Solving Triangular Systems More Accurately and Efficiently

Ph. Langlois, N. Louvet
DALI-LP2A Laboratory. Université de Perpignan.
52, avenue Paul Alduy. F-66860 Perpignan cedex.
philippe.langlois,nicolas.louvet@univ-perp.fr

Abstract - The aim of the proposed paper is to present an algorithm that solves linear triangular systems accurately and efficiently. By accurately, we mean that this algorithm should yield a solution as accurate as if it is computed in twice the working precision. By efficiently, we mean that its implementation should run faster than the corresponding XBLAS routine with the same output accuracy.

Keywords— **IEEE-754 floating point arithmetic, error-free transformations, extended precision, XBLAS, triangular linear system, substitution algorithm.**

I. Improving the result accuracy

When we perform computations with finite-precision arithmetic, *i.e*, floating point arithmetic, the computed values of the intermediate variables often suffer from the rounding errors introduced by each arithmetic operator $+, -, \times, /, \sqrt{}$. These rounding errors contribute to the inaccuracy of the results computed with numerical algorithms.

A. General multiprecision libraries

One natural way to improve this accuracy is to increase the working precision which is often the IEEE-754 double precision format [IEE 85]. For this purpose, many *multiprecision* libraries have been developed. These libraries that only use the fixed precision available on the computer can be divided into two groups according to the chosen representation of the multiprecision quantities.

- Some multiprecision libraries store the numbers in a *multiple-digit* format. It means that the mantissa is represented as a sequence of digits of arbitrary length, coupled with an exponent. Examples of such an implementation are Bailey's MPFUN [BAI 93], Brent's MP [BRE 78] or INRIA's MPFR[MPF] libraries.
- Other libraries store a number as the unevaluated sum of ordinary floating point numbers. This *multiple-component* format is implemented by Priest [PRI 91] and Shewchuck [SHE 97] but have been introduced (with slight differences) from a long time in [MØL 65], [DEK 71], [PIC 72], [LIN 81] for example.

These two types of libraries provide a multiprecision format either with an *a priori* fixed length or an arbitrary (dynamic) length. Allowing an arbitrary precision can drastically reduce the time-performances of the algorithms but is sometimes necessary, for example when computing ultimate digits of numerical constants or quantities.

B. Fixed multiprecision libraries

On the other hand, some applications in scientific computing can get full benefit from only using a small multiple of the working precision. A very illustrating example of such a need is the iterative refinement technique for improving the accuracy of the computed solution $\widehat{x}$ to a linear system $Ax = b$. Quoting Higham [HIG 02, p.232]: "if double the working precision is used in the computation of the residual $b - A\,\widehat{x}$, and the matrix A is not too ill-conditioned, then the iterative refinement produces a solution correct to working precision". Other examples illustrating the benefits of such a targeted increase of the precision are presented in [LI 02].

Well known implementations of these fixed multiprecision software are Bailey's *double-double* and *quad-double* libraries [HID 01]. These double-double or quad-double software provide floating point numbers (and the associated arithmetic) implemented in the multiple-component format using respectively two and four IEEE-754 double-precision numbers. Bailey's double-double algorithms are used by the authors of [LI 02] to implement efficiently the XBLAS library. These extended basic linear algebra subroutines provide the same set of routines as the well known BLAS but allow intermediate computation in extended precision. Double-double numbers implement this extended precision that improves the accuracy and the convergence of some BLAS and LAPACK subroutines [AND 99].

C. Targeted algorithms

Another way to enhance the accuracy of computed results exists but is less general than the previous ones since every improvement has to be designed for a given algorithm. A classic example of such targeted accuracy improvement is the summation of n floating point numbers : many algorithms have been designed to improve the accuracy of the computed sum, *e.g.*, Knuth-Kahan compensated summation, Kahan-Priest double compensated summation, Malcolm or Kulisch large accumulators, ...(see [HIG 02, chap.4] for entries).

These highly accurate algorithms compute correcting terms which take into account the rounding errors accumulated during the calculus. For instance, let x be the (unknown) exact result of a given computation and $\widehat{x}$ be the approximate of x computed in working precision. First we approximate the *global forward error*, $\Delta x = x - \widehat{x}$, and then we

correct x by computing $\widehat{x} + \Delta x$. The corrected result $\overline{x}$ is expected to be more accurate than $\widehat{x}$. Of course both the evaluation of the correcting term Δx and the corrected result $\overline{x}$ have to be performed in finite precision arithmetic: actually $\widehat{\Delta} x = fl(\Delta x)$ and $\overline{x} = fl(\widehat{x} + \widehat{\Delta} x)$ are computed ($fl()$ denotes floating point computations). The computation of these correcting terms relies on well known results about the elementary rounding errors in the arithmetic operators named *error free transformations* by Ogita *et al.* in [OGI 05].

D. The CENA method

Since the propagation of these elementary rounding errors is tedious to describe for large numerical algorithms, methods that automatically bound or even compute a correcting term have been developed [LIN 83], [KUB 96], [LAN 01]. The CENA method provides an automatic linear correction of the global rounding error (together with a bound of the corrected accuracy) thanks to the following properties [LAN 01].

Suppose that the function f is evaluated at the data $x = (x_1, \ldots, x_n)$. For simplicity, we assume that f is a real function, and that all the x_i are floating point numbers. A vector function will be considered componentwise.

$f(x)$ is evaluated by an algorithm $\widehat{f}$ using intermediate variables $\widehat{x}_{n+1}, \ldots, \widehat{x}_N$. Each $\widehat{x}_k$ is assumed to be the result of an elementary operation $+$, $-$, $\times$, $/$ or $\sqrt{}$. We denote by δ_k the corresponding elementary rounding error.

Then, the numerical evaluation of $f(x)$ is $\widehat{x}_N = \widehat{f}(x, \delta)$, that is, a function of the data x and the elementary rounding errors $\delta = (\delta_{n+1}, \ldots, \delta_N)$. A first-order mathematical expression of the global forward error Δx is formally given by the following Taylor expansion with respect to the elementary rounding errors

$$\Delta x_N = f(x) - \widehat{f}(x, \delta) = \sum_{k=n+1}^{N} \frac{\partial \widehat{f}}{\partial \delta_k}(x, \delta) \cdot \delta_k + E_L$$

with E_L the linearization error. The derivatives in this Taylor expansion are efficiently computable using automatic differentiation [GRI 00] in finite precision arithmetic, and the elementary rounding errors are computable efficiently either exactly, or with a good accuracy, also in finite precision arithmetic. Thus a correcting term $\widehat{\Delta} x = fl(\Delta x)$ is calculated using only the working precision available, and a corrected result $\overline{x} = fl(\widehat{x} + \widehat{\Delta} x)$ is produced. Moreover the residual error between the corrected result and the exact result is dynamically bounded using running error analysis.

This linear correction is suitable when the global forward error is dominated by the first-order terms and in particular for *linear algorithms*, *i.e.* algorithms such that its global forward error is a linear function of the elementary rounding errors. In that case, the linearization error E_L is equal to zero, so that the linear correction is optimal. These linear algorithms have been identified in [LAN 01]; let us cite for example the dot-product, the Horner scheme for polynomial evaluation and the substitution algorithm for triangular systems. Experimental results exhibit that a corrected result with the CENA method has a similar accuracy than the one of the not corrected computation performed in twice the working precision [LAN 04]. It means that the corrected accuracy is of the order of the condition number times the square of the working precision. The CENA correction of the classic substitution algorithm for triangular systems satisfies this accuracy estimate. Recent results from Ogita *et al.* propose the first proof of this kind of behavior for the summation and the dot product algorithms [OGI 05].

E. Our goal

One of the main time overhead introduced by the CENA method is the computation of the derivatives with respect to the elementary rounding errors using automatic differentiation –current implementations of the CENA method use overloaded operators to implement the reverse mode of automatic differentiation [GRI 00]. The algorithm we propose here is an optimized instantiation of the CENA correction applied to the substitution algorithm for triangular systems.

In Section II, we first introduce the notations we use. Then we recall some properties of the error free transformations, and next we explicit the correcting term the CENA method computes dynamically. This term represents the global forward error generated by the substitution algorithm when solving a triangular system $Tx = b$. We in-line it to provide a corrected version of the substitution algorithm. This algorithm verifies a complexity in $O(n^2)$ where n is the dimension of the unknown vector. In Section III we present our experimental results, concerning both the accuracy and the timing of our algorithm. Comparisons with the corresponding XBLAS algorithm are proposed. These experimental results show that our algorithm runs about twice faster than the corresponding XBLAS routine with the same output accuracy.

II. An efficient improvement of the result accuracy for triangular linear systems

In the sequel of this paper, we assume a floating point arithmetic adhering to the IEEE-754 floating point standard [IEE 85], with rounding to the nearest floating point value. All the computations are performed in the same working precision. We make the standard assumption that neither overflow, nor underflow occur during the calculus. $\mathbf{F}$ denotes the set of the normalized floating point numbers. Let

$\mathbf{u}$ be the relative error unit: if q is the number of bits of the mantissa in the working precision, then $\mathbf{u} = 2^{-q}$. For instance, if the working precision is IEEE-754 double precision, then $q = 53$ and $\mathbf{u} = 2^{-53} \approx 1.11\,10^{-16}$.

As in the previous section, $fl(.)$ denotes the result of a floating point computation, where all the operations inside the parenthesis are performed in the working precision. We also introduce the symbols $\oplus$, $\ominus$, $\otimes$ and $\oslash$, representing respectively the implemented addition, subtraction, multiplication and addition. We adopt MATLAB like notations for our algorithms. Computed quantities (*i.e.* that suffer of rounding errors) wear a hat.

A. Error free transformations

In this subsection, we review well know results concerning the error free transformations of the elementary floating point operations.

Let $\circ$ be an operation in $\{+, -, \times, /\}$, a and b be two floating point numbers, and $\widehat{x} = fl(a \circ b)$ ($b \neq 0$ when $\circ = /$). The *elementary rounding error* in the computation of $\widehat{x}$ is

$$y = (a \circ b) - fl(a \circ b), \tag{1}$$

that is the difference between the exact result and the computed result of the operation. In particular, for $\circ \in \{+, -, \times\}$, the elementary rounding error y belongs to $\mathbf{F}$, and is computable using only the operations defined on $\mathbf{F}$ [LAN 01]. Thus, for $\circ \in \{+, -, \times\}$, any pair of inputs $(a, b) \in \mathbf{F}^2$ can be transformed into an output pair $(\widehat{x}, y) \in \mathbf{F}^2$ such that

$$a \circ b = \widehat{x} + y \quad \text{and} \quad \widehat{x} = fl(a \circ b).$$

Ogita *et al.* [OGI 05] call such a transformation an *error free transformation* because no information is lost.

For the addition ($\circ = +$), we have a well known algorithm by Knuth [KNU 98], generally called TwoSum (Algorithm 1). TwoSum requires 6 flops (floating point operation).

Algorithm 1: Error free transformation of the sum of two floating point numbers.
function $[x, y]$ = TwoSum(a, b)
$x = a \oplus b$
$z = x \ominus a$
$y = (a \ominus (x \ominus z)) \oplus (b \ominus z)$

For the error free transformation of the product, we first need to split the input arguments into two parts. It is done using Algorithm 2 by Dekker [DEK 71]. If q is the number of bits of the mantissa, let $r = \lceil \frac{q}{2} \rceil$. Algorithm 2 splits a floating point number a into two parts x and y, both having at most $r - 1$ nonzero bits, such that $a = x + y$. For example, with the IEEE-754 double precision, $q = 53$, $r = 27$, therefore the output number have at most $r - 1 = 26$ bits. The trick is that one bit sign is used for the splitting.

Algorithm 2: Splitting of a floating point number into two parts.
function $[x, y]$ = Split(a)
$z = a \otimes (2^r + 1)$
$x = z \ominus (z \ominus a)$
$y = a \ominus x$

Then, Algorithm 3 by Veltkamp (see [DEK 71]) can be used for the error free transformation of the product. This algorithm is commonly called TwoProduct and requires 17 flops.

Algorithm 3: Error free transformation of the product of two floating point numbers
function $[x, y]$ = TwoProduct(a, b)
$x = a \otimes b$
$[a_h, a_l] = \mathsf{Split}(a)$
$[b_h, b_l] = \mathsf{Split}(b)$
$y = a_l \otimes b_l \ominus (((x \ominus a_h \otimes b_h) \ominus a_l \otimes b_h) \ominus a_h \otimes b_l)$

TwoProduct can be rewritten in a very straightforward way with a processor that provides a Fused-Multiply-and-Add operator, such as with Intel Itanium or with IBM PowerPC. For $a, b, c \in \mathbf{F}$, the result of $\mathsf{FMA}(a, b, c)$ is the exact result $a \times b + c$ rounded to the nearest floating point value. Thus $y = a \cdot b - a \otimes b = \mathsf{FMA}(a, b, -(a \otimes b))$ and TwoProduct can be replaced by Algorithm 4, requiring only 2 flops.

Algorithm 4: Error free transformation of the sum of two floating point numbers with an FMA
function $[x, y]$ = TwoProductFMA(a, b)
$x = a \otimes b$
$y = \mathsf{FMA}(a, b, -x)$

Concerning the division, the elementary rounding error is generally not a floating point number, so it cannot be computed exactly. Hence we cannot expect to obtain an error free transformation for the division. However an approximate elementary error, and an approximation bound can be obtained [PIC 76], [LAN 01]. Given $a, b \in \mathbf{F}$, Algorithm 5 computes an approximation $\widehat{y}$ of the elementary rounding error $y = a/b - a \oslash b$ such that

$$|y - \widehat{y}| \leq \mathbf{u}|y|.$$

This means that the computed approximation is as good as we can expect in the working precision. The key is that the numerator of y belongs to $\mathbf{F}$ [PIC 76]. Thus the previous error bound only take into account the rounding error of the last division by b. ApproxTwoDiv requires 21 flops. If a Fused-Multiply-and-Add operation is available, then this flop count drops to 6.

Algorithm 5: Transformation of the division of two floating point numbers
function $[x, y]$ = ApproxTwoDiv(a, b)
$x = a \oslash b$
$[v, w] = \mathsf{TwoProduct}(x, b)$
$y = (a \ominus v \ominus w) \oslash b$

We notice that algorithms TwoSum, TwoProduct, TwoProductFMA and ApproxTwoDiv require only well optimizable floating point operations. They do not use branches, nor access to the mantissa that can be time consuming.

B. Explicit formulation of the forward errors

For the sake of simplicity, the discussion will be made only for lower triangular systems. So, we consider a linear system of the form:

$$\begin{pmatrix} t_{1,1} & & \\ \vdots & \ddots & \\ t_{n,1} & \cdots & t_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

We suppose that all the coefficients of that linear system are representable floating point numbers. Throughout the rest of the paper, n will denote the dimension of the triangular system, *i.e.* the size of the unknown vector. The exact solution $x = (x_1, \ldots, x_n)^T$ can be theoretically computed using the well known substitution formulas

$$x_i = \frac{1}{t_{i,i}} \left[b_i - \sum_{j=0}^{i-1} t_{i,j} x_j \right], \tag{2}$$

for $i = 1 : n$. Using this formula, we can write the classic forward substitution algorithm (Algorithm 6), involving n^2 flops.

Algorithm 6: Substitution algorithm
for $i = 1 : n$
$\widehat{s}_{i,0} = b_i$
for $j = 1 : i - 1$
$\widehat{p}_{i,j} = t_{i,j} \otimes \widehat{x}_j$ { rounding error $\pi_{i,j}$}
$\widehat{s}_{i,j} = \widehat{s}_{i,j-1} \ominus \widehat{p}_{i,j}$ { rounding error $\sigma_{i,j}$}
end
$\widehat{x}_i = \widehat{s}_{i,i-1} \oslash t_{i,i}$ { rounding error δ_i}
end

Algorithm 6 computes an approximation $\widehat{x}$ of the exact solution to the triangular system. Each element $\widehat{x}_i$ of the vector $\widehat{x}$ may be affected by a global forward error $\Delta x_i = x_i - \widehat{x}_i$. We denote by $\Delta x = x - \widehat{x}$ the vector of these forward errors. If the CENA method was used to correct the computed solution $\widehat{x}$, it would first compute an approximation of the vector Δx using automatic differentiation. But to improve the time performance of the correction process, we want here to avoid the use of automatic differentiation. Our goal is thus to obtain explicit formulas for the computation of the vector $\widehat{\Delta} x$.

In Algorithm 6, we indicate after each arithmetic statement the symbol denoting the corresponding elementary rounding error. Thus, for $i = 1 : n$ and $j = 1 : i - 1$

$$\hat{p}_{i,j} = t_{i,j}\hat{x}_i - \pi_{i,j}, \tag{3}$$

$$\hat{s}_{i,j} = \hat{s}_{i,j-1} - \hat{p}_{i,j} - \sigma_{i,j}, \tag{4}$$

and for $i = 1 : n$

$$\hat{x}_i = \hat{s}_{i,i-1}/t_{i,i} - \delta_i. \tag{5}$$

Lemma 1: Let x be the exact solution of the triangular linear system considered, and let $\widehat{x}$ be the approximation of this solution computed by Algorithm 6. Then, for $i = 1 : n$

$$\widehat{x}_i = \frac{1}{t_{i,i}} \left[b_i - \sum_{j=1}^{i-1} t_{i,j}\, \widehat{x}_j + \pi_{i,j} - \sigma_{i,j} \right] - \delta_i.$$

Proof: Inserting equation (3) into equation (4) leads to

$$\hat{s}_{i,j} = \hat{s}_{i,j-1} - t_{i,j}\hat{x}_i + \pi_{i,j} - \sigma_{i,j}.$$

Since $\hat{s}_{i,0} = b_i$ we have

$$\hat{s}_{i,j} = b_i - \sum_{j=1}^{i-1} t_{i,j}\, \widehat{x}_j + \pi_{i,j} - \sigma_{i,j}.$$

Applying equation (5) finally gives the result. ■

Now, we can state the following proposition.

Proposition 2: We use the same notations as in Lemma 1. $\Delta x_i = x_i - \widehat{x}_i$ denotes the forward error affecting the computed result $\widehat{x}_i$. Then, for $i = 1 : n$ the following equation holds

$$\Delta x_i = -\frac{1}{t_{i,i}} \left[\sum_{j=1}^{i-1} t_{i,j} \Delta x_j + \pi_{i,j} - \sigma_{i,j} \right] + \delta_i.$$

Proof: Since $\Delta x_i = x_i - \widehat{x}_i$, from equation (2) and Lemma 1 it follows that

$$\Delta x_i = -\frac{1}{t_{i,i}} \left[\sum_{j=1}^{i-1} t_{i,j}(x_j - \widehat{x}_j) + \pi_{i,j} - \sigma_{i,j}. \right] + \delta_i,$$

which proves the result. ■

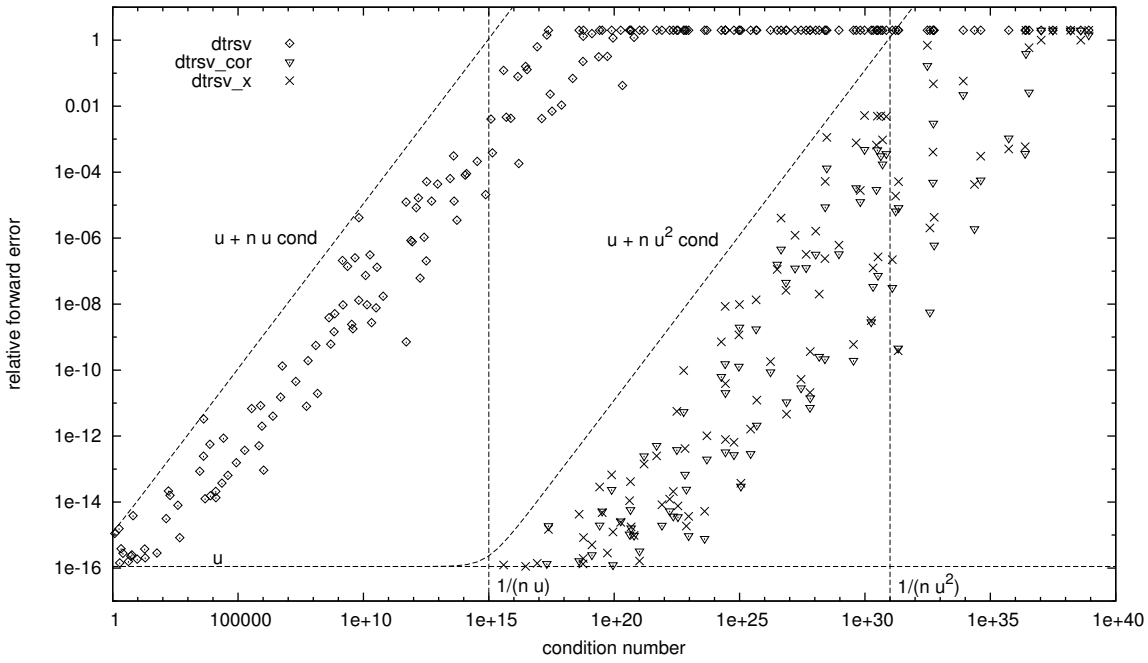

Fig. 1. Result accuracy (Y-axis) for dtrsv (substitution algorithm with IEEE-754 double precision), dtrsv_x (XBLAS routine) and dtrsv_x (corrected substitution) with respect to the condition number (X-axis).

C. The corrected algorithm

Proposition 2 gives explicit formulas to compute the elements of the vector Δx. Moreover, the quantities $\pi_{i,j}$ and $\sigma_{i,j}$ are computable exactly in finite precision arithmetic using algorithms TwoProduct and TwoSum. The rounding errors δ_k affecting each division in the substitution algorithm are also computable with a good accuracy with algorithm ApproxTwoDiv. Thus it is possible to write an algorithm that in-lines the correction of the CENA method. First we compute a floating point approximation $\widehat{\Delta x}$ of Δx according to Proposition 2. Then we compute a corrected solution $\bar{x} = \widehat{x} \oplus \widehat{\Delta x}$.

We can see that it is not useful to correct $\widehat{x}_1$. Indeed, the computed result $\hat{x}_1$ is already the best floating point approximation of x_1. Nevertheless, the correcting term $\widehat{\Delta x}_1$ will play a part in the computation of the other corrected results $\bar{x}_2, \ldots, \bar{x}_n$. All these remarks leads to Algorithm 7.

Algorithm 7: Corrected substitution algorithm

for $i = 1 : n$
  $\widehat{s}_{i,0} = b_i$
  $\widehat{u}_{i,0} = 0$
  for $j = 1 : i-1$
    $(\widehat{p}_{i,j}, \pi_{i,j}) = \mathsf{TwoProduct}(t_{i,j}, \widehat{x}_j)$
    $(\widehat{s}_{i,j}, \sigma_{i,j}) = \mathsf{TwoSum}(\widehat{s}_{i,j-1}, -\widehat{p}_{i,j})$
    $\widehat{u}_{i,j} = \widehat{u}_{i,j-1} \ominus (t_{i,j} \otimes \widehat{\Delta x}_j \oplus \pi_{i,j} \ominus \sigma_{i,j})$
  end
  $(\widehat{x}_i, \widehat{\delta}_i) = \mathsf{ApproxTwoDiv}(\widehat{s}_{i,i-1}, t_{i,i})$
  $\widehat{\Delta x}_i = \widehat{u}_{i,i-1} \oslash t_{i,i} \oplus \widehat{\delta}_i$
end
for $i = 2 : n$
  $\bar{x}_i = \widehat{x}_i \oplus \widehat{\Delta x}_i$
end

The corrected substitution algorithm computes the expected accurate solution according to the CENA method in $O(n^2)$ floating point operations, *i.e.*, in the same theoretical complexity as the uncorrected substitution algorithm. Algorithm 7 involves $\frac{27}{2}n^2 + \frac{21}{2}n - 1$ flops, with n the dimension of the triangular system. If TwoProductFMA is used instead of TwoProduct, then the flop count drops to $6n^2 + 3n - 1$. It also has to be noticed that an extra storage of n floating point numbers is required for the vector $\widehat{\Delta x}$.

III. Experimental results

A. Resulting accuracy of the corrected algorithm

An appropriate condition number for linear system solving has been introduced by Skeel (see [HIG 02, chapter 7]). If $Ax = b$ is a linear system with real coefficients, its condition number is

$$\mathrm{cond}(A, x) = \frac{\| |A^{-1}||A||x| \|_\infty}{\|x\|_\infty}.$$

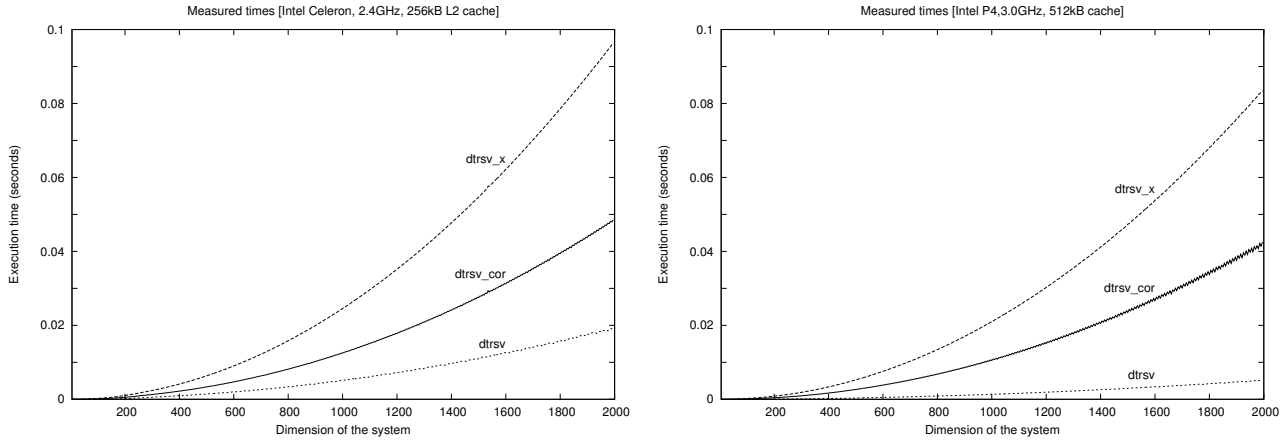

Fig. 2. Measured execution times on the two architectures

In the particular case of a linear triangular system $Tx = b$, we have the following inequality, true to the first order

$$\frac{\|x - \widehat{x}\|_\infty}{\|x\|_\infty} \lesssim n \times \text{cond}(T, x) \times \mathbf{u}, \qquad (6)$$

where $\mathbf{u}$ denotes the rounding unit, and n the size of the linear system. But if the computations are performed internally using twice the working precision (*i.e.* $\mathbf{u}^2$) and then rounded to the working precision (*i.e.* $\mathbf{u}$), then we have to take this final rounding into account. The relative accuracy of the computed solution $\widehat{x}$ now satisfies

$$\frac{\|x - \widehat{x}\|_\infty}{\|x\|_\infty} \lesssim \mathbf{u} + n \times \text{cond}(T, x) \times \mathbf{u}^2. \qquad (7)$$

Inequalities (6) and (7) are used to discuss the accuracy of the computed solutions.

In all our experiments, we use the IEEE-754 double precision as working precision. A family of very ill-conditioned triangular systems has been introduced in [LAN 01]. To perform our experiments here, we have written a random generator of very ill-conditioned triangular systems. This generator has been carefully designed to ensure that both the matrix T and the left hand-side vector b are double floating point numbers. The exact solution x is computed thanks to symbolic computation then carefully rounded to double precision x_d.

We experiment both the classic substitution algorithm performed in double precision dtrsv, the XBLAS dtrsv_x and our corrected routine dtrsv_cor. dtrsv_x is the substitution algorithm with internal computation processed inlining double-double subroutines [HID 01]. The Skeel condition numbers vary from 10 to 10^{40} (these huge condition numbers have a sense since here both T and b have been designed to be exact floating point numbers). Figure 1 presents the relative accuracy $\|\widehat{x} - x_d\|_\infty / \|x_d\|_\infty$ of the computed solution $\widehat{x}$ compared to the condition number range. We set relative errors greater than one to the value one, which means that almost no useful information is left. We observe that both the XBLAS and our corrected substitution algorithms exhibit the expected behavior: the relative accuracy is proportional to the square of the double precision $\mathbf{u}$. The full precision solution is computed as long as the condition number is smaller than $1/(n\,\mathbf{u})$. Then the computed solution has an accuracy of the order $n \times \text{cond} \times \mathbf{u}^2$ for systems with a condition number smaller than $1/(n\,\mathbf{u}^2)$. At last, no computed digit remains exact for condition number up to $1/(n\,\mathbf{u}^2)$.

B. Time efficiency of the corrected algorithm

We compare the actual execution of the proposed corrected algorithm dtrsv_cor with the classic substitution algorithm dtrsv, and with the reference implementation of the XBLAS dtrsv_x routine [LI 02]. The XBLAS library we use comes from the reference web site [XBL]. As the XBLAS library is only available in the C language, all algorithms were tested in C code. All the routines have been tested using IEEE-754 double precision as working precision. The presented tests have been performed with the following environments:

1) Intel Celeron: 3.4GHz, 256kB L2 cache,

2) Intel Pentium 4: 3.0GHz, 1024kB L2 cache.

On the two computers we used the GNU Compiler Collection (gcc-3.4.1), with Linux Kernel 2.6.8 and Glibc 2.3.3.

Figure 2 displays the timing of the three routines (dtrsv, dtrsv_x and dtrsv_cor). The dimension n of the triangular system vary from 5 to 2000. On Figure 3 we present the measured ratios of the actual computing times of the XBLAS dtrsv_x and of our corrected routine dtrsv_cor over the classic substitution dtrsv. The minimum, the mean

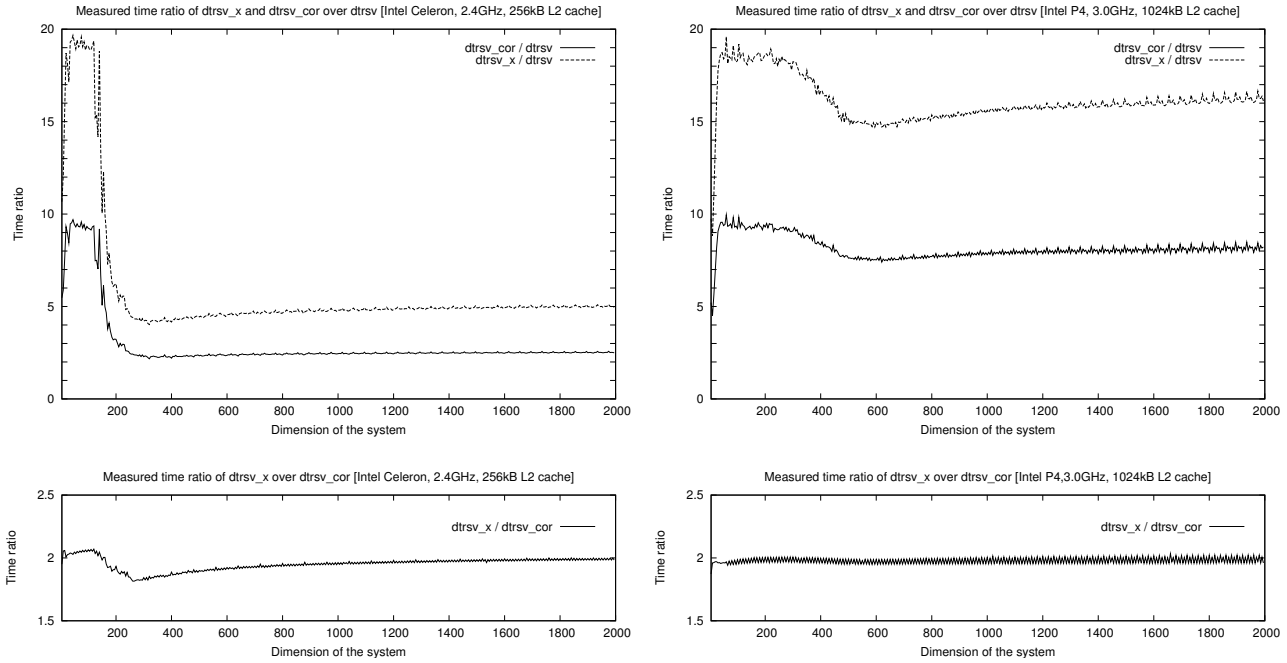

Fig. 3. Measured ratios on the two architectures

TABLE I
MEASURED AND THEORETICAL TIME RATIO ON THE TWO ENVIRONMENTS

Intel Celeron				
ratio	min.	mean	max.	theoret.
dtrsv_cor/dtrsv	2.17	2.95	9.70	13.5
dtrsv_x/dtrsv	4.02	5.81	19.66	22.5
dtrsv_x/dtrsv_cor	1.81	1.95	2.07	1.67
Intel Pentium 4				
ratio	min.	mean	max.	theoret.
dtrsv_cor/dtrsv	4.50	8.14	9.95	13.5
dtrsv_x/dtrsv	8.82	16.10	19.60	22.5
dtrsv_x/dtrsv_cor	1.87	1.98	2.03	1.67

and the maximum of these ratios are reported in Table I. The last column displays the theoretical ratios, resulting from the number of flops involved by the most inner loop of each routine. The Fused-Multiply-and-Add operation is not available in our experimental environments.

First, we have to notice that the actual time factor introduced either by dtrsv_x or dtrsv_cor is always significantly smaller than theoretically expected. This is an astonishing fact since the code for these functions is designed to be easily portable, and no algorithmic optimizations are performed, like data blocking, neither in dtrsv_cor, nor in dtrsv_x (see [LI 02]). This interesting property seems to be due to the fact that the classic substitution algorithm performs only one operation with each element of the matrix, causing suboptimal use of the cache, whereas dtrsv_x or dtrsv_cor perform much more floating point operations with each element of the matrix (see [LI 02], [OGI 05]). Most of these operations are performed at the register level, without incurring much memory traffic.

However, Figure 3 does not allow us to give a definitive conclusion about the time factors introduced either by dtrsv_x or dtrsv_cor. The slowdown factor is about 3 for our corrected algorithm in the first environment (Intel Celeron), whereas it is about 8 in the second (Intel Pentium 4). It is partly due to the fact that dtrsv_cor does not involve exactly the same memory traffic as dtrsv. Indeed, in our corrected algorithm we have to deal with the new vector $\widehat{\Delta}x$: at each execution of the most inner loop, one of its element is accessed . So the time ratio of dtrsv_cor over dtrsv strongly depends on the characteristics of the memory hierarchy of the computer. The same remark apply to dtrsv_x. But a straightforward analysis of the code of dtrsv_x and dtrsv_cor shows that the two routines involve almost the same memory traffic. Thus the time ratio of dtrsv_x over of dtrsv_cor is almost constant, as it can be seen from Figure 3 and Table I. The corresponding theoretical ratio is 1.67 and the measured value is on the average close to 2. From a practical point of view, we can state that the proposed algorithm is about twice faster than the reference time given by the XBLAS current implementation.

IV. CONCLUDING REMARKS

We presented a new algorithm that solves triangular systems accurately. We used only one working precision for

the floating point computations. The only assumption we made was that the floating-point arithmetic available on the computer conformed to the IEEE-754 standard. Experimental results exhibit that our algorithm provides corrected results as accurate as if computed by the classic algorithm using twice the working precision. Its low requirements make it highly portable, and our corrected algorithm could be easily integrated into numerical libraries.

The algorithm is an instantiation of the CENA correction applied to the substitution algorithm. However, we avoid the use of the automatic differentiation to compute the correction terms, which was one the main time overhead introduced by the CENA method. As a result, the corrected algorithm is efficient. We show that it is fast, not only in terms of flops, but also in terms of measured computed time. In particular, the measured slowdown factor introduced by the correction process is much smaller than the expected one. Our experimental results show that our algorithm runs about twice faster than the corresponding XBLAS routine. Even more interesting results should be obtained on computers with a Fused-Multiply-and-Add operator.

REFERENCES

[AND 99] ANDERSON E., BAI Z., BISCHOF C., BLACKFORD S., DEMMEL J., DONGARRA J., CROZ J. D., GREENBAUM A., HAMMARLING S., MCKENNEY A., SORENSEN D., *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, third dition, 1999.

[BAI 93] BAILEY D. H., *Algorithm 719, multiprecision translation and execution of Fortran programs, ACM Transactions on Mathematical Software*, vol. 19, n3, p. 288–319, 1993.

[BRE 78] BRENT R. P., *A fortran multiple-precision arithmetic package, ACM Transactions on Mathematical Software*, vol. 4, n1, p. 57–70, 1978.

[DEK 71] DEKKER T. J., *A Floating-Point Technique for Extending the Available Precision, Numerische Mathematik*, vol. 18, n3, p. 224–242, 1971.

[GRI 00] GRIEWANK A., *Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000, Reprinted in SIGPLAN Notices, 22(2):9-25, 1987.

[HID 01] HIDA Y., LI X. S., BAILEY D. H., Algorithms for quad-double precision floating point arithmetic, BURGESS N., CIMINIERA L., Eds., *Proceedings of the 15th Symposium on Computer Arithmetic*, Vail, Colorado, p. 155–162, 2001.

[HIG 02] HIGHAM N. J., *Accuracy and Stability of Numerical Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second dition, 2002.

[IEE 85] IEEE STANDARDS COMMITTEE 754, *IEEE Standard for binary floating-point arithmetic, ANSI/IEEE Standard 754-1985*, Institute of Electrical and Electronics Engineers, New York, 1985, Reprinted in SIGPLAN Notices, 22(2):9-25, 1987.

[KNU 98] KNUTH D. E., *The Art of Computer Programming: Seminumerical Algorithms*, vol. 2, Addison-Wesley, Reading MA, third dition, 1998.

[KUB 96] KUBOTA K., PADRE2 - Fortran Precompiler for Automatic Differentiation and Estimates of Rounding Error, BERZ M., BISCHOF C., CORLISS G., GRIEWANK A., Eds., *Computational Differentiation: Techniques, Applications, and Tools*, p. 367–374, SIAM, Philadelphia, Penn., 1996.

[LAN 01] LANGLOIS P., *Automatic linear correction of rounding errors, BIT Numerical Mathematics*, vol. 41, n3, p. 515–539, 2001.

[LAN 04] LANGLOIS P., *More accuracy at fixed precision, Journal of Computational and Applied Mathematics*, vol. 162, n1, p. 57–77, 2004.

[LI 02] LI X. S., DEMMEL J. W., BAILEY D. H., HENRY G., HIDA Y., ISKANDAR J., KAHAN W., KANG S. Y., KAPUR A., MARTIN M. C., THOMPSON B. J., TUNG T., YOO D. J., *Design, implementation and testing of extended and mixed precision BLAS, ACM Transactions on Mathematical Software*, vol. 28, n2, p. 152–205, 2002.

[LIN 81] LINNAINMAA S., *Software for doubled precision floating point computations, ACM Transactions on Mathematical Software*, vol. 7, n3, p. 272–283, 1981.

[LIN 83] LINNAINMAA S., *Error linearization as an effective tool for experimental analysis of the numerical stability of algorithms, BIT*, vol. 23, n3, p. 346–359, 1983.

[MØL 65] MØLLER O., *Quasi Double-Precision in Floating Point Addition, Nordisk tidskrift for informationsbehandling*, vol. 5, n1, p. 37–50, 1965.

[MPF] The MPFR Library, Available at http://www.mpfr.org.

[OGI 05] OGITA T., RUMP S. M., OISHI S., *Accurate Sum and Dot Product, SIAM Journal of Scientific Computing*, 2005, (to appear).

[PIC 72] PICHAT M., *Correction d'une Somme en Arithmétique à Virgule Flottante. (French) [Correction of a Sum in Floating-Point Arithmetic], Numerische Mathematik*, vol. 19, p. 400–406, 1972.

[PIC 76] PICHAT M., Contributions à l'étude des erreurs d'arrondi en arithmétique à virgule flottante. (French) [Contributions to the error analysis of rounding errors in floating-point arithmetic], Thèse, Université Scientifique et Médicale de Grenoble, 1976.

[PRI 91] PRIEST D. M., Algorithms for arbitrary precision floating point arithmetic, KORNERUP P., MATULA D. W., Eds., *Proceedings of the 10th IEEE Symposium on Computer Arithmetic (Arith-10)*, Grenoble, France, IEEE Computer Society Press, Los Alamitos, CA, p. 132–144, 1991.

[SHE 97] SHEWCHUK J. R., *Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates, Discrete and Computational Geometry*, vol. 18, n3, p. 305–363, 1997.

[XBL] A Reference Implementation for Extended and Mixed Precision BLAS, Available at http://crd.lbl.gov/~xiaoye/XBLAS/.