Available online at www.eccomasproceedia.org
Eccomas Proceedia UNCECOMP (2019) 487-497

ECCOMAS
Proceedia

UNCECOMP 2019
3rd ECCOMAS Thematic Conference on
Uncertainty Quantification in Computational Sciences and Engineering
M. Papadrakakis, V. Papadopoulos, G. Stefanou (eds.)
Crete, Greece, 24-26 June 2019

COMPUTING WITH UNCERTAINTY: INTRODUCING PUFFIN THE AUTOMATIC UNCERTAINTY COMPILER

Nick Gray, Marco De Angelis and Scott Ferson

Institute for Risk and Uncertainty, University of Liverpool, United Kingdom
e-mail: nickgray@liverpool.ac.uk

Abstract. *Although engineers often recognise the advantages of applying uncertainty analysis to their complex simulations, they often lack the time, patience or expertise to undertake that analysis. We describe a software tool, named puffin, that takes existing code and converts in to uncertainty aware code in the same language making use of intrusive uncertainty propagation techniques. It can work either automatically or with user specification of the uncertainties involved in the system.*

Keywords: Uncertainty Quantification, Uncertainty Compiler,

ISSN:2623-3339 © 2019 The Authors. Published by Eccomas Proceedia.
Peer-review under responsibility of the organizing committee of UNCECOMP 2019.
doi: 10.7712/120219.6354.18702

1 INTRODUCTION

Modern engineering is all about numerical calculation, with the inexorable growth of computer power more of these calculations are being undertaken with ever more complex computer simulations. These developments means that new technologies, as digital twins [1], have begun to be explored. Engineers need to make calculations even when there is uncertainty about the quantities involved.

There are two types of uncertainty *aleatory* and *epistemic* with in the numerical calculations essential to engineering. Aleatory uncertainty arises from the natural variability in dynamical environments and material properties, errors in manufacturing processes or inconsistencies in the realisation of systems. Aleatory uncertainty cannot be reduced by empirical effort. Epistemic uncertainty is caused by measurement imperfections or lack of perfect knowledge of a system. This could be due to not knowing the full specification of a system in the early phases of engineering design.

Imperfect scientific understanding of the underlying physics or biology involved, would cause uncertainty in the future performance of a system even after the design specifications have been decided. If uncertainties are small they can often be neglected or swept away by looking at the worst-case scenarios. However, in situations where the uncertainty is large, or would affect an engineering decision, this approach is suboptimal or impossible. Instead, a comprehensive strategy of accounting for the two kinds of uncertainty is needed that can propagate imprecise and variable numerical information through calculations.

Many engineers work with legacy computer codes that do not take full account of uncertainties. Because analysts are typically unwilling to rewrite their codes, various simple strategies have been used to remedy the problem, such as elaborate sensitivity studies or wrapping the program in a Monte Carlo loop. These approaches treat the program 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.

Strategies are needed that automatically translate original source into code with appropriate uncertainty representations and propagation algorithms. We have developed an uncertainty compiler for this purpose, named Puffin[1], along with an associated language. It handles the specifications of input uncertainties and inserts calls to an object-oriented library of intrusive uncertainty quantification (UQ) algorithms. We use ANTLR [2], a parser/lexer generator, and Python to translate *uncertainty näive* code into code with a full account of uncertainty in the same language. In theory, the approach could work with any computer language. We currently support Python and later versions will handle FORTRAN, C, R and MATLAB languages.

2 PUFFIN LANGUAGE

In order to develop Puffin it was first essential to build an uncertainty language. Puffin language enables users to specify the uncertainties involved in their code before compiling it into pre-existing scripts. Currently enables uncertainty analysis with five types:

- Interval (unknown value or values for which sure bounds are known), [3]
- Probability distribution (random values varying according to specified law such as normal, lognormal, Weibull, etc., with known parameters),

[1]In ornithology puffins belong to the family auks, as we are making an automatic uncertainty compiler (auc) puffin seemed like a fitting name

- P-box (random values for which the probability distribution cannot be specified exactly but can be bounded),[4]
- Confidence box (confidence structure that is a representation of inferential uncertainty about a parameter compatible with both Bayesian and frequentist paradigms), [5]
- Natural language expressions (such as *about 7.2* or *at most 12*)

We are also planning other . For each language that the compiler is to support a library of intrusive UQ code is required that will allow these types of numbers to be freely mixed together in mathematical expressions to reflect what is known about each quantity. Such libraries already exist for MATLAB and R and a python equivalent is currently in active development.

In Puffin language, if compiling into languages with immutable values and editable variables, -> will be used for immutable values and = for editable variables, in languages where this isn't the case they can be used interchangeably. # are used for comments. Guillemets surround code snippits from the target language. Both single and double quotation marks can be used in Puffin, although if the target language is pernickety about which one is used then the user will have to be aware of this themselves.

2.1 Intervals

An interval is an uncertain number representing values obeying 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. Interval arithmetic computes with ranges of possible values, as if many separate calculations were made under different scenarios. However, the actual computations the software does are made all at once, so they are very efficient. As shown in Figure 1, there are several different formats for specifying intervals. All types of intervals are defined using square brackets, this simplest definition is for the lower bound and upper bound to be comma separated within the square brackets. Plus minus intervals can be defined with either a positive number or a percentage. They can also be defined by a single number within the brackets in which case the significant digits are used for the bounds of the interval. There may sometimes be uncertainty about the endpoints, this can be specified using nested intervals such as shown in line 5

```
[1] a -> [1,2]
[2] b -> [1±2]          #[-1,3]
[3] c -> [1±2%]         #[0.98,1.02]
[4] d -> [1.0]          #[0.95,1.05]
[5] e -> [[0,1],[2,3]]  #[0,3]
```

Figure 1: Syntax for defining intervals in Puffin language, $\pm$ can be substituted with +- or -+. The comments show what the interval is taken to be when compiling.

2.2 Distributions and P-Boxes

Probability distributions are specified by their shape and parameters, such as gaussian(5,1), uniform(0,9), or weibull(3,6). A non-exhaustive list of distributions available in the langauge is shown in table 1, however we are planning to add more. As with all keywords in Puffin they can be defined in either all caps, all lower or sentence case. For distributions with common short names then these will also be accessible, for example *N* for the normal distribution. P-boxes can be specified as probability distributions with intervals for one or more of their parameters. If the shape of the underlying distribution is not known, but some parameters such as the mean, mode, variance, etc. can be specified (or given as intervals), the software will construct distribution-free p-boxes whose bounds are guaranteed to enclose the unknown distribution subject to constraints specified.

Bernoulli	beta	binomial	Cauchy
chi-squared	delta	empirical distributions	exponential
F distribution	Frechet	gamma	geometric
Gaussian	Gumbel	Laplace	logistic
lognormal	logtriangular	normal	Pareto
Pascal	Poisson	power function	rayleigh
reciprocal	Simpson	Student-t	trapzoidal
triangular	uniform	Wakeby	Weibul

Table 1: Some of the distributions available in the uncertainty language

Probability bounds analysis integrates interval analysis and probabilistic convolutions which are often implemented with Monte Carlo simulations. It uses p-boxes, which are bounds around probability distributions, to simultaneously represent the aleatory uncertainty about a quantity and the epistemic uncertainty about the nature of that variability. Probability distributions are special cases of p-boxes, so one can do a traditional probabilistic analysis with the add-in as well. The calculations the software does are very efficient and do not require Monte Carlo replications.

Figure 2 shows several difference distribution and p-box assignments.

```
a -> t(1,2)     #student-t distribution
b -> beta(2,3)
c -> normal([0±0.5],1)
d -> U([1,2],3) #Uniform distribution
```

Figure 2: Syntax for defining p-boxes in Puffin language

2.3 C-Boxes

Confidence boxes (c-boxes) are imprecise generalisations of traditional confidence distributions, which, like Student's t–distribution, encode frequentist confidence intervals for parameters of interest at every confidence level. They are analogous to Bayesian posterior distributions in that they characterise the inferential uncertainty about distribution parameters estimated from sparse or imprecise sample data, but they have a purely frequentist interpretation that makes them useful in engineering because they offer a guarantee of statistical performance through repeated use. Unlike confidence intervals which cannot usually be used in mathematical calculations, c-boxes can be propagated through mathematical expressions using the ordinary machinery of probability bounds analysis, and this allows analysts to compute with confidence, both figuratively and literally, because the results also have the same confidence interpretation. For instance, they can be used to compute probability boxes for both prediction and tolerance distributions. C–boxes can be computed in a variety of ways directly from random sample data. There are c-boxes both for parametric problems (where the family of the underlying distribution from which the data were randomly generated is known to be normal, lognormal, exponential, binomial, Poisson, etc.), and for nonparametric problems in which the shape of the underlying distribution is unknown. Confidence boxes account for the uncertainty about a parameter that comes from the inference from observations, including the effect of small sample size, but also the effects of imprecision in the data and demographic uncertainty which arises from trying to characterise a continuous parameter from discrete data observations.

In Puffin language, c-boxes can be defined using dot notation shown in Figure 3. All distributions that work with p-boxes are also available in c-box form.

```
[1] a -> cbox.uniform([0,1],[2,3])
[2] b -> cbox.beta(2,3)
[3] c -> cbox.edf(X,Y)
```

Figure 3: Syntax for defining c-boxes in Puffin language. Line 3 shows the definition from an empirical distribution function where X and Y would represent arrays of data

2.4 Hedge Words

In order to make uncertainty analysis as simple as possible for the end user Puffin language allows for users to be able to input their uncertainties using natural language expressions such as *about* or *almost*. Table 2, lists some the allowed hedge words and their possible interpretations.

Hedge words can be interpreted as intervals [6] or or c-boxes. [7]

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 *or* from x to y	$[x, y]$
x out of y	cbox.beta(a,b)

Table 2: Hedge expressions and their mathematical equivalent. Note: d is the number of significant figures of x

2.5 Dependence assumptions

By default, the language assumes that each newly specified probability distribution or p-box is stochastically independent of every other. Users can change this assumption by specifying nature of the dependence using the syntax shown in Figure 4.

In addition, Puffin language automatically tracks calculations that were used to compute uncertain numbers and will modify the default assumption of independence if appropriate. For instance, an increasing monotone function (such as log, exp, and sqrt) of a distribution creates an uncertain number that is perfectly dependent on the original distribution. Reciprocation creates an uncertain number that is oppositely dependent on the original distribution. When the function that transforms an uncertain number is complex and the relationship between the original distribution and the result cannot be educed, the two are assigned the unknown dependence. If the two later are used in a calculation, Fréchet convolution, which makes no assumption about the dependence between the arguments, is used to combine them. Fréchet convolution must be used because an assumption of independence would be untenable, because one argument is a direct function of the other. Generally, Fréchet convolution creates p-boxes from precise probability distributions, or widens the results from p-boxes relative to convolutions that assume independence or some other precise dependence function. The extra width represents the additional uncertainty arising from not knowing the dependence function. Users can countermand the languages automatic tracking of dependence and specify the assumption to be used in any particular convolution.Âż

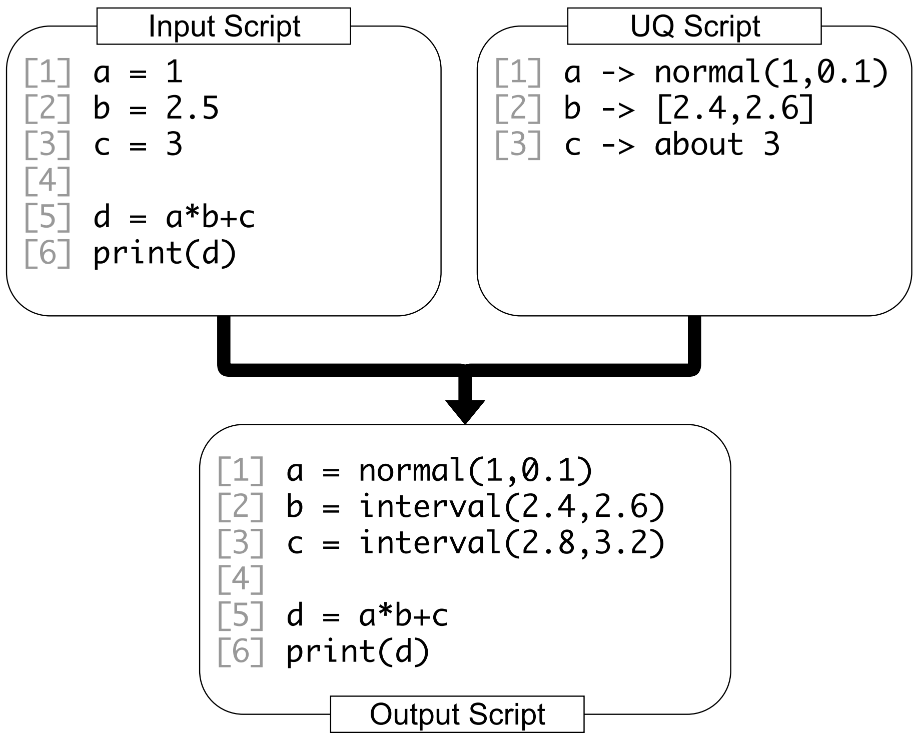
```
a -> uniform(0,1) !dep(b)
b -> normal(2,3) !dep(c)
c -> normal(4,5) !dep(a,b)
```

Figure 4: Syntax for adding dependance between variables in Puffin Language

3 PUFFIN COMPILER

The process for taking the uncertainty naive code and adding in appropriate uncertainty analysis can be done in two different ways: the automatic approach and the second is to specify the uncertainty using the language described above. Currently the compiler can only be used from the terminal or command line. We are planning on developing a user interface for the compiler that is based on the open source *winmerge*[2] software. It will allow the user to be able to see the differences between the original and uncertainty code.

To be perhaps more useful the second method allows the end user to specify the uncertainty manually using the uncertainty language described above. It is possible to generate the Puffin langauge file by running the *–getpuffin* command when using the compiler, this will parse over the file and get all the variable declarations within the script. Figure 5 shows an example of using the compiler.



```
a = 1
b = 2.5
c = 3

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

```
a -> normal(1,0.1)
b -> [2.4,2.6]
c -> about 3
```

```
a = normal(1,0.1)
b = interval(2.4,2.6)
c = interval(2.8,3.2)

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

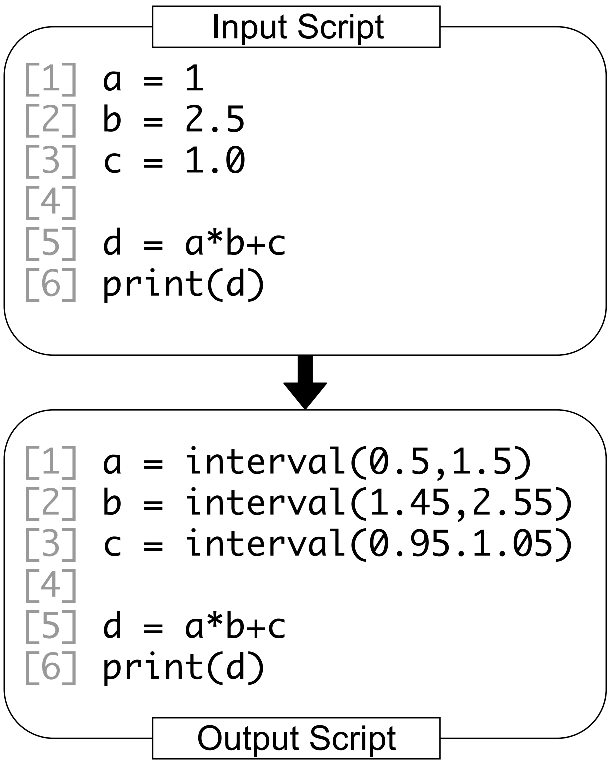
Figure 5: The result of using the compiler whilst defining the uncertainty in Puffin langauge on a simple pseudocode script.

3.1 Automatic Uncertainty Analysis

The automatic approach takes the significant figures of the assignments and uses that information as a proxy for the uncertainty for an example see Figure 3.1. When using this mode the compiler will need to *tread carefully* around mathematical constants such as π or e for which there is no uncertainty. For example if the variable had value equal to 3.14159 then it would be pretty clear that it is referring to the mathematical constant however if the value was 3.1 then it could be ambiguous. 3.141 could also cause problems, the correct rounding of π to 3 decimal places is 3.142 however 3.141 is so ubiquitous as the start of π that it would be a simple error

[2]winmerge.org/

for an analysis to make when creating code.



```
a = 1
b = 2.5
c = 1.0

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

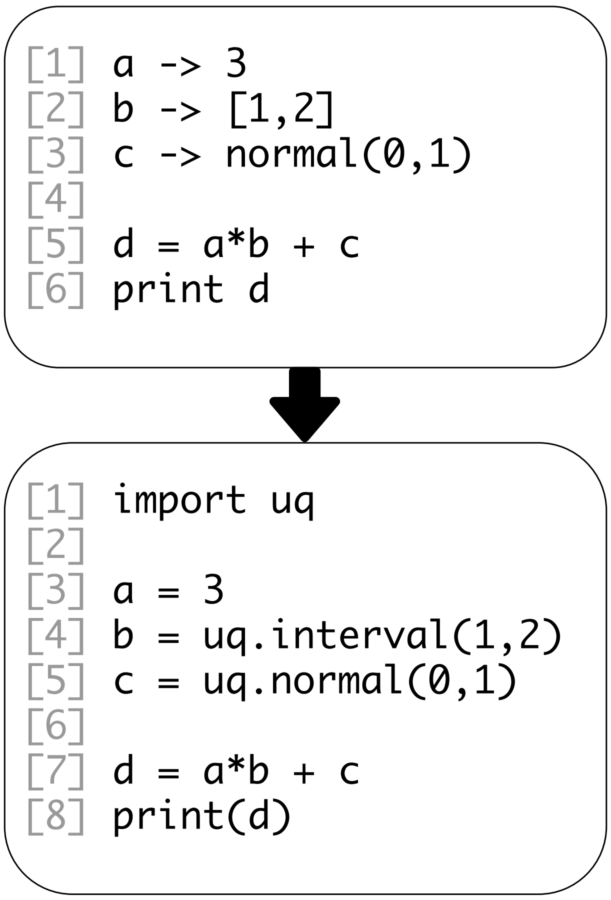
```
a = interval(0.5,1.5)
b = interval(1.45,2.55)
c = interval(0.95.1.05)

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

Figure 6: The result of using the compiler in automatic mode on a simple pseudocode script.

3.2 Direct Compiler

Once Puffin language has been fully developed we are intending to create a direct compiler that allows creation of scripts in the code. Initially the compiler will turn the Puffin code directly into Python 3 code. Figure 7 shows direct translation from Puffin language to Python.



```
a -> 3
b -> [1,2]
c -> normal(0,1)

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

```
import uq

a = 3
b = uq.interval(1,2)
c = uq.normal(0,1)

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

Figure 7: Direct translation from Puffin language to Python 3

4 REPEATED VARIABLE PROBLEM

A limitation of using the Puffin compiler to incorporate uncertainty analysis into numerical calculations arise from multiple occurrences of an uncertain variable in a mathematical expression. Let $a = [1, 2]$, $b = [-1, 1]$ and $c = [3, 4]$. Applying interval arithmetic naively gives

$$ab + ac = [1, 10] \tag{1}$$

but also

$$a(b + c) = [2, 10] \tag{2}$$

One would expect that the results of equation 1 and equation 2 would be the same as, algebraically, $ab + ac \equiv a(b + c)$ however the distributive law of real numbers does not generally hold for uncertain numbers. In the case of intervals, the expression with repeated uncertain quantities may be wider than the one with no such repetitions, even when they would be equivalent for real values, the uncertain number appearing twice in the first formulation means, in effect, the uncertainty it represents is entered twice into the resulting calculation.

This problem besets most uncertainty quantification methods, although an advantage of Monte Carlo methods is that they can escape this problem. This uncertainty inflation would also occur if a calculation is conducted in multiple steps. For instance, if the first term ab in the example above is calculated on one line and on a new line ac is calculated before the final sum is calculated in a third line, the uncertainty of a will have been introduced into the final result twice leading to the inflated uncertainty shown in equation 1.

If possible, the number of repetitions of uncertain variables should be reduced by algebraic manipulation to avoid possible inflation of the uncertainty. This would apply whether the uncertain parameter is an interval, distribution, p-box or c-box. It should be noted that only the repeated variable matters when reducing the expression because other variables can be as arbitrarily complex, with as many repeats required. For instance if x is the only uncertain in equation 3 then the fact that b is repeated five times is irrelevant.

$$(a+bx)^n(c+dx)^n = \left(\frac{(2bdx+ad+bc)^2-(ad+bc)^2+4abcd}{4bd}\right)^n \tag{3}$$

Unfortunately, it is not always possible to reduce all multiple occurrences. For example, equation 4 cannot be reduced to a single instance of x. In such cases, special or ad hoc strategies must be devised, even partial solutions improve calculations.

$$x^3+x^2+x+1 \tag{4}$$

When computing with intervals, we are guaranteed that the result of a computation with repeated variables will have width no smaller than the correct answer. Therefore, even if multiple occurrences of the variable cannot be reduced, a conservative estimate of the final value can still be calculated. In risk assessment such an estimate may meet the practical needs of an analysis. For probability distributions and p-boxes, this guarantee holds when using FrÃl'chet convolutions, however it does not extend to cases where independence has been assumed or precise dependancies have been specified between the variables. The size of any error cause by repeated variables is dependant on the particulars of the mathematical expression as well as the quantities involved

A useful extension to the compiler would be to automatically detect and simplify mathematical expressions with repeated uncertain variables, even over multiple lines, in a way that reduces the repetitions of parameters containing uncertainty or in situations where they cannot be simplified then display a warning to the user. An example of how this might look when generating the Puffin langauge file form a script is show in Figure 8. Although this problem is known to be NP-hard in general, software strategies can be designed to find expressions with fewer repetitions of the same variable. In instances where no solution could be found then the compiler should issue appropriate warnings to the user. We are exploring a strategy that repeatedly applies mathematical identities that reduce the number of appearances of uncertain parameters. There are many such reducing templates. The approach is to parse an expression into a binary tree, and search for matches with a reducing template in each subtree. The search is iterated over all the templates and over all subtrees, and it is repeated until no further reduction occurs. To shorten the list of reducing templates, the matching algorithms automatically test multiple rearrangements of the subtree that are implied by associativity and commutativity of basic operators.

```
#! Automatically reduced number of repetitions
#! of variable x in line:
x = x*a+x*b -> x = x*(a+b)

#! Automatically reduced number of repetitions
#! of variables x and y in line:
z = (x+y)/(1-xy) -> z = tan(arctan(x)+arctan(y))

#!! Can't find repeated variable reduction for x
#!! in line:
z = (a+x)/(b+x)
#!! May cause artificial uncertainty inflation
```

Figure 8: Example Puffin syntax showing how a generated Puffin file could highlight repeated variables to the user and automatically reduce where possible

Acknowledgement

The authors would like to acknowledge the support of the UK Engineering and Physical Science Research Council through grant number EP/R006768/1

REFERENCES

[1] Mike Shafto, Mike Conroy, Rich Doyle, Ed Glaessgen, Chris Kemp, Jacqueline LeMoigne, and Lui Wang. Modeling, Simulation, Information Technology and Processing Roadmap - Technology Area 11. Technical report, National Aeronautics and Space Administration, 2012.

[2] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, Dallas, USA, 2012.

[3] Ramon E Moore, R Baker Kearfott, and Michael J Cloud. *Introduction to Interval Analysis*, volume 110. Society for Industrial and Applied Mathematics, Philadelphia, USA, 2009.

[4] Scott Ferson, Vladik Kreinovich, Lev Ginzburg, Davis S Myers, and Kari Sentz. Constructing Probability Boxes and Dempster-Shafer Structures. Technical Report January, Sandia National Lab.(SNL-NM),, Albuquerque, United States, 2003.

[5] Michael Scott Balch. Mathematical foundations for a theory of confidence structures. *International Journal of Approximate Reasoning*, 53(7):1003–1019, 2012.

[6] Scott Ferson, Jason O'Rawe, Andrei Antonenko, Jack Siegrist, James Mickley, Christian C. Luhmann, Kari Sentz, and Adam M. Finkel. Natural language of uncertainty: numeric hedge words. *International Journal of Approximate Reasoning*, 57:19–39, feb 2015.

[7] Scott Ferson, Michael Balch, Kari Sentz, and Jack Siegrist. Computing with Confidence. In *Proceedings of the Eighth International Symposium on Imprecise Probability: Theory and Applications*, Compiègne, France, 2013.