Single Interval Mathematics Package for Octave

December 18, 2008

Contents

1 Author and version

I am Simone Pernice and I am the writer of the SIMP package. If you find any bug or you want to add features, you can contact me at pernice@libero.it. You can find further details on my web site: simonepernice.freehostia.com

List of versions of this document:

- Version 1.0, Turin 14th December 2008, initial draft

2 Preface

I am an electrical engineer, I worked both on hardware and software fields. Designing an electrical circuit is quite complex because the characteristics of the components involved have huge tolerance some about 5%, while others 80%. The target is to have the circuit working in the worst case of the components and of the environment! That involves to compute several times the same equation putting inside the worst or best case values. Some time ago I was doing the usual design when I though, what if I use intervals instead of numbers: I may get the results in just one step! Then I went on the Internet looking for Interval Mathematics: I discovered it was invented in the '50s. Eventually I looked for an open source calculator able to manage intervals instead of numbers. Unfortunately I was not able to find anything. There are just some libraries for C++ language and packages for proprietary software. I was thinking to develop a calculator on my own, but I was not that happy because it would have required a lot of effort. Then I discovered Octave. After a look at its tutorial I decided to write a package to work on intervals in Octave. It was the right choice, because it took just three days to make SIMP while building a new program from scratch would have taken weeks. The package extends the basic arithmetic functions to single interval mathematics, it is called SIMP. Unfortunately it is not possible to overload Octave operators with m files, therefore I made new functions working on matrix for that purpose.

Section 3 shows why numbers are not the best tool when computing real world entities. Section 4 explains what is an interval and how a function can be extended to work on intervals. Section 5 explains some issues to be aware when making interval calculations, which are not present in the number computation. Section 6 shows some example to explain how the SIMP package can be used. Eventually the Section 7 provides the detailed list of functions available.

3 Introduction

Every day we need to compute the result of a lot of simple mathematical equations. For example the cost of the apples bought at the supermarket is given by the apple cost per kilo times the number of kilos bought: $applePrice = appleCostPerKilo \times kilosOfAppleBought$.

When we need the result of those mathematical expressions, we put the values on the right side of the equation and we got its result on the left side. Well, we usually put wrong numbers on the right side and therefore there is no doubt we get wrong results. There are a lot of reasons why we put wrong values, below some of them follows:

1. Most of the values are measured, therefore they are known within a given tolerance (looking for accuracy and precision on Wikipedia will provide interesting information on that matter);
2. Some values have an infinite number of digits after the decimal point, like for π (the ratio between a circumference and its diameter);
3. Some values change with time or samples or whatever, like the weight of a person (which can change of 5% during a day) or the current gain of a BJT (which can change of 50% from the samples of the same series);
4. Some value are estimation or guess: something like between a minimum and a maximum.

For example if a pipe brakes and you want to buy a new one you need its diameter. If you do not have a caliber, you may measure its circumference and divide it by π (3.1415...):$diameter = \frac{circumference}{\pi}$.

Here there are two errors: the circumference is knows with the tolerance given by your meter, moreover π has an infinite number of digits while only few of them can be used in the operation. You may think the error is negligible, the result is enough accurate to buy a new pipe in a hardware shop. However the not infinite accuracy of those operation avoid the use of computers as automatic theorem demonstration tools and so on...

This kind of issue is quite common on engineer design. What engineers do is to be sure their design will work in the worst case or in most of the cases (usually more than 99.9%). Here a simple example. Let us say you want to repaint the walls of your living room completely messed up by your children. You need to compute how many paint cans you need to buy. The equation is quite simple:

$$paintCans = \frac{2 \times (roomWidth + roomLength) \times roomHeight}{paintLitersPerCan \times paintEfficiency} \quad (1)$$

where paintEfficiency is how many square meters of surface can be painted with a liter of paint. The problem here is that usually we do not have a long enough meter to measure the room width and length, it is much simpler to count the number of steps to go through it (1 step is about 1m, let us say from 0.9 to 1.1m). Moreover the paint provider usually declare a paint efficiency range. Let us put below the data:

- roomWidth = 6
- steps roomLength = 4

- steps roomHeight = 3
- meters paintEfficiency = from 0.7 to 1.3 square meters per liter (1 liter per square meter in average)
- paintLitersPerCan = 40

To compute the average result just put average values in (1). We get: $paintCans = \frac{2\times(6+4)\times3}{40\times1} = 1.5$

paint cans, which means two unless you are able to buy just half of a can.

Are you satisfied with that result? I am not. What if I have underestimated something? As every good engineer I would check what happens in the worst case, which means 1.1m step and efficiency of just 0.8. Again just substituting those values in (1), we get: $paintCans = \frac{2\times(6.6+4.4)\times3}{40\times0.7} = 2.36$. That is really interesting: in the worst case I would miss 0.36 cans, it makes sense to buy three cans to avoid to go back to the hardware shop to buy one more (in the worst case).

More happy with the result now? I am not completely satisfied, I am asking myself: what if in the best case I need just 1 can? In that case probably I need more accurate data because the result range would be too wide. Eventually from (1) we get: $paintCans = \frac{2\times(5.4+3.6)\times3}{40\times1.3} = 1.04$. Which means two cans. I am satisfied, I have to buy at least two cans, but probably I may need one more. In the next paragraph you will see how to do all this stuff in one step using Octave and SIMP!

4 Intervals

As you can see in the example above a lot of computations are required to get an idea of the result. To get the worst (and sometime it is required also the best) case, you need to think carefully at the equation because some time you need to put the higher value (for steps) while other times the smallest (for efficiency) to get the worst case and vice versa for the best. There is a much simpler way to work with that issue. You can use intervals instead of number. An interval is:

$$[a1, a2] = \{\forall x \in \mathbb{R} \mid a1 \leqslant x \leqslant a2\}; a1, a2 \in \mathbb{R} \cup \{-\infty, +\infty\}; a1 \leqslant a2; \quad (2)$$

In few words, [a, b] is the set of real numbers among a and b. Please note that intervals suit perfectly in all the cases where numbers do not fit (some were showed in the Section 3). Moreover if it is correct to use a number, it is possible to use a degenerate interval like [a, a]. It is possible also to define functions that work on interval instead of numbers:

$$f([a1, a2], [b1, b2], ...) = \{\forall f(a, b, ...) \in \mathbb{R} \mid \exists a \in [a1, a2], b \in [b1, b2], ...\} \quad (3)$$

Please note, with that definition it is possible to extend every function (like addition, multiplication, square, square root, ...) to work on intervals.

Note also sometime a function may generate several intervals as result, but in this package we will concentrate on single interval function. I will give you an example: $[4,4]/[-2,2] = [-\infty,-2] \cup [2,+\infty]$. However the result is a double interval, for that reason if you try to divide for a interval containing 0 in SIMP, you will get an error.

Eventually note that usually compute the interval result of a function applied to intervals is a long task. Therefore sometime we are satisfied by a bigger interval containing the correct interval. However the target is always to get the smallest interval containing the solution. In the example above you would have:

- roomWidth = [5.4, 6.6] meters
- roomLength = [3.6, 4.4] meters
- roomHeight = 3 = [3, 3] meters
- paintEfficiency = [0.7, 1.3] square meters per liter
- paintLitersPerCan = 40 = [40, 40] liters per can

Now you have just to compute the equation once with extended function to get [1.04, 2.36] cans. SIMP, the Octave package I developed, extends all the basic mathematical functions and it provides functions to extend all other available Octave functions.

5 Issues with interval computation

There are few issues you need to know about interval computations.

5.1 Variables independence

All the variables you use to make a computation are independent from each other, although they are the same instance of variable for you. For example the perimeter of a rectangle (which was also used in a piece of the equation 1) can be written in several ways. If the dimensions of the rectangle are b end h we can write:

$$perimeter = 2(h + b) \tag{4}$$

$$perimeter = h + h + b + b \tag{5}$$

In standard mathematics they are the same, while in interval mathematics we have the interval $equation(4) \subseteqq equation(5)$. The reason is that every interval that appears in the equation is not dependent from each other, therefore in equation 5 is like having four different dimensions (h, h', b and b').

What is important to remember is that: *Every variable should appear just once in the equation if possible in order to get the smallest interval possible as result.*

5.2 Hyper-cube Overlapping

The result of a set of equation involving intervals is *always* an Hyper-cube. For example if we have just two equations in the unknowns x and y the result will be a couple of intervals which draw a rectangle on the Cartesian plane. However in general the solution will be smaller, for example just a segment. In that case the rectangle will contain the solution. There is no way to solve that, just remember the solution is usually a super set of the actual one.

6 Introduction on how the package works

6.1 Installation

First of all you have to install the SIMP package in Octave, so that Octave will automatically load the new functions at every start. To do that just run Octave from the directory where was downloaded the package and execute the following command:

- pkg install simp.tar.gz

6.2 Some example

SIMP works on matrix, interpreted as intervals. The basic interval is: [min, max], which in Octave is a row vector made by two elements. The added functions work on those matrix. For example to add the intervals [5.4, 6.6] and [3.6, 4.4] it is possible to write:

- addInt ([5.4, 6.6], [3.6, 4.4])

$>$ ans = [9, 11]

addInt stands for add intervals. It is possible to use scalar number instead of intervals. Every scalar is computed like: $s = [s, s]$. Therefore the following computations give the same result:

- addInt ([5.4, 6.6], 3.6)

$>$ ans = [9, 10.2]

- addInt ([5.4, 6.6], [3.6, 3.6])

$>$ ans = [9, 10.2]

Octave supports variable declaration, therefore it is possible to write:

- roomWidth = [5.4, 6.6];
- roomLength =[3.6, 4.4];
- addInt (roomWidth, roomLength)

> ans = [9, 11]

Now we know enough from Octave to compute the equation 1:

- roomHeight = 3;
- paintEfficiency = [0.7, 1.3];
- paintLitersPerCan = 40;
- divInt(mulInt(2, addInt (roomWidth, roomLength), roomHeight), mulInt(paintLitersPerCan, paintEfficiency))

> ans = [1.0385, 2.3571]

Sometime the intervals are expressed in terms of tolerance. The electronic resistors are sold in several tolerances: 1%, 2%, 5%. There is a function to easily make an interval from a tolerance. For example a resistor of 10KOhms with 5% of tolerance can be converted in an interval:

- tollToInt (10000, 5)

> ans = [9500, 10500]

If the tolerance is not symmetric, for example +5% and -10%:

- tollToInt (10000, 5, -10)

> ans = [9000, 10500]

All the computations are done among intervals, therefore if a result with tolerance is required it is possible to get back the value and tolerance of an interval in the hypothesis that the tolerance is symmetrical:

- engFormInt([9500, 10500])

> 10K+-5%

Eventually those functions work on vectors of intervals which may be useful to compute functions. It is possible to compute operation on vectors of the same size (element by element) or scalar by vector. In that case the scalar is expanded in a vector of the sime size of the others. That is really useful for the functions where a x vector can be used to compute a y in just one time.

- vect1 = [1 2;3 4;5 6]

> vect1 =

> 1 2

> 3 4

> 5 6

- addInt (vect1, vect1, 1)

> vect1 =

> 3 5

> 7 9

> 11 13

As you can see SMIP can be used just like a calculator. The basic operations are applied on intervals instead of scalar. That makes really simple to evaluate equations where some data is uncertain.

7 Functions

This section describe the functions provided in the package.

7.1 Intervals

An interval is described by a matrix with two columns and a row. The matrix is composed by real numbers. The first value is the interval minimum, the second is the interval maximum. A real number is interpreted like an interval with the same maximum and minimum. It is also possible to write a vector of intervals: it is a matrix with two columns and n rows, where every row is an interval.

7.2 addInt (a, b, c, ...)

addInt adds the given vector of intervals. It returns $a + b + c + \ldots$. They must have the same number of rows or be scalars. At least two vectors are required.

7.3 negateInt (a)

negateInt negates the given vector of intervals. It returns $-a$.

7.4 subInt (a, b, c, ...)

subInt subtracts the given vector of intervals. It returns $a - b - c - \ldots$. They must have the same number of rows or be scalar. At least two vectors are required.

7.5 mulInt (a, b, c, ...)

mulInt multiplies the given vector of intervals. It returns $a \cdot b \cdot c \cdot \ldots$. They must have the same number of rows or be scalar. At least two vectors are required.

7.6 invertInt (a)

invertInt inverts the given vector of intervals. It returns $\frac{1}{a}$.

7.7 invertAddInt (a, b, c, ...)

invertAddInt adds the inverse of the given vector of intervals and then inverts again the result. It returns $\frac{1}{\frac{1}{a}+\frac{1}{b}+\frac{1}{c+\ldots}}$. They must have the same number of rows or be scalar. At least two vectors are required.

7.8 divInt (a, b, c, ...)

divInt divides the given vector of intervals. It returns $\frac{\frac{a}{b}}{c} \ldots$. They must have the same number of rows or be scalar. At least two vectors are required.

7.9 powerInt (a, n)

powerInt rise the given vector of intervals to the power of n. It returns a^n.

7.10 sqrInt (a)

sqrInt squares the given vector of intervals. It returns a^2.

7.11 sqrtInt (a)

sqrtInt square roots the given vector of intervals. It returns $\sqrt{a}$.

7.12 sqrAddInt (a, b, c, ...)

addSqrInt adds the square of the given vector of intervals and then it squares root the result. It returns $\sqrt{a^2+b^2+c^2+\ldots}$. They must have the same number of rows or be scalar. At least two vectors are required.

7.13 engFormInt (a)

engFormInt prints the intervals in the vector a in engineer format. It prints the average between minimum and maximum expressed with engineer notation and add the tolerance in percentage.

7.14 dB10ToLinInt (a)

db10ToLinInt rises 10 to the power of a tens. It returns $10^{\frac{a}{10}}$.

7.15 linToDB10Int (a)

linToDB10Int produces 10 times the logarithm of a. It returns $10log_{10}(a)$.

7.16 dB20ToLinInt (a)

db10ToLinInt rises 10 to the power of a twenties. It returns $10^{\frac{a}{20}}$.

7.17 linToDB20Int (a)

linToDB10Int produces 20 times the logarithm of a. It returns $20log_{10}(a)$.

7.18 valtol100ToInt (v, ptol, ntol)

valtol100ToInt produces an interval with center value given by the scalar v, and positive and negative tolerance express in percentage given by ptol and ntol. ntol is optional, if not present it is assumed equal to ptol.

7.19 intToTol (a)

intToTol produces the tolerance of the vector of interval a, in the hypothesis that the actual value is in the average point.

7.20 intToTol100 (a)

intToTol100 produces the tolerance (expressed in percentage) of the vector of interval a, in the hypothesis that the actual value is in the average point.

7.21 intToVal (a)

intToVal produces the central value of the vector of intervals a, in the hypothesis that the actual value is in the average point.

7.22 linSpaceTol100Int (begin, end, numberOfIntervals, tolerance)

linSpaceTol100Int produces a vector of numberOfIntervals intervals equally spaced between begin and end, with the given tolerance.

7.23 logSpaceTol100Int (begin, end, numberOfIntervals, tolerance)

linSpaceTol100Int produces a vector of numberOfIntervals intervals equally spaced between 10^{begin} and 10^{end}, with the given tolerance.

7.24 monotonicFunctionInt (f, x)

monotonicFunctionInt produces the y interval obtained applying the monotonic function f to the interval x. If f is not monotonic, y may be wrong.

7.25 functionInt (f, x, nOfPoint)

functionInt try to produces the y interval obtained applying the function f to the interval x. The function is checked for monotonicity on the given number of point. nOfPoint is optional, default is 10.

7.26 plotInt (x, y)

plotInt plots two curves. Those curves, for every x interval center point, goes through ymin and ymax. x and y must have the same rows.

7.27 errorBarInt (x, y)

For every x center point, plots the y tolerance.

8 Biography

1. http://en.wikipedia.org/wiki/Interval_arithmetic
2. http://www.cs.utep.edu/interval-comp/hayes.pdf
3. http://www-mdp.eng.cam.ac.uk/web/CD/engapps/octave/octavetut.pdf
4. http://www.gnu.org/software/octave/docs.html