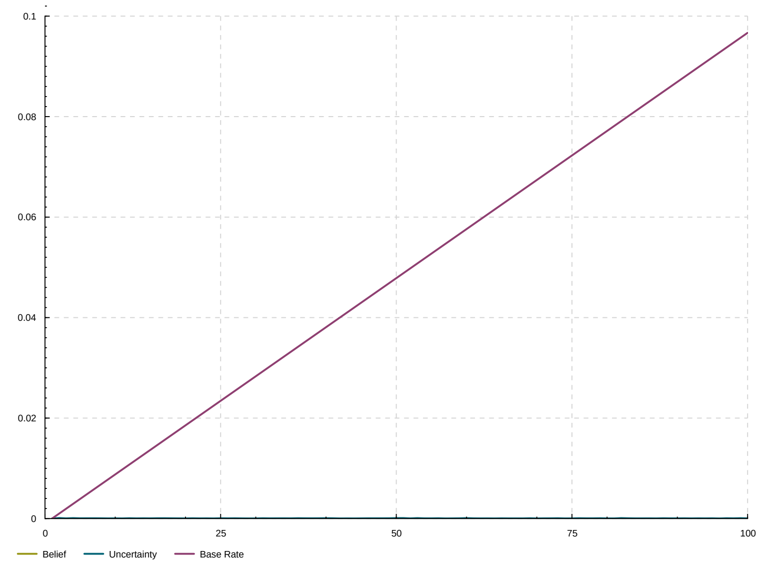
Final interval width:

```
b = 0x1.9389c2fcad615p-8
u = 0x1.27d2426208c5cp-3
a = 0x1.8c00000000018p-4
```

Telephone-Packed-MostlyNo-binary64



Values

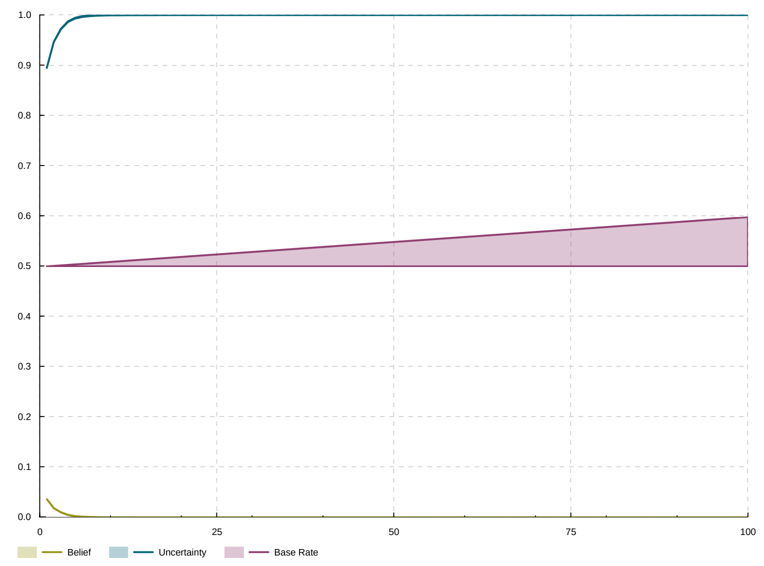


Interval Width

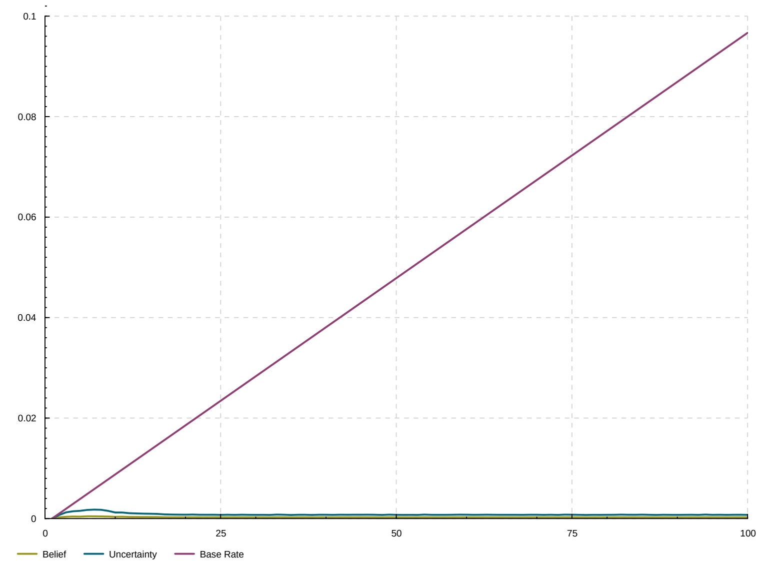
Final interval width:

```
b = 0x1.83671a91dc015p-16
u = 0x1.83671a91ecp-15
a = 0x1.8c00000000018p-4
```

Telephone-Packed-MostlyUnknown-binary64



Values

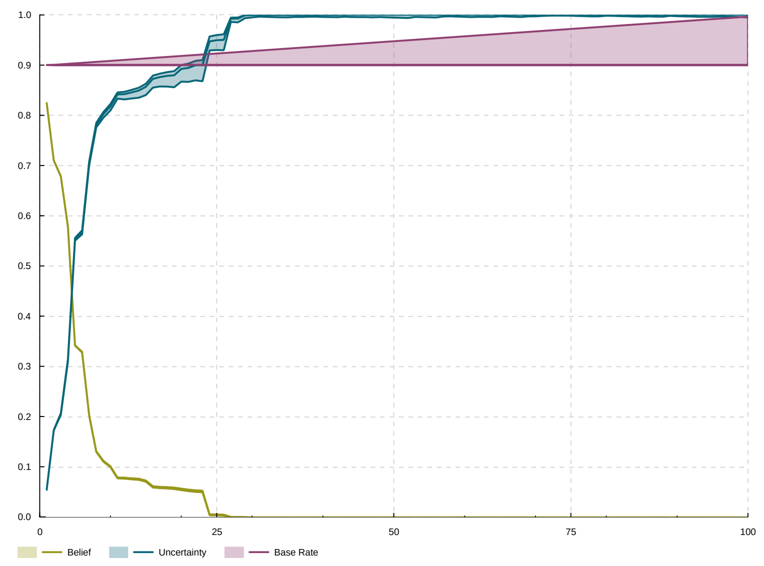


Interval Width

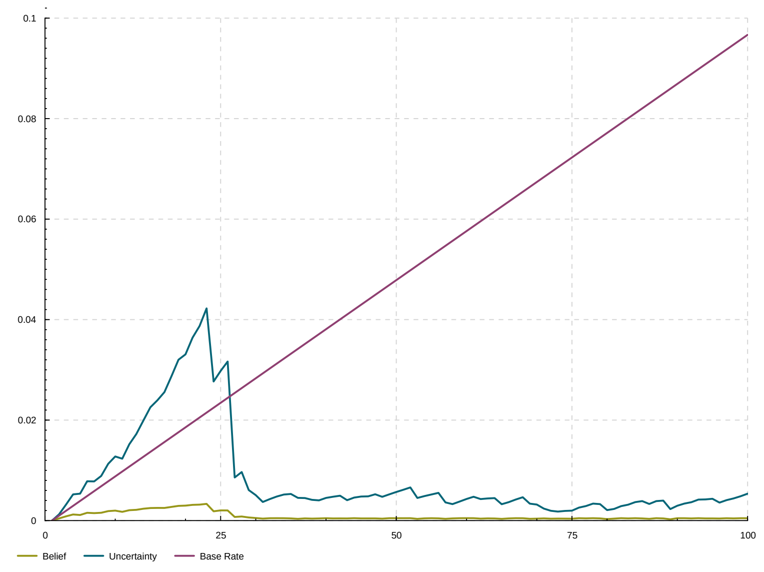
Final interval width:

```
b = 0x1.01d2b39d816d4p-12
u = 0x1.82bc0d6c438p-11
a = 0x1.8c00000000018p-4
```

Telephone-Packed-SpanishInquisition-binary64



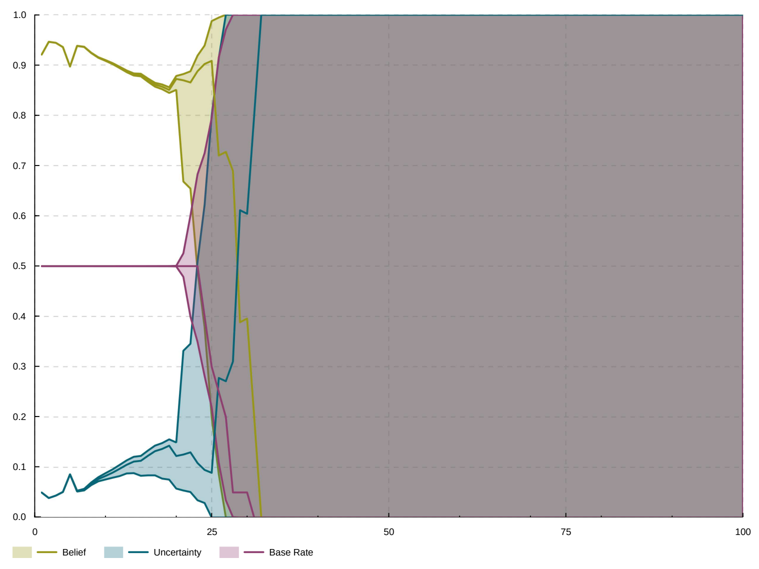
Values



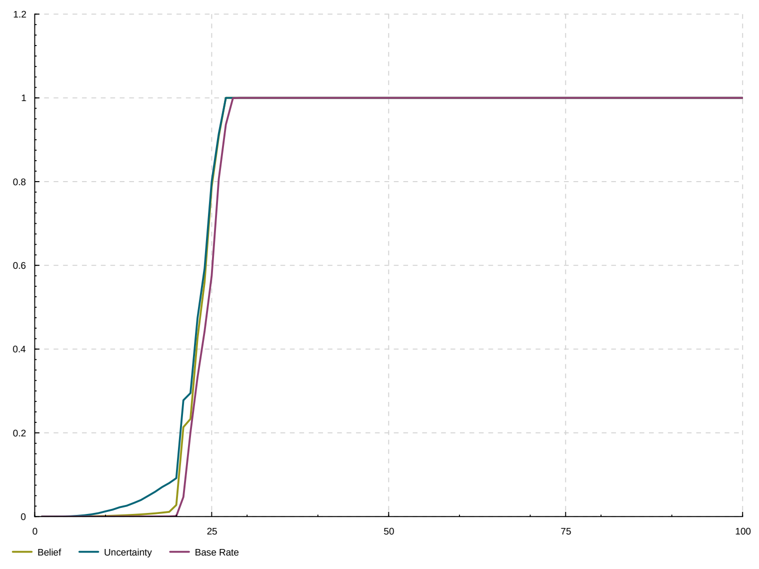
Interval Width

Final interval width:
```
b = 0x1.ee54d2c7d7046p-12
u = 0x1.5ece25c5e4f8p-8
a = 0x1.8c00000000018p-4
```

Platoon-Local-MostlyYes-binary32



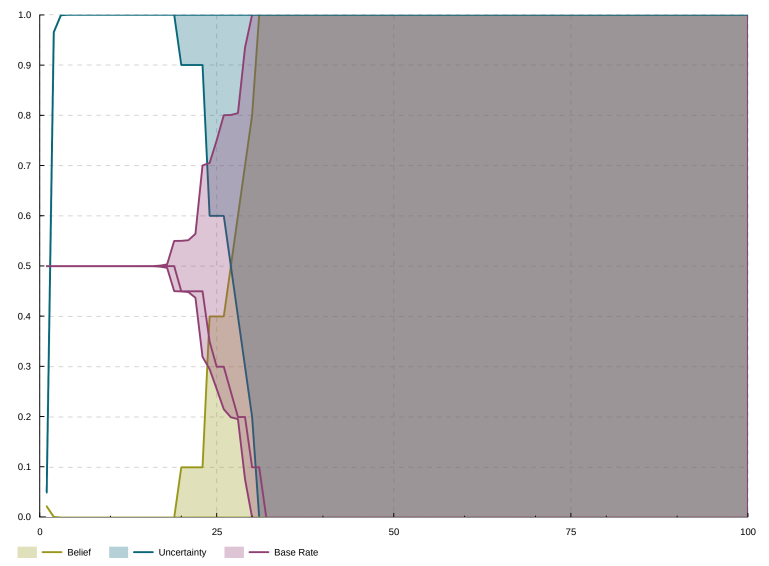
Values



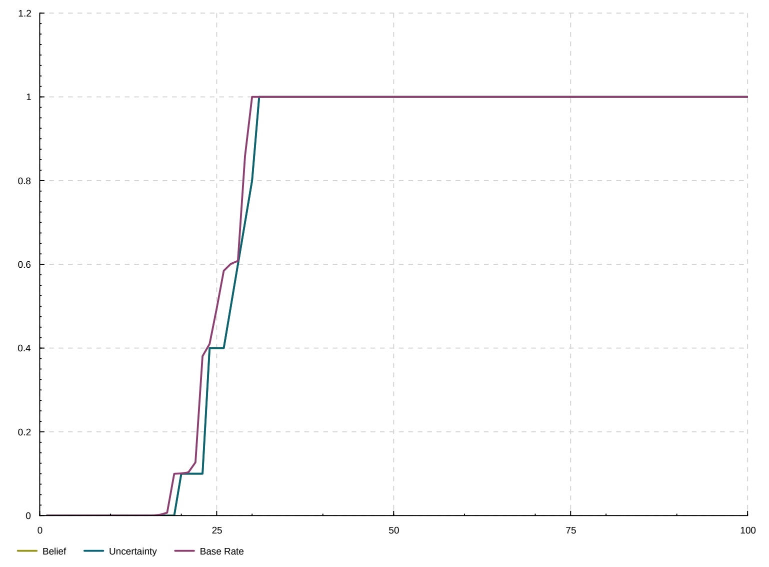
Interval Width

Final interval width:
```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.000004p0
```

Platoon-Local-MostlyNo-binary32



Values

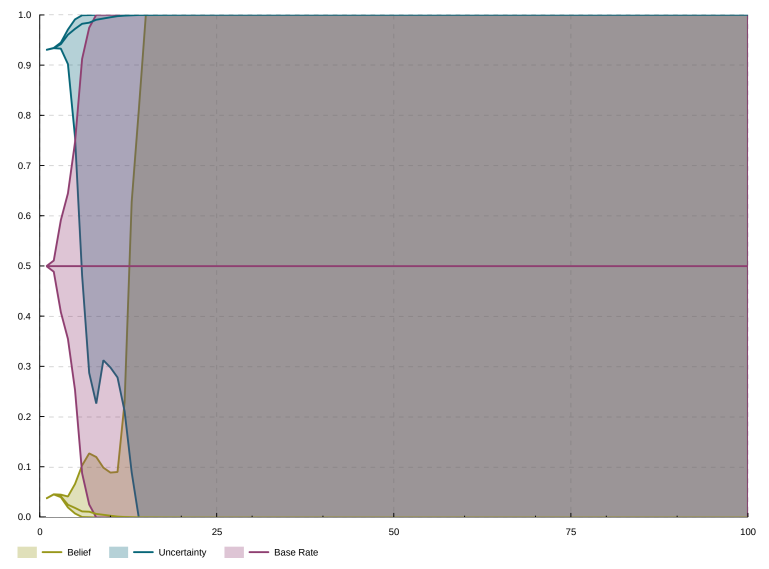


Interval Width

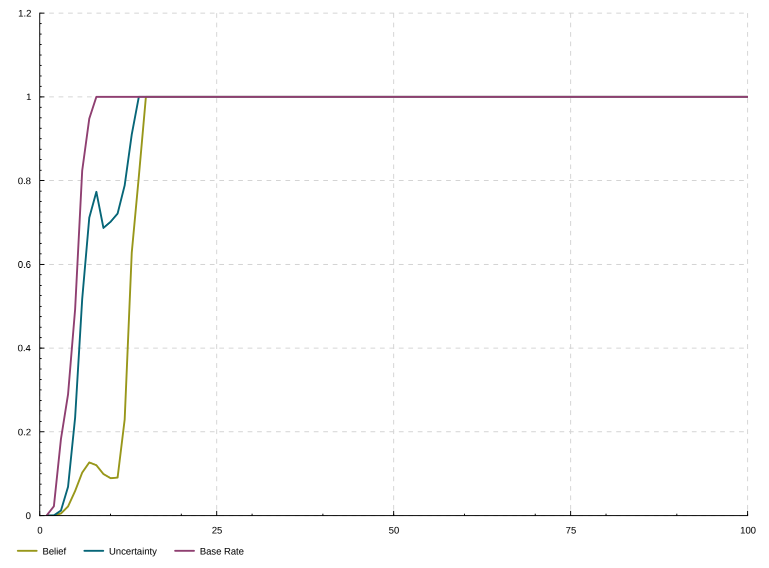
Final interval width:

```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.000004p0
```

Platoon-Local-MostlyUnknown-binary32



Values

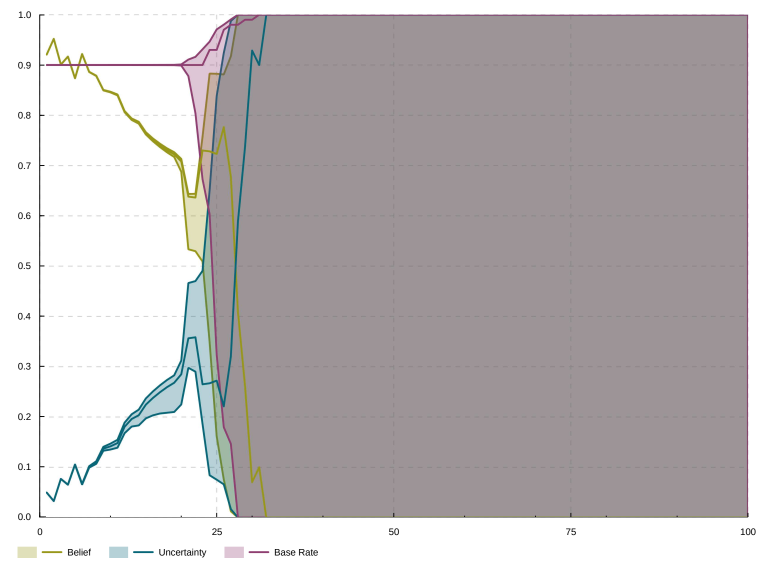


Interval Width

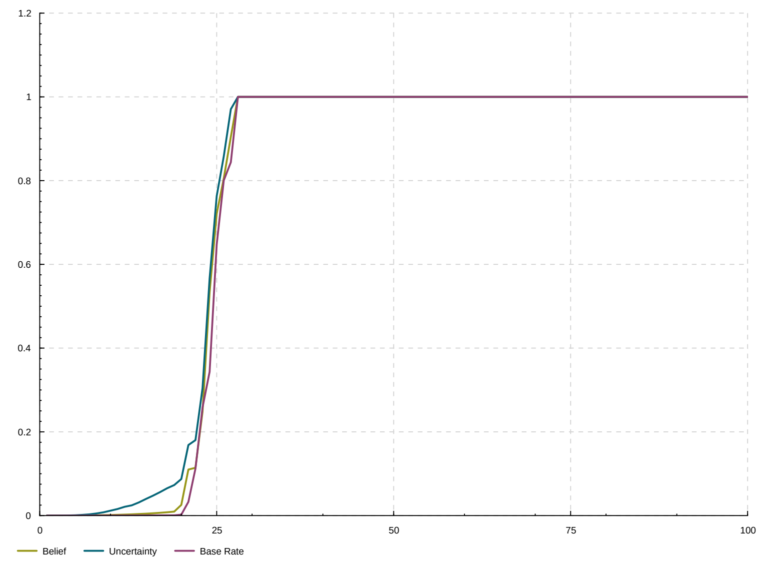
Final interval width:

```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.000004p0
```

Platoon-Local-SpanishInquisition-binary32



Values

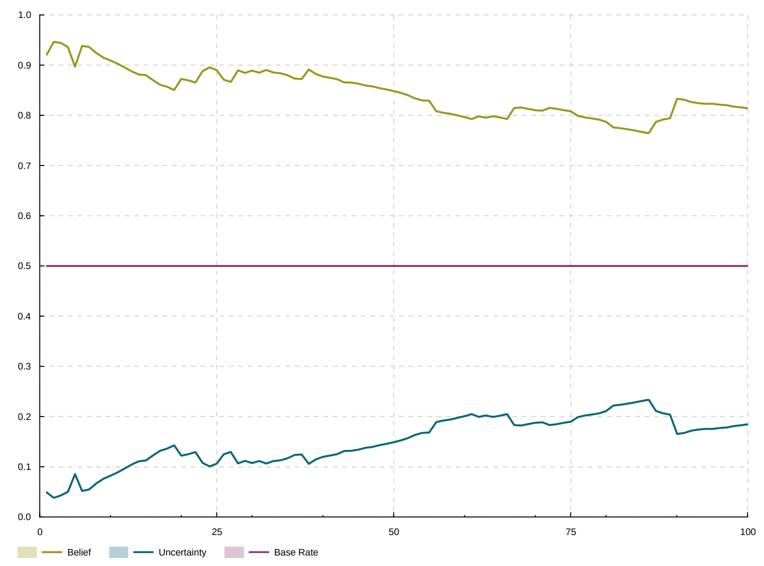


Interval Width

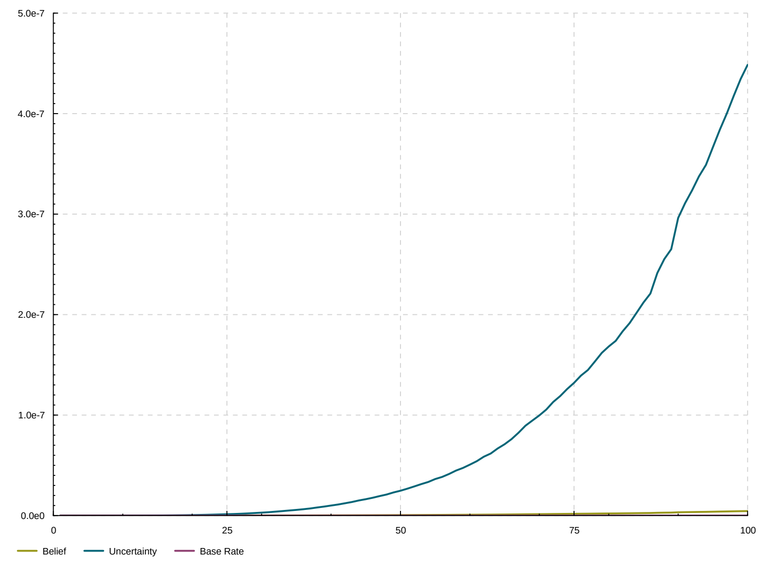
Final interval width:

```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.000004p0
```

Platoon-Local-MostlyYes-binary64



Values

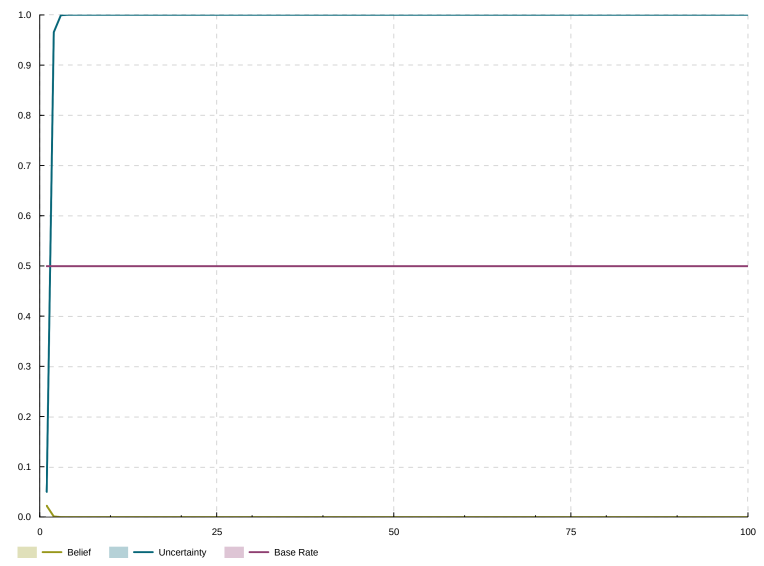


Interval Width

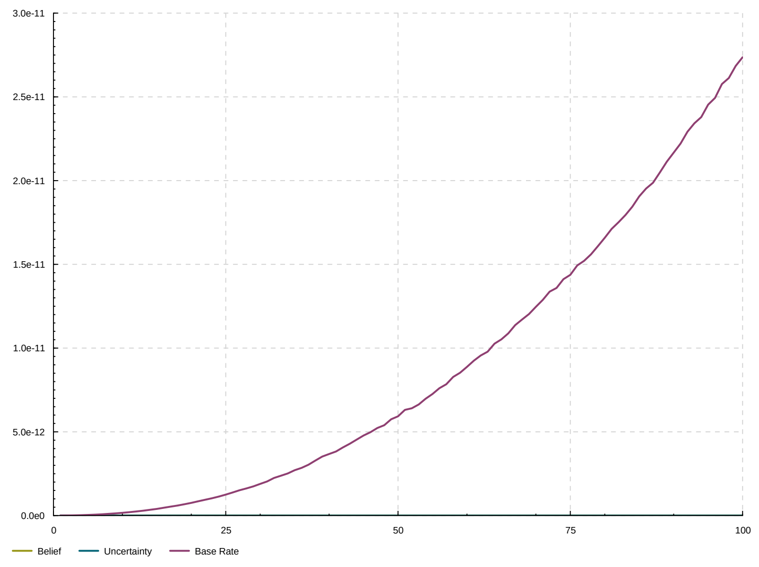
Final interval width:

```
b = 0x1.3438ea8p-28
u = 0x1.e18ed61bp-22
a = 0x1.e50e8p-36
```

Platoon-Local-MostlyNo-binary64



Values

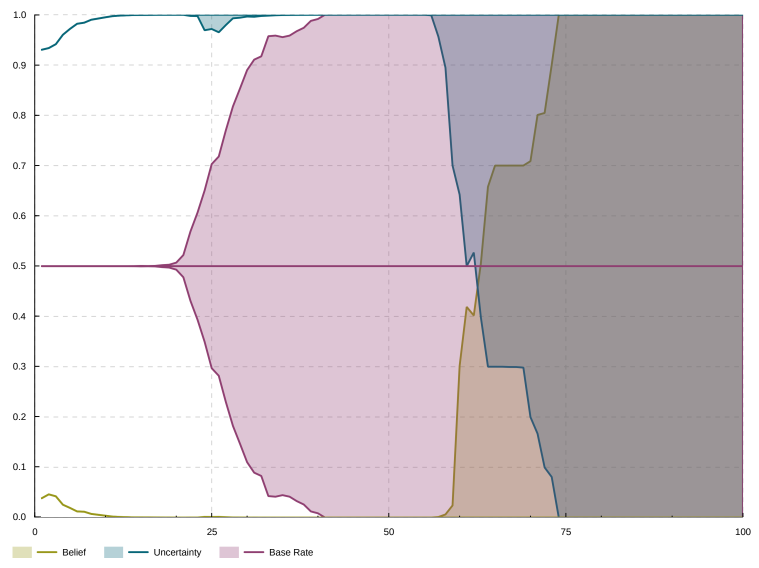


Interval Width

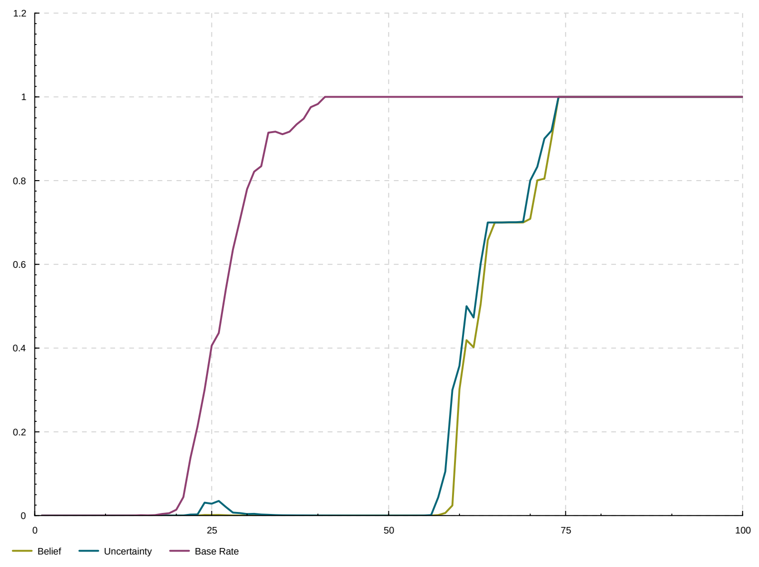
Final interval width:

```
b = 0x1.8115d38p-710
u = 0x1p-51
a = 0x1.e16cp-36
```

Platoon-Local-MostlyUnknown-binary64



Values

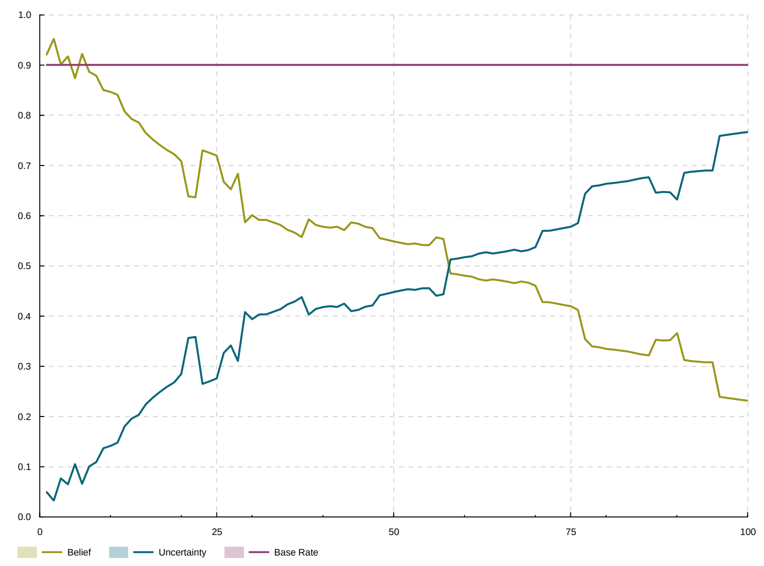


Interval Width

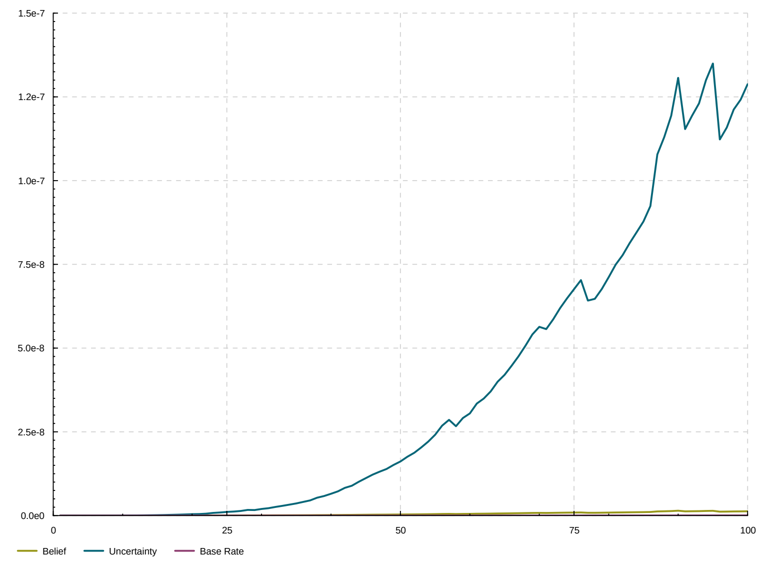
Final interval width:

```
b = 0x1.0000000000002p0
u = 0x1.0000000000001p0
a = 0x1.0000000000002p0
```

Platoon-Local-SpanishInquisition-binary64



Values

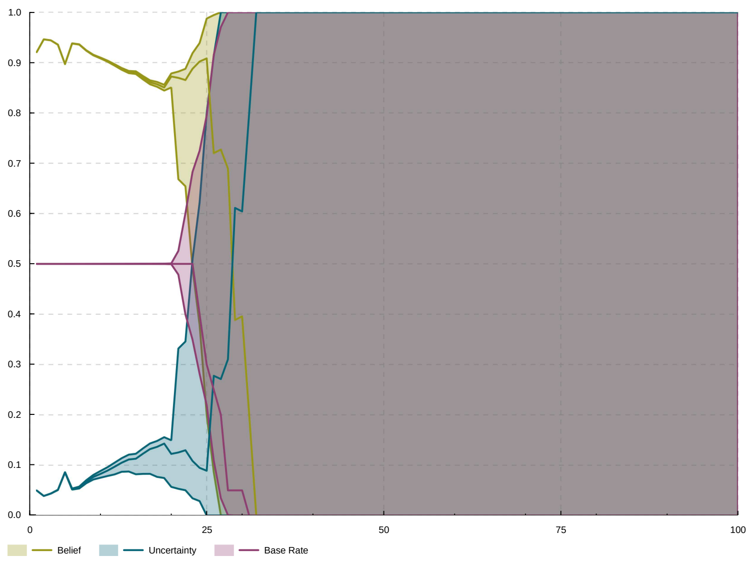


Interval Width

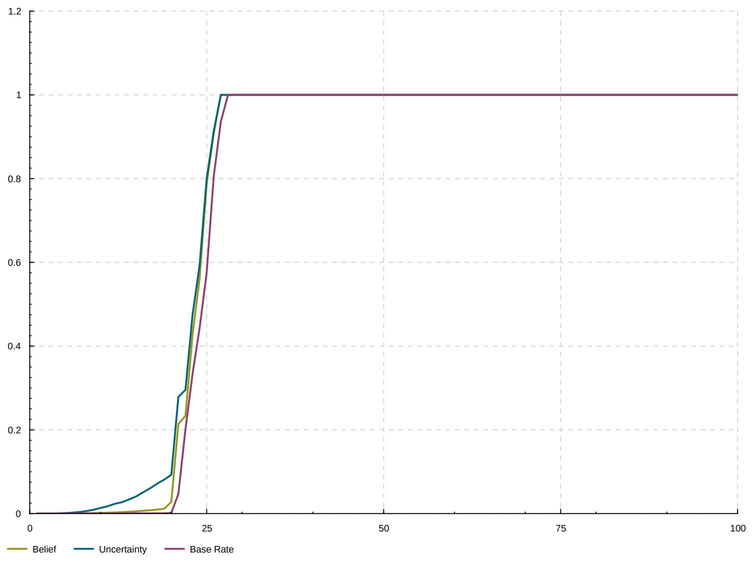
Final interval width:

```
b = 0x1.61fab7p-30
u = 0x1.148a8b94p-23
a = 0x1.bab38p-35
```

Platoon-UQ0.16-MostlyYes-binary32



Values

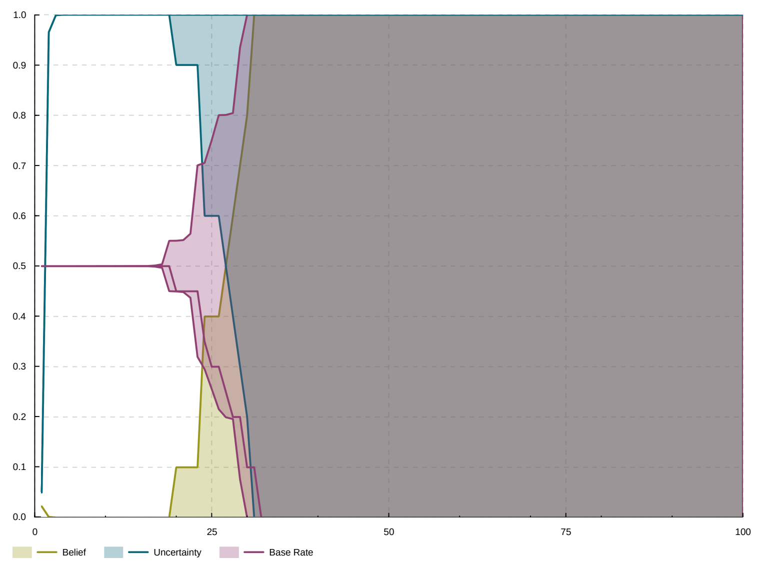


Interval Width

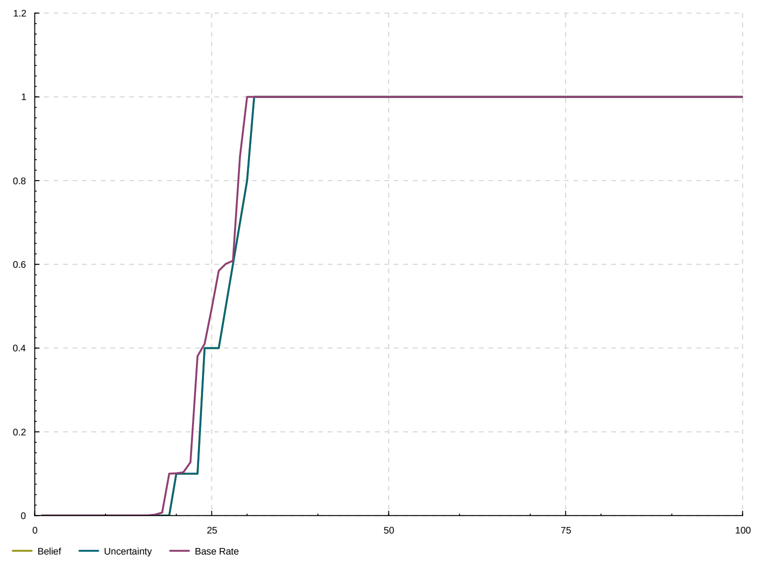
Final interval width:

```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.fffe06p-1
```

Platoon-UQ0.16-MostlyNo-binary32



Values

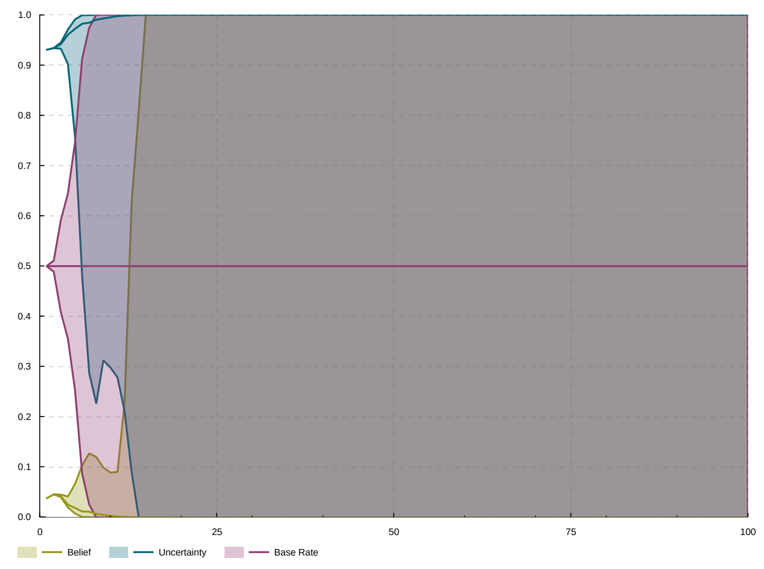


Interval Width

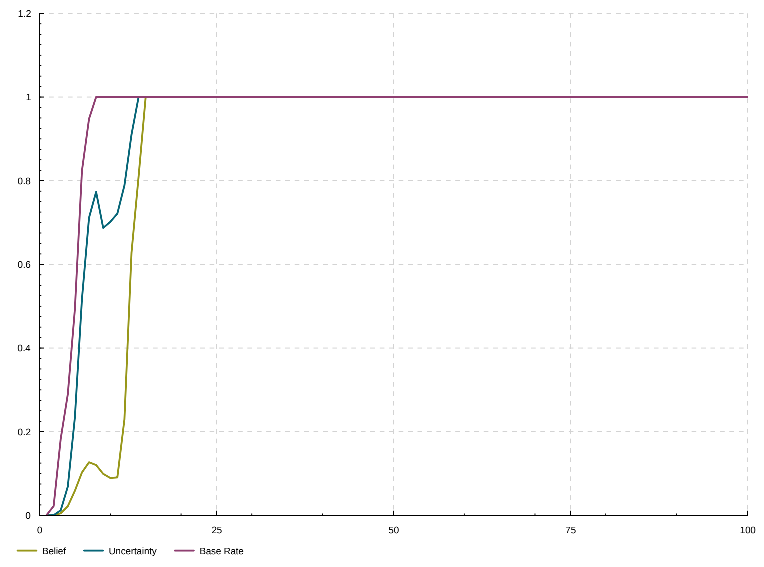
Final interval width:

```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.fffe06p-1
```

Platoon-UQ0.16-MostlyUnknown-binary32



Values

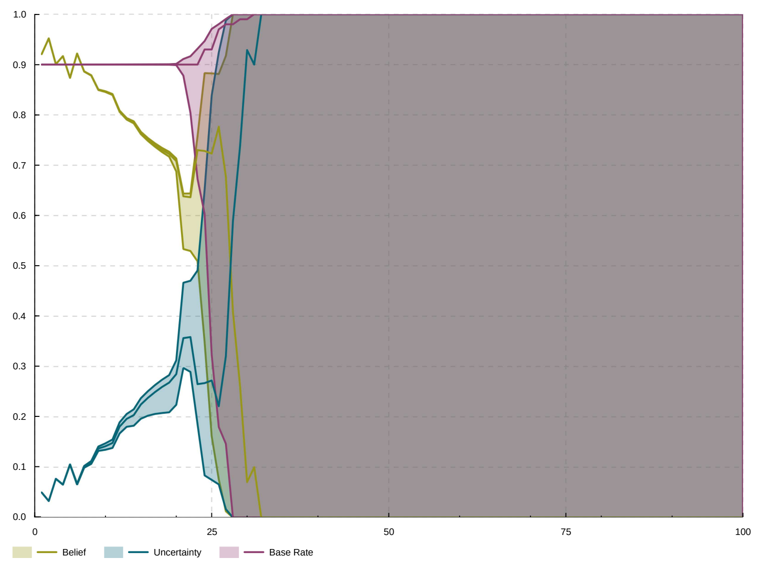


Interval Width

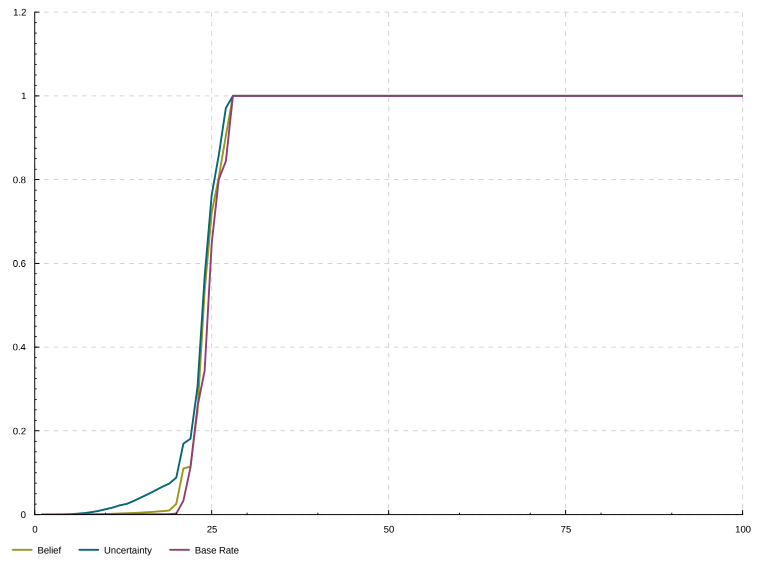
Final interval width:

```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.fffe06p-1
```

Platoon-UQ0.16-SpanishInquisition-binary32



Values

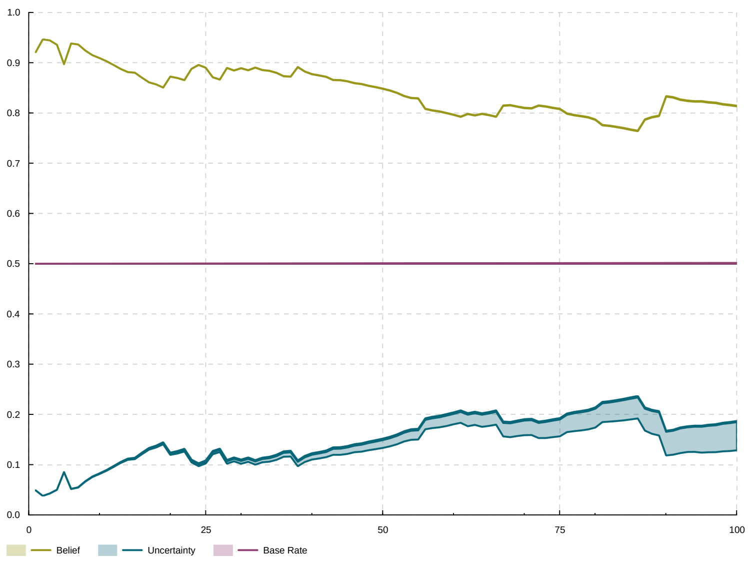


Interval Width

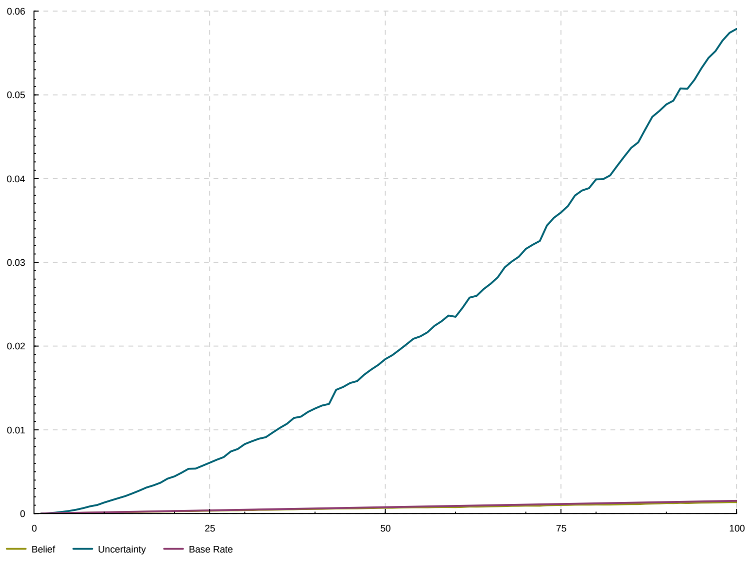
Final interval width:

```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.fffe06p-1
```

Platoon-UQ0.16-MostlyYes-binary64



Values

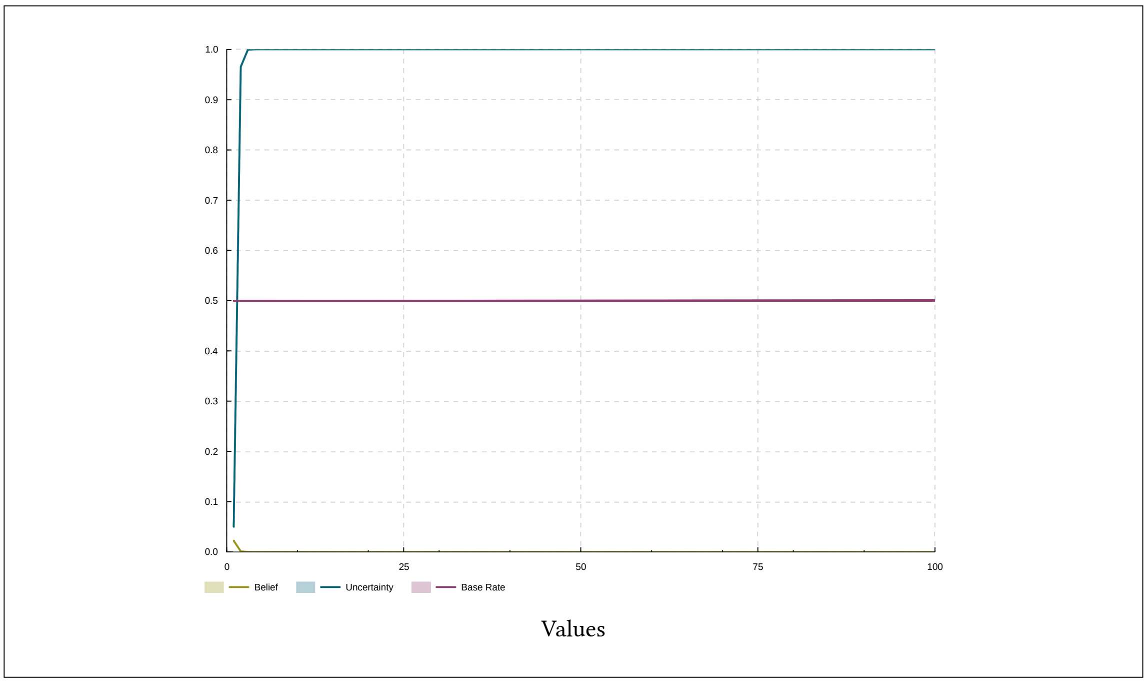


Interval Width

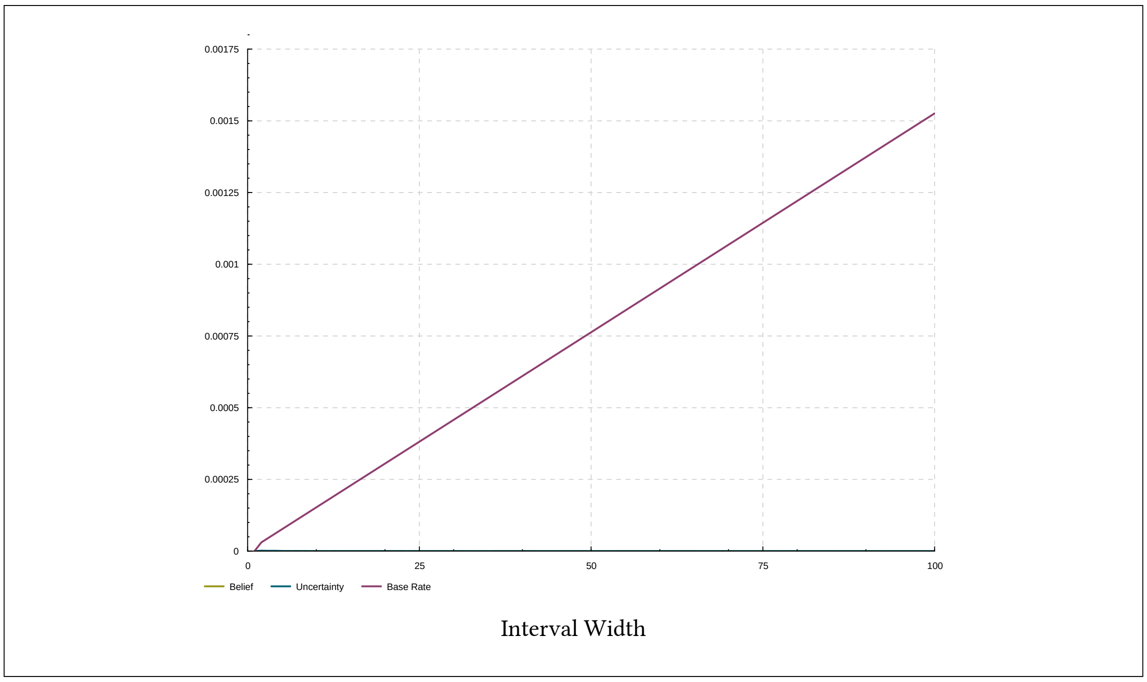
Final interval width:

```
b = 0x1.65a8e75106p-10
u = 0x1.da22a43aa5bc4p-5
a = 0x1.90000000006p-10
```

Platoon-UQ0.16-MostlyNo-binary64



Values

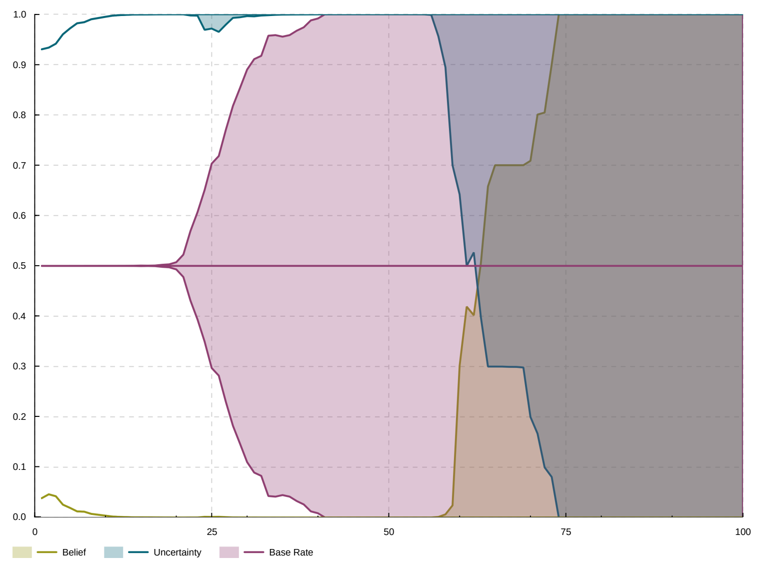


Interval Width

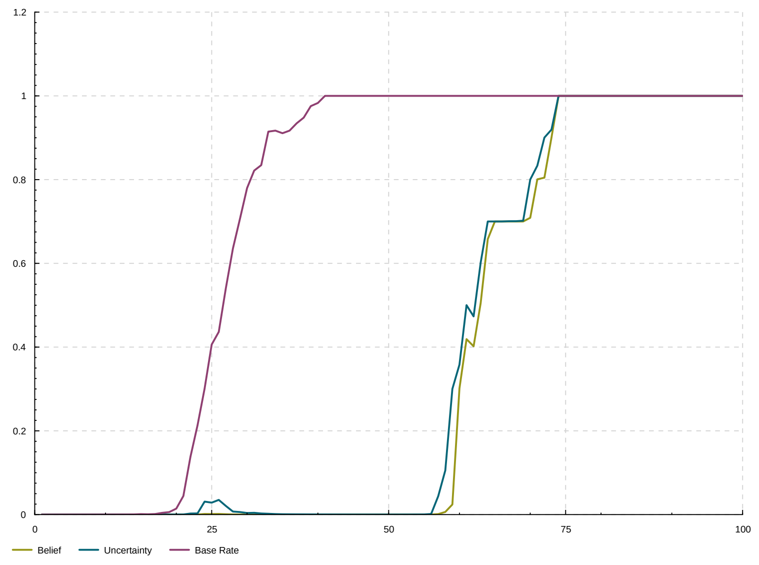
Final interval width:

```
b = 0x1.3557900413ae1p-25
u = 0x1.3557902p-24
a = 0x1.90000000006p-10
```

Platoon-UQ0.16-MostlyUnknown-binary64



Values

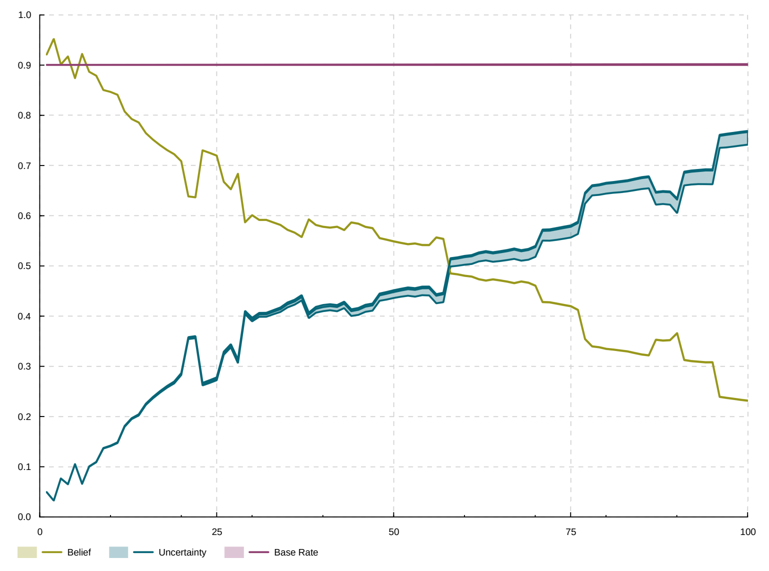


Interval Width

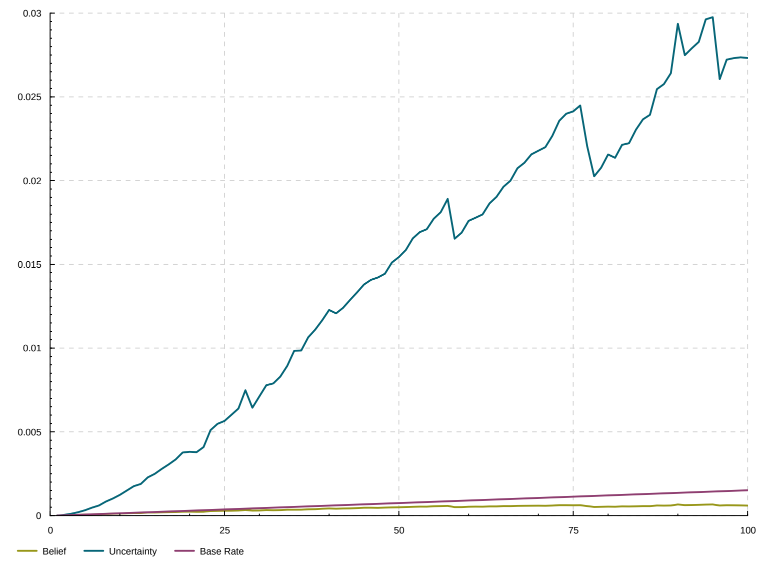
Final interval width:

```
b = 0x1.0000000000002p0
u = 0x1.0000000000001p0
a = 0x1.fffe000000003p-1
```

Platoon-UQ0.16-SpanishInquisition-binary64



Values

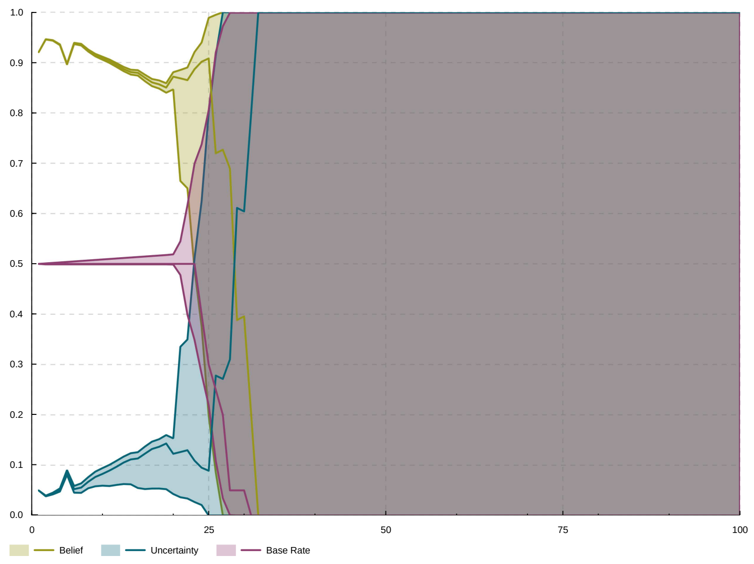


Interval Width

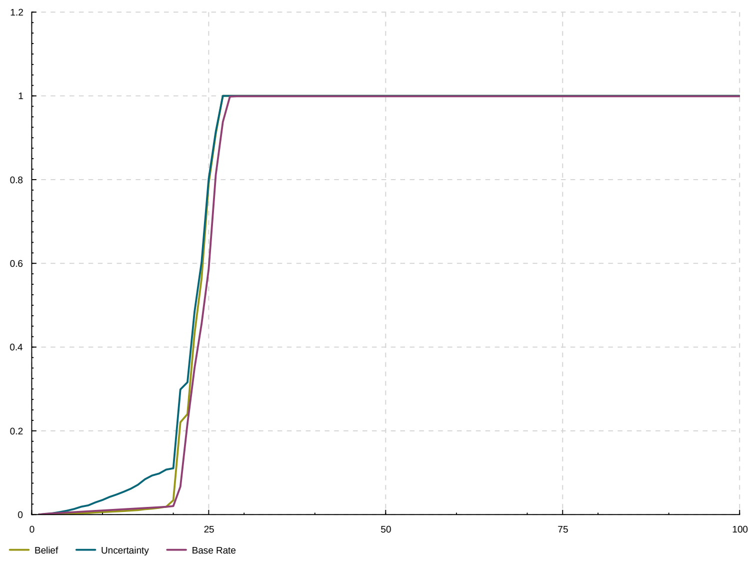
Final interval width:

```
b = 0x1.38c2d93f39ep-11
u = 0x1.bf9ee87cdd66p-6
a = 0x1.8c000000006p-10
```

Platoon-Packed-MostlyYes-binary32



Values

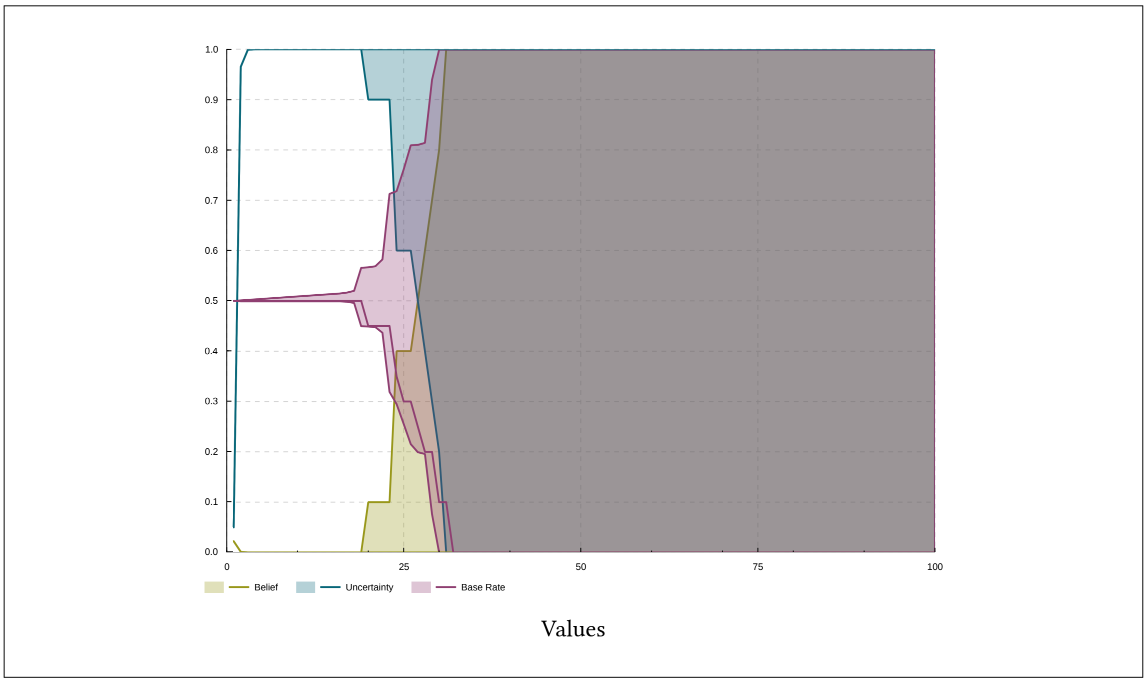


Interval Width

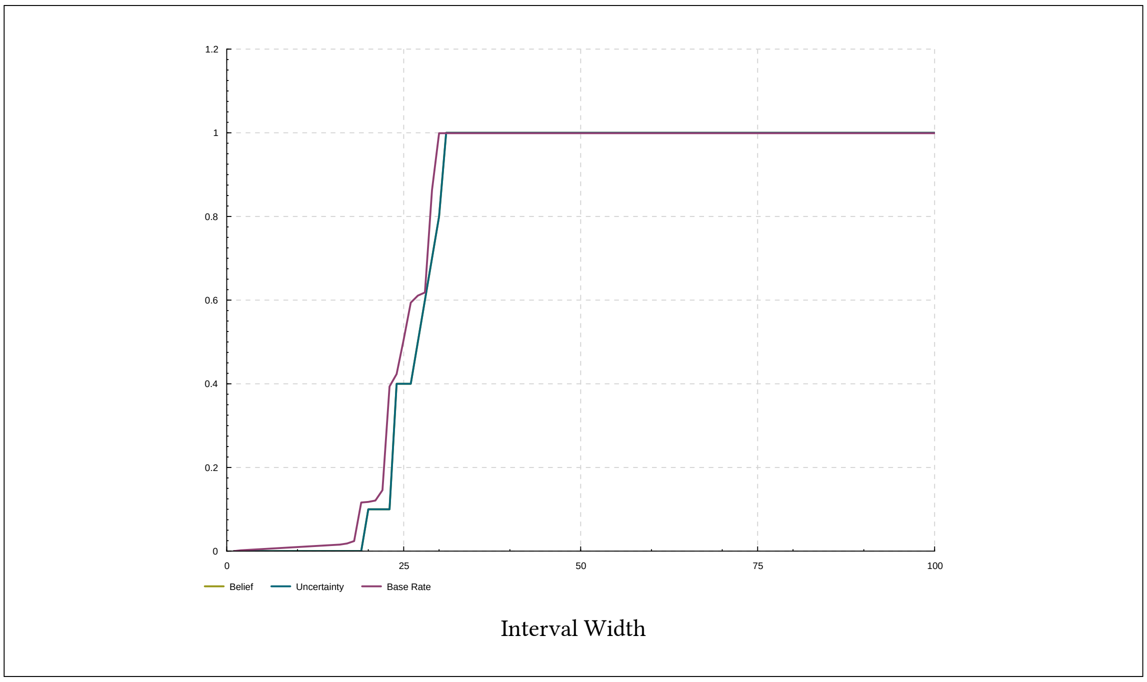
Final interval width:

```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.ff8006p-1
```

Platoon-Packed-MostlyNo-binary32



Values

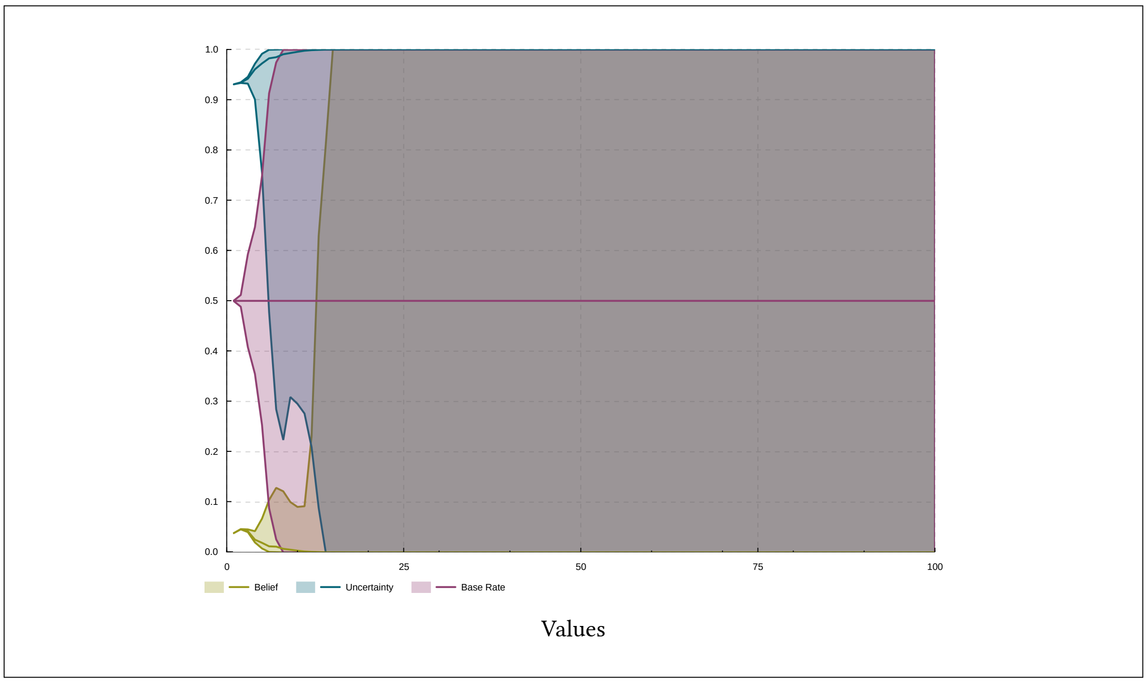


Interval Width

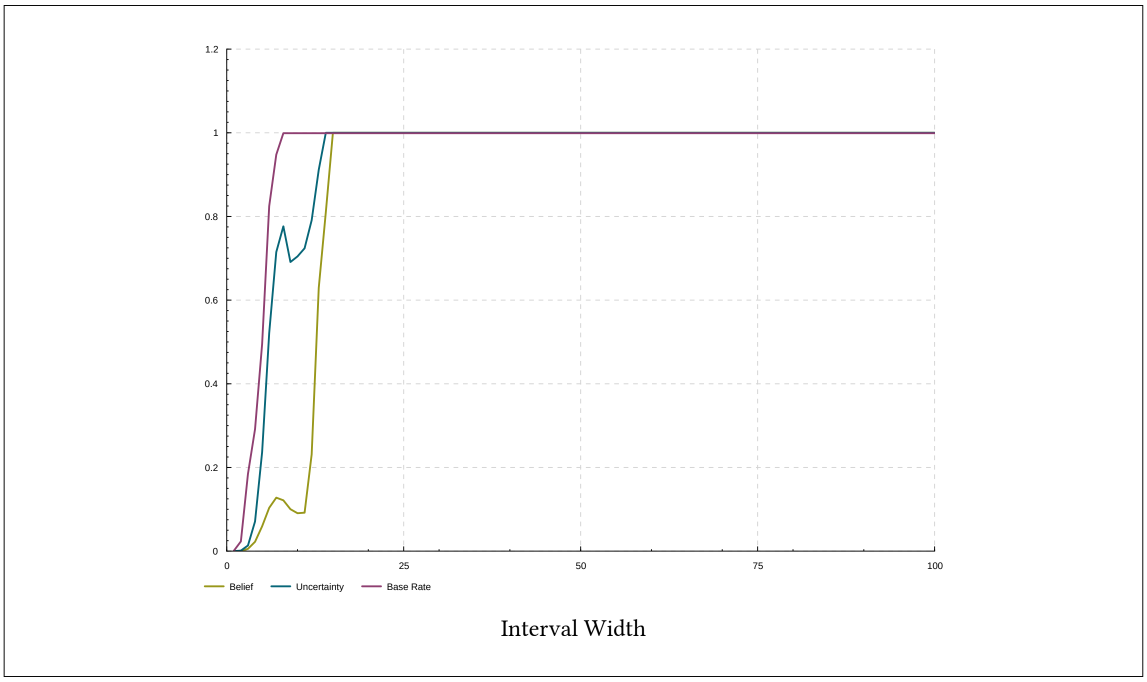
Final interval width:

```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.ff8006p-1
```

Platoon-Packed-MostlyUnknown-binary32



Values

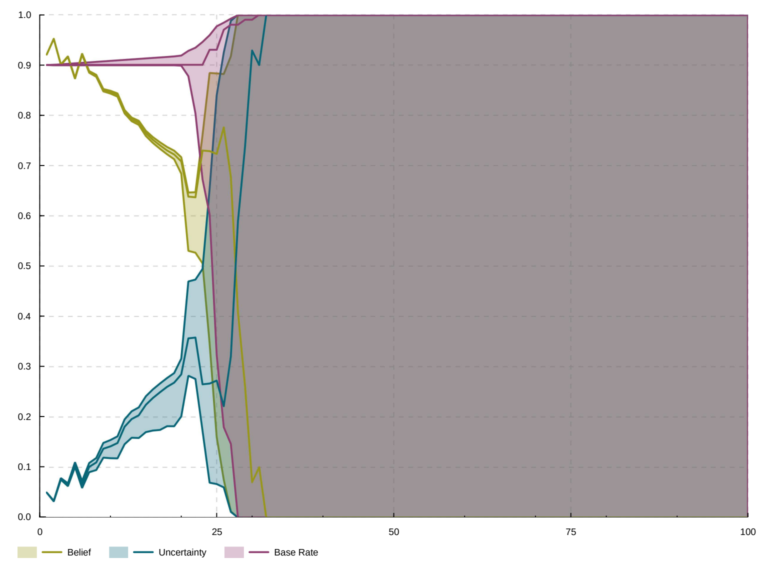


Interval Width

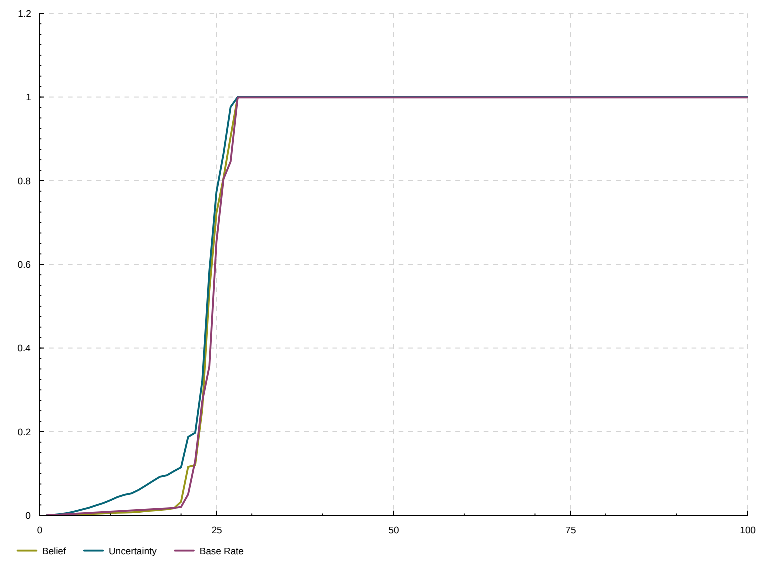
Final interval width:

```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.ff8006p-1
```

Platoon-Packed-SpanishInquisition-binary32



Values

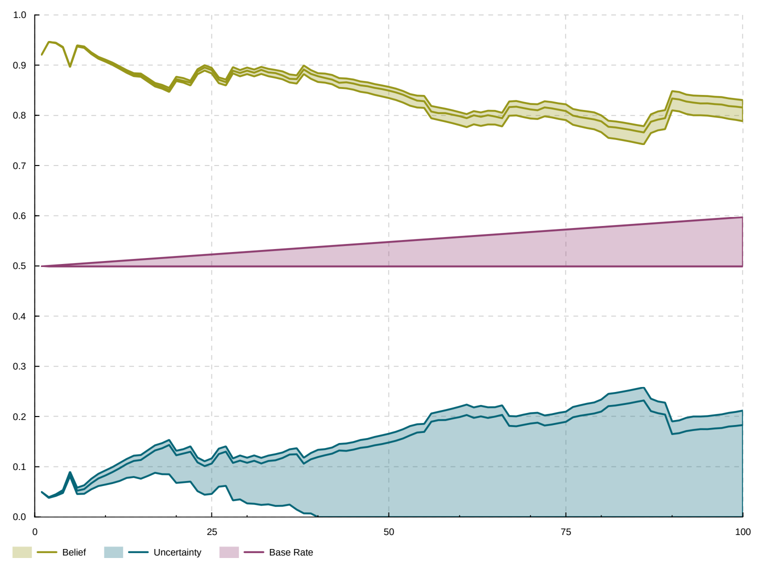


Interval Width

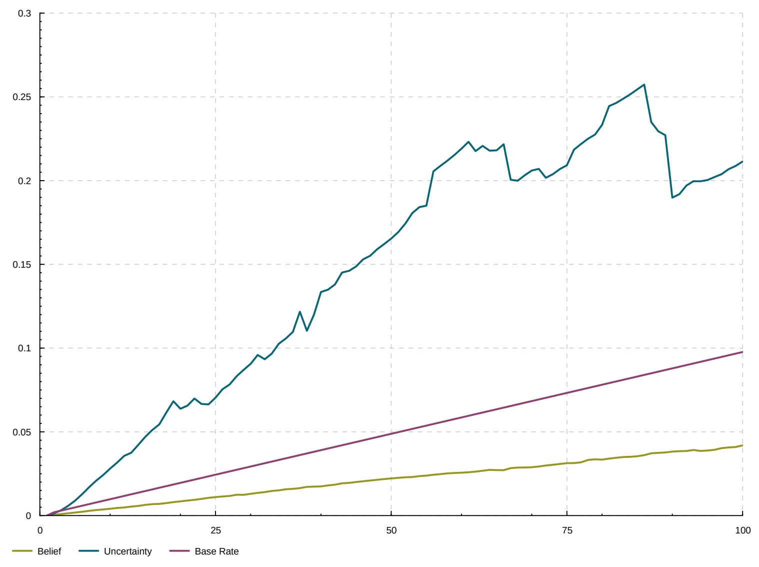
Final interval width:

```
b = 0x1.000004p0
u = 0x1.000002p0
a = 0x1.ff8006p-1
```

Platoon-Packed-MostlyYes-binary64



Values

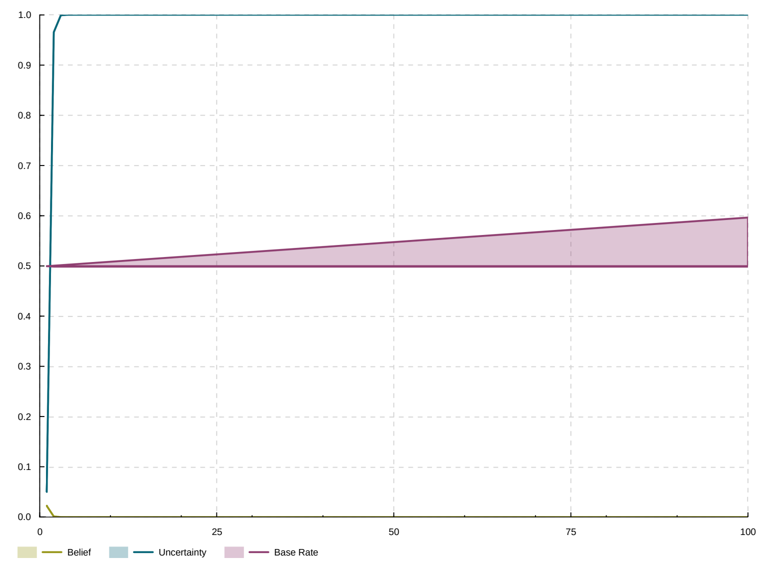


Interval Width

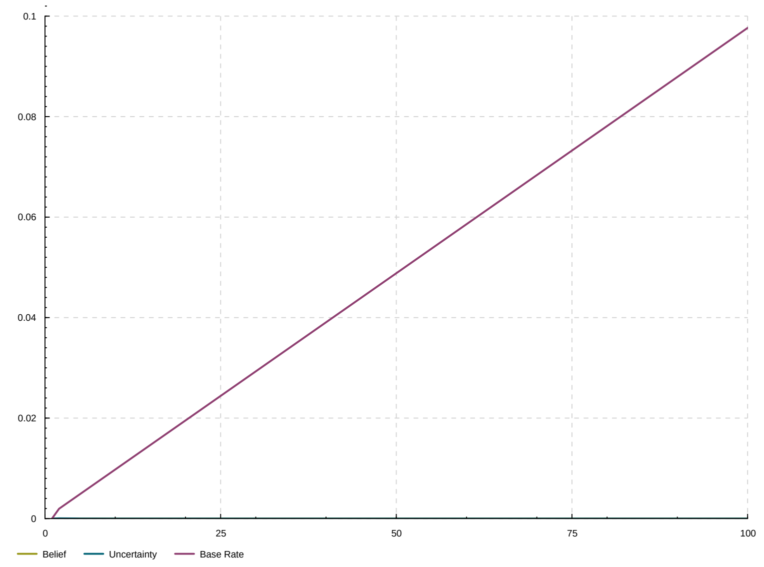
Final interval width:

```
b = 0x1.5742773b4faep-5
u = 0x1.b0e9bcb1510e9p-3
a = 0x1.9000000000018p-4
```

Platoon-Packed-MostlyNo-binary64



Values

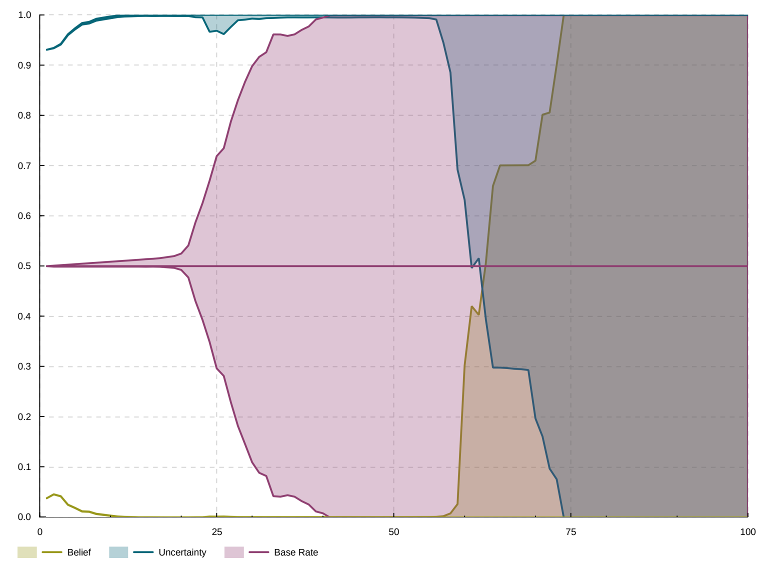


Interval Width

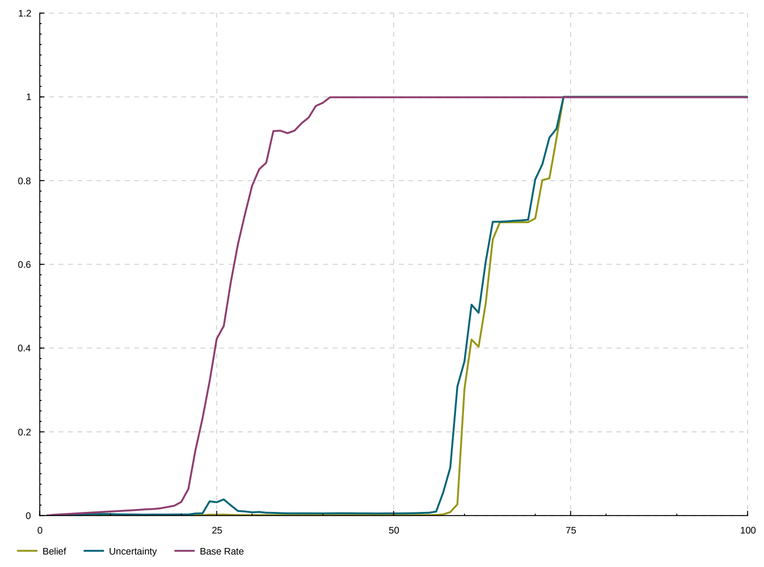
Final interval width:

```
b = 0x1.3557900413ae1p-20
u = 0x1.35579005p-19
a = 0x1.9000000000018p-4
```

Platoon-Packed-MostlyUnknown-binary64



Values

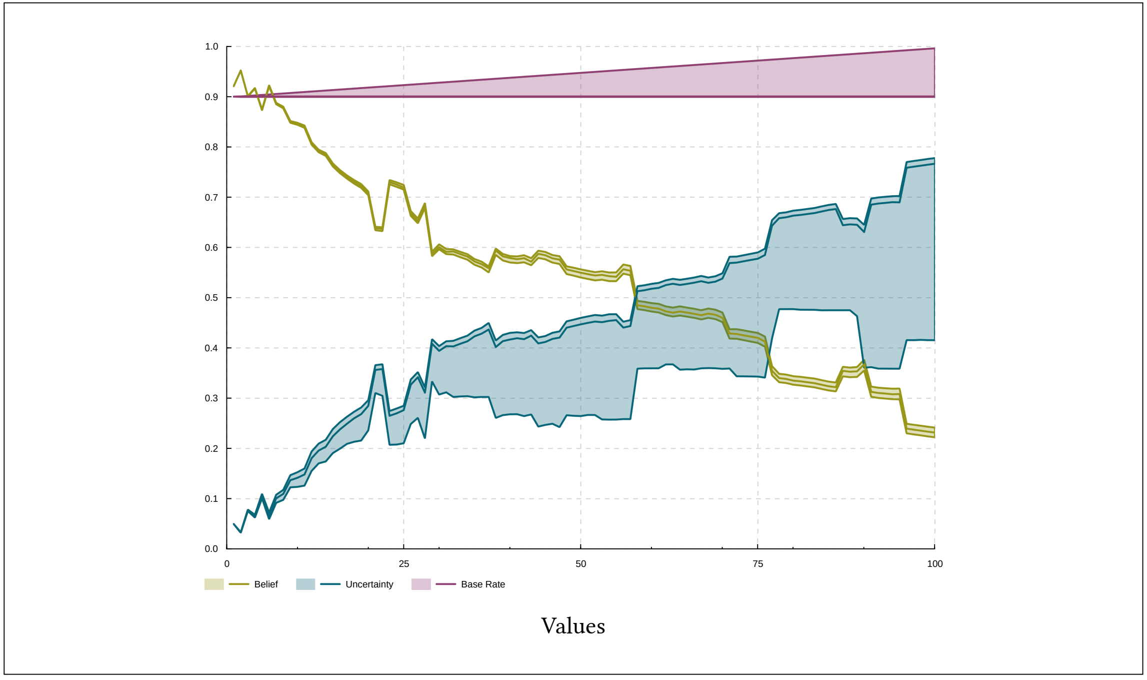


Interval Width

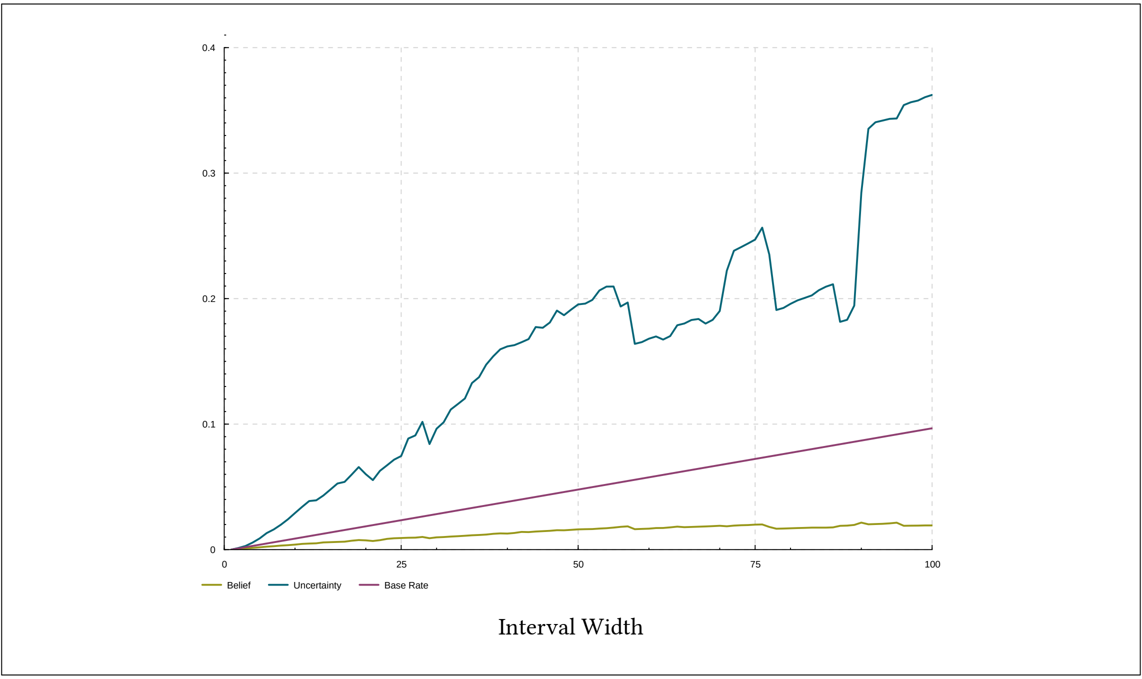
Final interval width:

```
b = 0x1.0000000000002p0
u = 0x1.0000000000001p0
a = 0x1.ff80000000003p-1
```

Platoon-Packed-SpanishInquisition-binary64



Values



Interval Width

Final interval width:

```
b = 0x1.3be9e677cd51p-6
u = 0x1.7305908b1f2fep-2
a = 0x1.8c00000000018p-4
```

References

[1] Frank Kargl, Nataša Trkulja, Artur Hermann, Florian Sommer, Anderson Ramon Ferraz de Lucena, Alexander Kiening, and Sergej Japs, "Securing Cooperative Intersection Management through Subjective Trust Networks," *2023 IEEE 97th Vehicular Technology Conference (VTC2023-Spring)*, IEEE (2023).
https://doi.org/10.1109/VTC2023-Spring57618.2023.10200789

[2] Randall Munroe, "Self-Driving Issues," *XKCD*(1958).
https://xkcd.com/1958/

[3] Mingxi Cheng, Chenzhong Yin, Junyao Zhang, Shahin Nazarian, Jyotirmoy Deshmukh, and Paul Bogdan, "A General Trust Framework for Multi-Agent Systems," *AAMAS '21: Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*(20), ACM (2021).
https://dl.acm.org/doi/10.5555/3463952.3463996

[4] Audun Jøsang, *Subjective Logic: A Formalism for Reasoning under Uncertainty*, Springer International Publishing AG (2016). ISBN: 978-3-319-42335-7.
https://doi.org/10.1007/978-3-319-42337-1

[5] K. Garlichs, A. Willecke, M. Wegner, and L. C. Wolf, "TriP: Misbehavior Detection for Dynamic Platoons using Trust" in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pp. 455-460 (2019).
https://doi.org/10.1109/ITSC.2019.8917188

[6] Nataša Trkulja, Artur Hermann, Ana Petrovska, Alexander Kiening, Anderson Ramon Ferraz de Lucena, and Frank Kargl, "In-vehicle Trust Assessment Framework," *21th escar Europe : The World's Leading Automotive Cyber Security Conference* (2023).
https://horizon-connect.eu/wp-content/uploads/2024/02/5_escar_Europe_2023_paper_5784.pdf

[7] T. Hickey, Q. Ju, and M. H. Van Emden, "Interval arithmetic: From principles to implementation," *Journal of the ACM* **48**(5), ACM (2001).
https://doi.org/10.1145/502102.502106

[8] International Organization for Standardization and International Electrotechnical Commission, *ISO/IEC 9899:1999/Cor3:2007: Programming Languages - C* (2007).
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

[9] David Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.* **23**(1), pp. 5-48, Association for Computing Machinery (1991).
https://doi.org/10.1145/103162.103163

[10] "IEEE Standard for Floating-Point Arithmetic" in *IEEE Std 754-2019* (2019).
https://doi.org/10.1109%2FIEEESTD.2019.8766229

[11] A.H. Robinson and C. Cherry, "Results of a prototype television bandwidth compression scheme," *Proceedings of the IEEE* **55**(3), pp. 356-364, IEEE (1967).
https://doi.org/10.1109/PROC.1967.5493

[12] David A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE* **40**(9), pp. 1098-1101, IEEE (1952).
https://doi.org/10.1109/JRPROC.1952.273898

[13] John W. Eaton, David Bateman, Søren Hauberg, and Rik Wehbring, *GNU Octave version 10.1.0 manual: a high-level interactive language for numerical computations.*
https://www.gnu.org/software/octave/doc/v10.1.0/

[14] Rens Wouter van der Heijden, *Misbehavior detection in cooperative intelligent transport systems,* Ulm University (2018).

[15] Nataša Trkulja, Artur Hermann, Paul Lukas Duhr, Echo Meißner, Michael Buchholz, Frank Kargl, and Benjamin Erb, "Vehicle-to-Everything Trust: Enabling Autonomous Trust Assessment of V2X Data by Vehicles," *Proceedings of the 2nd Cyber Security in CarS Workshop*(8), pp. 1-14 (2025). https://doi.org/10.1145/3736130.3762691

[16] Anna Angelogianni and Thanassis Giannetsos, *D2.1: Operational Landscape, Requirements and Reference,* CONNECT Project, UBITECH (2023). https://horizon-connect.eu/wp-content/uploads/2024/04/CONNECT-D2.1-PU-M12.pdf

[17] Artur Hermann, *D3.2: CONNECT Trust & Risk Assessment and CAD Twinning Framework,* CONNECT Project, Ulm University (2022). https://horizon-connect.eu/wp-content/uploads/2024/04/CONNECT-D3.2-PU-M18.pdf

[18] A. Jøsang, D. Wang, and J. Zhang, "Multi-source fusion in subjective logic," *20th International Conference on Information Fusion (Fusion),* pp. 1-8 (2017). https://doi.org/10.23919/ICIF.2017.8009820