



Getting started with uncertainty evaluation using the Monte Carlo method in R

Adriaan M. H. van der Veen¹ · Maurice G. Cox²

Received: 24 October 2019 / Accepted: 15 April 2021 / Published online: 6 June 2021
© The Author(s) 2021

Abstract

The evaluation of measurement uncertainty is often perceived by laboratory staff as complex and quite distant from daily practice. Nevertheless, standards such as ISO/IEC 17025, ISO 15189 and ISO 17034 that specify requirements for laboratories to enable them to demonstrate they operate competently, and are able to generate valid results, require that measurement uncertainty is evaluated and reported. In response to this need, a European project entitled “Advancing measurement uncertainty—comprehensive examples for key international standards” started in July 2018 that aims at developing examples that contribute to a better understanding of what is required and aid in implementing such evaluations in calibration, testing and research. The principle applied in the project is “learning by example”. Past experience with guidance documents such as EA 4/02 and the Eurachem/CITAC guide on measurement uncertainty has shown that for practitioners it is often easier to rework and adapt an existing example than to try to develop something from scratch. This introductory paper describes how the Monte Carlo method of GUM (Guide to the expression of Uncertainty in Measurement) Supplement 1 can be implemented in R, an environment for mathematical and statistical computing. An implementation of the law of propagation of uncertainty is also presented in the same environment, taking advantage of the possibility of evaluating the partial derivatives numerically, so that these do not need to be derived by analytic differentiation. The implementations are shown for the computation of the molar mass of phenol from standard atomic masses and the well-known mass calibration example from EA 4/02.

Keywords Measurement uncertainty · GUM · Uncertainty propagation · Monte Carlo · R · Calibration · Testing

Introduction

One of the complicating factors in the evaluation and propagation of measurement uncertainty is the competence in mathematics and statistics required to perform the calculations. Nevertheless, standards such as ISO/IEC 17025 [1], ISO 15189 [2] and ISO 17034 [3] that specify requirements

for laboratories to enable them to demonstrate they operate competently, and are able to generate valid results, require that measurement uncertainty is evaluated and reported. The well-known law of propagation of uncertainty (LPU) from the Guide to the expression of uncertainty in measurement (GUM) [4] requires the calculation of the partial derivatives of the measurement model with respect to each of the input variables.

In this primer, we (re)introduce the Monte Carlo method of GUM Supplement 1 (GUM-S1) [5], which takes the same measurement model and the probability density functions assigned to the input variables as the LPU to obtain (an approximation to) the output probability density function. We show, based on some well-known examples illustrating the evaluation of measurement uncertainty, how this method can be implemented for a single measurand and how key summary outputs, such as the estimate (measured value), the associated standard uncertainty, the expanded uncertainty

Presented at the workshop “Mathematical and Statistical Methods for Metrology 2019”, Torino, Italy, 30–31 May 2019.

✉ Adriaan M. H. van der Veen
avdveen@vsl.nl

Maurice G. Cox
maurice.cox@npl.co.uk

¹ VSL Unit Chemistry Mass Pressure and Viscosity,
Thijssseweg 11, 2629, JA, Delft, The Netherlands

² NPL Management Ltd, Hampton Road, Teddington,
Middlesex TW11 0LW, UK

and a coverage interval for a specified coverage probability, can be obtained.

The Monte Carlo method of GUM-S1 [5] is a versatile method for propagating measurement uncertainty using a measurement model. It takes the measurement model, i.e. the relationship between the output quantity and the input quantities, and random samples of the probability density functions of the input quantities to generate a corresponding random sample of the probability density function of the output quantity.

The use of probability density functions is well covered in the GUM [4] and further elaborated in GUM-S1 [5]. In this primer, the emphasis is on setting up an uncertainty evaluation using the Monte Carlo method for a measurement model with one output quantity (a “univariate” measurement model). GUM Supplement 2 (GUM-S2) [6] provides an extension of the Monte Carlo method to measurement models with two or more output quantities (“multivariate” measurement models) as well as giving a generalisation of the LPU to the multivariate case.

The vast majority of the uncertainty evaluations in calibration and testing laboratories are performed using the LPU [4]. This mechanism takes the estimates (values) of the input quantities and the associated standard uncertainties to obtain an estimate for the output quantity and the associated standard uncertainty. The measurement model is used to compute (1) the value of the output quantity and (2) the sensitivity coefficients, i.e. the first partial derivatives of the output quantity with respect to each of the input quantities (evaluated at the estimates of the input quantities). The second part of the calculation involving the partial derivatives is perceived as being cumbersome and requires skills that are often beyond the capabilities of laboratory staff and researchers. The computation of the sensitivity coefficients can also be performed numerically [7, 8]. One of the advantages of the Monte Carlo method is that sensitivity coefficients are not required. All that is needed is a measurement model, which can be in the form of an algorithm, and a specification of the probability distributions for the input quantities. These probability distributions (normal, rectangular, etc.) are typically already specified in uncertainty budgets when the LPU is used.

In this tutorial, we show how the Monte Carlo method of GUM-S1 can be implemented in R [9]. This environment is open-source software and specifically developed for statistical and scientific computing. Most of the calculations in laboratories, science and elsewhere are still performed using mainstream spreadsheet software. An example of using the Monte Carlo method of GUM-S1 with MS Excel is given in the Eurachem/CITAC Guide on measurement uncertainty [10]. It is anticipated that this tutorial will also be useful for those readers who would like to get started using other software tools or other languages.

Monte Carlo method

The heart of the Monte Carlo method of GUM-S1 can be summarised as follows [5]. Given a measurement model of the form

$$Y = f(X_1, \dots, X_N)$$

and probability density functions assigned to each of the input quantities X_1, \dots, X_N , generate M sets of input quantities $X_{1,r}, \dots, X_{N,r}$ ($r = 1, \dots, M$) and use the measurement model to compute the corresponding value for Y_r . M , the number of sets of input quantities should be chosen to be sufficiently large so that a representative sample of the probability density function of the output quantity Y is obtained. The approach here applies to independent input quantities and a scalar output quantity Y . For its extension to dependent input quantities, see GUM-S1 [5], and a multivariate output quantity, see GUM-S2 [6].

GUM-S1 [5, clause 6.4] describes the selection of appropriate probability density functions for the input quantities, thereby supplementing the guidance given in the GUM [4, clause 4.3]. GUM-S1 also provides guidance on the generation of pseudo-random numbers. Pseudo-random numbers rather than random numbers are generated by contemporary software since the latter are almost impossible to obtain. However, comprehensive statistical tests indicate that the pseudo-random numbers generated cannot be distinguished in behaviour from truly random numbers.

Considerable confidence has been gained by the authors over many years concerning the performance of the Monte Carlo method of uncertainty evaluation from a practical viewpoint. For measurement models that are linear in the input quantities, for which the law of propagation of uncertainty produces exact results, agreement with results from the Monte Carlo method to the numerical accuracy expected has always been obtained. Thus, weight is added to the above point: there is evidence that the effects of working with pseudo-random numbers and truly random numbers are identical.

If needed, the performance of a random number generator can be verified [11, 12]. For the purpose of this tutorial, it is assumed that the built-in random number generator in R is fit for purpose.

A refinement of the Monte Carlo method concerns selecting the number of trials automatically so as to achieve a degree of assurance in the numerical accuracy of the results obtained. An adaptive Monte Carlo procedure for this purpose involves carrying out an increasing number of Monte Carlo trials until the various results of interest have stabilised in a statistical sense. Details are provided in [5, clause 7.9] and since then an improved method in [13].

In many software environments, random number generators for most common probability density functions are already available; if not, they can be readily developed using random numbers from a rectangular distribution [5, annex C]. (The rectangular distribution is also known as the uniform distribution.) Should even a random number generator for the rectangular distribution not be available in the software environment, then the one described in GUM-S1 can be implemented as a basis for generating random numbers. The default random number generator in R is the Mersenne Twister [14], which is also implemented in many other programming environments, including MATLAB and Microsoft Excel (since version 2010, see [15]). Based on this random number generator, there are generators available for a number of probability distributions [9].

The output of applying the Monte Carlo method is an array (vector) Y_1, \dots, Y_M characterising the probability density function of the output quantity. This sample is, however, not in the form in which a measurement result is typically communicated. From the output Y_1, \dots, Y_M , the following can be computed:

- the measured value, usually taken as the arithmetic mean of Y_1, \dots, Y_M
- the standard uncertainty, usually computed as the standard deviation of Y_1, \dots, Y_M
- a coverage interval containing the value of the output quantity with a stated probability, obtained as outlined below;
- the expanded uncertainty;
- the coverage factor.

The last two items apply when the coverage interval can be reasonably approximated by a symmetric probability density function. Care should be taken in cases of asymmetric output probability density functions in which the mean of Y_1, \dots, Y_M might result in an unreliable estimator of the measurand, behaving worse than other estimators (such as the mode or the median) and even worse than the simple estimate obtained from $y = f(x_1, \dots, x_n)$ provided by the measured model f directly applied to the input estimates x_1, \dots, x_n .

The most general way of representing a coverage interval is by specifying its upper and lower limits. This representation is always appropriate whether the output distribution is symmetric or not. In many instances, however, the output probability density function is (approximately) symmetric, and then, the expanded uncertainty can be computed as the half-width of the coverage interval. The coverage factor can be computed from the expanded uncertainty $U(y)$ and the standard uncertainty $u(y)$, i.e. $k = U(y)/u(y)$. The symmetry of the output probability density function can be verified by examining a histogram of Y_1, \dots, Y_M , or obtaining a kernel density plot [16, 17], a smooth approximation to the probability density function.

Software environment

R is an open-source language and environment for statistical computing and graphics. It is a GNU project, similar to the S language and environment, which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. R can be considered as a different implementation of S [9]. It is available for Windows, MacOS and a variety of UNIX platforms (including FreeBSD and Linux) [9].

Users of Windows, MacOS and a number of Linux distributions may also wish to download and install RStudio [18], which provides an integrated development environment, in which code can be written, the values of variables can be monitored, and separate windows for the console and graphics output are available. The R code provided in this primer has been developed in RStudio (version 1.2.1335, build 1379 (f1ac3452)).

Generating random numbers

In R, it is straightforward to generate a sample of random numbers from most common probability density functions. For example, the following code generates a sample of a normal distribution with mean $\mu = 10.0$ and standard deviation $\sigma = 0.2$ and a sample size $M = 10,000$:

```
M = 10000
mu = 10.0
sigma = 0.2

set.seed(2926)
X1 = rnorm(M, mu, sigma)
```

The function to be called to generate an array (vector) of random numbers having a normal distribution with mean μ and standard deviation σ is called `rnorm`. The line `set.seed(2926)` is useful for debugging purposes, as it ensures that the random number generator starts at the same point every time. Any other value for the seed would also ensure the exact reproduction of the series of numbers obtained from the random number generator. If that is not required, the line can be omitted. In this tutorial, the `seed` is set, so that the reader can exactly reproduce the output. The output is collected in a variable named `X1`. It is an array with 10,000 elements.

The following code snippet shows the mean and standard deviation of the 10,000 generated numbers, using R's built in functions `mean` and `sd`, respectively.

```
mean(X1)
## [1] 10.00131

sd(X1)
## [1] 0.2006594
```

Using R's functions `plot` and `hist`, a histogram of variable `X1` can be plotted (Fig. 1). The code to generate Fig. is as follows:

```
plot(hist(X1,plot=FALSE),xlab = 'X1',ylab = "frequency",main = "")
```

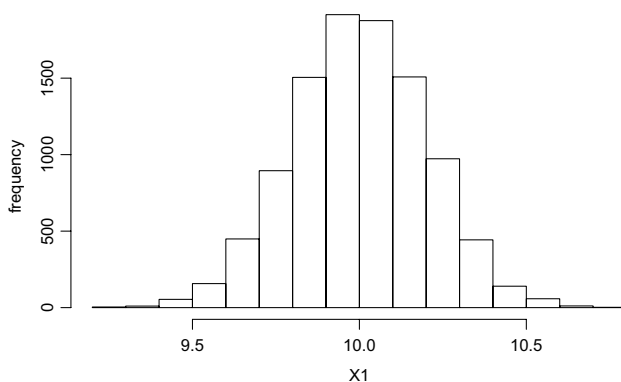


Fig. 1 Histogram of $M = 10,000$ samples of the random variable $X1$ having a normal distribution with mean 10.0 and standard deviation 0.2

where `hist` calculates the histogram from the array `X1` and `plot` generates the figure. The plotted histogram resembles that of a normal distribution. The larger is the number of samples drawn from the random number generator, the closer the resemblance to the normal distribution will be.

From the first code fragment in this section, it is readily seen that R has a function for generating random numbers with a normal distribution. It also has functions for generating random numbers with a rectangular distribution (`runif`), the t distribution (`rt`), exponential distribution (`rexp`) and gamma distribution (`rgamma`). There exist a package (extension) called “trapezoid” [19] implementing among others the trapezoidal distribution, a package called “mvtnorm” [20] implementing the multivariate normal distribution (useful when some of the input quantities are dependent [5]) and a package called

“triangle” [21] implementing the triangular distribution. So, apart from the curvilinear trapezoidal distribution and the arc sine distribution, random numbers for all

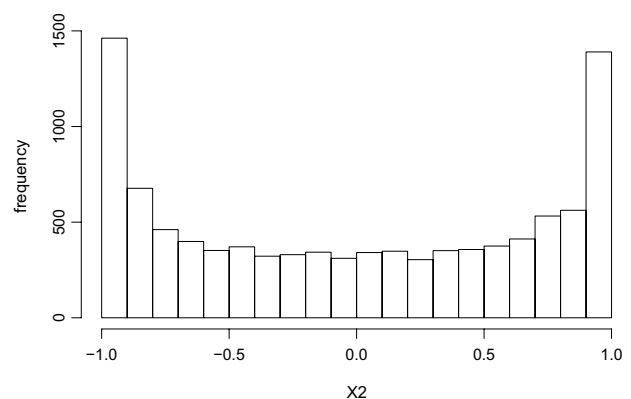


Fig. 2 Histogram of the random variable $X2$ containing $M = 10,000$ samples having an arcsine distribution between -1 and 1

probability density functions mentioned in GUM-S1 [5, table 1] are available in R.

The arc sine distribution can be implemented as follows in R. According to GUM-S1 [5, clause 6.4.6.1], a U-shaped random variable X on the interval $[a, b]$ can be obtained through

$$X = \frac{a+b}{2} + \frac{b-a}{2} \sin \Phi,$$

where Φ is a random variable with a rectangular distribution on $[0, 2\pi]$. In R, a function `rarcsin` that provides such a random variable, and a call to that function, can be coded as follows:

```
rarcsin <- function(n,a,b) {
  X = (a+b)/2 + (b-a)/2 * sin(runif(n,0,2*pi))
  return(X)
}

X2 = rarcsin(M,-1.0,1.0)
```

The argument n determines the number of random numbers returned; a and b denote the lower and upper limits, respectively, of the interval over which the arcsine distribution has a nonzero density. If $n > 1$, the function returns an array; if $n = 1$, it returns a single number. This behaviour mimics the behaviour of the other functions implemented in R to generate random numbers.

The last line in the code snippet creates an array $X2$ of M elements ($M = 10,000$ in this instance) of a random variable having an arcsine distribution over the interval $[-1, 1]$. A histogram (obtained through the R function `hist`) is shown in Fig. 2.

Implementation of the Monte Carlo method

Molar mass of phenol

In this example, the molar mass of phenol (molecular formula C_6H_5OH) is computed. The example shows how an output quantity with an uncertainty is obtained from input quantities with uncertainty. There is no experiment involved. The example is pivotal for many calculations involving

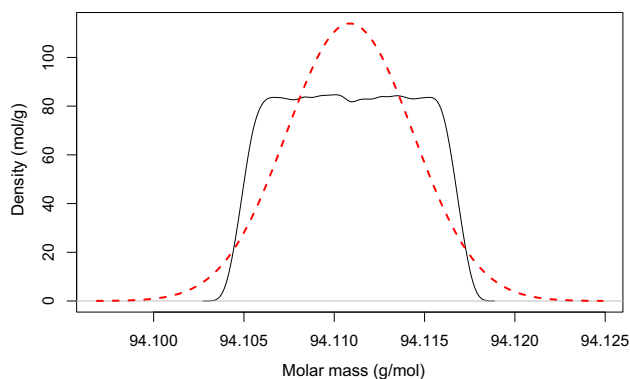


Fig. 3 Output probability density function of the molar mass of phenol and, superimposed, a normal distribution with the same mean and standard deviation

reference data, such as atomic weights, molar masses and enthalpies of formation.

The molar mass is computed from the standard atomic masses and the coefficients appearing in the molecular formula, which for the elements involved are 6 for carbon, 6 (5+1) for hydrogen and 1 for oxygen. The current relative atomic masses are used as published by IUPAC (International Union of Pure and Applied Chemistry) [22]. The relative atomic masses that apply to “normal materials” are called standard atomic weights [22, 23]. Their interpretation is described in an IUPAC technical report [24]. From this interpretation, it follows that the standard atomic masses are modelled using the rectangular distribution.

The molecular weight of phenol (chemical formula C_6H_5OH) is computed as

$$M_r(C_6H_5OH) = 6A_r(C) + 6A_r(H) + A_r(O)$$

The Monte Carlo method is implemented in R using $M = 100,000$ trials. The R code that performs the evaluation reads as

```

M = 100000
C = runif(M, 12.0096, 12.0116)
H = runif(M, 1.00784, 1.00811)
O = runif(M, 15.99903, 15.99977)
MW = 6*C + 6*H + O
MW.val = mean(MW)
MW.unc = sd(MW)
MW.Unc = (quantile(MW, probs = 0.975) -
           quantile(MW, probs = 0.025))/2.0
k = MW.Unc/MW.unc

```

The first line declares a variable M that holds the number of trials to be carried out by the Monte Carlo method. Then, for each of the elements, M samples are drawn using the rectangular distribution (using R's function `runif`) and the lower and upper limits provided by the standard atomic weights of IUPAC [22]. These arrays have, respectively, the names C , H and O for the atomic masses of carbon, hydrogen and oxygen. The molar mass is then computed in the line defining MW . R is very efficient with vectors (arrays) and matrices (tables) [25]. The value of the molar mass ($MW.val$) is computed by taking the average of MW , the standard uncertainty by taking the standard deviation of MW and the expanded uncertainty by taking the half-width of the 95% coverage interval. The latter is obtained by calculating the 0.025 and 0.975 quantiles (which provides a probabilistically-symmetric coverage interval) [5, clause 7.7]. In the last line, the coverage factor is computed as the ratio of the expanded uncertainty and the standard uncertainty.

The code to plot the output probability density function of the molar mass (MW) and to superimpose a normal distribution with the same mean and standard deviation is given as follows:

```

x = seq(from = MW.val-4*MW.unc, to=MW.val+4*MW.unc, by=8*MW.unc/100)
hx = dnorm(x, MW.val, MW.unc)
plot(density(MW), xlab = "Molar mass (g/mol)",
     ylab = "Density (mol/g)", main="",
     xlim=c(min(x), max(x)), ylim=c(0, max(hx)))
lines(x, hx, lwd=2, lty=2, col="red")

```

The first two lines compute the relevant part of the normal distribution around the mean ± 4 standard deviations. The subsequent lines plot the output probability density function and the normal distribution, respectively.

The graph is shown in Fig. 3. It is obvious that the normal distribution is not an appropriate approximation of the probability density function of the output quantity, which is close in appearance to a rectangular distribution and much narrower than the normal distribution. The molar mass is $94.1108 \text{ gmol}^{-1}$ with standard uncertainty 0.0035 gmol^{-1} . The expanded uncertainty is 0.0059 gmol^{-1} . The coverage factor is 1.67, which is much closer to the 95% coverage factor of 1.69 for a rectangular distribution than that for a normal distribution (1.96).

Mass example from EA 4/02

In this section, the mass calibration example of EA 4/02 [26] is revisited. The evaluation using the Monte Carlo method rests on the same assumptions for the input quantities as in that example. In this example, we show how the Monte Carlo method can be implemented for any explicit measurement model with a single output quantity. The measurement model is coded in the form of a function, which promotes writing tidy code. It also allows iterative calculations to be readily implemented when the measurement model is defined implicitly [6]. This example describes the calibration of a 10

kg weight by comparison with a standard 10 kg weight. The weighings are performed using the substitution method. This method is implemented in such a way that three mutually independent observations for the mass difference between the two weights are obtained.

The measurement model is given by [26, S2]:

$$m_X = m_S + \delta m_D + \delta m + \delta m_C + \delta B, \quad (1)$$

where the symbols have the following meaning:

m_X	conventional mass of the weight being calibrated,
m_S	conventional mass of the standard,
δm_D	drift of the value of the standard since its last calibration,
δm	observed difference in mass between the unknown mass and the standard,
δm_C	correction for eccentricity and magnetic effects,
δB	correction for air buoyancy.

For using the Monte Carlo method, probability density functions are assigned to each of the five input quantities [5]. These probability density functions are described in the original example [26].

The conventional mass of the standard m_S is modelled using the normal distribution with mean 10,000.005 g and standard deviation 0.0225 g. The standard deviation (standard uncertainty) is calculated from the expanded uncertainty and the coverage factor provided on the calibration certificate. This interpretation is also described in GUM-S1 [5, 6.4.7]. The drift of the mass of the standard weight δm_D is modelled using a rectangular distribution, centred at 0.000 g and with a half-width of 0.015 g. The corrections for eccentricity and magnetic effects and that for air buoyancy are both modelled using a rectangular distribution with midpoint 0.000 g and half-width 0.010 g.

The mass difference δm between the two weights computed from the indications of the balance is calculated as the mean of $n = 3$ independent observations. EA 4/02 explains that the associated standard uncertainty is computed from a pooled standard deviation 0.025 g, obtained from a previous mass comparison, divided by \sqrt{n} .

In the implementation of the Monte Carlo method, the three observations are simulated using normal distributions with means of the observed values (i.e. 0.010 g, 0.030 g and 0.020 g, respectively) and a standard deviation of 0.025 g for each. The mass difference is formed by calculating the arithmetic average of the three simulated observations.

The measurement model [Eq. (1)] can be coded in R as follows:

```
# measurement function
mass.x <- function(m.std, dm.d, diff, dm.c, dm.B) {
  m.std + dm.d + diff + dm.c + dm.B
}
```

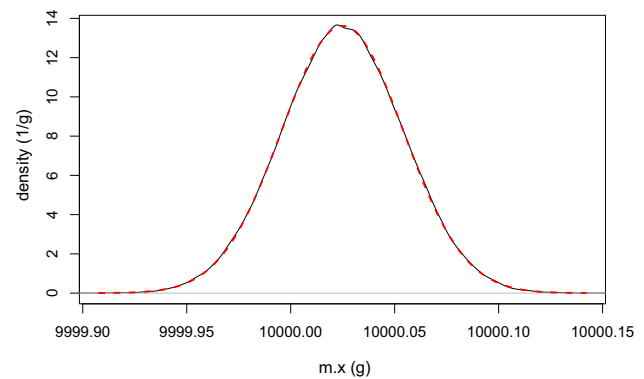


Fig. 4 Probability density function of the output quantity m_X and superimposed a normal distribution with the same mean and standard deviation

where $m.std$ denotes the conventional mass of the standard weight, $dm.d$ the drift correction of the conventional mass of the standard weight, $diff$ the mass difference obtained from the substitution weighing, $dm.c$ the correction due to eccentricity and magnetic effects and $dm.B$ the correction due to air buoyancy. The function is called `mass.x` and returns the value of the output quantity m_X .

Most programming languages implement a “for” loop, which enables executing a block of code a defined number of times. Anyone familiar with this “for” loop in computer programming would now use this kind of loop to code the recipe given in GUM-S1 clause 7.2.2 [5]. An implementation of the Monte Carlo method with a fixed value for the number of samples M would then read as follows:

```

# implementation of the procedure of GUM-S1 with fixed M
prob = 0.95
M = 10000 * ceiling(1.0/(1.0-prob)) # GUM-S1 7.2.2
m.x = numeric(M)
m.data = numeric(3)
for (i in 1:M) {
  m.std = rnorm(1,10000.005,0.0225)
  dm.d = runif(1,-0.015,+0.015)
  dm.c = runif(1,-0.010,+0.010)
  dm.B = runif(1,-0.010,+0.010)
  m.data[1] = rnorm(1,0.01,0.025)
  m.data[2] = rnorm(1,0.03,0.025)
  m.data[3] = rnorm(1,0.02,0.025)
  m.diff = mean(m.data)
  m.x[i] = mass.x(m.std,dm.d,m.diff,dm.c,dm.B)
}

```

On the first line, the probability level of the coverage interval (prob) is defined to be 0.95. In accordance with the guidance in clause 7.2.2 of GUM-S1 [5], M is calculated using the built-in function `ceiling` which returns the smallest integer not less than its argument. With `prob = 0.95`, the net effect of calling `ceiling` is that the floating point number is converted to an integer, as the result of $1/(1-\text{prob})$ of 20; hence, the minimum number of Monte Carlo trials is $M = 10,000 \times 20 = 200,000$. Then, an array (vector) `m.x` is declared that will hold the values calculated for the mass of the weight being calibrated. The vector `m.data` is a temporary storage for simulating the mass differences between the standard weight and the weight being

calibrated. In the `for` loop, at each iteration a sample is drawn of the input quantities m_s (`m.std`), δm_D (`dm.d`), δm_C (`dm.c`) and δB (`dm.B`). The mass difference from comparing the two weights (`m.diff`) is simulated by drawing from a normal distribution with different means, but the same standard deviations, the three readings and taking the average. The measured value of the output quantity m_X (`m.x`) is finally obtained by calling the measurement model with as arguments the input quantities.

Running the above code provides the following output for the mean, standard deviation (standard uncertainty) and the coverage interval of m_X :

```

print(mean(m.x),digits = 9)

## [1] 10000.025

print(sd(m.x),digits = 2)

## [1] 0.029

quantile(m.x,probs = c(0.025,0.975))

##      2.5%      97.5%
## 9999.968 10000.082

```


where the argument `probs` holds the probabilities corresponding to the lower and upper ends of the probabilistically symmetric 95% coverage interval.

This way of coding an implementation of the Monte Carlo method would work in a large number of computer languages, including Python, MATLAB, Fortran, C, C++ and Pascal. While the above code in R does what is intended, the same task can be performed with greater effectiveness in R, exploiting the fact that R is very efficient in working with vectors and matrices [25]. Computational efficiency is especially important with more complex models and larger numbers of Monte Carlo trials, as it can greatly reduce the required computing time. The following code implements the same simulation, using vectors and matrices where possible:

simulated data which have been generated using the normal distribution with mean 0 and standard deviation 0.025. The mass differences are computed by calculating the row means and storing these in `m.diff` using the R function `apply`. Note also that the measurement model can be called with vectors rather than scalars as arguments (last line of the code); in this case also, `m.x` is a vector of length M .

The second code runs in less than half the time of the first implementation. For this simple example, the difference is a matter of a few seconds, but for more complex models the difference in speed will be of more practical significance. Especially the steps that are repeated often should be carefully thought about. Another issue is memory use. The second implementation consumes appreciably more memory (for it holds all generated values for the input quantities)

```
# implementation of the procedure of GUM-S1 with fixed M
prob = 0.95
M = 10000 * ceiling(1.0/(1.0-prob)) # GUM-S1 7.2.2
m.std = rnorm(M,10000.005,0.0225)
dm.d = runif(M,-0.015,+0.015)
dm.c = runif(M,-0.010,+0.010)
dm.B = runif(M,-0.010,+0.010)
m.data = matrix(rep(c(0.01,0.03,0.02),M), nrow = M, byrow = TRUE)
m.data = m.data + matrix(rnorm(3*M,0,0.025),nrow = M,byrow = TRUE)
m.diff = apply(m.data,1,mean)
m.x = mass.x(m.std,dm.d,m.diff,dm.c,dm.B)
```

Now the variables `m.std`, `dm.d`, `dm.c` and `dm.B` are vectors holding all M values for the input quantities. The data from comparing the weights are summarised in a matrix called `m.data` of M rows and 3 columns. The matrix is constructed by adding the means (0.01, 0.03 and 0.02) to the

than the first (which only holds the last value for each of the input quantities).

The second code provides the following output for the mean, standard deviation (standard uncertainty) and the coverage interval of m_X :

```
print(mean(m.x),digits = 9)
## [1] 10000.0249
print(sd(m.x),digits = 2)
## [1] 0.029
quantile(m.x,probs = c(0.025,0.975))
##      2.5%      97.5%
## 9999.967 10000.082
```

The output probability density function is shown in Fig. 4. Its form closely resembles that of a normal distribution with mean 10,000.025 g and standard deviation 0.029 g. The following code computes the expanded uncertainty by taking the half-width of the 95% coverage interval and the coverage factor by dividing the expanded uncertainty by the standard uncertainty:

```
m.x.Unc = (quantile(m.x, probs = 0.975) - quantile(m.x, probs = 0.025))/2.0
m.x.k = m.x.Unc/sd(m.x)
```

The expanded uncertainty is 0.057 g, and the coverage factor is 1.96. This coverage factor is that of a 95% coverage interval of the normal distribution. The coverage factor differs from that used in EA 4/02, which uses $k = 2$ for obtaining (at least) 95% coverage probability. The close agreement is readily explained, as the dominating uncertainty contributions are modelled using the normal distribution, and the sum of two normal distributions is also normally distributed (see also the measurement model, Eq. (1)). That the output quantity has an (approximately) normal distribution is reflected in the coverage factor obtained from the Monte Carlo method.

Now all results are obtained that commonly appear on a calibration certificate (as well as in many test reports), as described in ISO/IEC 17025 [1]:

- the measured value (= value of the output quantity);
- the expanded uncertainty;
- the coverage factor.

In this case, one might also be willing to state that the output probability density function is a normal distribution, whereas in this case such a statement can reasonably be made, in most cases the output probability density function cannot directly be approximated by a well-known analytic probability density function. Comparison of the three results listed above with those from the GUM would imply that for comparable data the GUM would be fit for purpose in a subsequent uncertainty evaluation. In a measurement model with m_x as one of the input quantities, the above information suffices to apply the GUM [4].

```
# measurement function
mass2.x <- function(x) {
  m.std = x[1]; dm.d = x[2];
  diff = x[3]; dm.c = x[4]; dm.B = x[5]
  m.std + dm.d + diff + dm.c + dm.B
}
```

Law of propagation of uncertainty

The law of propagation of uncertainty (LPU) is the most widely used mechanism for propagating uncertainty. Whereas with the Monte Carlo method the lack of computing and programming skills can form a bottleneck, with the

LPU it is often the calculation of the sensitivity coefficients, i.e. the partial derivatives of the output quantity with respect to the input quantities, that provides a difficulty. Most guidance documents, such as the GUM [4], GUM-S2 [6] and EA 4/02 [26] direct their readers to analytic differentiation of the measurement model to obtain the expressions for calculating the sensitivity coefficients. While this guidance is fully appropriate, it is not always practicable, for many people have lost their skills in differentiation. The fact that there are tables with derivatives of common functions (such as [27, 28]) is barely mentioned in such documents. Numerical approximation of the sensitivity coefficients [7, 8, 10, 29] is a very good alternative, provided that it is done properly. In this section, we show how to use numerical differentiation and the law of propagation of uncertainty to perform the uncertainty evaluation of the mass example of EA 4/02 [26].

The R package `numDeriv` provides the function `grad` (from gradient) that returns from a function a generally good approximation, using Richardson extrapolation [30], of the partial derivatives of the input variables. The function returns a vector holding the values of these partial derivatives. The function passed to `grad` should have only one argument, namely a vector holding all input variables. Hence, the measurement model needs to be reformulated as follows:

where x denotes the vector with input variables. For clarity and convenience, in the function body of `mass2.x` the same symbols have been used as in `mass.x` shown previously. The convenience extends to easier debugging the code as necessary. The penultimate line calculates the result of the function as the sum of the five input variables, just as in the case of the Monte Carlo method.

The uncertainty evaluation itself can be coded as follows:

`m.x.unc`. Again, this last line shows the flexibility of R working with vectors.

The mass of the calibrated weight is 10,000.025 g with standard uncertainty 0.029 g. Using a coverage factor $k = 2$, the expanded uncertainty becomes 0.059 g. These results reproduce those in example S.2 of EA 4/02 to the number of decimal digits given.

The values of the sensitivity coefficients are

```
require(numDeriv)
m.std = 10000.005; dm.d = 0.0; diff = mean(c(0.01,0.03,0.02))
dm.c = 0.0; dm.B = 0.0;
sens = grad(func=mass2.x,x=c(m.std,dm.d,diff,dm.c,dm.B))
m.std.u = 0.0225
dm.d.u = 0.015/sqrt(3); dm.c.u = 0.010/sqrt(3)
```

```
diff.u = 0.025/sqrt(3); dm.B.u = 0.010/sqrt(3)
m.x = mass2.x(c(m.std,dm.d,diff,dm.c,dm.B))
m.x.unc = sqrt(sum(sens^2*c(m.std.u,dm.d.u,diff.u,dm.c.u,dm.B.u)^2))
```

The first line loads the package `numDeriv` (which needs to be installed in RStudio). The next two lines define

```
## [1] 1 1 1 1 1
```

the values of the input quantities. The vector `sens` on the fourth line holds the sensitivity coefficients returned by calling `grad`. The subsequent three lines calculate the standard uncertainties associated with the five input quantities. The penultimate line calculates the estimate of the output quantity `m.x` and the last line its associated standard uncertainty

and are identical to those given in EA 4/02 [26]. The code is also valid for measurement models with non-trivial sensitivity coefficients [8].

The approach described also works with correlated input variables. In that case, the calculation of the standard uncertainty associated with m_X is performed as follows:

```
D = diag(c(m.std.u,dm.d.u,diff.u,dm.c.u,dm.B.u))
CM = D %*% D
tmp = t(sens) %*% CM %*% sens
m.x.unc = sqrt(tmp[1,1])
```

The first two lines form the covariance matrix, diagonal in this case, associated with the five input quantities. (These are only needed to create the covariance matrix; if there were correlations between the five input variables, the code for creating it would have to be adapted accordingly.) The actual implementation of the LPU for correlated input variables is given in the last two lines of the previous code. By vector/matrix multiplication (see also the law of propagation of uncertainty in GUM-S2 [6]), a covariance matrix of dimension 1×1 associated with the output quantity is returned (τ_{mp}). The last line takes the square root of the only element in this matrix (holding the variance of m_X) to obtain the standard uncertainty associated with m_X . This standard uncertainty is 0.029 g.

Interfacing with Word, Excel and LaTeX

This article has been developed with R Sweave using the `knitr` package [31, 32]. R Sweave is implemented in RStudio [18] and enables LaTeX to be combined with R (and other systems). Upon compilation, the PDF (portable document format) reflects the output after executing the R code, compiling the resulting LaTeX file. It has the great advantage that the consistency between data, code and output is preserved [33].

Data collected in mainstream spreadsheet software can be processed in R, for instance, using the `openxlsx` package [34] which provides functions for reading and writing entire Excel workbooks. It supports the `xlsx`-format and also allows R to create new workbooks, fill these and write these.

Producing output in Microsoft Word or HTML format is possible using R markdown [35] and also available in RStudio [18]. The syntax resembles that of Word, with the possibility of including tables, graphs and calculations involving R. Formulæ can be incorporated using a syntax that is similar to that in Word and LaTeX and upon compilation to Word, results in editable formulæ, compatible with the equation editor of MS Word 2007 and later (using the `docx` file format).

Discussion and conclusions

In this tutorial, we have shown how straightforwardly the Monte Carlo method for propagating distributions can be implemented in software. True, it requires some programming skills, but the code provided can serve as a template for more complex applications. Two versions have been given, one working with most programming languages and a second optimised for R.

The framework shown for the mass example can readily be extended to nonlinear measurement models and models

with multiple output quantities. The flexibility also extends to skewed probability density function for the output quantity. In this case, the Monte Carlo method may (correctly) provide a different estimate from just using the measurement model and the estimates of the input quantities. Also, a different 95% coverage interval (see GUM-S2 [6]) may be preferred to the probabilistically symmetric one as used in the examples (which both have an (approximately) symmetric output probability density function for the measurand).

We also have shown how the law of propagation of uncertainty can be used without the need for deriving the analytic expressions for the sensitivity coefficients (the partial derivatives of the output quantity with respect to the input quantity). Code, again in the form of a template, has been provided that after adaptation works with any univariate measurement model and also for the case of correlated input quantities.

Acknowledgements The project 17NRM05 “Examples of Measurement Uncertainty Evaluation” leading to this application has received funding from the EMPIR programme co-financed by the Participating States and from the European Union’s Horizon 2020 research and innovation programme.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. ISO/IEC 17025 (2017) General requirements for the competence of testing and calibration laboratories, 3rd edn. ISO, International Organization for Standardization, Geneva, Switzerland
2. ISO 15189 (2012) Medical laboratories—requirements for quality and competence, 3rd edn. ISO, International Organization for Standardization, Geneva, Switzerland
3. ISO 17034 (2016) General requirements for the competence of reference material producers, 1st edn. ISO, International Organization for Standardization, Geneva, Switzerland
4. BIPM, IEC, IFCC, ILAC, ISO, IUPAC, IUPAP, OIML (2008a) Guide to the expression of uncertainty in measurement, JCGM 100:2008, GUM 1995 with minor corrections. BIPM
5. BIPM, IEC, IFCC, ILAC, ISO, IUPAC, IUPAP, OIML (2008b) Supplement 1 to the ‘Guide to the expression of uncertainty in measurement’—propagation of distributions using a Monte Carlo method, JCGM 101:2008. BIPM
6. BIPM, IEC, IFCC, ILAC, ISO, IUPAC, IUPAP, OIML (2011) Supplement 2 to the ‘Guide to the expression of uncertainty in measurement’—extension to any number of output quantities, JCGM 102:2011. BIPM

7. Boudjemaa R, Cox MG, Forbes AB, Harris PM (2004) Automatic differentiation and its applications to metrology. In: Ciarlini P, Cox MG, Pavese F, Rossi GB (eds) *Advanced mathematical and computational tools in metrology VI*. World Scientific, Singapore, pp 170–179
8. Possolo A (2012) Five examples of assessment and expression of measurement uncertainty. *Appl Stoch Models Bus Ind* 29(1):1–18. <https://doi.org/10.1002/asmb.1947>
9. R Core Team (2019) *R: a language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. <https://www.R-project.org/>
10. *Quantifying Uncertainty in Analytical Measurement* (2012) EURACHEM/CITAC Guide QUAM:2012.P1, 3rd edn
11. Knuth D (2001) *The art of computer programming: semi-numerical algorithms*. Addison-Wesley, Boston
12. Press WH, Flannery BP, Teukolsky SA, Vetterling WT (1992) *Numerical recipes in C: the art of scientific computing*, 2nd edn. Cambridge University Press, Cambridge
13. Wübbeler G, Harris PM, Cox MG, Elster C (2010) A two-stage procedure for determining the number of trials in the application of a Monte Carlo method for uncertainty evaluation. *Metrologia* 47(3):317–324. <https://doi.org/10.1088/0026-1394/47/3/023>
14. Matsumoto M, Nishimura T (1998) Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans Model Comput Simul* 8(1):3–30. <https://doi.org/10.1145/272991.272995>
15. Mélard G (2014) On the accuracy of statistical procedures in Microsoft Excel 2010. *Comput Stat* 29(5):1095–1128. <https://doi.org/10.1007/s00180-014-0482-5>
16. Silverman BW (1986) *Density estimation for statistics and data analysis*. Taylor & Francis Ltd, Abingdon-on-Thames
17. Venables WN, Ripley BD (2002) *Modern applied statistics with S*, 4th edn. Springer, New York
18. R Studio Team (2019) *RStudio: integrated development environment for R*. RStudio Inc, Boston
19. Hetzel JT (2012) Trapezoid: the trapezoidal distribution. <https://CRAN.R-project.org/package=trapezoid>. R package version 2.0-0
20. Genz A, Bretz F (2009) *Computation of multivariate normal and t probabilities*. Lecture notes in statistics. Springer, Heidelberg
21. Carnell R (2017) Triangle: provides the standard distribution functions for the triangle distribution. <https://CRAN.R-project.org/package=triangle>. R package version 0.11
22. Meija J, Coplen TB, Berglund M, Brand WA, Bièvre PD, Gröning M, Holden NE, Irrgeher J, Loss RD, Walczyk T, Prohaska T (2016) Atomic weights of the elements 2013 (IUPAC technical report). *Pure Appl Chem*. <https://doi.org/10.1515/pac-2015-0305>
23. Cohen E, Cvitas T, JG Frey, Holmström B, Kuchitsu K, Marquardt R, Mills I, Pavese F, Quack M, Stohner J, Strauss H, Takami M, Thor A, (2008) *Quantities, units and symbols in physical chemistry*, IUPAC Green Book, 3rd edn. IUPAC & RSC Publishing, Cambridge
24. Possolo A, van der Veen AMH, Meija J, Hibbert DB (2018) Interpreting and propagating the uncertainty of the standard atomic weights (IUPAC technical report). *Pure Appl Chem* 90(2):395–424. <https://doi.org/10.1002/asmb.19470>
25. Bloomfield V (2014) *Using R for numerical analysis in science and engineering*. CRC Press, Taylor & Francis Group, Boca Raton
26. EA Laboratory Committee (2013) EA 4/02 Evaluation of the uncertainty of measurement in calibration. European Cooperation for Accreditation
27. Swishchuk A (2019) Table of basic derivatives. http://people.ualgary.ca/~aswish/AMAT219TABLES_W11.pdf. Accessed 2019-06-05
28. Weast RC (1984) *CRC handbook of chemistry and physics: a ready-reference book of chemical and physical data*, 64th edn. CRC Press Inc, Boca Rato
29. Ellison Stephen LR (2018) metRology: support for metrological applications. R package version 0.9-28-1
30. Richardson LF (1911) IX. The approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam. *Philos Trans R Soc Lond Ser A Contain Pap Math Phys Character* 210(459–470):307–357
31. Xie Y (2018) knitr: a general-purpose package for dynamic report generation in R. <https://yihui.name/knitr/>. R package version 1.21
32. Xie Y (2015) *Dynamic documents with R and knitr*, 2nd edn. Chapman and Hall/CRC, Boca Raton
33. Xie Y (2014) knitr: a comprehensive tool for reproducible research in R. In: Stodden V, Leisch F, Peng RD (eds) *Implementing reproducible computational research*. Chapman and Hall/CRC, Boca Raton
34. Walker A (2018) openxlsx: read, write and edit XLSX Files. <https://CRAN.R-project.org/package=openxlsx>. R package version 4.1.0
35. Allaire J, Horner J, Xie Y, Marti V, Porte N (2018) markdown: 'Markdown' Rendering for R. <https://CRAN.R-project.org/package=markdown>. R package version 0.9

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.