Contents lists available at ScienceDirect

International Journal of Approximate Reasoning

journal homepage: www.elsevier.com/locate/ijar

Towards an automatic uncertainty compiler

Nicholas Gray[a,*], Marco de Angelis[b,1], Scott Ferson[a]

[a] Institute for Risk and Uncertainty, University of Liverpool, Liverpool, L69 7XZ, United Kingdom
[b] Civil and Environmental Engineering, University of Strathclyde, Glasgow, G1 1XQ, United Kingdom

ARTICLE INFO

Article history:
Received 8 March 2023
Received in revised form 8 May 2023
Accepted 24 May 2023
Available online xxxx

Keywords:
Automatic uncertainty analysis
Uncertainty compiler
Probability bounds analysis

ABSTRACT

An uncertainty compiler is a tool that automatically translates original computer source code lacking explicit uncertainty quantification into code containing appropriate uncertainty representations and uncertainty propagation algorithms. It handles the specifications of input uncertainties, and inserts calls to intrusive uncertainty quantification algorithms. In theory, one could create an uncertainty compiler for any scientific programming language. The uncertainty compiler can apply intrusive uncertainty propagation methods to codes or parts of codes and, therefore, more comprehensively and flexibly address epistemic and aleatory uncertainties. This paper explores the concept and the practicalities of creating such a compiler.

1. Introduction

Modern science and engineering are all about numerical calculation, and increasingly work is being done taking advantage of the power of computers. The algorithms used often rely on precise, abstract numbers – without any associated uncertainty. Scientists and engineers need to make calculations even when there is uncertainty about the quantities involved, yet the tools they commonly use do not allow this to be done intrusively. As a result, many analysts work with deterministic computer codes that do not fully account for uncertainties.

Because analysts are typically unwilling to rewrite their codes – due to the time constraints or a lack of understanding of the uncertainty calculi available – various simple strategies have been used to remedy the problem, such as elaborate sensitivity studies or wrapping the programme in a Monte Carlo loop, this allows *non-intrusive uncertainty propagation*. These approaches treat the programme like a black box because users consider it uneditable. However, whenever it is possible to look inside the source code, it is better characterised as a crystal box because the operations involved are clear but fixed and unchangeable in the mind of the current user. An alternative approach is *intrusive uncertainty propagation* where the uncertain variables are replaced with objects that characterise their uncertainty and have calculations directly performed on them. Other changes to the code may be required to allow for optimal performance, these changes depend of the programming language used: whether it is duck-typed or static typed, compiled or interpreted, etc.

An uncertainty compiler is a tool that automatically translates original computer source code lacking explicit uncertainty analysis into code containing appropriate uncertainty representations and uncertainty propagation algorithms. It handles the specifications of input uncertainties, and inserts calls to intrusive uncertainty quantification algorithms in the library. The

* Corresponding author.
E-mail addresses: nickgray@liverpool.ac.uk (N. Gray), marco.de-angelis@strath.ac.uk (M. de Angelis), ferson@liverpool.ac.uk (S. Ferson).
[1] This author was at the University of Liverpool when work was undertaken.

https://doi.org/10.1016/j.ijar.2023.108951

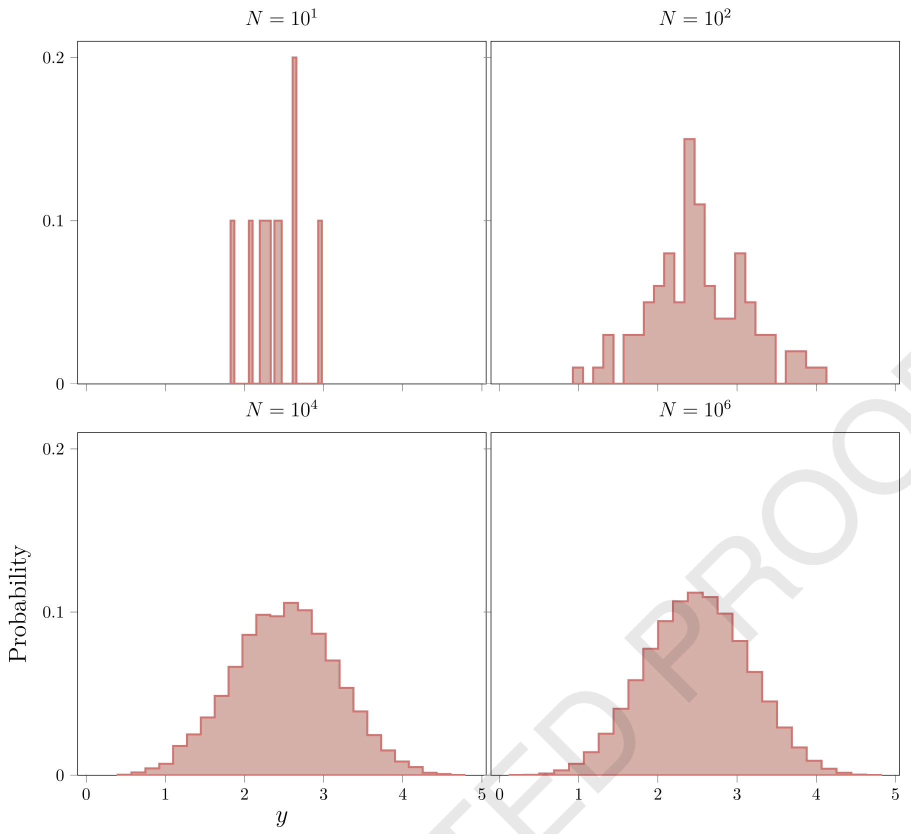


Fig. 1. Normalised histogram for the Monte Carlo simulation of Equation (1) for increasing number of iterations.

uncertainty compiler can apply intrusive uncertainty propagation methods to codes or parts of codes and, therefore, more comprehensively and flexibly address epistemic and aleatory uncertainties.

2. The problem with Monte Carlo

The most common approach to quantifying uncertainty is to wrap code within a Monte Carlo (MC) shell. In this approach, the calculations are repeated with random values for selected input variables. This is done for many iterations, and the distribution of resulting outputs can be analysed. Such tools exist in many programming languages: DAKOTA for C++ [1], COSSAN [2] and UQLab [3] for MATLAB or UQpy for Python [4]. [4] give an excellent overview of many more software packages available for non-intrusive uncertainty quantification.

Several pitfalls make MC analyses unsuitable for uncertainty propagation, especially for epistemic uncertainty [5,6]. To highlight these, we shall consider an example.

Suppose we have five variables, $x_1, x_2, \ldots x_5$, which are known to all have a value between 0 and 1, but no further information is known about the values. Suppose we need to perform the calculation,

$$y = x_1 + x_2 + x_3 + x_4 + x_5, \tag{1}$$

with the knowledge that some bad thing will happen if $y \geq 4.5$. Numbers can be randomly generated for $x_1, x_2, \ldots,$ and these can be used in order to calculate the value of y for N iterations. After this is complete, we can plot a histogram to show the distribution for y. Since we do not have any information about the distribution for $x_1, x_2, \ldots x_5$, it seems sensible to assume that all values are equally likely and use a uniform distribution. Fig. 1 shows these histograms for various N. From this, we can see that as $N \to \infty$, the histogram resembles a normal distribution.

Whatever the number of replications used in the simulation, we can estimate the probability of the bad thing happening. With 10^6 replications, this estimate is $\Pr(y \geq 4.5) = 2.53 \times 10^{-4}$. However, it seems reasonable to consider whether we have confidence that the event is so rare. We had no information about the distributions of the five values except that they were between 0 and 1. Nor did we know what dependencies there might be between the variables. From this information, we cannot rule out the possibility that each x value is much more likely to be closer to 1 than 0. Nor can we exclude that there is some dependence between the x values implying that if x_1 is high, then all the others are also likely to be high.

Several engineering failures were due in part to underestimating risks in ways similar to this example [7,8]. Before the 1986 Challenger Disaster, NASA management had predicted the probability of failure with loss of vehicle and crew as 1 in 10^5 flights [9]. This turned out to be a gross underestimation of the actual risk, which after the fleet's retirement stood at 2 in 135. The Fukushima Daiichi nuclear disaster was due in part to underestimating the risk of a tsunami of the magnitude that caused the disaster and failing to understand that collocating the backup generators created dependence that destroyed the planned engineered redundancy when the site was flooded during the event [10, p. 48].

The MC method presented in Fig. 1 is a simplistic approach. There are many different approaches that could have been taken to use MC to propagate the uncertainty through Equation (1) that are more advanced. It is also possible to use copulas to introduce dependencies between the random samples [11,12]. There are several different sampling techniques that could be used to generate the MC samples and offer analytical strategies for computing distributions

Latin Hypercube Sampling (LHS) is a popular method which aims to be representative of the true variability of the inputted numbers [13–15]. LHS is performed by dividing the range of each random variable into N bins with equal probability mass, where N is the required number of samples, generating one sample per bin, and then randomly pairing the samples. The samples generated are uniformly distributed over each marginal distribution [4].

There are also stratification methods that aim to divide the parameter space into a set of disjoint and space-filling strata that draw samples from the strata. The aim is to improve the space-filling properties of the sampling and the method allows for weighted samples [16]. Refined stratification adaptively updates the stratification by dividing the existing strata in order to improve convergence [16].

Aside from MC, there are other probabilistic approaches that could be used for uncertainty propagation. For instance, adaptive stochastic collocation techniques may have been better at finding the extreme values in the above example [17]. Or polynomial chaos expansion could have been used [18]. This technique may require modifying the original source code, although non-intrusive methods do exist [1,19].

Depending on the needs of the analyst, it may be the case that MC simulations may be suitable to solve their uncertainty quantification problems. After all, MC techniques are easy, efficient, and give insights into the aleatory uncertainty that may be present within simulations [20]. However, if there is epistemic uncertainty since all of these different approaches ultimately rely on treating all the uncertainties in a probabilistic way [6,21] false confidence may occur [22].

Some models include MC simulations intrinsically with the code. For instance, stochastic physics-based codes, such as particle simulations used for radiation transport [23] incorporate randomness of the particle trajectory, and stochastic demography models used in population viability analysis [24,25] simulate random births and deaths of organisms. Running these codes multiple times with the same parameters yields different results. The modelling of chaotic systems such as in weather forecasting also requires MC-style repetitions. However, the traditional probabilistic interpretation may still be questionable if the uncertainty is at least partially epistemic [26]. Stochastic codes could therefore pose some additional challenges for uncertainty compilers.

In general, performing uncertainty analysis by simply wrapping deterministic simulation code in a MC loop may not give a complete account of the uncertainties present within a simulation. The probabilities of extreme events are difficult to correctly estimate when there is no information about the distributions of input variables or any inter-variable dependencies.

An alternative to MC is intrusive uncertainty analysis, where the uncertainty is included directly within the code and calculations. This requires changing the code directly and thus it would be useful to have a compiler that is able to transform the code automatically. Especially since making such modifications is not as simple as replacing the variable definitions.

3. Intrusive uncertainty analysis

An alternative to Monte Carlo is intrusive uncertainty analysis, where the uncertainty is included directly within the code and calculations. There are numerous uncertain objects that could be used to define this uncertainty, including, but not limited to: probability bound analysis (PBA) [27–29], meta-distributions or second- or higher-order distributions [30–32], fuzzy numbers [33], possibility distributions [34], consonant structures [35], info-gap models [36]. All of which have numerous applications within engineering [21]. However, given the aim of this paper is to discuss the feasibility of an automatic uncertainty compiler we shall limit this discussion to PBA.

Within PBA there are a number of key objects:

- Intervals: unknown value or values for which certain bounds are known [37].
- Probability distribution: random values varying according to specified law such as normal, uniform, binomial, etc., with known parameters [38].
- Probability box (p-box): random values for which the probability distribution cannot be specified exactly but can be bounded [27].

As intervals embody epistemic uncertainty and probability distributions represent aleatory uncertainty, p-boxes can be used to propagate both types of uncertainty through calculations. Libraries that add some/all of these objects are available in C++ [39], Python [29], MATLAB [40], R [[41] or [42]] and Julia [43].

3.1. Intervals

An interval is an uncertain number representing values from an unknown distribution over a specified range, or perhaps a single value that is imprecisely known even though it may, in fact, be fixed and unchanging. Intervals thus embody epistemic uncertainty. Intervals can be specified by a pair of scalars corresponding to the lower and upper bounds of the interval, such as $[0, 1]$ or $[4, 5]$. They can also be expressed as a value plus or minus some error, such as $[5 \pm 2]$, equivalent to $[3, 7]$.

Interval arithmetic computes with ranges of possible values as if many separate calculations were made under different scenarios. However, the actual computations made by the software are done all at once, so they are very efficient. Basic binary operations $(+,\ -,\ \times,\ \div)$ can be performed using interval arithmetic:

$$[a,b]+[c,d]=[a+c,b+d], \tag{2}$$

$$[a,b]-[c,d]=[a-d,b-c], \tag{3}$$

$$[a,b]\times[c,d]=[\min(ac,ad,bc,bd)\,,\max(ac,ad,bc,bd)] \tag{4}$$

and

$$[a,b]\div[c,d]=[a,b]\times\left[\frac{1}{d},\frac{1}{c}\right]. \tag{5}$$

If $0 \in [c, d]$ then care has to be taken with the division to avoid division-by-zero levels

Intervals can be propagated through common mathematical functions such as exp, sin, log, etc. This is relatively straightforward if the function is monotonic as this implies that the endpoints of the input interval correspond to the endpoints of the output interval. For example, when calculating the exponential of an interval,

$$\exp([0,1])=[\exp(0),\exp(1)]\approx[1,2.718] \tag{6}$$

For non-monotonic functions, such as sine or cosine, it is not necessarily the case that the endpoints of the interval correspond to the endpoints of the output function. For example, it is not the case that

$$\sin([0,\pi])=[\sin(0),\sin(\pi)]=[0,0] \tag{7}$$

because there are many x values such that $\sin(x) > 0$ for $x \in [0, \pi]$. The true width of the interval can be calculated using the maxima of the function within the domain, in this case, $\frac{\pi}{2}$. Hence,

$$\sin([0,\pi])=[0,1]. \tag{8}$$

We can return to the example above to demonstrate the advantage of intervals over Monte Carlo for dealing with epistemic uncertainty. In the example, each variable is only known to have values between 0 and 1; therefore, they are best represented by the interval $[0, 1]$. This has the advantage of not assuming that all values within the range are equally likely nor making any other assumptions about the calculation. In this situation, we can calculate that $y = [0, 5]$, and would not make any further assumptions about the most-likely value from within this range.

3.2. Probability distributions and probability boxes

A probability distribution is a mathematical function that gives the probabilities of occurrence for different possible values of a random variable. A discrete probability distribution is a mathematical function that gives the probability of each possible value of the variable. For a continuous random variable, the distribution specifies, by way of areas under the curve, the probability that the variable falls within a particular interval [44].

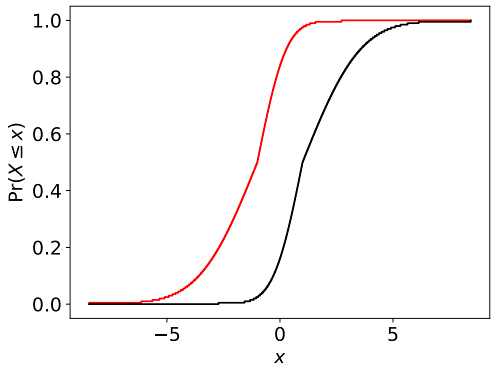
PBA integrates interval analysis and probability distributions using probability boxes (p-boxes) [27]. They can be considered as interval bounds on a (cumulative) probability distribution, probability distributions with interval parameters or as the set of possible distributions that are valid for the variable. One can think of a probability distribution as a special case of a p-box where the two bounds are identical.

P-boxes can be defined in situations where the shape of the distribution is unknown, but some empirical evidence about the data is known, such as the minimum, maximum, mean, and standard deviation. In this situation, bounds can be created such that they are consistent with all the available information [45,29].

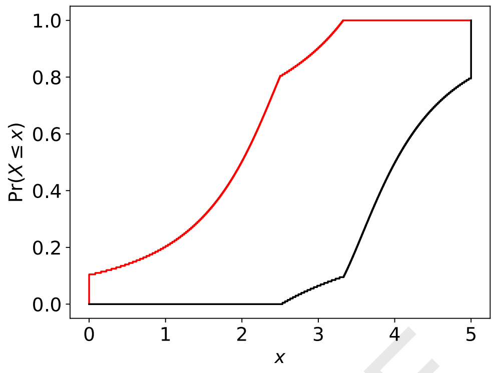
Fig. 2 shows two different probability boxes, a is a normal distribution with $\mu = [-1, 1]$ and $\sigma = [1, 2]$ whereas b has been created only knowing partial information about the variable, in this case, minimum = 0, maximum = 5, mean = 3 and standard deviation = 1.

P-boxes characterise both epistemic and aleatory uncertainty. A p-box can be expressed mathematically as

$$F(x)=[\underline{F}(x),\overline{F}(x)],\ \underline{F}(i)\leq\overline{F}(i)\ \forall x\in\mathbb{R} \tag{9}$$



(a) P-box for a normal distribution with $\mu = [-1, 1]$ and $\sigma = [1, 2]$.



(b) P-box defined by knowing that minimum = 0, maximum = 5, mean = 3 and standard deviation = 1.

Fig. 2. Two p-boxes. (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

where $\underline{F}(x)$ is the function that defines the left bound of the p-box (the red in Fig. 2a) and $\overline{F}(x)$ defines the right bound of the p-box (the black line in Fig. 2a).

As with intervals, standard arithmetic operations can be performed on p-boxes (and therefore probability distributions). For two p-boxes $A(x) = [\underline{A}(x), \overline{A}(x)]$ and $B(x) = [\underline{B}(x), \overline{B}(x)]$, then

$$C(x) = A(x) \circ B(x) = [\underline{C(x)}, \overline{C(x)}] \tag{10}$$

where

$$\underline{C(z)} = \inf_{z=x\circ y} \left[\min\left(\underline{A(x)} \circ \underline{B(y)}, 1\right)\right] \tag{11a}$$

$$\overline{C(z)} = \sup_{z=x\circ y} \left[\max\left(\overline{A(x)} \circ \underline{B(y)} - 1, 0\right)\right] \tag{11b}$$

if $\circ \in [+, \times]$, or

$$\underline{C(z)} = 1 + \inf_{z=x\circ y} \left[\min\left(\underline{A(x)} \circ \overline{B(y)}, 0\right)\right] \tag{12a}$$

$$\overline{C(z)} = \sup_{z=x\circ y} \left[\max\left(\overline{A(x)} \circ \underline{B(y)}, 0\right)\right] \tag{12b}$$

if $\circ \in [-, \div]$. Naturally, the division is only valid if $0 \notin B$ [28, p. 89].

For distributions and p-boxes, if the dependence between the variables is known, then there can be a significant tightening of the bounds when performing the calculations [28]. If the dependence is not known, then Fréchet bounds can be used to make no assumptions about the dependencies; they are the most general case and are guaranteed to bound the correct answer. Fréchet is the methodology shown in equations (10), (11) and (12).

Fig. 3 shows the result of adding two separate p-boxes, $A = \text{U}([0, 1], [2, 3])$ and $B = \text{U}([4, 6], [5, 7])$, together with different dependencies between A and B. The Fréchet bounds enclose all the other dependencies, as it is the operation that is defined in equations (10), (11) and (12). Perfect, or comonotonic, is where there is perfect positive dependence between the two variables, with the highest possible correlation coefficient. Opposite, or countermonotonic, is perfect negative dependence between the two variables with the lowest possible correlation coefficient. Independence is where there is no dependence between the two variables. It should not be assumed that variables are independent unless this is known because wrongly assuming independence can lead to incorrectly reducing the amount of uncertainty and understating tail risks.

3.3. *Logical operations with uncertain objects*

When making decisions, it is often the case that two values need to be compared to one another. Asking whether an observed value is greater than, equal to, or less than some threshold value is fundamental. For example, if a decision relies on some observed value X being less than 1 when we know the value of X accurately, it is easy to make such a comparison. However, if there is some uncertainty about the value of X, then this comparison may not be so easy.

For intervals, $X = [a, b]$ and $Y = [c, d]$, then

$$X < Y = \begin{cases} 1 & b < c \\ 0 & a \geq c \\ [0, 1] & \text{otherwise} \end{cases} \tag{13}$$

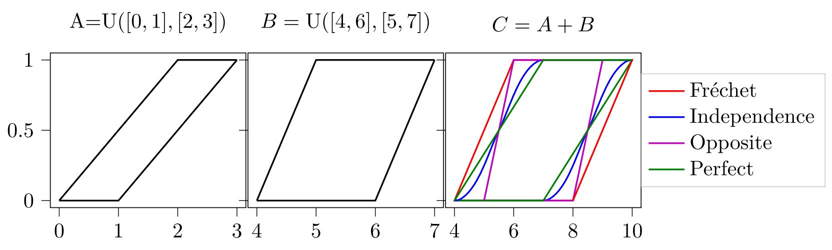


Fig. 3. Addition of two p-boxes with different dependencies.

and

$$X > Y = \begin{cases} 0 & b \le c \\ 1 & a > d \\ [0,1] & \text{otherwise} \end{cases} \tag{14}$$

with 0 and 1 denoting true and false respectively, and [0,1] being the Boolean equivalent of "I don't know". We can call [0,1] the *dunno* interval. This implies that we cannot say whether an uncertain value characterised by an interval is larger or smaller than another unless the interval is entirely greater or less than the other interval. For the equality comparison,

$$X == Y = \begin{cases} [0,1] & a \in Y \text{ or } b \in Y \\ 0 & \text{otherwise,} \end{cases} \tag{15}$$

when asking for equivalence between intervals it is never possible to say that one value is equal to another. We can introduce a new Boolean operator (===) to test whether two uncertain numbers are equivalent in form,

$$X === Y = \begin{cases} 1 & a = c \text{ and } b = d \\ 0 & \text{otherwise.} \end{cases} \tag{16}$$

The dunno interval can be converted into a true Boolean using operators such as ALWAYS or SOMETIMES

$$\text{ALWAYS}\left([0,1]\right) = 0 \tag{17a}$$

$$\text{SOMETIMES}\left([0,1]\right) = 1 \tag{17b}$$

so that we can get

$$\text{ALWAYS}\left(X < Y\right) = \begin{cases} 1 & b < c \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

$$\text{SOMETIMES}\left(X < Y\right) = \begin{cases} 1 & a < d \\ 0 & \text{otherwise.} \end{cases} \tag{19}$$

Some methods can deal with more nuanced ways of using logical operations with intervals; see [46] as an example.

If A and B are p-boxes, equivalence can be calculated as

$$A > B = \Pr\left((A - B) > 0\right) \tag{20}$$

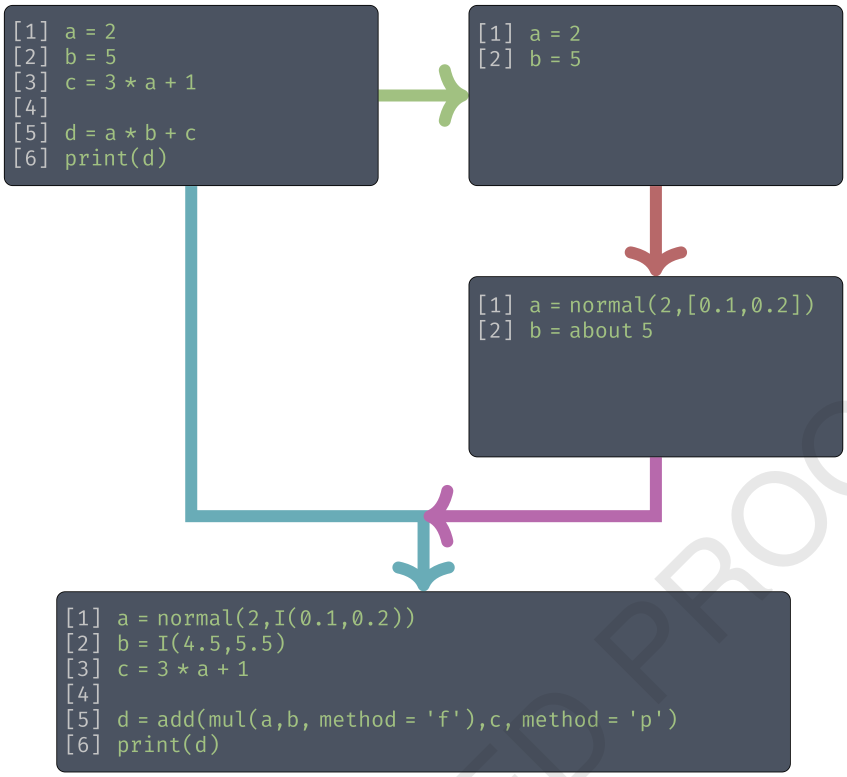
$$A < B = \Pr\left((A - B) < 0\right) \tag{21}$$

returning an interval probability. This requires different logic systems such as fuzzy logic that could be used in order to make logical operations [47].

4. An automatic uncertainty compiler?

Strategies are needed that automatically translate source code into code with appropriate uncertainty representations and propagation algorithms. [48] introduced a MATLAB toolbox to perform automatic uncertainty propagation based upon the unscented transform. However, more general approaches are needed. This section discusses the prospect of creating an automatic uncertainty compiler to perform intrusive uncertainty analysis with crystal box codes (when the source code can be viewed but is considered fixed and uneditable). Such a compiler would handle the specifications of input uncertainties and insert calls to an object-oriented library of intrusive uncertainty quantification (UQ) algorithms. This approach could theoretically work with any computer language and any flavour of intrusive uncertainty propagation.

Such a compiler will have to perform several tasks:



```
[1] a = 2
[2] b = 5
[3] c = 3 * a + 1
[4]
[5] d = a * b + c
[6] print(d)
```

```
[1] a = 2
[2] b = 5
```

```
[1] a = normal(2,[0.1,0.2])
[2] b = about 5
```

```
[1] a = normal(2,I(0.1,0.2))
[2] b = I(4.5,5.5)
[3] c = 3 * a + 1
[4]
[5] d = add(mul(a,b, method = 'f'),c, method = 'p')
[6] print(d)
```

Fig. 4. The steps that an uncertainty compiler needs to take in order to take the simple pseudocode script in the top left, extract the assignments (green line), add in the user-specified uncertainty (red line), translate the uncertainties into the original language (pink line) and then merge into the original script (blue line), making appropriate additional changes so that the code runs optimally.

1. Read the input source code to identify the variables in any assignment operations which may have uncertainty associated with them.
2. Replace or modify some or all of these assignments according to options and specifications provided by the user,
3. Translate the expression trees, with amended assignments, into the target language equipped with its intrusive UQ library, and
4. Optimise the output code to ensure that code will run robustly and not be artifactually inflated.[2]
5. Once this trans-compilation has occurred, the output script can be run identically to the original code. As the uncertainty quantification is included directly within the simulation, the outputs will contain expressions of the associated uncertainty and be guaranteed to include the correct answer.

Further details of the precise requirements for each of these steps will follow, but, at this stage, it is helpful to consider a simple pseudocode script, as shown in the top left corner of Fig. 4. In this plot, the first step (represented by the blue line) requires the compiler to read the input script and then detect and extract the assignment operations. These include lines 1 and 2, but not those that assign a value based upon a mathematical expression (line 5), a function or directly from another variable. In theory, such variables could also be edited by the user.

These extracted variables then need to have any required uncertainty added. These uncertainties must then be translated to the source language and merged with the original script to produce a new script, as shown at the bottom of Fig. 4. This translation may include altering any functions that depend on the amended variables. In this case, the infix operators in the definition of `d` in line 5 (`+`, `*`) have been recognised and replaced with an explicit call to the UQ library functions (`add`, `mul`) which also have as an argument the dependence operation that is to be used. The lower panel of Fig. 4, the value 'f' of this argument corresponds to making no assumption about the intervariable dependence between `a` and `b`, and perfect dependence (comonotonicity) between their product and the variable `c`.

[2] *Artifactual uncertainty* is when the size of the outputted uncertainty is overly wide as an artifact of the calculation and not as a result of the true uncertainty present.

Table 1
Matrix showing dependencies between several variables. (f - Fréchet, i - Independence, o - Opposite, ≡ - Equal in value, 0 - implies that there is no correlation between the two variables.)

	a	b	c	d	e
a		i	≡	f	o
b	i		i	f	0
c	≡	i		f	o
d	f	f	f		f
e	o	0	o	f	

5. Reading and modifying

The first stage that the compiler will need to perform is to be able to read the script and detect the assignment operators that might have some uncertainty about them. Only numeric objects should be highlighted, not characters, strings or other non-numeric classes. In strongly typed programming languages like C, FORTRAN, and Pascal, the task of distinguishing numeric from other types of objects is easy. In Python, R, Julia or any other dynamic duck-typed language, the object type is not detectable until runtime and can even change during execution. This means that it may be difficult for the reader to detect.[3]

Objects that are collections of numeric values, such as arrays, lists and dictionaries, will require special attention. It may be necessary to look at what is inside the lists and highlight those with numeric objects within to allow the individual objects to have any uncertainties added. For vectors and matrices in general, one could consider the elements of the vector or matrix as value for which there could be uncertainty. For example, if

$$\mathbf{x} = \begin{pmatrix} 5 & 3 \\ 2 & 1 \end{pmatrix}. \tag{22}$$

The compiler would have to extract the individual elements the uncertainty about **x** could be expressed. It would therefore be possible for the matrix to be made up of mixed uncertainty objects, e.g.

$$\mathbf{x} = \begin{pmatrix} [4.5, 5.5] & N(3,1) \\ \text{ABOUT } 2 & 1 \end{pmatrix}. \tag{23}$$

Alternatively, it could be the case that all elements have the same uncertainty, something which should be possible to specify. This approach would, in theory, enable large linear algebra operations to be performed, although computational expense and dependency issues may make these operations particularly challenging to implement and require alternative algorithms to be used.

Once these objects have been extracted, they should be expressed within a language that enables users to specify the uncertainties associated with the extracted variables. This language should be simple and independent of the source language and does not need to be a textual programming language; instead, it could be a visual language as part of a graphical user interface. Every type of uncertainty that is expressible within this language must be supported by an object constructor in the uncertainty library written in the source language.

Additionally, if there is any dependency between any two variables, then the language needs a way for users to specify what these are. A natural way to do this is for the reader to produce a matrix of all linked variables and their associated dependencies. For example, as line 3 of Fig. 4 specified that `c = 3 * a` the compiler could detect that there is perfect dependence between them. The user can then edit this matrix to specify any known dependencies, as shown in Table 1. Fréchet would be used for calculations between variables that do not have specified or detectable dependencies.

In general, dependence between uncertain quantities can be expressed through the use of correlation coefficients or copulas or bounds on copulas more generally [49–51,12]. This can include named copulas such as independence, opposite and perfect (as shown in Fig. 3) or other copula families parameterised by a numerical correlation coefficient [29]. Independence implies the correlation is zero, although zero correlation does not imply independence. Likewise, a correlation of one implies perfect dependence, but, depending on the copula family, perfect dependence may not imply correlation one. The symbol ≡ can be used to indicate that the variables are equal in value, i.e., equal in distribution and perfectly positively correlated.

The dependency matrix can be checked for feasibility by asserting that it is positive semi-definite and that there are no conflicting dependencies within the table. For example, the matrix shown in Table 2 is not logically consistent for continuous variables. This is because a high value of x implies a high value of z, since they are positively dependent on each other. Meanwhile, a high value of z implies the value of y must be low due to their opposite dependence. Hence, there must also be dependence between x and y, not the independence specified in the table.

In order to make specifying the uncertainty as simple as possible, users should be able to input their uncertainties using natural language expressions such as *about* or *almost*. Humans are more likely to express their uncertainties in terms of

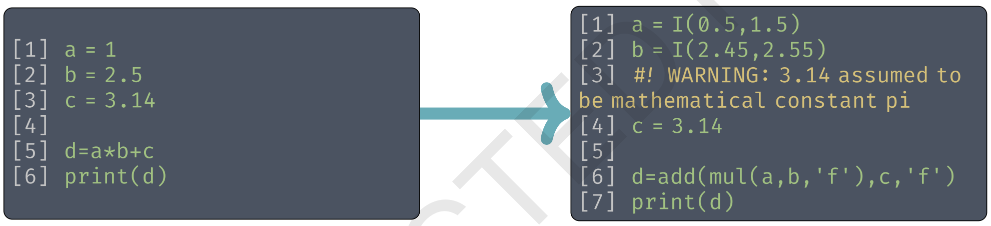
[3] However, when it comes to translating and writing, duck typed languages are easier to implement.

Table 2
Dependency matrix that does not make logical sense. (i - Independence, p - Perfect, o - Opposite).

	x	y	z
x		i	p
y	i		o
z	p	o	

Table 3
Hedge expressions and their mathematical equivalent. Note: d is the number of significant figures of x.

Hedged Numerical Expression	Possible Interpretation
ABOUT(x)	$[x \pm 2 \times 10^{-d}]$
AROUND(x)	$[x \pm 10 \times 10^{-d}]$
COUNT(x)	$[x \pm \sqrt{x}]$
ALMOST(x)	$[x - 0.5 \times 10^{-d}, x]$
OVER(x)	$[x, x + 0.5 \times 10^{-d}]$
ABOVE(x)	$[x, x + 2 \times 10^{-d}]$
BELOW(x)	$[x - 2 \times 10^{-d}, x]$
AT MOST(x)	$[0, x]$
AT LEAST(x)	$[x, \infty]$
ORDER(x)	$[x/2, 5x]$
BETWEEN x AND y	$[x, y]$
k-out-of-n	KN(k, n)



```
a = 1
b = 2.5
c = 3.14

d=a*b+c
print(d)
```

```
a = I(0.5,1.5)
b = I(2.45,2.55)
#! WARNING: 3.14 assumed to be mathematical constant pi
c = 3.14

d=add(mul(a,b,'f'),c,'f')
print(d)
```

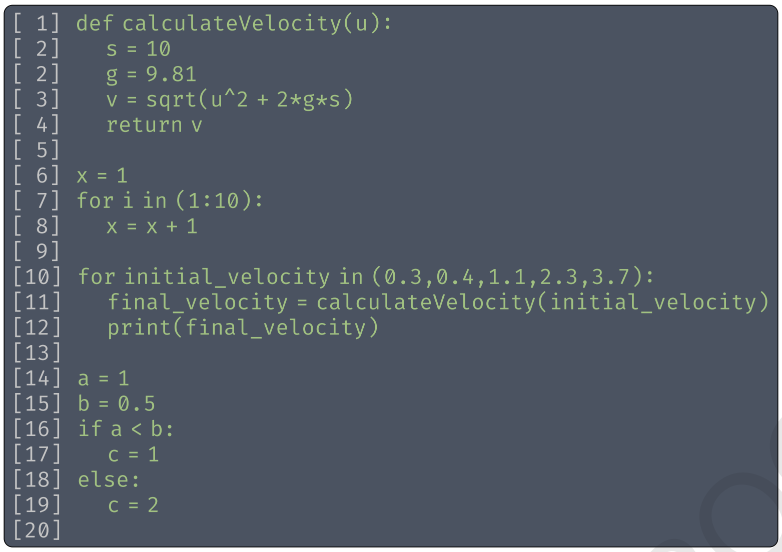
Fig. 5. Result of using the automatic uncertainty compiler on a simple pseudocode script.

hedged expressions around a round number rather than as a percentage or probability. Table 3 lists some hedge words and their possible interpretations. Hedge words can be interpreted as intervals or p-boxes [52,53].

Often it is the case that all code is not contained within one script. For example, classes and functions are often placed in other files to improve readability or avoid repetitions. Ideally, the compiler would be able to read several scripts simultaneously and allow users to express the uncertainty whilst remembering the context for all the individual objects. It is also often the case that scripts read data from other files when running. Under this scenario, it would be difficult to express the uncertainty directly within the script, although import functions could be modified to add in the uncertainties. For instance, anytime a floating-point number is read from the file, its significant digits could be interpreted to specify an interval around the value. So, for example, the value '3.56' would be understood as the interval [3.555, 3.565]. Another approach might be to parse the data file and add the uncertainties into the file directly. Doing so would require changing the import function to be able to handle uncertain data files.

It is also possible to completely automate the uncertainty compilation. Under default settings, this would replace floating-point constants with intervals interpreted from the significant figures used in the source code assignments and use that information as a proxy for the uncertainty (for an example, see Fig. 5). In this mode, each individual task happens concurrently without requiring any further input from an end-user. When using this mode, the compiler must tread carefully around mathematical constants such as π or e for which there is no uncertainty. In the example, the compiler has printed a warning above `c = 3.14` to highlight the assumption that has been made. The compiler would allow users to specify what values are precise constants. It is also worth noting that since `c` has no uncertainty the addition operation does not need to be replaced with a function with specified dependence arithmetic. Fréchet has been used for the dependence between `a` and `b`,

For-loops and functions are potential stumbling blocks for any uncertainty compiler. Fig. 6 shows a simple pseudocode script with a function and a for-loop. Within the first for-loop, each `i` is simply a control variable with a start and end variable. The individual value of `i` is irrelevant and, as such, would have no uncertainty about it. The second for-loop is a "for each"-loop, implying that the code needs to do something for each value within some iterable object. Under this scenario, it may be the case that there is uncertainty about the object within the list. In Fig. 6, the code is setting the value of `initial_velocity` as each value within the list for each iteration of the for-loop. It may be the case that there is



```
[ 1] def calculateVelocity(u):
[ 2]    s = 10
[ 2]    g = 9.81
[ 3]    v = sqrt(u^2 + 2*g*s)
[ 4]    return v
[ 5]
[ 6] x = 1
[ 7] for i in (1:10):
[ 8]    x = x + 1
[ 9]
[10] for initial_velocity in (0.3,0.4,1.1,2.3,3.7):
[11]    final_velocity = calculateVelocity(initial_velocity)
[12]    print(final_velocity)
[13]
[14] a = 1
[15] b = 0.5
[16] if a < b:
[17]    c = 1
[18] else:
[19]    c = 2
[20]
```

Fig. 6. Pseudocode script with functions, if-statements and for-loops.

uncertainty within the object, ergo the compiler should recognise this and allow users to change the code such that the objects within the list can have uncertainty added to them.

Local variables within functions will also have to be handled. For example, the function in Fig. 6 has two local variables, `s` for the distance that the object travels and `g` for the acceleration due to gravity. It is conceivable that both of these variables have some uncertainty associated with them, and the compiler needs to be able to detect the variables and offer the ability to edit them so that uncertainty is handled.

If-statements and other logical control structures may also pose issues for such an uncertainty compiler. In line 16 of Fig. 6, the logical operators within the statement would need to change to ensure that the statement runs as expected (see Section 3.3). Ideally, the analyst would decide what should happen if the statement `a < b` returned a [0,1] result by using the adverb operators discussed in Section 3.1. This may require additional editing to deal with situations where an uncertain result should be handled differently from a certain true or false or when p-boxes return probabilities for the comparison.

6. Translating and optimising

Once the uncertainty has been specified, it will need to be translated into the target language. Assuming that there exist appropriate uncertainty objects within the source language, this need not be difficult. The more difficult task is modifying the code such that it runs optimally.

The nature of these modifications depend on the specifics of the language the code is written in. In languages such as C++, it may be the case that the modifications required are small because native languages features such as operator overloading and templates can be used to increase the flexibility of the existing code [54,55]. With operator overloading, the behaviour of operators can be customised for specific types of data, allowing for more efficient calculations that are tailored to the needs of the uncertainty objects desired. Templates, on the other hand, enable generic programming, which means that a single function or class can be used with multiple data types, reducing the amount of code that would need to be modified by the compiler to enable intrusive uncertainty analysis.

When performing uncertainty analysis it would be ideal to always obtain best possible results that are guaranteed to bound the true value without overestimating the uncertainty. The uncertainty can be inflated or artifactually high if careful consideration of the dependence between, and repetition of, uncertain numbers is not undertaken.

There are a couple of approaches that could be used to handle dependency issues. The simplest approach is to replace the infix operators with the function calls that take as an argument the dependency calculi needed for the operation. For instance, in Fig. 4, the calculation

```
d = a * b + c
```

has been replaced by the compiler to

```
d = add(mul(a, b, method = 'f'), c, method = 'f')
```

where the `add` and `mul` contain keyword arguments that tell the code what arithmetic to use. In this approach, care has to be taken to make sure that precedence, left recursion and associativity of the operators are handled correctly [56, pp. 69–72].

Any dependence between `a` and `b` would have to be initialised. This could be done by adding a function after both `a` and `b` have been assigned or by modifying the assignment for `b`. In this case, as no dependence is known to the compiler, Fréchet has been used.

Smart dependency tracking – where the variables themselves work out how to handle the uncertainty arithmetic with other uncertain objects at runtime – would be the zenith when wrangling the issue. Whilst initial dependencies would still have to be specified, under smart dependency tracking, no other changes would need to be made to the code by the user or compiler for the dependencies to be handled optimally. In the output script of Fig. 4, line 5 would not need to be changed as the variables would remember which other variables they depend on and, when the calculations are performed, know what dependency calculus to use. This knowledge would have the be included within the class definitions for the uncertain objects and requires all objects to store the ancestry of themselves, know the dependence between themselves and their ancestors and be able to access the dependencies between the initial variables. This approach would have demands on memory and computational time. The result of `a * b` would know that it depends on `a` and `b`. When `c` is added to the product, it would be detected that `a * b` and `c` both have `a` as a common ancestor and therefore use the appropriate addition methodology would be ascertained and used.

The repeated variable problem is caused when artifactual uncertainty is introduced as a result of uncertain objects being repeated within the calculation. For example, if $a = [1, 2]$, $b = [-1, 1]$ and $c = [3, 4]$, then

$$\begin{aligned} ab + ac &= [1,2] \times [-1,1] + [1,2] \times [3,4] \\ &= [-2,2] + [3,8] \\ &= [1,10] \end{aligned} \tag{24}$$

but

$$\begin{aligned} a(b + c) &= [1,2]\,([-1,1] + [3,4]) \\ &= [1,2] \times [2,5] \\ &= [2,10]. \end{aligned} \tag{25}$$

Although algebraically these two expressions are equal, the uncertainty associated with $ab + ac$ is greater than the uncertainty about $a(b + c)$. This is because the uncertain variable a is repeated within the former but appears only once in the latter. In essence, the uncertainty about a has been considered twice when performing the first calculation. The amount of this artifactual uncertainty can be reduced by transforming the original equation into a single-use expression where uncertain variables are only used once. If this is not possible, other techniques can be used to reduce this artifactual uncertainty (e.g. [57–60]). The repeated variable problem appears to be ubiquitous to many, if not all, uncertainty calculi [21].

The compiler will need to be able to rearrange equations such that repeated variables are removed whenever possible. The most basic approach will require having a directory of multi-use to single-use expressions. Another smarter approach would be to have a symbolic algebra system that can rearrange into single-use expressions on the fly. The simplest version of this is for it to happen just across one line, for instance, replacing

$$d = ab + ac \tag{26}$$

with

$$d = a(b + c). \tag{27}$$

These rearrangements are often not obvious and may require algebraic tricks, i.e.

$$w = \frac{x + y}{1 - xy} \tag{28}$$

can be rearranged into the single-use expression

$$d = \tan\left(\arctan(x) + \arctan(y)\right). \tag{29}$$

This problem is made more difficult if there are repetitions across multiple lines. In Fig. 4, `c = 3 * a + 1` is used in the calculation `d = a * b + c`. This equation could be rewritten as

```
d = a * b + 3 * a + 1
```

and the reduced to the single-use equation

```
d = a * (b + 3) + 1.
```

Care would need to be taken to ensure that the right variable gets the rearrangement. Take the following kinematics equation to find the position s of a particle at time t,

$$s = ut + \frac{1}{2}at^2, \tag{30}$$

where u is the initial velocity and a is the acceleration of the particle. This equation has a single repetition for both u and a but t is repeated. If there is uncertainty associated with t then this equation can be rearranged into a single-use expression

$$s = \left(\sqrt{\frac{a}{2}}t + \frac{u}{\sqrt{2a}}\right)^2 - \frac{u^2}{2a}. \tag{31}$$

This equation contains repetitions of a and u and, as such, may only be preferred if there is no uncertainty associated with either a or u. If there is uncertainty associated with either, then it may be best not to perform the rearrangement or to intersect possible rearrangements to obtain the best possible expression.

There are additional issues that the compiler could arise when rearranging equations. For example in Equation (31), if the uncertainty about a includes negative numbers, then $\sqrt{2a}$ is likely to be problematic. Additionally, there could be problems if a straddles 0 because this would result in a division by zero. A strategy for dealing with this may be to perform the calculations using both Equation (30) and (31) and intersect them to remove any artifactual uncertainty.

Many computer languages that are not purely functional support functions that specify their parameters with "call by reference" or "call by object reference", meaning the memory location of a value is passed to the function rather than a copy of the actual value. This convention can allow the function to change the values of those parameters in the calling routine, not just locally within the function. For example, Python passes the object reference to functions for all non-primitive objects, such as lists, data frames and NumPy arrays. Primitive objects in Python are: integers, floating-point numbers, Boolean, and strings. Handling languages that use the call-by-reference method of passing arguments will require care when translating.

In Fig. 7, the simple python function takes as an input a numerical value, adds 1 to it using the in place `+=` operator and then returns the addition. In the case where `x` is a primitive object (an integer), the value passes to the function and returns the calculation without modifying the original `x` value.

If the code was translated using an uncertainty compiler then this may cause an issue. In *Translation #1*, since `x` is now a (non-primitive) interval object then the object's reference gets passed to the function. This means that it *may* be the case that as the local variable `a` gets modified, the global variable `x` also gets modified. This behaviour can be seen within the example, the function has returned the addition but `x` has also been changed because the object is not copied or cloned—its reference has been shared. This issue has been fixed in *Translation #2* by the compiler recognising that `a += 1` may pose a particular problem and replacing it with `a = a + 1` as this reassigns a new object to the variable rather than to the location it references.[4]

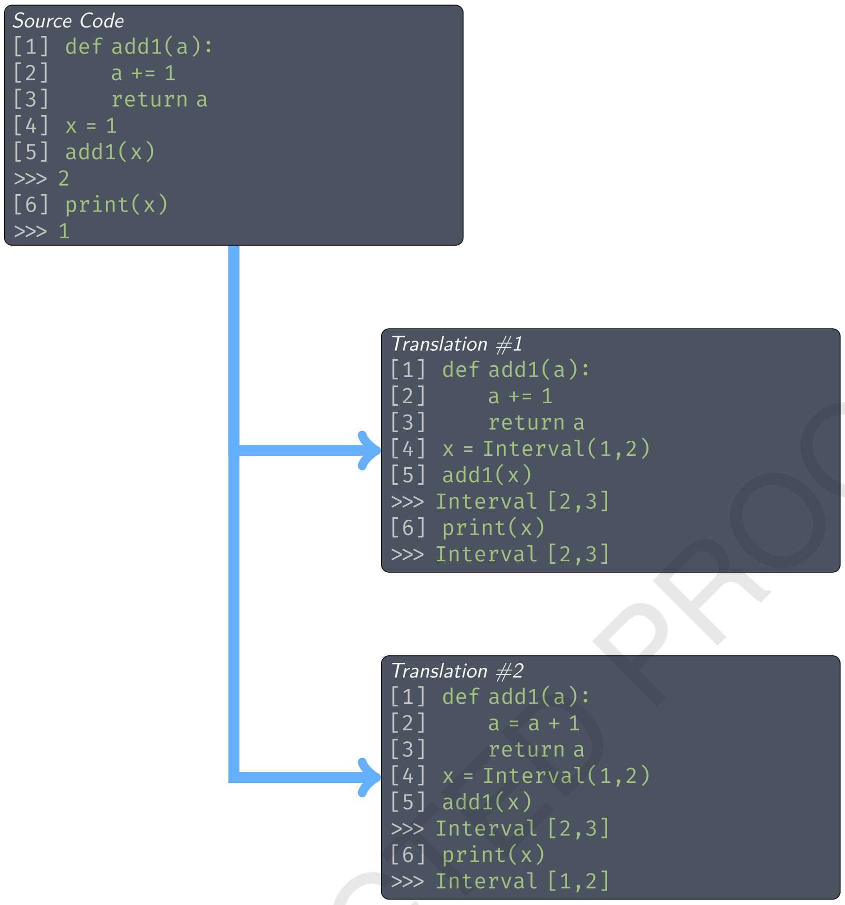
6.1. Hermeneutic problems

Several problems could occur when translating a script because it can be difficult to understand a programmer's intent from the code. An example would be the assignment `c = a` as there are several different interpretations as to what such a command implies when it comes to the uncertainty:

1. `a` and `c` are the same object but have been given different names for some reason. Under this scenario, they should be considered equivalent to each other, and therefore the calculation `a + c` could be rearranged to the single-use expression `2 * a`.
2. `c = a` could have been written as `c = 1 * a` and the `1` has been dropped as it would have had no mathematical impact on the calculation, this implies that they are perfectly dependent on each other in the same way that `c = -1 * a` implies negative dependence. Therefore, the calculation `a + c` would need to be performed using perfect dependence.
3. `c` is a copy of `a`; they have the same uncertainty, but their realisations are not necessarily related to each other whilst having the same distribution shape. This leads to two more distinct possibilities:
   (a) `a` and `c` are independent of each other and `a + c` should be convolved using independence.
   (b) No assumption is made about the dependence between `a` and `c` so `a + c` should be convolved using Fréchet.

Knowledge of which of these scenarios is correct depends on the context of the script, something a compiler should not make an assumption about by itself.

[4] It is worth noting that this may not always be desired behaviour. In Python `foo = foo + bar` is interpreted as `foo.__add__(bar)` whereas `foo += bar` is interpreted as `foo.__iadd__(bar)`. There may be differences between these two functions and it is not unimaginable the two functions have different outputs. It is likely to be the case that this difference would impact the overhead of the calculation – which itself may have lead to the analyst preferring one form over the other (especially for NumPy objects).



```
[1] def add1(a):
[2]     a += 1
[3]     return a
[4] x = 1
[5] add1(x)
>>> 2
[6] print(x)
>>> 1
```

```
[1] def add1(a):
[2]     a += 1
[3]     return a
[4] x = Interval(1,2)
[5] add1(x)
>>> Interval [2,3]
[6] print(x)
>>> Interval [2,3]
```

```
[1] def add1(a):
[2]     a = a + 1
[3]     return a
[4] x = Interval(1,2)
[5] add1(x)
>>> Interval [2,3]
[6] print(x)
>>> Interval [1,2]
```

Fig. 7. Simple Python script modified by the uncertainty compiler. In Translation #2 the compiler has modified `a += 1` to `a = a + 1` to guard against errors caused by call by object reference.

Another potential interpretation problem can occur because people naturally favour making their code readable when creating code. For example, the equation of motion for a damped harmonic oscillator can be given by

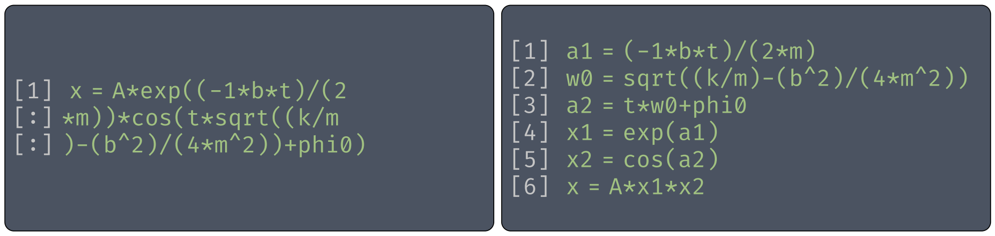
$$x''(t) + \frac{b}{m}x'(t) + \frac{k}{m}x(t) = 0 \tag{32}$$

where b is a damping constant, m is the mass of the oscillator, and k is the spring constant. This equation can be solved analytically to find

$$x(t) = A\exp\left(\frac{-bt}{2m}\right)\cos\left(t\sqrt{\frac{k}{m} - \frac{b^2}{4m^2}} + \phi_0\right) \tag{33}$$

where A is a constant and ϕ_0 is the initial angle. In Fig. 8 the equation has been coded in two different ways. In 8a the equation has been coded on a single line. As the equation is quite complicated, the programmer coding the equation would likely want to split it into multiple parts, as has been done in 8b. There are no mathematical differences between the two approaches as they will lead to the same value. Care would have to be taken breaking up equations in such a way that strategies would be needed to ensure that breaking the lines up would not have a detrimental effect on how the code operated. The dependency tracking throughout the split equation must also be assessed correctly. For example, if there were uncertainty about the damping constant b, it would be difficult to assess the dependence between lines 4 and 5, especially since the cosine function is not monotonic.

It may be better to use other techniques to solve the ODE (Equation (32)) numerically rather than analytically, using methods such as VSPODE or VNODE [61–65]. In general, any uncertainty compiler should be accompanied by a suite of bespoke UQ libraries for common but computationally complex problems that have many repeated variables. Some of these problems are NP-hard in the general case, but solutions for many important special cases have already been implemented in convenient software. In addition to the libraries for nonlinear differential equations mentioned above, special libraries would



```
[1] x = A*exp((-1*b*t)/(2
[:] *m))*cos(t*sqrt((k/m
[:] )-(b^2)/(4*m^2))+phi0)
```

(a) Coded in a single line

```
[1] a1 = (-1*b*t)/(2*m)
[2] w0 = sqrt((k/m)-(b^2)/(4*m^2))
[3] a2 = t*w0+phi0
[4] x1 = exp(a1)
[5] x2 = cos(a2)
[6] x = A*x1*x2
```

(b) Coded across multiple lines

Fig. 8. Two different ways in which Equation (33) might be coded.

be needed for computing matrix inversion [66,67], solving finite element models,[5] optimisation problems,[6] and calculating statistics from data containing intervals [73–75]. For instance, computing even statistical variance and linear regressions are NP-hard, and they probably should not be attempted without using specialised algorithms.[7] In theory, the compiler could try to detect such calculations and swap in code from these libraries as appropriate. At a minimum, the compiler should be bundled with the suite of ready solutions to the common problems to which the programmer could link.

7. Writing and running

Once the transcompilation has been performed, a new source code will have been created which can be compiled or run in the same way as the original script. Running this code may take longer and require more computational resources. However, it may still be beneficial in terms of computational cost over Monte Carlo as it only needs to be run once and not many times.

Once the code has been run, any outputs will themselves be uncertain. There are numerous ways in which this uncertainty can be expressed depending on the exact requirements specified by the analyst. Outputs could be the most likely value if this is sufficient for the analysis. Another simple approach would be to use the most appropriate hedged expression. For instance, it could return `about 3.2` if that is the best expression. This hedge would have to be the closest possible to the actual range without understating the uncertainty, whilst not being so large that it overstates the uncertainty.

For distributions, a simple approach would be to present the range or a 95% range (2.5th and 97.5th percentiles) of the output. This could go alongside other statistics about the distribution to present, for example, a five-number output that characterises the distribution (min, first quartile, median, third quartile, max). If the distribution family is known, it might be helpful to output the range, mean and variance.

Another way of characterising the distribution is to provide a verbose statement presenting a sentence or paragraph to describe the uncertainty. For instance, the output could be expressed as “a normal probability distribution p-box between -1.7 and 11 with a mean of 4.5 and a variance between 4 and 5” or “an uncertain number with an unspecified shape between 0 and 10 with a median of at most 2.4, a mean of 1.2 and a variance of at most 10.6”. Additionally, these statements could also come with a graphical representation of the output, such as shown in Fig. 2.

8. Conclusion

The idea of an automated uncertainty compiler is unlikely to be realised. Many codes are not simply single-line equations, and uncertainty propagation is not as simple as replacing every elementary operation with an interval or p-box equivalent. Numerous wrinkles have to be addressed, including:

- input specification (shapes and details),
- computational burden,
- repeated variables,
- dependencies,
- programmer intent, and
- suitable output.

It is also likely to be the case that no automatic uncertainty compiler would introduce perfect uncertainty translations. Manually editing code or creating a new uncertainty-aware script from scratch will likely outperform the automatic changes since, in general, hand-coding is always likely to outperform source-to-source translation [77]. It is, therefore, evident that any uncertainty compiler will likely be semi-automatic at best.

[5] See https://en.wikipedia.org/wiki/Interval_finite_element for numerous approaches to solving interval FEM problems.
[6] See [68–72] as examples of optimisations with imprecise numerical values.
[7] For example, the approach for interval linear regression in [76] would have to be used instead of standard least-squares linear regression.

Perhaps the greatest challenge is that rearrangements will need to be made to equations to prevent artifactual inflation of the uncertainty. The compiler will also need strategies to handle inputs from other files, non-standard inputs, and code that spans multiple interconnected lines. The repeated variable problem and, more generally, incorrect handling of dependencies can artifactually inflate the uncertainty of complex computation. Even if the calculations are technically correct in the sense that they enclose the true uncertainty, the naive application of intrusive uncertainty quantification can sometimes yield results with massively inflated uncertainty that renders them practically useless. Having massive uncertainty is not the problem itself; the true uncertainty may actually be considerable. The problem is when the uncertainty artifactually depends on how the analysis was structured and does not reflect the features of the underlying computational problem.

An uncertainty compiler is perhaps the exact opposite of an optimising compiler, which aims to minimise a programme's execution time and memory requirements. Both of these will almost certainly increase by replacing objects with uncertain equivalents. For instance, if intervalising calculations increase the computational time five-fold,[8] a simulation limited by computational time may need to be scaled back. Distributions or p-boxes would be still more burdensome. Of course, efficiency is not always a critical issue, and this extra computational effort does pay for global uncertainty propagation and what computer scientists call automatic result verification [78]. Moreover, implementing modern uncertainty quantification techniques could be more efficient and comprehensive than simply embedding a deterministic computation inside a Monte Carlo shell with millions of replications.

Despite these issues, it is important to work towards the goal of an *automatic uncertainty compiler*. It is worth remembering that the analysts who will need one most may not be sure what normal distributions or intervals are and have probably never heard of a p-box. Nor do they have the time to sift through hundreds or thousands of lines of code to extract and manipulate the various parameters and make the changes required to enable the code to handle the uncertainty optimally. For such analysts, a software tool that can hand-hold them through making the appropriate edits will be of enormous value, and even partial solutions will be beneficial. Adding intrusive uncertainty analysis to simple problems would benefit analysts who could compute with what they know rather than make assumptions or rely on methods they may feel uncomfortable doing.

The authors envisage the creation of an uncertainty compiler through a grassroots effort. Such an effort would require creating a standard interface to be used for all languages and a translation syntax that tells the compiler what components of the code need to be changed. Once this has been created, for any new language that would be supported, the language would require packages and libraries with uncertain objects and uncertainty propagation algorithms and a translation file that told the compiler what translations it would need to perform. Once a framework is in place, user-developers could create uncertainty compilers in whatever languages they wanted to use. One could imagine that such an endeavour would be similar to the creation of NumPy where there was "a sense of building something consequential together for the benefit of many others" [79].

The complexity of creating these elements would depend on the exact specifications of the language in question. Languages such as C++ would require fewer changes to the source code as more of the calculations could be enabled through overloading and templates, whereas languages such as MATLAB would require more wholesale changes to the original code. This is not to say that an uncertainty compiler would be better for one language over another, uncertainty compilers could work with any language.

The general problem of uncertainty analysis is hard, and it is difficult to create software that comprehensively solves all these problems. The development of an uncertainty compiler is likely to be difficult. However, many practical problems are more straightforward than the most general problem, and when this is the case, it will be extremely useful to use a tool which is able to handle uncertainty analysis intrusively within code. Progress towards the creation of an uncertainty compiler would benefit both the scientific computing and uncertainty quantification communities even if the compiler itself remains elusive.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgements

This work has been funded by the Engineering and Physical Science Research Council (EPSRC) through the programme grant "Digital twins for improved dynamic design", EP/R006768/1.

[8] These numbers are for entirely illustrative purposes and should not prime readers' expectations for the increased computational burden PBA methods introduce.

References

[1] B.M. Adams, W.J. Bohnhoff, K.R. Dalbey, J.P. Eddy, M.S. Eldred, D.M. Gay, K. Haskell, P.D. Hough, L.P. Swiler, DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 5.0 Reference Manual, Tech. Rep., Sandia National Laboratories, Albuquerque, NM, United States, 2010.

[2] E. Patelli, COSSAN: a multidisciplinary software suite for uncertainty quantification and risk management, in: R. Ghanem, D. Higdon, H. Owhadi (Eds.), Handbook of Uncertainty Quantification, Springer International Publishing, Cham, 2015, pp. 1–69.

[3] S. Marelli, B. Sudret, UQLab: a framework for uncertainty quantification in Matlab, in: Vulnerability, Uncertainty, and Risk, American Society of Civil Engineers, Liverpool, UK, 2014, pp. 2554–2563.

[4] A. Olivier, D.G. Giovanis, B. Aakash, M. Chauhan, L. Vandanapu, M.D. Shields, UQpy: a general purpose Python package and development environment for uncertainty quantification, J. Comput. Sci. 47 (2020) 101204, https://doi.org/10.1016/j.jocs.2020.101204.

[5] S. Ferson, What Monte Carlo methods cannot do, Hum. Ecol. Risk Assess. 2 (4) (1996) 990–1007, https://doi.org/10.1080/10807039609383659.

[6] S. Ferson, L.R. Ginzburg, Different methods are needed to propagate ignorance and variability, Reliab. Eng. Syst. Saf. 54 (2–3) (1996) 21.

[7] W. Oberkampf, Simulation informed decision making [Conference Presentation], in: Virtual Conference on Epistemic Uncertainty in Engineering, 2021, https://www.youtube.com/watch?v=i4L3fUpr59s.

[8] E. Paté-Cornell, On "black swans" and "perfect storms": risk analysis and management when statistics are not enough, Risk Anal. 32 (11) (2012), https://doi.org/10.1111/j.1539-6924.2011.01787.x.

[9] R.P. Feynman, Appendix F - Personal observations on reliability of shuttle, in: Report of the Presidential Commission on the Space Shuttle Challenger Accident, vol. 2, US Government Printing Office, Washington, DC, USA, 1986, https://history.nasa.gov/rogersrep/v2appf.htm.

[10] Y. Amano, The Fukushima Daiichi Accident Report by the Director General, Tech. Rep., International Atomic Energy Agency, Vienna, Austria, 2015.

[11] P.K. Trivedi, D.M. Zimmer, Copula modeling: an introduction for practitioners, Found Trends Econom. 1 (1) (2006) 1–111, https://doi.org/10.1561/0800000005.

[12] A. Gray, D. Hose, M. De Angelis, M. Hanss, S. Ferson, Dependent possibilistic arithmetic using copulas, in: Proceedings of the Twelfth International Symposium on Imprecise Probabilities: Theories and Applications, vol. 147, Proceedings of Machine Learning Research, Granada, Spain (Virtual), 2021, pp. 173–183.

[13] M.D. McKay, R.J. Beckman, W.J. Conover, A comparison of three methods for selecting values of input variables in the analysis of output from a computer code, Technometrics 21 (2) (1979) 239–245, https://doi.org/10.2307/1268522, American Statistical Association.

[14] J. Helton, F. Davis, Latin hypercube sampling and the propagation of uncertainty in analyses of complex systems, Reliab. Eng. Syst. Saf. 81 (1) (2003) 23–69, https://doi.org/10.1016/S0951-8320(03)00058-9.

[15] M.D. Shields, J. Zhang, The generalization of Latin hypercube sampling, Reliab. Eng. Syst. Saf. 148 (2016) 96–108, https://doi.org/10.1016/j.ress.2015.12.002.

[16] M.D. Shields, K. Teferra, A. Hapij, R.P. Daddazio, Refined Stratified Sampling for efficient Monte Carlo based uncertainty quantification, Reliab. Eng. Syst. Saf. 142 (2015) 310–325, https://doi.org/10.1016/j.ress.2015.05.023.

[17] J. Jakeman, M. Eldred, D. Xiu, Numerical approach for quantification of epistemic uncertainty, J. Comput. Phys. 229 (12) (2010) 4648–4663, https://doi.org/10.1016/j.jcp.2010.03.003.

[18] S. Yang, F. Xiong, F. Wang, Polynomial chaos expansion for probabilistic uncertainty propagation, in: J.P. Hessling (Ed.), Uncertainty Quantification and Model Calibration, InTech, 2017.

[19] M. Eldred, Recent advances in non-intrusive polynomial chaos and stochastic collocation methods for uncertainty analysis and design, in: 50th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, American Institute of Aeronautics and Astronautics, Palm Springs, California, 2009.

[20] D.P. Kroese, T. Brereton, T. Taimre, Z.I. Botev, Why the Monte Carlo method is so important today, WIREs Comput. Stat. 6 (6) (2014) 386–392, https://doi.org/10.1002/wics.1314.

[21] M. Beer, S. Ferson, V. Kreinovich, Imprecise probabilities in engineering analyses, Mech. Syst. Signal Process. 37 (1–2) (2013) 4–29, https://doi.org/10.1016/j.ymssp.2013.01.024.

[22] M.S. Balch, R. Martin, S. Ferson, Satellite conjunction analysis and the false confidence theorem, Proc. R. Soc. A, Math. Phys. Eng. Sci. 475 (2019) 20180565, https://doi.org/10.1098/rspa.2018.0565.

[23] B. Kirk, Overview of Monte Carlo radiation transport codes, Radiat. Meas. 45 (10) (2010) 1318–1322, https://doi.org/10.1016/j.radmeas.2010.05.037.

[24] A.J. McKane, T.J. Newman, Stochastic models in population biology and their deterministic analogs, Phys. Rev. E 70 (4) (2004) 041902, https://doi.org/10.1103/PhysRevE.70.041902.

[25] M.A. Burgman, S. Ferson, H.R. Akcakaya, Risk Assessment in Conservation Biology, 1st edition, Population and Community Biology Series, vol. 12, Chapman & Hall, London; New York, 1993.

[26] N. Le Carrer, Possibly extreme, probably not: is possibility theory the route for risk-averse decision-making?, Atmos. Sci. Lett. 22 (7) (Jul. 2021), https://doi.org/10.1002/asl.1030.

[27] S. Ferson, V. Kreinovich, L. Ginzburg, D.S. Myers, K. Sentz, Constructing Probability Boxes and Dempster-Shafer Structures, Tech. Rep. January, Sandia National Laboratories, Albuquerque, NM, United States, 2003.

[28] S. Ferson, R.B. Nelsen, J. Hajagos, D.J. Berleant, J. Zhang, W.T. Tucker, L.R. Ginzburg, W.L. Oberkampf, Dependence in probabilistic modeling, Dempster-Shafer theory, and probability bounds analysis, Tech. Rep. 19094, Sandia National Laboratories, Albuquerque, NM, USA, 2004.

[29] N. Gray, S. Ferson, M. De Angelis, A. Gray, F. Baumont de Oliveira, Probability bounds analysis for Python, Softw. Impacts 12 (2022) 100246, https://doi.org/10.1016/j.simpa.2022.100246.

[30] M. Haenggi, Meta distributions–part 1: definition and examples, IEEE Commun. Lett. (2021) 1, https://doi.org/10.1109/LCOMM.2021.3069662.

[31] M. Haenggi, Meta distributions–part 2: properties and interpretations, IEEE Commun. Lett. (2021) 1, https://doi.org/10.1109/LCOMM.2021.3069681.

[32] H.E. Kyburg, Higher order probabilities and intervals, Int. J. Approx. Reason. 2 (3) (1988) 195–209, https://doi.org/10.1016/0888-613X(88)90116-8.

[33] J.G. Dijkman, H.V.A.N. Haeringen, S.J.D.E. Lange, Fuzzy Numbers, Fuzzy numbers, J. Math. Anal. Appl. 92 (1983) 301–341.

[34] D. Dubois, H. Prade, Interval-valued fuzzy sets, possibility theory and imprecise probability, in: Proceedings of the Joint 4th Conference of the European Society for Fuzzy Logic and Technology and the 11th Rencontres Francophones Sur La Logique Floue et Ses Applications, Barcelona, Spain, 2005.

[35] M.S. Balch, New two-sided confidence intervals for binomial inference derived using Walley's imprecise posterior likelihood as a test statistic, Int. J. Approx. Reason. 123 (2020) 77–98, https://doi.org/10.1016/j.ijar.2020.05.005.

[36] Y. Ben-Haim, Info-Gap Decision Theory: Decisions Under Severe Uncertainty, 2nd edition, Academic Press, Oxford, UK, 2006.

[37] R.E. Moore, R.B. Kearfott, M.J. Cloud, Introduction to Interval Analysis, vol. 110, Society for Industrial and Applied Mathematics, Philadelphia, USA, 2009.

[38] A.H.-S. Ang, W.S. Tang, Probability Concepts in Engineering: Emphasis on Applications in Civil & Environmental Engineering, 2nd edition, John Wiley & Sons, Ltd, Hoboken, N.J., USA, 2007.

[39] W.F. Mascarenhas, Moore: interval arithmetic in C++20, in: G.A. Barreto, R. Coelho (Eds.), 37th Conference of the North American Fuzzy Information Processing Society, vol. 831, Springer International Publishing, Fortaleza, Brazil, 2018, pp. 519–529.

[40] Probability Bounds Analysis for MATLAB, https://github.com/Institute-for-Risk-and-Uncertainty/pba-for-matlab.

[41] Probability Bounds Analysis for R, https://github.com/ScottFerson/pba.r.
[42] HYRISK, https://cran.r-project.org/web/packages/HYRISK/index.html.
[43] A. Gray, S. Ferson, E. Patelli, ProbabilityBoundsAnalysis.jl: arithmetic with sets of distributions, in: JuliaCon, Virtual Conference, 2021, p. 12.
[44] B. Everitt, A. Krondal, The Cambridge Dictionary of Statistics, 4th edition, Cambridge University Press, Cambridge, United Kingdom, 2010.
[45] S. Ferson, RAMAS Risk Calc 4.0 Software: Risk Assessment with Uncertain Numbers, Lewis Publishers, Boca Raton, Florida, USA, 2002, https://books.google.co.uk/books?id=tKz7UZRs0CEC.
[46] V. Kreinovich, Decision making under interval uncertainty (and beyond), in: P. Guo, W. Pedrycz (Eds.), Human-Centric Decision-Making Models for Social Sciences, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 163–193.
[47] L.A. Zadeh, Fuzzy logic, Computer 21 (4) (1988) 83–93, https://doi.org/10.1109/2.53.
[48] D.A. Perez, H. Gietler, H. Zangl, Automatic uncertainty propagation based on the unscented transform, in: 2020 IEEE International Instrumentation and Measurement Technology Conference (I2MTC), IEEE, Dubrovnik, Croatia, 2020, pp. 1–6.
[49] P. Embrechts, F. Lindskog, A. Mcneil, Modelling dependence with copulas and applications to risk management, in: Handbook of Heavy Tailed Distributions in Finance, Elsevier, 2003, pp. 329–384.
[50] R.B. Nelsen, An Introduction to Copulas, 2nd edition, Springer Series in Statistics, Springer, New York, New York, USA, 2006.
[51] H. Joe, Dependence Modeling with Copulas, Chapman & Hall/CRC, Boca Raton, Florida, USA, 2014.
[52] S. Ferson, J. O'Rawe, A. Antonenko, J. Siegrist, J. Mickley, C.C. Luhmann, K. Sentz, A.M. Finkel, Natural language of uncertainty: numeric hedge words, Int. J. Approx. Reason. 57 (2015) 19–39, https://doi.org/10.1016/J.IJAR.2014.11.003.
[53] D. Hose, M. Hanss, A universal approach to imprecise probabilities in possibility theory, Int. J. Approx. Reason. 133 (2021) 133–158, https://doi.org/10.1016/j.ijar.2021.03.010.
[54] E. Phipps, R. Pawlowski, Efficient expression templates for operator overloading-based automatic differentiation, http://arxiv.org/abs/1205.3506, May 2012, arXiv:1205.3506.
[55] E. Phipps, R. Pawlowski, C. Trott, Automatic differentiation of C++ codes on emerging manycore architectures with sacado, ACM Trans. Math. Softw. 48 (4) (2022) 1–29, https://doi.org/10.1145/3560262.
[56] T. Parr, The Definitive ANTLR 4 Reference, The Pragmatic Bookshelf, Dallas, USA, 2012.
[57] L.H. De Figueiredo, J. Stolfi, Affine arithmetic: concepts and applications, Numer. Algorithms 37 (2004) 147–158.
[58] E. Goubault, S. Putot, A zonotopic framework for functional abstractions, Form. Methods Syst. Des. 47 (3) (2016) 302–360, https://doi.org/10.1007/s10703-015-0238-z.
[59] A. Gray, M. De Angelis, S. Ferson, E. Patelli, What's Z-X, when Z = X+Y? Dependency tracking in interval arithmetic with bivariate sets, in: 9th International Workshop on Reliable Engineering Computations, Virtual Conference, 2021, pp. 27–28.
[60] W. Krämer, Generalized intervals and the dependency problem, Proc. Appl. Math. Mech. 684 (2006) 683–684, https://doi.org/10.1002/pamm.200610.
[61] N.S. Nedialkov, K.R. Jackson, G.F. Corliss, Validated solutions of initial value problems for ordinary differential equations, Appl. Math. Comput. (1999) 48.
[62] N.S. Nedialkov, K.R. Jackson, J.D. Pryce, An effective high-order interval method for validating existence and uniqueness of the solution of an IVP for an ODE, Reliab. Comput. 7 (6) (2001) 17.
[63] N. Nedialkov, Interval tools for ODEs and DAEs, in: 12th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN 2006), IEEE, Duisburg, Germany, 2006, p. 4.
[64] Y. Lin, M.A. Stadtherr, Validated solutions of initial value problems for parametric ODEs, Appl. Numer. Math. 57 (2007) 1145–1162, https://doi.org/10.1016/j.apnum.2006.10.006.
[65] J.A. Enszer, D.M. Andrei, M.A. Stadtherr, Probability bounds analysis for nonlinear population ecology models, Math. Biosci. 267 (2015) 97–108, https://doi.org/10.1016/j.mbs.2015.06.012.
[66] J. Rohn, Inverse interval matrix, SIAM J. Numer. Anal. 30 (3) (1993) 864–870, https://doi.org/10.1137/0730044.
[67] J. Rohn, R. Farhadsefat, Inverse interval matrix: a survey, Electron. J. Linear Algebra 22 (Jan. 2011), https://doi.org/10.13001/1081-3810.1468.
[68] A.T. Latipova, On solving optimization problems with inexact data, IFAC Proc. Vol. 46 (9) (2013) 1234–1239, https://doi.org/10.3182/20130619-3-RU-3018.00510.
[69] N. Le Carrer, P.L. Green, A possibilistic interpretation of ensemble forecasts: experiments on the imperfect Lorenz 96 system, Adv. Sci. Res. 17 (2020) 39–45, https://doi.org/10.5194/asr-17-39-2020.
[70] B. Stankewitz, N. Mücke, L. Rosasco, From inexact optimization to learning via gradient concentration, http://arxiv.org/abs/2106.05397, Nov. 2021, arXiv:2106.05397.
[71] B. Stankewitz, N. Mücke, L. Rosasco, From inexact optimization to learning via gradient concentration, Comput. Optim. Appl. 84 (1) (2023) 265–294, https://doi.org/10.1007/s10589-022-00408-5.
[72] F. Stonyakin, A. Tyurin, A. Gasnikov, P. Dvurechensky, A. Agafonov, D. Dvinskikh, M. Alkousa, D. Pasechnyuk, S. Artamonov, V. Piskunova, Inexact model: a framework for optimization and variational inequalities, Optim. Methods Softw. 36 (6) (2021) 1155–1201, https://doi.org/10.1080/10556788.2021.1924714.
[73] V. Kreinovich, L. Longpré, S.A. Starks, G. Xiang, J. Beck, R. Kandathi, A. Nayak, S. Ferson, J. Hajagos, Interval versions of statistical techniques with applications to environmental analysis, bioinformatics, and privacy in statistical databases, J. Comput. Appl. Math. 199 (2) (2007) 418–423, https://doi.org/10.1016/j.cam.2005.07.041.
[74] V. Kreinovich, Interval computations and interval-related statistical techniques: tools for estimating uncertainty of the results of data processing and indirect measurements, in: F. Pavese, A.B. Forbes (Eds.), Data Modeling for Metrology and Testing in Measurement Science, Birkhäuser Boston, Boston, 2009, pp. 1–29.
[75] K. Tretiak, S. Ferson, Should data ever be thrown away? Pooling interval-censored data sets with different precision, Int. J. Approx. Reason. 156 (2023) 114–133, https://doi.org/10.1016/j.ijar.2023.02.007.
[76] K. Tretiak, G. Schollmeyer, S. Ferson, Neural network model for imprecise regression with interval dependent variables, Neural Netw. 161 (2023) 550–564, https://doi.org/10.1016/j.neunet.2023.02.005.
[77] D.A. Plaisted, Source-to-source translation and software engineering, J. Softw. Eng. Appl. 06 (04) (2013) 30–40, https://doi.org/10.4236/jsea.2013.64A005.
[78] E. Adams, U. Kulisch, Scientific Computing with Automatic Result Verification, Academic Press, Boston, MA, USA, 1993.
[79] C.R. Harris, K.J. Millman, S.J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith, R. Kern, M. Picus, S. Hoyer, M.H. van Kerkwijk, M. Brett, A. Haldane, J.F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T.E. Oliphant, Array programming with NumPy, Nature 585 (7825) (2020) 357–362, https://doi.org/10.1038/s41586-020-2649-2.