ANNIE CUYT[1] AND BRIGITTE VERDONK[2]

On the integration of floating-point, interval, rational and symbolic computation

In choosing an appropriate computer representation for real numbers to perform scientific computations, two main aspects are usually taken into account: the speed of the computation and the reliability of the output. As is well-known, increasing one usually goes at the expense of the other and there is a trade-off to be made.
Recently several commercially available products have tried to combine the best of several worlds as far as scientific computing is concerned. We refer for instance to communication tools between technical computing environments such as Matlab, symbolic computing environments such as Mathematica and Maple and the traditional programming languages Fortran and C. Two very common remarks about this type of integration are the following. First of all, one cannot freely combine packages and sources of one's own choice. Second but not least, speed is a serious drawback because no executable code of a mixed floating-point and exact program can be generated.
Therefore we have initiated the development of the following two tools. `NumSciLab` *allows for an easy integration of code (C, Fortran, Pascal, Ada, Mathematica, ...) into a customizable library and concentrates on the data integration of different arithmetic representations for numeric and symbolic entities.* `FlPtSim` *was originally designed as a didactical environment offering simulation of floating-point code in different precisions, but now comes with added features such as multiple precision, interval, rational and rational interval arithmetic, all available directly in C++.*
We assume throughout that the reader is familiar with the notions of floating-point, rational, interval and multiple precision arithmetic. In a first section we briefly discuss an example that motivates the need for having different machine representations of numbers at hand. The second and third section respectively summarize the new developments mentioned above.

1. Motivation

The problem we consider, consists of finding the zeroes of the polynomial

$$p_1(x) = (x-1)(x-2)\dots(x-20) - 2^{-23}x^{19} = p_0(x) - 2^{-23}x^{19}$$

This problem was first discussed by Wilkinson [1] who reports that the roots of $p_1(x)$ are given by

1.000000000	$10.095266145 \pm 0.643500904i$
2.000000000	$11.793633881 \pm 1.652329728i$
3.000000000	$13.992358137 \pm 2.518830070i$
4.000000000	$16.730737466 \pm 2.812624894i$
4.999999928	$19.502439400 \pm 1.940330347i$
6.000006944	
6.999697234	
8.007267603	
8.917250249	
20.846908101	

Modifying the coefficient of x^{19} in $p_0(x)$ in the 31^{st} bit apparently drastically influences the roots, shifting 10 of the 20 roots to the complex plane. As explained in [1], this indicates that the problem we are dealing with is very ill-conditioned. There are different ways to obtain reliable bounds for the roots of $p_1(x) = 0$. We have not looked for a specific algorithm for polynomial root finding because our aim is to discuss issues which are independent of the chosen algorithm but related to the particular computer arithmetic representation.

In our discussion we use regula falsi to locate the real roots of $p_1(x) = 0$, because it is a broadly-used and simple method which provides bounds for the roots. Given $(x_\ell, p_1(x_\ell)), (x_h, p_1(x_h))$ such that $p_1(x_\ell) < 0$, $p_1(x_h) > 0$, the regula falsi iteration is given by

$$x_n = x_h + p_1(x_h)\frac{x_h - x_\ell}{p_1(x_\ell) - p_1(x_h)}$$

$$\text{if } p_1(x_n) < 0 \text{ then } x_\ell = x_n \text{ else } x_h = x_n$$

Let us first consider the implementation of regula falsi in floating-point arithmetic. We remark that the coefficients of $p_1(x)$ cannot be represented exactly in IEEE double precision (at least 57 bits precision are needed). But even with the coefficients stored exactly, we still need to evaluate $p_1(x)$ at each iteration point x_n (Horner's scheme is applied). The accumulation of data and rounding errors implies that in floating-point arithmetic, we do not compute the roots of $p_1(x) = 0$, but rather of a perturbed polynomial $\tilde{p}_1(x) = 0$. While the regula falsi method mathematically guarantees to generate bounds for the roots, its implementation in floating-point arithmetic falls short of this goal as can be seen from the following output. We display the bounds $\tilde{x}_\ell$ and $\tilde{x}_h$ generated by the double precision implementation of the regula falsi iteration to locate the root $z_9 = 8.917250249$. Apparently

$$z_9 \notin [\tilde{x}_\ell, \tilde{x}_h] = [8.9172245567595, 8.9172249342899]$$

because

$$\tilde{p}_1(\tilde{x}_\ell) > 0 \ , \ \tilde{p}_1(\tilde{x}_h) < 0 \ \text{ whereas } \ p_1(\tilde{x}_\ell) > 0 \ , \ p_1(\tilde{x}_h) > 0$$

Hence, we need to look at other arithmetic representations in which to implement the regula falsi method, if we want to maintain its mathematical properties.

A possible choice of representation to avoid the above problems is to perform the computations in rational arithmetic. With rational coefficients and only the 4 basic operations $\{+, -, \times, /\}$ involved, all computed entities are rational and no approximations need to be made. Instead of a problem of accuracy, here efficiency causes the method to be non-applicable. This becomes quite evident if we take into account that the numerator and denominator of $p_1(x_\ell)$ and $p_1(x_h)$ quickly become very large. Rational arithmetic clearly needs to be combined with polynomial root-finding algorithms adapted to this representation. Several such algorithms are implemented in Saclib, a library of C programs for computer algebra [2].

Yet another arithmetic representation that can be used to tackle the above problem is interval arithmetic. Using this approach, the coefficients of $p_1(x)$ are represented by enclosing intervals and all computations are performed in interval arithmetic. However, since the problem we consider is very ill-conditioned, the application of regula falsi yields very large intervals which do not give any information on the location of the roots.

As in floating-point arithmetic, we could choose to increase the precision of the representation and perform the evaluations in multiple precision interval arithmetic. This allows to push the rounding error further back to less significant bits and hence the effect of intervals becoming too large is slowed down. Like in multiple precision floating-point arithmetic however, the question of how to determine the precision in order to keep the intervals reasonably bounded is not a trivial one in general.

Also note that feeding the polynomial $p_1(x)$ into a symbolic computing environment does not solve the problem since $p_1(x)$ cannot be factored explicitly.

Conclusions that can be drawn from our analysis do, as can be expected, not favor one approach above another in general. However, what can be concluded is that environments in which different representations are supported in a uniform framework (in such a way that the choice of the representation can be made dependent upon the problem at hand), are to be favoured. Moreover, in order to obtain full-proof reliable answers it is often the case that a mixture of different arithmetic implementations is needed. All this has led us to believe that there is a need for environments in which different arithmetic representations are supported, together with the necessary algorithms adapted to the representations.

2. NumSciLab

Our aim in developing NumSciLab [3], a laboratory for Numeric and Scientific computation, has been to achieve the integration of commercial as well as user–developed numeric, symbolic and graphics software, providing a plug-and-play like desktop-tool where users can experiment with routines from several sources (C, Fortran, Mathematica, ...). In NumSciLab one can set up one's own favourite library, where routines written in several languages can be included. In a so-called description-file one specifies for each routine the origin of its source, the type of the input/output parameters, possible default values etc. Each description file is interpreted by the library manager-program of NumSciLab, which generates from it an executable software library that includes the object codes of all described routines whose source is a traditional programming language. We include an extract of the description file containing some routines for the solution of the above problem:

```
library-name:  NonLinearEqs // Solution of nonlinear equations
sources:       fortran Mathematica C
objects:       regfalsi.o
include:       nonlineareqs.h

function:      regfalsi
source:        C
args:          {dfunc double} f - // f(x)=0
               double a -
               double b -
               double eps -+
               double root +
return-type:   void
solves:        F1 // refers to GAMS [4] taxonomy

function:      Solve
source:        Mathematica
args:          expression eqns -
               expression vars -
return-type:   expression
solves:        F1
solves:        F2
...
```

NumSciLab provides a customizable browser which, upon selection of a particuliar routine by the user, displays a tailor-made communication window for the chosen routine. The user can give input in any format (symbolic, floating-point, rational, interval, hexadecimal, binary, ...). The system converts the user input to the format needed by the routine, and builds the call of the routine fully automatically. In this way one can easily experiment and re-use the same input, once for a Fortran program and afterwards for a Mathematica routine. Data integration is achieved by using a uniform data representation mechanism which is based on parse trees with annotations. This representation makes it possible to easily tackle a problem both with numeric and symbolic routines, or with non–standard computer arithmetic representations for real numbers. The system has full graphic capabilities and lots of extras. It is written in Tcl (Tool Command Language), which combined with Tk, a toolkit for the X11 window system based on Tcl, allows for a uniform development of both the functionality and the graphical user interface of NumSciLab.

3. FlPtSim

FlPtSim, or FLoating-PoinT Simulator, was originally developed as a didactical environment (through a C^{++}-class). In order for the students to get thoroughly acquainted with the notions of computer arithmetic, FlPtSim lets them experiment with the basic operations, conversions and elementary functions, while they can vary the precision and the exponent range of the binary floating-point number set as well as the rounding mode. For every performed operation a lot of information can be provided by FlPtSim: the internal binary representation in every rounding mode including interval mode, the IEEE flags, the exact decimal equivalent of the internal representation as well as its conversion to decimal. Within the simulator it is also possible to execute entire C-programs in a specified set of floating-point numbers rather than in single or double precision and output as much information on the program variables as in the interactive system. In this way students get a feeling for rounding errors, instability, ill-conditioning and more.

The didactical tool FlPtSim has been extended with the aim to become a full-fledged integrating environment which allows to compute in different arithmetic representations [5]. To this end the following C^{++}-classes are provided: the class $Fltps(t, e)$ implements multiple precision floating-point arithmetic where t stands for the precision and e is the number of bits provided for the exponent, the class $Interval(t, e)$ for multiple-precision interval arithmetic, *Rational* for exact rational arithmetic and *RInterval* for interval arithmetic with rational endpoints. Through these C^{++}-classes, different number representations can be combined in a single program. It is important to note in this respect, that the conversion from floating-point to rational is exact, in that every floating-point number is a binary rational number, i.e. a rational number whose denominator is a positive power of 2. Irrational numbers occurring in exact computations are reliably represented by enclosing binary rational intervals and further computations are carried out in rational interval arithmetic. The implementation of these different arithmetic representations as C^{++}-classes makes it platform independent.

Coming back to our motivating example, we remark that the mathematical properties of the regula falsi method can be maintained in its implementation if the function evaluation is performed exactly, while still implementing the iteration formula in floating-point arithmetic. Such a hybrid implementation is now possible using the C^{++}-classes described above, with only very little performance cost compared to a full rational implementation of regula falsi. We list part of the code that realizes this mathematically correct implementation of regula falsi:

```
#include "flpts.h"
#include "rational.h"
....
rational coef[NR_OF_COEF];

flpts f(flpts xx)
{
rational result, x(xx);

result=coef[DEGREE];
for(int i=DEGREE-1;i>=0;i--) {
  result=x*result+coef[i]; }
return result.to_flpts(53,11);
}

void regfalsi(flpts x1,flpts x2, flpts (*f)(flpts x),flpts eps)
{
flpts xh, xl, fl, fh, xnew, fxnew;

...
for (int count=0;count<MAXIT;count++) {
  xnew=xh+fh*(xh-xl)/(fl-fh);
  fxnew=f(xnew);
  if (fxnew<0) {
    xl=xnew;
    fl=fxnew; }
  else {
    xh=xnew;
    fh=fxnew; }
  ...   }
}
```

4. References

1 Wilkinson J. Rounding errors in algebraic processes, Prentice-Hall, Englewood Cliffs, 1963.
2 Collins G. et al. SACLIB user's guide, RISC Technical Report 93-19, 1993.
3 Cuyt A. and Verdonk B. On the integration of software tools for scientific computation, in preparation.
4 Boisvert R., Howe S, and Kahaner D. The guide to available mathematical software problem classification system, *Communications in Statistics – Simulation and Computation*, **20** (1991), 811–842 (http://gams.nist.gov).
5 Cuyt A. and Verdonk B. Using different number representations in a single programming environment, in preparation.

Addresses: Annie Cuyt, Brigitte Verdonk Dept Mathematics and Computer Science, Universiteit Antwerpen (UIA), Universiteitsplein 1, B–2610 Wilrijk-Antwerpen (Belgium) , email: Annie.Cuyt@uia.ua.ac.be, Brigitte.Verdonk@uia.ua.ac.be