A computational system for uncertainty propagation of measurement results

(short title: "A system for uncertainty propagation")

Luca Mari
Università Cattaneo – LIUC
C.so Matteotti, 22 – 21053 Castellanza (VA), Italy

Abstract

This paper discusses some design issues in the implementation of the law of uncertainty propagation according to an automatic differentiation strategy in the context of a simulation engine supporting the construction and the interactive testing of models of dynamic systems. The proposed solution propagates not only the partial derivatives, as usual in automatic differentiation, but also the input uncertainties, so to make their various modifications visible to the user of the evaluation system, and give him the opportunity to analyze the partial contributions to the standard uncertainty of the output measurand. A tool for uncertainty propagation in a general, user-oriented, computational system, instead of a software library or a dedicated system, makes uncertainty propagation transparently computable also for vector / matrix measurands, even in the case of dynamic systems, and makes uncertainty evaluation an inherent component of computational processes instead of an optional, ad hoc, addendum to them.

Keywords: uncertainty propagation; automatic differentiation; computational methods for measurement

1. Introduction

“There are certain human activities which apparently have perfect sharpness. The realm of mathematics and of logic is such a realm, par excellence. Here we have yes-no sharpness. But this yes-no sharpness is found only in the realm of things we say, as distinguished from the realm of things we do. Nothing that happens in the laboratory corresponds to the statement that a given point is either on a given line or it is not” [1].

While that measurement results are not “yes-no sharp” is a commonly accepted standpoint nowadays, the

epistemological background for this position is still highly debated, as the multiplicity of (not always clearly compatible) concepts in the *International Vocabulary of Metrology* [2] (2.13 measurement accuracy; 2.14 measurement trueness; 2.15 measurement precision; 2.16 measurement error; 2.26 measurement uncertainty, just to mention a few of them) witnesses. This confusion state did not prevent a wide agreement in the metrological community around a formal procedure to express "degrees of non-yes-no sharpness" in measurement, modeled in terms of uncertainties, as specified by the *Guide to the Expression of Uncertainty in Measurement* (GUM) [3]. The preliminary hypothesis for the application of this procedure is that many measurement processes include a data processing stage, aimed at computing a function $f : R^n \rightarrow R$ whose argument is a n-tuple $x_1, ..., x_n=[x_i]$ of input quantities, typically comprising some influence quantities, and whose value y is the (output) measurand. Since the values x_i have generally an uncertainty associated, the issue arises on how the function f propagates such uncertainty to the value y. The basic procedure recommended by the GUM to compute the function f, in this context called the "measurement function" ("function of quantities, the value of which, when calculated using known quantity values for the input quantities in a measurement model, is a measured quantity value of the output quantity in the measurement model" [2, 2.49], being a measurement model the "mathematical relation among all quantities known to be involved in a measurement" [2, 2.48], a peculiar terminology taken from [1, 3.1.6]), is based on the so called law of uncertainty propagation ("LUP" henceforth), which (i) requires each quantity to be expressed as a couple x=quantity value, $u(x)$=standard uncertainty of x, (ii) assumes that y and $u(y)$ are computed separately, and (iii) derives the standard uncertainty $u(y)$ of y from the first-order approximation of the Taylor series of the function f at $[x_i]$, i.e., in terms of the contributions $u(x_i)$, each of them weighed by the sensitivity coefficient $\partial f/\partial x_i$, i.e., the ratio of change of the function f at the n-dimensional point $[x_i]$ along the i-th dimension. Specifically, the LUP states that:

$$u(y) = \sqrt{\sum_{i=1}^{n} \left(\frac{\partial f}{\partial x_i} \right)^2 u^2(x_i)} \qquad (1)$$

under the hypothesis of null covariances, or:

$$u(y) = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n}\frac{\partial f}{\partial x_i}\frac{\partial f}{\partial x_j}u(x_i, x_j)} \tag{2}$$

equivalent to:

$$u(y) = \sqrt{\sum_{i=1}^{n}\left[\frac{\partial f}{\partial x_i}\right]^2 u^2(x_i) + 2\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{\partial f}{\partial x_i}\frac{\partial f}{\partial x_j}u(x_i, x_j)} \tag{2b}$$

if one or more covariances $u(x_i, x_j)$ are non null (by exploiting the fact that $u(x_i, x_j)=u(x_j, x_i)$), sometimes also written:

$$u(y) = \sqrt{\sum_{i=1}^{n}\left[\frac{\partial f}{\partial x_i}\right]^2 u^2(x_i) + 2\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{\partial f}{\partial x_i}\frac{\partial f}{\partial x_j}\rho(x_i, x_j)u(x_i)u(x_j)} \tag{2c}$$

where $\rho(x_i,x_j)\in[-1,1]$ is the correlation coefficient of x_i and x_j.

For a given measurement function f, provided that the standard uncertainties $u(x_i)$ and the covariances $u(x_i,x_j)$, or the correlation coefficients $\rho(x_i,x_j)$, are given (the hypotheses at the basis of the LUP hold independently of the technique, either "type A" or "type B" as denoted by the GUM, adopted to evaluate the standard uncertainties), Eq. (1) shows that the substantial problem of computing the LUP is to obtain the partial derivatives $\partial f/\partial x_i$. Two common methods to this goal are symbolic differentiation and numerical differentiation by finite differences. Both these techniques have known drawbacks. With the aim of overcoming them a computational technique known as *automatic differentiation* (AD) [4] has been proposed. AD exploits the possibility for many functions f to be decomposed into a sequence of elemental functions, any one of which may be trivially differentiated. These partial derivatives, evaluated at a given argument, can be then combined in accordance with the chain rule, as specified in derivative calculus, to generate the required derivative for f. It should be noted that this process yields exact (to numerical accuracy, i.e., round-offs) numerical values of derivatives. Because the symbolic transformation occurs only at the most basic level (a computer program could implement differentiation of elemental functions by a simple table lookup), AD avoids the computational problems inherent in complex symbolic computation.

The application of AD for computing the LUP has already been considered ([5], [6]), and some software

libraries implementing it have been developed (e.g., [7], who presents his work in terms of a specialized version of the Interpreter design pattern he calls “the GUM tree”, and [8]). It is also plausible that some kind of AD is adopted in software packages specialized in measurement uncertainty management, such as [9] (see [10]), that typically allow the user interactively specifying at least:

- the measurement function as a string that can contain variable symbols, standing for input quantities, together with the basic polynomial, trigonometric, … functions;
- the value of each input quantity and its standard uncertainty (either stated explicitly or computed from a probability distribution whose parameters are entered);
- the table of correlation coefficients,

and from this data compute the output measurand value and its uncertainty, both as combined standard uncertainty and expanded uncertainty provided that a coverage factor has been also specified.

This paper discusses some design issues arising in the implementation of an algorithm for computing the AD-based LUP in the context of a simulation engine [11] supporting the construction and the interactive testing of models of dynamic systems (at the address http://www.liuc.it/persone/lmari/stgraph such a system can be freely accessed / downloaded). The driving hypothesis is that the introduction of tools for uncertainty propagation in general, user-oriented, computational systems, instead of software libraries or dedicated systems, could allow making uncertainty evaluation an inherent component of computational processes instead of an optional, ad hoc, addendum to them, thus contributing to increase the awareness, even outside the metrological community, that experimental data should generally computed together with their uncertainty (hence as non “yes-no sharp” data). A specific problem is also particularly taken into account in this paper, and a possible solution is presented in the context of the mentioned software system: the AD-aware definition of non-derivable or even non-continuous elemental function, accomplished by a fuzzy smoothing at the threshold of different standard uncertainties.

2. Automatic differentiation for uncertainty propagation

The first steps to make a function *f* computable by an automatic executor, such as the interpreter of a programming language or a spreadsheet, are to parse its expression, recognize the tokens by which it is

constituted and organize them into a hierarchical structure, e.g., a parenthesized list or a tree structure whose leaves are the function arguments and the nodes are the elemental functions recognized in the tokenization stage. The same holds in the case in which together with the function also its partial derivatives must be computed. As simple example, consider at first a univariate function $f : R \to R$ such as $y=f(x)=\sin(x^2+1)$. This function is parsed to the tree shown in Figure 1 and then typically evaluated by means of a stack structure as in Figure 2 (to highlight that the evaluation process is numerical, instead of symbolic, a given value for the function argument will be assumed, e.g., $x=3.0$; note that measurement units will be maintained implicit from now on).

Figure 1 – Tree resulting from the parsing of a simple function.

Figure 2 – Evaluation trace of a simple function (to be read bottom-up).

Exactly the same structure and process are exploited by the so called Forward-Mode AD to compute $\partial f/\partial x$ in parallel to $y=f(x)$. Instead of as a scalar, each variable (the input quantity x, the intermediate results, the output measurand y) is now formalized as a non-primitive type, so to maintain not only the variable value but also the (partial) derivative of the computed function, i.e., the couple $\langle x, Dx\rangle$ (let us denote $\partial f/\partial x$ as Dx for short). The data pushed to and popped from the stack are now such couples. Correspondingly, the elemental functions must be extended to operate on such objects. In our example:

$$\begin{aligned} &\langle x, Dx\rangle^n = \langle x^n, nx^{n-1}Dx\rangle \\ &\langle x_1, Dx_1\rangle + \langle x_2, Dx_2\rangle = \langle x_1 + x_2, Dx_1 + Dx_2\rangle \\ &\sin(\langle x, Dx\rangle) = \langle \sin(x), \cos(x)Dx\rangle \end{aligned} \tag{3}$$

and so on, according to the chain rule of differentiation. The recursive evaluation process must be seeded by setting the derivative component of its inputs:

– for the variable x: $\langle x=3.0, Dx=1\rangle$ (the derivative of a variable with respect to itself is 1);

– for the constant 1: $\langle 1, D1=0\rangle$ (the derivative of a constant is null).

Hence in this case the evaluation trace becomes as shown in Figure 3.

Figure 3 – AD-enabled evaluation trace of a simple monadic function.

The output $\langle y=-0.54, Dy=-5.03\rangle$ is finally combined by the LUP with the standard uncertainty of the input quantity, let us assume for example $u(x)=0.1$, to obtain the standard uncertainty (called "combined uncertainty" by the GUM) of the output measurand. Eq. (1) leads in this case trivially to

$$u(y) = |Dy\, u(x)| = 0.50 .$$

This basic version must be extended to deal with $f : R^n \to R$ functions. In this case Eqs. (1) and (2) require all the n partial derivatives $\partial f/\partial x_i = D_i x$ to be computed, a process that can be effectively performed in parallel by maintaining together with each variable value the whole vector $[D_1 x, ..., D_n x]$ of its partial derivatives, thus extending the non-primitive type of the variables to $\langle x, [D_i x]\rangle$ (note that an increase in space, i.e., memory, efficiency would be obtained by performing n swaps, one for each partial derivative to be evaluated, thus reducing the logical parallelism of the algorithm we are going to present; we will adopt here the sequential version mainly for reasons of clarity in the presentation). The elemental functions must now compute all the elements of the derivative vector(s), e.g.:

$$\begin{aligned}\sin\left(\left\langle x,\left[D_i x\right]\right\rangle\right) &= \left\langle \sin(x),\left[\cos(x)D_1 x,\ldots,\cos(x)D_n x\right]\right\rangle = \\ &= \left\langle \sin(x),\cos(x)\left[D_i x\right]\right\rangle \end{aligned} \tag{4}$$

for a single argument function, and, e.g.:

$$\begin{aligned}\left\langle x_1,\left[D_i x_1\right]\right\rangle + \left\langle x_2,\left[D_i x_2\right]\right\rangle &= \left\langle x_1 + x_2,\left[D_1 x_1 + D_1 x_2,\ldots, D_n x_1 + D_n x_2\right]\right\rangle = \\ &= \left\langle x_1 + x_2,\left[D_i x_1 + D_i x_2\right]\right\rangle \end{aligned} \tag{4b}$$

for a dyadic function.

In the simple example of the function $y = f(x_1, x_2) = \sin\left(x_1^2 x_2\right)$, the evaluation process (assuming $x_1=3.0$ and $x_2=2.0$) is shown in Figure 4.

Figure 4 – AD-enabled evaluation trace of a simple dyadic function.

Together with the standard uncertainties $u(x_i)$ and covariances $u(x_i, x_j)$, the results of this process are finally fed into Eq. (1) or (2) to obtain the combined uncertainty $u(y)$. It should be noted that the initial values of the derivative vectors, by which the process is seeded, are in the example $\langle 1, 0\rangle$ for x_1 and $\langle 0, 1\rangle$

for x_2, and in general $D_i x_j = \delta_{ij}$ ($\delta_{ij}=1$ if $i=j$ and =0 otherwise). In the case an input quantity occurs more than once in the function expression, these seeds must be assigned consistently. For example, the function $y = f(x_1, x_2) = x_1 \sin(x_1^2 x_2)$ must be seeded as shown in Figure 5, a structure actually corresponding to the acyclic directed graph shown in Figure 6.

Figure 5 – AD seeding for a simple dyadic function represented as a tree.

Figure 6 – AD seeding for a simple dyadic function represented as a directed graph.

This graph gives the hint for a further extension of the AD-based uncertainty propagation process. It should be noted, indeed, that what is propagated through the graph are the partial derivatives, not the uncertainties, for which the graph itself remains a black box. On the other hand, it could be sometimes interesting to have also uncertainties propagated through the graph, so to make their progressive modifications visible to the user of the evaluation system, and give him the opportunity to introduce locally the possible covariances and to analyze the partial contributions to the final result, i.e., the standard uncertainty of the output measurand.

3. Propagating uncertainties through graphs

When a function f is complex, and possibly formalizes a multi-measurand / multivariate problem, $f : R^n \rightarrow R^m$, an effective technique to formalize and to compute it is to define it as a set of intermediate functions combined in an acyclic directed graph. This way, the computation of f is split into intermediate steps, one for each node of the graph. As a (trivial, indeed) example, the function $y=\sin(x_1)\cos(x_2)$ can be alternatively defined by the four graphs shown in Figure 7.

Figure 7 – Four possible definition graphs for the same dyadic function.

We will call them (user generated) *definition graphs* (d-graphs for short), in contrast to the (parser generated) *evaluation graphs* (e-graphs) introduced in the previous section. A d-graph is such that:

- each node represents a quantity or an intermediate function, which is in its turn parsed to an e-graph;
- each arrow from a node f_i to a node f_j represents the functional dependence of (the variable associated

with) f_j on (the variable associated with) f_i;

- nodes without incoming arrows ("input nodes") are associated with input quantities x_i;
- nodes without outgoing arrows ("output nodes") are associated with output quantities y_j.

Hence, (a) and (d) are respectively a black box and a white box d-graph, since (a) hides the whole function definition in the output node y, while (d) splits f in all possible intermediate functions.

In a simulation engine supporting the graph-based definition and editing of model functions, a non-black box d-graph (i.e., graphs (b), (c), and (d) in the example) allows the user to get information on the computational state of the intermediate functions, and therefore on partial results including, if suitably implemented, their uncertainty: a white box d-graph is indeed formally equivalent to the corresponding e-graph but, on the contrary of the e-graph, it allows the user to interact with intermediate functions. To this goal, each node f_i of a d-graph must be dealt with as an object characterized in each time instant not only by a value but also by a standard uncertainty and a covariance matrix, $f_i=\langle v_i, u(v_i), [u(v_j,v_k)]\rangle$. For implementing a multi-step LUP computation the following constraints are proposed, also on the basis of the experience acquired in the design and implementation work done in the context of the mentioned simulation engine we are developing:

C1. *uncertainties as inputs*: input nodes, and only input nodes, have both their value and standard uncertainty assigned as exogenous data, whereas their covariance matrix is identically null;

C2. *local definition of covariances*: non-input nodes functionally depending on $m \geq 2$ nodes have a $m \times m$ covariance matrix related to such nodes (as a trivial consequence, the covariance matrix of any node depending on a single node is null) and assigned as exogenous data.

The constraints C1 and C2 guarantee the distinction between input uncertainties and propagated uncertainties, and in particular prevent an already partially computed propagated uncertainty to be overridden by an input uncertainty.

Let us assume that the engine is already provided with a d-graph compliant evaluator, i.e., an executor able to evaluate the nodes f_i in the proper sequence (*), and that the implementation of the elemental

(*) Let $\{\boldsymbol{F}_j\}$ be a partition on the set $\boldsymbol{F}$ of nodes of a d-graph such that:
- $\boldsymbol{F}_0$ is the subset of input nodes;
- $\boldsymbol{F}_1$ is the subset of nodes directly connected only from input nodes, i.e., nodes in $\boldsymbol{F}_0$;

functions is already AD-aware, in the sense that the data they deal with are couples $\langle x,[D_i x]\rangle$, as discussed in the previous section. Such an evaluator is then suitably extended to compute the standard uncertainties of all non-input nodes of the d-graph by the following recursive procedure (Greek letters are references to the Java implementation, as presented in the Appendix):

- if f_i is an input node (recursion base):
  - get / evaluate the value v_i (standard procedure)
  - get the standard uncertainty $u(v_i)$ (exogenous data: see constraint C1) //α
  - store $f_i=\langle v_i, u(v_i)\rangle$ as node value //β
- else (f_i is a non-input node, directly depending on the m nodes $[f_j]$) (recursion step):
  - for each f_j:
    - get the value v_j (which includes standard uncertainty: see constraint C1)
    - generate the m seeds $D_k f_j=\delta_{kj}$ //γ
    - make $\langle v_j,[D_k f_j]\rangle$ available to the evaluator as value of f_j
  - evaluate the value $\langle v_i,[D_k f_i]\rangle$ (AD-aware procedure) //δ
  - given the m standard uncertainties $u(v_j)$, the m partial derivatives $D_k f_j$, and the $(m^2-m)/2$ covariance values $u(v_j,v_k)$ (covariances are exogenous data: see constraint C2):
    - compute $u(v_i)=\sqrt{\sum_{j=1}^{m}(D_j f_j)^2 u^2(v_j)+2\sum_{j=1}^{m-1}\sum_{k=j+1}^{m}(D_j f_j)(D_k f_k)u(v_j,v_k)}$ (a local equivalent of Eq. (2b)) //ε
  - store $f_i=\langle v_i, u(v_i)\rangle$ as node value //β

Hence, all graph nodes are recursively evaluated and the evaluation of the combined standard

---

- $\boldsymbol{F}_2$ is the subset of nodes not in $F_0 \cup F_1$ and directly connected only from nodes in $F_0 \cup F_1$;
- $\boldsymbol{F}_j$ is the subset of nodes not in $\bigcup_{i=0}^{j-1} F_i$ and directly connected only from nodes in $\bigcup_{i=0}^{j-1} F_i$,

the maximum index j such that $F_j \neq \varnothing$ being called the computational width w of the graph.

Hence, a function f is computed through the d-graph by w+1 macro-steps (each of them including one or more steps which can be in principle executed in parallel) as follows:

- macro-step 0: each input node in $\boldsymbol{F}_0$ is evaluated from its externally assigned value;
- macro-step j, j=1,...,w: each node in $\boldsymbol{F}_j$ is evaluated from the values of the nodes in $\bigcup_{i=0}^{j-1} F_i$.

uncertainties is completed.

As a simple case of this procedure, let us consider the evaluation of the function $y=\sin(x_1)\cos(x_2)$, for example for $x_1=1.0$, $u(x_1)=0.1$ and $x_2=2.0$, $u(x_2)=0.2$, from which the result $y=-0.35$, $u(y)=0.15$ is obtained. On the other hand, if the version (d) of the d-graph is implemented, the partial uncertainties for $z_1=\sin(x_1)$ and $z_2=\cos(x_2)$ can be obtained, $u(z_1)=0.05$ and $u(z_2)=0.18$, thus highlighting that the main contribution to the combined standard uncertainty comes from the second term.

A subtle problem must be finally considered, related to possible "hidden" covariances, generated by the introduction of the intermediate steps in the evaluation procedure. In a variation of the previous example, let us suppose that the function under evaluation is $y=\sin(x)\cos(x)$, by means of the d-graph shown in Figure 8.

Figure 8 – Definition graph of a monadic function, showing a case of the "hidden" covariances.

The given procedure gives the intermediate standard uncertainties $u(z_1)=0.05$ and $u(z_2)=0.08$, and the combined standard uncertainty $u(y)=0.08$, a wrong value (the correct one is 0.04) which does not take into account the fact that the intermediate variables z_1 and z_2 are indeed correlated because of their common dependence on x. Since the LUP is based on the first order approximation of the Taylor series of the measurement function, the correlation coefficient $\rho(z_1,z_2)$ is equal to either +1 or −1, dependently on the trend (i.e., the sign of the first derivative) of the involved functions at the point x, i.e., $\rho(z_1,z_2)=Z/|Z|$

where $Z=\frac{\partial z_1}{\partial x}\frac{\partial z_2}{\partial x}$. Once this "auto" correlation has been identified, Eq. (2c) is then particularly suitable to the evaluation procedure of the LUP.

4. Validation

The presented AD-based LUP implementation can be validated directly within the simulation engine by exploiting its flexibility in operating in parallel with different computational processes. Let us assume that a measurement function $Y=f(X_1,...,X_k)$ has been implemented, and that for each input quantity X_i an average value x_i and its standard uncertainty $u(x_i)$ are given. Moreover, let us hypothesize that the

population from which the values x_i and $u(x_i)$ have been obtained follows a known probability distribution. Hence, the same function f can be operated in three distinct cases:

- *"theoretical LUP"*: the set of couples $\langle x_i, u(x_i)\rangle$ is taken as input, from which the LUP is computed as discussed and the couple $\langle y_{thLUP}, u(y_{thLUP})\rangle$ is obtained;
- *"experimental LUP"*: for each X_i a set $\{x_{i,j}\}_j$ of n values is generated by a Montecarlo sampling on the given distribution with the given x_i and $u(x_i)$; for each $\{x_{i,j}\}$ the sample average $\bar{x}_i = \sum_j x_{i,j} / n$ and its standard uncertainty $u(\bar{x}_i) = \sqrt{\sum_j (x_{i,j} - \bar{x}_i)^2 / (n(n-1))}$ are computed; the set of couples $\langle \bar{x}_i \, , \, u(\bar{x}_i) \rangle$ is then taken as input, from which the LUP is computed and the couple $\langle y_{exLUP}, u(y_{exLUP})\rangle$ is obtained;
- *"Montecarlo propagation"*: the same sets $\{x_{i,j}\}_j$ generated in the previous case are taken into account; for each of the k-tuples $x_{1,j},\ldots, x_{k,j}$ the value $y_j=f(x_{1,j},\ldots, x_{k,j})$ is computed, thus obtaining a set $\{y_j\}$ of n values, from which the sample average $y_{MC} = \sum_j y_j / n$ and its standard uncertainty $u(y_{MC}) = \sqrt{\sum_j (y_j - y_{MC})^2 / (n(n-1))}$ are computed.

Hence, a comparison of the three couples $\langle y_{thLUP}, u(y_{thLUP})\rangle$, $\langle y_{exLUP}, u(y_{exLUP})\rangle$, and $\langle y_{MC}, u(y_{MC})\rangle$ is informative on the correctness of the LUP implementation:

- first of all, the relative error between y_{thLUP} and y_{exLUP} and between $u(y_{thLUP})$ and $u(y_{exLUP})$ can be interpreted as a validator of the goodness of the random generator from which the sample sets are produced;
- under the hypothesis of a positive result in the first test, the relative error between y_{exLUP} and y_{MC} and between $u(y_{exLUP})$ and $u(y_{MC})$ (note that such values are obtained from the *same* sample data set) can

be interpreted as a validator for the LUP implementation.

As an example, Figure 9 shows the graph implementing the validator for the simple function $Y=X_1/X_2$, with X_1 associated with a Gaussian distribution with average 1.0 and standard deviation 0.05, and X_2 associated with a rectangular distribution with bounds 1.5 and 2.0. The number n of samples is set to 1000. Nodes `deltaCheck` and `deltaChecku` compute the relative percentage error between theoretical and experimental LUP; nodes `delta` and `deltau` compute the relative percentage error between experimental LUP and Montecarlo propagation.

Figure 9 – Graph implementing the validator for the simple function $Y=X_1/X_2$

This validation strategy can be particularly useful in the case of non linear functions, to obtain some general hints about the reliability of an uncertainty evaluation by means of the LUP.

5. On the definition of AD-aware elemental functions

A specific problem arising in the introduction of the AD-based LUP in the context of a general simulation engine, such as the one we are developing, is related to the definition of the AD-aware elemental functions. While software packages specialized in measurement uncertainty management typically deal with only a small set of basic elemental functions, such as arithmetic operators and trigonometric functions, simulation engines commonly include a rich set of functions to enable an efficient and flexible modeling process. Many of such functions are complex in their definition (as a paradigmatic case consider a function implementing the Fast Fourier Transform of its vector argument) and / or are not everywhere differentiable, e.g., the function $\min(x_1,x_2)$, or even not continuous, e.g. the function $\text{if}(x_1,x_2,x_3)$, which expresses in functional form the conditional operator that in the C / C++ / Java programming languages is written:

```
result = x1 ? x2 : x3;
```

interpreted as “if the condition `x1` is true, assign to `result` the value of `x2`; otherwise, the value of `x3`”. While these functions cannot be directly fed into Eqs. (1) and (2), they can be computationally dealt with as in the following example, taking into account how an AD-aware implementation of the function $\min(x_1,x_2)$ can be done despite of the non-differentiability of the function itself. The function $\min(x_1,x_2)$

can be defined as:

$$\min(x_1, x_2) = \begin{cases} x_1 & if x_1 \le x_2 \\ x_2 & otherwise \end{cases}$$

so that:

$$\frac{\partial \min(x_1, x_2)}{x_1} = \begin{cases} 1 & if x_1 \le x_2 \\ 0 & otherwise \end{cases} \text{ and } \frac{\partial \min(x_1, x_2)}{x_2} = \begin{cases} 0 & if x_1 \le x_2 \\ 1 & otherwise \end{cases}$$

and for the chain rule (writing the conditional operator according to the Java syntax):

$$\min(\langle x_1, Dx_1 \rangle, \langle x_2, Dx_2 \rangle) = \langle \min(x_1, x_2), x_1 \le x_2 ? Dx_1 : Dx_2 \rangle \quad (5)$$

showing that in this case the uncertainty propagation consists actually in the selection between the standard uncertainties of the input quantities. As function of x_1 and x_2, $u(y)$ has a discontinuity even if $y=\min(x_1,x_2)$ is continuous. This is conceptually questionable, as a simple case shows (see Figure 10). Let us suppose that x_2 has a constant value, say $x_2=5.0$, with $u(x_2)=0$, and consider the behavior of $u(\min(x_1,x_2))$ while x_1 changes its value in the interval [4.0, 6.0] with a constant standard uncertainty $u(x_1)=0.1$. According to the previous definitions, the switch of $u(y)=u(\min(x_1,x_2))$ at the value $x_1=5.0$ implies that the standard uncertainty of the output measurand is null if, e.g., $x_1=5.1$, a conclusion that contradicts the fact that the value for the input quantity x_1 is uncertain, so that it could have been instead expressed as 4.9 with a non-null probability. In fact, the boolean rule prevents in this case the uncertainty $u(x_1)$ to be propagated to $u(y)$.

Figure 10 – Chart of (a) $y=\min(x_1,x_2)$ and (b) $u(y)$ for $x_1 \in [4.0,6.0]$ along the x-axis, $x_2=5.0$, and $u(x_1)=0.1$ and $u(x_2)=0.0$.

This problem can be solved by borrowing the concept of threshold smoothing from the fuzzy set theory: while maintaining the standard implementation for $y=\min(x_1,x_2)$ (in the opposite case a defuzzification step would be however required to maintain the logic underlying the LUP), the partial derivatives can be propagated by fuzzifying the transition from Dx_1 to Dx_2. Instead of the definition in Eq. (5), i.e.,

$x_1 \le x_2 ? Dx_1 : Dx_2$, in the case of linear transition the following rule is adopted:

$$\text{given } k = k_{\%}\frac{Dx_2 - Dx_1}{200},$$

$$\begin{cases} Dx_1 & \text{if } x_1 \le (x_2 - |k|) \\ Dx_2 & \text{if } x_1 > (x_2 + |k|) \\ Dx_1 + \dfrac{100}{k_{\%}}\left[-\text{sign}(k)(x_2 - x_1) + k\right] & \text{otherwise} \end{cases} \tag{6}$$

where $k_{\%}$ is a system parameter ranging from 0 (crisp transition) to 100 (smooth transition, with slope from Dx_1 to Dx_2 equals to ± 1 . Hence, the previous chart for $k_{\%}$=50 and =100 becomes respectively as shown in Figure 11 (it should be noted that this fuzzy rule also solves the subtle problem of deciding which standard uncertainty should be assigned to the singular point, x=5.0 in this case).

Figure 11 – Chart of $u(\min(x_1,x_2))$ in the case the "fuzzification factor" $k_{\%}$ is equal to 50 (a) and 100 (b).

As a little bit more complex and interesting example, consider $y=\max(\sin(x_1),x_2)$ with x_2=0.0, $u(x_2)$=0.05, and x_1 varying in the interval [0.0, 5.0] with $u(x_1)$=0.1. The charts in Figure 12 show, in function of x_1, the value of y, the value of $u(y)$ for $k_{\%}$=0, and the value of $u(y)$ for $k_{\%}$=100.

Figure 12 – Chart of (a) $y=\max(\sin(x_1),x_2)$ and $u(y)$ with $k_{\%}$=0 (b) and $k_{\%}$=100 (c) for $x_1 \in [0.0,5.0]$ along the x-axis, x_2=0.0, and $u(x_1)$=0.1 and $u(x_2)$=0.05.

These results, and specifically the value of $u(y)$ as a function of $k_{\%}$, can be compared to the standard uncertainty computed by propagating the assumed distributions of the input quantities by the Montecarlo method (the simulation engine we are developing is able to propagate distributions, as exploited in the validation strategy presented above and as discussed in [11]: hence, the following comparison has been directly performed by the engine).

Let us consider once more the case of computing $u(y)=u(\min(x_1,x_2))$, for example from the input quantities x_1=4.8, $u(x_1)$=0.1, and x_2=5.0, $u(x_2)$=0.0. From the proposed rule, Eq. (6), if $k_{\%}$=50 then $u(y)$=0.09, and if $k_{\%}$=100 then $u(y)$=0.07, both of them correctly being included in the interval $[u(x2),u(x1)]$=[0.0,0.1].

By assuming, e.g., $x_1 \sim N(4.8;0.1)$, a Montecarlo sampling can be performed, and the computed distribution for y, in this case with 10 k samples, is as in Figure 13, clearly highlighting that the fraction of the sample

set such that x_1>5.0 was accumulated to y=5.0.

Figure 13 – The histogram distribution obtained by a Montecarlo sampling of min(x_1,x_2) with x_1~N(4.8,0.1) and x_2=5.0.

From this distribution the sample standard deviation can be computed, $u(y)$≈0.097, showing that in this Gaussian case the "smoothing parameter" $k_{\%}$ should be chosen with a value less than 50 (and specifically, after some fine tuning, ≈42).

An analogous conclusion is obtained when x_1=4.9, $u(x_1)$=0.1, in which the fact that x_1 is closer to the threshold x_2 makes the case even more critical, as the propagated distribution for y in Figure 14 shows (consider that in this case the peak at x=5.0 collects about 1.6 k samples).

Figure 14 – The histogram distribution obtained by a Montecarlo sampling of min(x_1,x_2) with x_1~N(4.9;0.1) and x_2=5.0.

Now, if $k_{\%}$=50 then $u(y)$=0.07, and if $k_{\%}$=100 then $u(y)$=0.06, while the sample standard deviation is $u(y)$≈0.086, corresponding to $k_{\%}$≈28.

These examples show that the proper choice of a value for $k_{\%}$ can be a matter of fine tuning for the model under analysis, but seem to confirm the substantial correctness of the logic underlying the presented procedure.

6. Conclusions

Of the general steps of which a measurement uncertainty analysis procedure consists:

1. formalize the measurement process as a function mapping one or more input quantities to one or more output measurands;

2. identify the uncertainty sources and (possibly) distributions;

3. evaluate the uncertainties of the input quantities;

4. combine such uncertainties to obtain the uncertainties of the output measurands;

5. report the analysis results,

this work mainly focused on step 4.

The proposed strategy enables a parser / evaluator to deal with uncertain data with minimal, and however localized, modifications with respect to its standard implementation: (i) data values and functions operating on them must be (re)defined to be AD-aware, i.e., so to be able to compute partial derivatives, and (ii) the standard evaluation procedure must be encapsulated by a pre-processing step, aimed at seeding the AD process, and a post-processing step, which actually computes the LUP. As a benefit of this approach, the computational features of the parser / evaluator are easily adapted to uncertainty management, thus enabling to propagate uncertainty to vector or matrix variables, to compute and visualize uncertainties for whole (time) series instead of single points only, and to propagate uncertainty also in sequential / dynamic (stateful) models, and not only in combinatorial / algebraic (memoryless) ones. Furthermore, the adoption of a graph-based simulation engine makes the uncertainty propagation process visible to the user, who by means of the graph defining the measurement function specifies the level of granularity to access the intermediate results in the propagation itself. Finally, the computational environment allows extending the LUP application to non everywhere differentiable or even non continuous elemental functions by means of their AD-aware redefinition.

The introduction of tools for uncertainty propagation in a general, user-oriented, computational system, instead of a software library or a dedicated system, allows making uncertainty evaluation an inherent component of computational processes instead of an optional, ad hoc, addendum to them. Furthermore, the generality of the simulation engine generates flexibility in decision about the technique to adopt for uncertainty formalization: indeed, the same tool can be used to formalize not only the LUP but also, e.g., the propagation of distributions by Montecarlo method [12] or an iterative, approximate, technique such as PUMA [13], and a comparison of their results can be easily performed, with the aim of validating them and possibly choosing the best strategy according to the problem structure and available data.

Appendix: implementation strategy in the Java programming language

The data pushed to and popped from the evaluation stack are objects of the `ADDouble` (“AD-enabled double”) class.

```
/** AD-enabled Double class. */
public class ADDouble extends Number implements Comparable<ADDouble> {
  /** The value of this object. */
```

```
  private double value;
  /** The partial derivatives of this object (in reference to a given expression). */
  private double[] der;

  // together with the accessor methods (get* / set*)
  // as in the JavaBeans design pattern
}
```

Correspondingly, the elemental functions must be extended to operate on such objects. To best exploit the the object-orientedness of the Java language, each of these functions can be implemented as a class extending (i.e., subclassing) a superclass which gathers all common behaviors. For example, a single argument function such as the one that computes sin(x) can be implemented as (*):

```
/** Class implementing the sin() elemental function. */
public class Sin extends Function {
  // ...

  /** Evaluate the elemental function.
   * @param input the function argument
   * @return the value of the function
   */
  final ADDouble exec(final ADDouble input) {
    double val = Math.sin(input.getValue());
    return getResultWithDer(input, val); // a method inherited from the superclass
  }

  /** Compute an element of the partial derivative vector.
   * @param input the function argument
   * @param i the index of the derivative to compute
   * @return the result
   */
  @Override
  double computeDer(final ADDouble input, final int i) {
    return Math.cos(input.getValue()) * input.getDer(i);
  }
}
```

the common superclass being:

```
/** Abstract class implementing the common features of the elemental functions. */
public abstract class GeneralFunction {
  // ...

  /** Get the result of a monadic function, including the partial derivatives.
   * @param input the function argument
   * @param val the function value
   * @return the result
```

(*) The shown implementation for the method `exec()` assumes the partial derivatives to be computed even in the case of models for which the standard uncertainty for input measurands is not specified, i.e., is null. Given the fact that AD typically at least doubles the computation times, this strategy is clearly inefficient. Hence, the user assignable model property `withUncertainty` has been introduced, so that the actual implementation of the mentioned method is the following:

```
final ADDouble exec(final ADDouble input) {
       double val = Math.sin(input.getValue());
       if(Model.isWithUncertainty()) { return ADDouble.valueOf(val); }
       return getResultWithDer(input, val);
}
```

```
   */
  ADDouble getResultWithDer(final ADDouble input, final double val) {
    double[] d = input.getDer();
    if(d == null) { return ADDouble.valueOf(val); }
    int size = d.length;
    double[] der = new double[size];
    for(int i = 0; i < size; i++) {
      der[i] = computeDer(input, i); // delegated to the subclasses
    }
    return ADDouble.valueOf(val, der);
  }
}
```

The implementation is analogously simple for the *n*-adic, $n \geq 2$, elemental functions, and only becomes more complex in the case of non everywhere differentiable or non continuous functions, such as min(x_1,x_2), for which the effect of threshold smoothing discussed in the section 4 is introduced.

The nodes of d-graphs are instances of the class:

```
/** d-graph node class. */
public class Node {
  /** The value of this node. */
  private double value;
  /** The standard uncertainty of this node. */
  private double unc;
  /** The data on correlations. */
  private ArrayList<CorrelData> correlData;

  // together with the accessor methods (get* / set*)
  // as in the JavaBeans design pattern,
  // together with several other methods (see below)
}
```

being:

```
/** Correlation data type. */
class CorrelData {
  /** Correlation value. */
  private double correl;
  /** Index of the first node. */
  private int indexOfNode1;
  /** Index of the second node. */
  private int indexOfNode2;
}
```

so that the AD-aware, LUP-enabled evaluator method (part of the `Node` class) has the following structure (Greek letters are references to the algorithm, as presented in section 3):

```
/** Set the value of this node by evaluating its defining expression.
 */
final void evaluate() {
  // initalization stage
  // ...
  uncPreProcessor(); // uncertainty pre-processor
  ADDouble value = evalExpression(); // standard (AD-enabled) evaluator [δ]
  double unc = uncPostProcessor(value); // uncertainty post-processor
  setValue(value.getValue()); [β]
  setUnc(unc); [β]
  // completion stage, and exception handling
  // ...
```

```
}

/** Uncertainty pre-processor: it seeds the evaluation process
  *  by setting the derivative component of the defining nodes of this node.
  */
private void uncPreProcessor() {
  if(!Model.isWithUncertainty()) { return; } // see the previous footnote
  int size = getDefiningNodes().size(); // number of defining nodes
  if(size == 0) { return; } // an input node: nothing to do
  for(int i = 0; i < size; i++) {
    EDouble v = getDefiningNodes().get(i).getValue(); // previously evaluated
    v.setDer(new double[size]); // initialize the array of partial derivatives
    for(int j = 0; j < size; j++) { v.setDer(j, 0.0); }
    v.setDer(i, 1.0); // seed for AD [γ]
  }
}


/** Uncertainty post-processor: it computes the standard uncertainty of this node.
 * @param value the value, as obtained without uncertainty
 * @return the standard uncertainty of the specified node
 */
private double uncPostProcessor(final ADDouble value) {
  if(!Model.isWithUncertainty()) { return 0.0; } // see the previous footnote
  if(isInput()) { return getUnc(); } // simply transfer the uncertainty [α]
  double unc = 0.0; // initialize the uncertainty value
  for(Node iNode : getDefiningNodes()) { // the first step of LUP computation
    unc += Math.pow(result.getDer(i), 2) * Math.pow(iNode.getUnc(), 2); [ε]
  }
  for(CorrelData data : getCorrelData()) { // the first step of LUP computation
    double correl = data.getCorrel(); // for adding the covariances
    int i1 = data.getIndexOfNode1();
    int i2 = data.getIndexOfNode2();
    unc += 2 * correl * getDefiningNodes().get(i1).getUnc() * value.getDer(i1)
           * getDefiningNodes().get(i2).getUnc()) * value.getDer(i2); [ε]
  }
  return Math.sqrt(u); [ε]
}
```

References

[1] P.W.Bridgman, How much rigor is possible in physics?, in: Henkin et al. (eds.), The Axiomatic method, North-Holland, 1959.

[2] ISO et al., ISO/IEC Guide 99:2007, International Vocabulary of Metrology – Basic and General Concepts and Associated Terms, 3rd ed., Geneva, 2007.

[3] International Organization for Standardization, Guide to the expression of uncertainty in measurement, Geneva, 1993, amended 1995 (published by ISO in the name of BIPM, IEC, IFCC, IUPAC, IUPAP and OIML) (now also ISO ENV 13005: 1999).

[4] L.B.Rall, The arithmetic of differentiation, Math. Mag., 59, 275, 1986.

[5] B.D.Hall, Calculating measurement uncertainty using automatic differentiation, Meas.Sci.Technol., 13

421–427, 2002.

[6] R.Boudjemaa, M.G.Cox, A.B.Forbes, P.M.Harris, Automatic differentiation and its application in metrology, in: Advanced Mathematical and Computational Tools in Metrology VI, P.Ciarlini, M.G.Cox, F.Pavese, G.B.Rossi (Eds.), World Scientific Publishing, 2004.

[7] B.D.Hall, The GUM tree design pattern for uncertainty software, in: Advanced Mathematical and Computational Tools in Metrology VI, P.Ciarlini, M.G.Cox, F.Pavese, G.B.Rossi (Eds.), World Scientific Publishing, 2004.

[8] M.T.Flanagan, Michael Thomas Flanagan's Java Scientific Library, http://www.ee.ucl.ac.uk/~mflanaga/java/index.html, last accessed: summer 2007.

[9] Danish Technological Institute, GUM Workbench, software package by Metrodata GmbH, 1999.

[10] S.Castrup, A comprehensive comparison of uncertainty analysis tools, 2004 Measurement Science Conference Anaheim, CA.

[11] L.Mari, D.Petri, Propagating uncertainty through discrete time dynamic systems, proc. IEEE International Workshop on Advanced Methods for Uncertainty Estimation in Measurement (AMUEM), 23-26, 2006.

[12] International Organization for Standardization, GUM Supplement 1: Numerical methods for the propagation of distributions.

[13] International Organization for Standardization, ISO 14253-2: Geometrical Product Specifications - Inspection by measurement of workpieces and measuring equipment. Part 2: Guide to the estimation of uncertainty in GPS measurement, in calibration of measuring equipment and in product verification, 1998.

Figure Captions

Figure 1 – Tree resulting from the parsing of a simple function.

Figure 2 – Evaluation trace of a simple function (to be read bottom-up).

Figure 3 – AD-enabled evaluation trace of a simple monadic function.

Figure 4 – AD-enabled evaluation trace of a simple dyadic function.

Figure 5 – AD seeding for a simple dyadic function represented as a tree.

Figure 6 – AD seeding for a simple dyadic function represented as a directed graph.

Figure 7 – Four possible definition graphs for the same dyadic function.

Figure 8 – Definition graph of a monadic function, showing a case of the "hidden" covariances.

Figure 9 – Graph implementing the validator for the simple model $Y=X_1/X_2$

Figure 10 – Chart of (a) $y=\min(x_1,x_2)$ and (b) $u(y)$ for $x_1 \in [4.0,6.0]$ along the x-axis, $x_2=5.0$, and $u(x_1)=0.1$ and $u(x_2)=0.0$.

Figure 11 – Chart of $u(\min(x_1,x_2))$ in the case the "fuzzification factor" $k_{\%}$ is equal to 50 (a) and 100 (b).

Figure 12 – Chart of (a) $y=\max(\sin(x_1),x_2)$ and $u(y)$ with $k_{\%}=0$ (b) and $k_{\%}=100$ (c) for $x_1 \in [0.0,5.0]$ along the x-axis, $x_2=0.0$, and $u(x_1)=0.1$ and $u(x_2)=0.05$.

Figure 13 – The histogram distribution obtained by a Montecarlo sampling of $\min(x_1,x_2)$ with $x_1 \sim N(4.8,0.1)$ and $x_2=5.0$.

Figure 14 – The histogram distribution obtained by a Montecarlo sampling of $\min(x_1,x_2)$ with $x_1 \sim N(4.9,0.1)$ and $x_2=5.0$.

Acknowledgments

I warmly thank Prof. Dario Petri and Prof. Sergio Sartori, who have variously contributed to this work with their expert and kind support.