Numerical validation in quadruple precision using stochastic arithmetic

S. Graillat\textsuperscript{1}, F. Jézéquel\textsuperscript{1,2}, R. Picot\textsuperscript{1,3}, F. Févotte\textsuperscript{3}, and B. Lathuilière\textsuperscript{3}

\textsuperscript{1} Sorbonne Université, CNRS, Laboratoire d’Informatique de Paris 6, LIP6, F-75005 Paris, France
\textsuperscript{2} Université Panthéon-Assas, 12 place du Panthéon, 75231 Paris CEDEX 05, France
\textsuperscript{3} EDF R&D, 7 boulevard Gaspard Monge, 91120 Palaiseau, France
\{Francois.Fevotte,Bruno.Lathuiliere\}@edf.fr

Abstract

Discrete Stochastic Arithmetic (DSA) enables one to estimate rounding errors and to detect numerical instabilities in simulation programs. DSA is implemented in the CADNA library that can analyze the numerical quality of single and double precision programs. In this article, we show how the CADNA library has been improved to enable the estimation of rounding errors in programs using quadruple precision floating-point variables, \ie having 113-bit mantissa length. Although an implementation of DSA called SAM exists for arbitrary precision programs, a significant performance improvement has been obtained with CADNA compared to SAM for the numerical validation of programs with 113-bit mantissa length variables. This new version of CADNA has been successfully used for the control of accuracy in quadruple precision applications, such as a chaotic sequence and the computation of multiple roots of polynomials. We also describe a new version of the PROMISE tool, based on CADNA, that aimed at reducing in numerical programs the number of double precision variable declarations in favor of single precision ones, taking into account a requested accuracy of the results. The new version of PROMISE can now provide type declarations mixing single, double and quadruple precision.

Keywords: accuracy, floating-point arithmetic, IEEE 754-2008 standard, numerical validation, quadruple precision, rounding errors, stochastic arithmetic.

1 Introduction

Most numerical simulation programs use the single or double precision binary floating-point format, respectively defined in the IEEE 754-2008 standard \cite{13} as \textit{binary32} and \textit{binary64}. However quadruple or arbitrary precision may be required for unstable problems, such as the computation of a chaotic sequence or the approximation of multiple roots of a polynomial. The IEEE 754-2008 standard also defines a quadruple precision binary floating-point format, named \textit{binary128}, with 113-bit mantissa length, including one implicit bit. The numerical stability of programs in single or double precision can be controlled using the CADNA\textsuperscript{1} library \cite{4, 14} that implements Discrete Stochastic Arithmetic (DSA) \cite{21}. CADNA

\textsuperscript{1}http://cadna.lip6.fr
Numerical validation in quadruple precision using stochastic arithmetic Graillat et al.

enables one to estimate rounding errors in a simulation program and to detect numerical instabilities that occur during its execution. CADNA has been successfully used for the numerical validation of academic and industrial simulation codes in various domains. Another library called SAM$^2$ [8] uses DSA to control the numerical stability of arbitrary precision programs. SAM is based on the MPFR$^3$ [6] arbitrary precision library.

In this paper we show how to estimate with CADNA rounding errors in quadruple precision programs. Although the numerical validation of simulations in quadruple precision is possible using SAM with 113-bit mantissa length numbers, the performance cost of arbitrary precision features makes the improvement of CADNA preferable. We describe in this paper the functionalities added to the CADNA library for the control of accuracy of binary128 results. Furthermore we show how these modifications have enabled the improvement of the PROMISE$^4$ software [9, 10] that is based on CADNA and aims at optimizing the precision of variables in numerical codes. Indeed, from a required accuracy of the results, PROMISE automatically determines the appropriate precision for each floating-point variable in a code. Thanks to the CADNA improvement described in this paper, PROMISE can now provide mixed-precision programs with not only declarations in single and double, but also in quadruple precision.

This paper is organized as follows. In Sect. 2 we briefly present the principles of DSA and describe software tools related to DSA: CADNA, SAM, and PROMISE. In Sect. 3 we compare several solutions for performing numerical simulation in quadruple precision: the binary128 floating-point type, the MPFR arbitrary precision library and the QD$^5$ package [11] that provides a double-double data type. In Sect. 4 we describe the new functionalities of CADNA related to the accuracy estimation of quadruple precision results. In Sect. 5 that is devoted to numerical experiments, we present results obtained in quadruple precision with CADNA for the computation of a chaotic sequence, the Hénon map, and the approximation of multiple roots of polynomials. In Sect. 5, we also show the mixed precision configurations provided by PROMISE from quadruple precision programs and the speedups obtained. Finally, we conclude and discuss future works in Sect. 6.

2 DSA and related software

2.1 Principles and validity of DSA

Discrete Stochastic Arithmetic (DSA) [21] enables one to estimate rounding errors during numerical simulations. It consists in executing each arithmetic operation three times with the random rounding mode: each result obtained is randomly rounded up or down with the probability 1/2. Then the number of exact significant digits in the computed result is estimated with a 95% confidence interval using Student’s test. It must be pointed out that in contrast to other approaches, such as interval arithmetic [1], that compute guaranteed error bounds, DSA provides an estimation of rounding errors.

DSA is based on a model that assumes that rounding errors are independent centered uniformly distributed random variables. With the random rounding mode, rounding errors are random variables, but in practice they are not rigorously centered and Student’s test may give a biased estimation of the computed result. However it has been proved [2] that, even if rounding errors are not rigorously centered, the estimation obtained with DSA can be considered correct.

---

$^2$http://www-pequan.lip6.fr/~jezequel/SAM
$^3$http://www.mpfr.org
$^4$http://promise.lip6.fr
$^5$http://crd-legacy.lbl.gov/~dhbailey/mpdist/
Numerical validation in quadruple precision using stochastic arithmetic Graillat et al.

up to one digit.

The accuracy estimation may be invalid if both operands in a multiplication or a divisor are not significant [2]. Therefore during the execution of a program with DSA, the accuracy of all multiplication operands and divisors should be controlled. This dynamical control of multiplications and divisions is the so-called self-validation of DSA.

2.2 Implementation of DSA

The CADNA [4, 14] library enables one to use DSA in programs written in C, C++, or Fortran. CADNA provides new numerical types: the stochastic types. A stochastic variable is composed of three perturbed floating-point values and an integer to store its accuracy. The CADNA library contains the definition of arithmetic operations, order relations, and mathematical functions involving stochastic variables. Operations can be performed on combinations of integers, classical floating-point variables and stochastic variables. However only the accuracy of stochastic variables can be estimated.

The SAM library [8] enables one to estimate with DSA rounding errors in arbitrary precision programs. SAM is based on the MPFR [6] arbitrary precision library and can be used in programs written in C/C++. SAM with 24-bit (resp. 53-bit) mantissa length numbers is similar to CADNA in single (resp. double) precision, except the range of the exponent is only limited by the machine memory. In a program using SAM, the number of exact significant digits in a result can be estimated with the probability 95%, whatever its precision.

Because of the definition of arithmetic operations, mathematical functions, and order relations for stochastic variables in CADNA and SAM, a few modifications are required in user programs to be controlled with DSA: mainly changes in variable declarations and in input/output statements. Both CADNA and SAM can detect numerical instabilities that occur during the program execution. Those instabilities are of different types: some are related to the self-validation of DSA, some are due to numerical noise involved in an order relation or a mathematical function, some are due to a cancellation, i.e. a severe loss of accuracy due to the subtraction of two nearly equal numbers. At the end of the run, each type of instability together with its occurrence is printed.

In the current C/C++ CADNA version, two stochastic types are available: float_st and double_st, respectively associated with single and double precision. The numerical stability of simulations in quadruple precision can be controlled with SAM using 113-bit mantissa length. However our aim is to add features to CADNA in order to enable the use of stochastic variables in quadruple precision, without the performance cost of an arbitrary precision library. Indeed this new CADNA version will provide a quadruple precision stochastic type based on the binary128 type defined in the IEEE 754-2008 standard.

2.3 The PROMISE software

The PROMISE (PRecision OptiMISE) software [9, 10] is based on CADNA and aims at optimizing floating-point type declarations in simulation codes. The sequel relies on Definition 1 that makes clear the notion of decimal significant digits in common between two numbers.

Definition 1. The number of decimal significant digits in common between two real numbers $a$ and $b$ is defined in $\mathbb{R}$ by: for $a \neq b$, $D_{a,b} = \log_{10} \left| \frac{a + b}{2(a - b)} \right|$ and for all $a \in \mathbb{R}$, $D_{a,a} = +\infty$.

Then $|a - b| = \left| \frac{a + b}{2} \right| 10^{-D_{a,b}}$. For instance, if $D_{a,b} = 3$, the relative difference between $a$
and $b$ is of the order of $10^{-3}$, which means that $a$ and $b$ have three significant decimal digits in common.

In the following, we denote by $C$ a set of variables to optimize in a program. A configuration is a bipartition $(C^s, C^d)$ of $C$. The set of all possible configurations is denoted by $\mathcal{R}$.

A configuration $(C^s, C^d)$ is said to be admissible if, when all variables in $C^s$ (resp. $C^d$) are declared in single (resp. double) precision, the resulting program satisfies the following criteria: (i) it is compilable, (ii) it runs without error, and (iii) it yields results meeting accuracy requirements. A testing function $\tau : \mathcal{R} \to \{\checkmark, \times\}$ determines if a configuration is admissible (✓) or not (✗).

The implementation of PROMISE relies on the Delta-Debugging (DD) algorithm [22], and more specifically its DD$\text{max}$ variant: being given a set of variables $C$ and a testing function $\tau$, calling DD$\text{max}(C, \tau)$ yields a configuration $(C^s, C^d)$ which is admissible and locally maximal in the sense that no variable in $C^d$ can be moved to $C^s$ without causing the configuration to become inadmissible.

PROMISE is implemented in two versions, which differ in how the source code is modified and how the accuracy requirements are checked for in the testing function. For $n$ required exact significant digits, the testing function $\tau$ can be defined in two ways:

- **Full stochastic:** for every tested configuration, a fully CADNA-instrumented version of the code is generated. A configuration is admissible if the number of exact significant digits estimated using DSA is at least $n$, and if these digits are in common, in the sense of Definition 1, with a reference DSA computation in double precision.

- **Stochastic reference:** only the reference result is computed in high-precision DSA. The evaluation of every tested configuration is performed in standard floating-point arithmetic. A configuration is admissible if it produces results which have at least $n$ significant digits in common, in the sense of Definition 1, with the reference results.

## 3 Quadruple precision arithmetic

### 3.1 The binary128 format and its implementation

Several binary types are specified in the IEEE 754-2008 standard, such as binary32, binary64, and binary128, defined respectively on 32, 64, and 128 bits. A binary128 floating-point number consists of a sign bit $s$, a 15-bit long exponent $e$, and a 112-bit long mantissa $m$. Although 112 bits are explicitly stored to represent the mantissa, its precision is actually 113 bits. Indeed like for binary32 and binary64 numbers, binary128 numbers benefit from an implicit bit that is set to 1, except for the representation of zero or subnormal numbers.

A few 128-bit processors exist: up to our knowledge, only SPARC V8 [19] or V9 [20] and IBM POWER9 [18] processors feature 128-bit registers. On other architectures, the binary128 floating-point type is emulated. We describe in this section how the binary128 format is implemented by the GCC compiler$^6$ on a 64-bit processor.

In the GCC compiler, a binary128 number is defined as a bit field structure that consists of a sign bit, a 15-bit long integer for the exponent, a 48-bit long integer for the high part of the mantissa, and a 64-bit long integer for its low part. As already mentioned, although the mantissa length is physically 112 bits, its actual length is 113 bits. Arithmetic operations on binary128 numbers require several steps. First, the 4 elements that define each binary128

---

$^6$GCC, the GNU Compiler Collection: [http://gcc.gnu.org](http://gcc.gnu.org)
Numerical validation in quadruple precision using stochastic arithmetic Graillat et al.

number are extracted from the bit field structure. From the exponents, particular cases specified in the IEEE 754 standard may be identified: some operations involving ±0, ±∞, or NaN (Not a Number) that is associated with an undefined or unrepresentable value. In such cases, the result is rapidly returned. Otherwise arithmetic operations are carried out on the integers representing the binary128 numbers. Rounding is performed before the creation of the bit field structure of the result. If an exception such as overflow or underflow occurs, a status flag may be raised.

3.2 The double-double format and its implementation

A double-double number \(a\) is a pair \((a_h, a_l)\) of IEEE-754 floating-point numbers where \(a_h\) and \(a_l\) do not overlap, i.e. \(a = a_h + a_l\) with \(|a_l| \leq 2^{-53}|a_h|\). Algorithms for arithmetic operations involving double-double numbers rely on error-free transformations. It is known that for the basic operations +, −, ×, the approximation error of a floating-point operation is still a floating-point number (see for example [3]). From a pair of floating-point operands, an error-free transformation provides both the result of the floating-point operation and the associated rounding term. However double-double results are not necessarily rounded as specified in the IEEE 754 standard.

There are several implementations for the double-double library. The difference lies in the fact that the lower-order terms are treated in different ways. The main reference is the QD library [11] that is used in the performance comparison presented in Sect. 3.3. In the QD library there exist several implementations of double-double operations, in particular, for addition and division, the sloppy version that performs better, at the price of a possibly higher error. These sloppy algorithms should be used with parcimony. Indeed with the sloppy version of the addition, there is no proof that the relative error on the result is bounded if the operands have opposite signs [15].

3.3 Performance comparison of quadruple precision programs

Performance comparison is performed on an Intel Xeon E5-2660 processor at 2.2 GHz and 32 GB RAM with the GNU compiler G++ 4.8.5 and the Intel compiler ICC 17. The MPFR arbitrary precision library, version 3.1.1, is used with 113-bit mantissa length numbers. Performances of two C++ codes are measured. Matrix performs a naive multiplication of two square matrices of size 1,000. Map computes the sequence defined by \(U_0 = 1\) and for \(i = 1, \ldots, n, U_i = (0.1 \times U_{i-1} - (1/3+U_{i-1})^2)/(1-U_{i-1})^3\) with \(n = 128,000,000\). This sequence has been defined in order to perform all arithmetic operations while avoiding optimizations such as vectorization. In this code, no raising to power is computed using the pow mathematical function, multiplications are performed instead. Figure 1 presents the performance ratio w.r.t. double precision (binary64) of the binary128 type provided by GCC or ICC, the double-double type of the QD library, and MPFR with 113-bit mantissa length. Compilation is performed with the O0 or the O3 option.

Whatever the compiler and the compilation option, MPFR version 3.1.1 with 113-bit mantissa length performs the worst. Its performance ratio w.r.t. binary128 varies from 4 to 13. As a remark, since our performance measurements, MPFR has benefited from performance improvements [16]. In particular, from version 3.1.5 to version 4.0-dev, the number of cycles for arithmetic operations (+, −, ×, ÷) on numbers with 113-bit mantissa length has been divided by at most 2. However in the performance tests presented in this section, MPFR 4 would still be costful compared to the binary128 type of GCC or ICC, or the double-double type of the QD library.
Numerical validation in quadruple precision using stochastic arithmetic Graillat et al.

Figure 1: Performance ratio w.r.t double precision (binary64) of binary128, MPFR with 113-bit mantissa length, and double-double

From Figures 1a and 1b, if the compilation option O0 is chosen, binary128 performs the best. However, in our experiments, its average cost w.r.t. binary64 is approximately a factor 12. Its performance ratio with respect to the double-double type varies from 1.5 to 4. It can be noticed that with the double-double type, the performance gain of the sloppy version is low: from 0.5% to 13%.

From Figure 1, for matrix multiplication, the performance ratio of binary128 over binary64 is 4 or 8 times higher, depending on the compiler, with the O3 option than with O0. Indeed, on the one hand, matrix multiplication is particulary well optimized with the O3 option for the binary32 or the binary64 format. On the other hand, whatever the optimization option, each arithmetic operation in binary128 implies to access data structures to read the operands and write the result.

From Figures 1c and 1d, because the double-double program for matrix multiplication also benefits from a particularly good optimization with the O3 option, it performs 1.5 or 3 times better, depending on the compiler, than its binary128 counterpart. However, with the Map application, performances of binary128 and double-double are similar. With double-double, the performance gain of the sloppy version is higher with the O3 option than with O0. With O3, it varies from 8% to 34%.

Although double-double performs better, in particular for matrix multiplication, than binary128 with the O3 option, it must be pointed out that the binary128 format is defined in the IEEE 754-2008 standard, whereas the double-double type of the QD library does not adhere to that standard. Indeed double-double computation cannot be performed with the rounding modes defined in that standard. Therefore, as described in Sect. 4, the quadruple precision

From Figures 1a and 1b, if the compilation option O0 is chosen, binary128 performs the best. However, in our experiments, its average cost w.r.t. binary64 is approximately a factor 12. Its performance ratio with respect to the double-double type varies from 1.5 to 4. It can be noticed that with the double-double type, the performance gain of the sloppy version is low: from 0.5% to 13%.

From Figure 1, for matrix multiplication, the performance ratio of binary128 over binary64 is 4 or 8 times higher, depending on the compiler, with the O3 option than with O0. Indeed, on the one hand, matrix multiplication is particularly well optimized with the O3 option for the binary32 or the binary64 format. On the other hand, whatever the optimization option, each arithmetic operation in binary128 implies to access data structures to read the operands and write the result.

From Figures 1c and 1d, because the double-double program for matrix multiplication also benefits from a particularly good optimization with the O3 option, it performs 1.5 or 3 times better, depending on the compiler, than its binary128 counterpart. However, with the Map application, performances of binary128 and double-double are similar. With double-double, the performance gain of the sloppy version is higher with the O3 option than with O0. With O3, it varies from 8% to 34%.

Although double-double performs better, in particular for matrix multiplication, than binary128 with the O3 option, it must be pointed out that the binary128 format is defined in the IEEE 754-2008 standard, whereas the double-double type of the QD library does not adhere to that standard. Indeed double-double computation cannot be performed with the rounding modes defined in that standard. Therefore, as described in Sect. 4, the quadruple precision

43
extension of CADNA is based on the binary128 format of the IEEE 754-2008 standard.

4 DSA and related software in quadruple precision

4.1 Extension of CADNA to quadruple precision

In order to extend CADNA to quadruple precision, i.e. in order to define a new quad type stochastic type associated to quadruple precision, a few non-trivial aspects need to be taken care of. First, a utility library is incorporated into CADNA, that allows manipulating quadruple precision numbers as easily as other standard floating-point numbers. Quadruple precision floating-point numbers are named differently in GCC (float128) and Intel (Quad) compilers. In order to provide uniform access to these types, we use the float128 type from the Boost::Multiprecision library. This type is a thin wrapper around the underlying quadruple precision floating-point type, and incurs no overhead with respect to the performance of native types. Moreover, all mathematical functions working on quadruple precision are suffixed with the letter q (e.g. \texttt{logq} or \texttt{cosq}). In order to allow to access mathematical functions under their usual names, we also provide overloaded definitions of standard functions for the float128 type.

With this being in place, stochastic arithmetic can be implemented for quadruple precision in the same way as single and double precision. Since version 2.0.0 of CADNA, for performance reasons, no explicit rounding mode switching is performed during program execution [4]. The CPU rounding mode is set towards plus infinity during program initialization, and rounding mode towards minus infinity is emulated taking advantage of properties such as \( a \oplus_{+\infty} b = -(a + b) \) and \( a \otimes_{-\infty} b = -(a \otimes b) \) where \( \oplus_{+\infty} \) and \( \otimes_{+\infty} \) (resp. \( \oplus_{-\infty} \) and \( \otimes_{-\infty} \)) are the floating-point operations rounded towards \( +\infty \) (resp. \( -\infty \)). Such a technique can easily be applied to quadruple precision numbers, provided that an efficient implementation is available to flip the sign of a number. In CADNA, this operation is performed by bitwise XORing the floating-point number with an appropriate mask.

Like in the single and double precision version of CADNA, arithmetic operations involving any combination of numerical types with at least one stochastic type are overloaded. Indeed it is possible to mix in arithmetic operations integers, classical floating-point or stochastic variables, in single, double or quadruple precision.

A last detail which must be accounted for is the formatting of quadruple precision numbers for output. Standard \texttt{printf}-like C functions and the standard C++ streaming operator (\texttt{<<}) indeed do not properly handle quadruple precision floating-point numbers. Therefore CADNA has been modified in order to enable to print quadruple precision classical or stochastic variables using the \texttt{printf} C function and the C++ streaming operator. These new functionalities are based on the specialized \texttt{quadmath_snprintf} function.

4.2 Performance of CADNA in quadruple precision

The performance of CADNA is measured on the workstation already mentioned in Sect. 3.3 with the same compilers and the compilation option O3. Table 1 presents the performance ratio of CADNA execution w.r.t. classic floating-point execution for both codes Matrix and Map previously described in Sect. 3.3. Three instability detection levels are chosen with CADNA: detection of no instability (instability detection is deactivated); self-validation (as introduced in Sect. 2.1, instabilities in multiplications and divisions are detected); detection of all kinds
of instabilities. As a remark, with both codes, unstable multiplications and cancellations are detected whatever the precision chosen (single, double or quadruple).

If instability detection is deactivated, CADNA overhead in single or in double precision varies from 7 to 20 depending on the code and the compiler. This overhead is mainly due to the extra computation inherent to CADNA, and furthermore codes with classic floating-point types in single or in double precision may benefit from better optimizations than their CADNA counterparts that use stochastic types. With the same instability detection level, CADNA overhead is lower in quadruple precision: it varies from 4 to 8. This better overhead in quadruple precision can be explained by the fact that the binary128 format is implemented as a data structure. Each arithmetic operation in classic floating-point quadruple precision requires accesses to such a structure. Therefore classic floating-point codes in quadruple precision do not take benefit from the same optimizations as in single or in double precision.

With CADNA, instability detection introduces extra execution time and this cost is not particularly higher in quadruple precision. The additional execution time due to self-validation (w.r.t. no instability detection) varies from 7% to 56% depending on the code, the compiler and the precision chosen. The overhead of the detection of all kinds of instabilities (w.r.t. no instability detection) is a factor 1.4 to 4.8. This cost is mainly due to the detection of cancellations that implies to compute for each addition or subtraction the number of correct digits of the operands and the result.

Table 2 presents the execution time of the codes Matrix and Map compiled with GCC and ICC using CADNA in quadruple precision and SAM with 113-bit mantissa length. As expected, whatever the compiler and the code, CADNA performs better than SAM. If instability detection is deactivated, the performance ratio varies from 3 to 7. This can be explained by the performance ratio between the binary128 computation and the MPFR library already mentioned in Sect. 3.3. Self-validation induces extra computation, and as a consequence, the ratio of the execution time with SAM over the one with CADNA increases and it increases more if all kinds of instabilities are detected: in this case, it varies from 6 to 30. As a remark, the performance ratio between CADNA and SAM is higher with ICC than with GCC. This difference is due to the better performance with ICC than with GCC of the codes using CADNA, whereas the execution times of the codes using SAM are similar with both compilers. As a remark, in our performance measurements, SAM is based on MPFR version 3.1.1. As already mentioned in Sect. 3.3, with 113-bit mantissa length numbers, SAM would benefit from the performance improvements of MPFR 4 [16]. However CADNA would remain competitive compared to SAM.
Table 2: Execution time in seconds using CADNA in quadruple precision and SAM with 113-bit mantissa length

<table>
<thead>
<tr>
<th></th>
<th>CADNA</th>
<th>SAM</th>
<th>CADNA</th>
<th>SAM</th>
<th>CADNA</th>
<th>SAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GCC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>matrix</td>
<td>240</td>
<td>679</td>
<td>251</td>
<td>754</td>
<td>1144</td>
<td>7387</td>
</tr>
<tr>
<td>map</td>
<td>133</td>
<td>547</td>
<td>207</td>
<td>2148</td>
<td>409</td>
<td>4684</td>
</tr>
<tr>
<td>ICC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>matrix</td>
<td>163</td>
<td>698</td>
<td>170</td>
<td>724</td>
<td>791</td>
<td>7593</td>
</tr>
<tr>
<td>map</td>
<td>90</td>
<td>597</td>
<td>133</td>
<td>2145</td>
<td>160</td>
<td>4754</td>
</tr>
</tbody>
</table>

4.3 Extension of PROMISE to quadruple precision

The PROMISE tool described in Sect. 2.3 can be extended to handle a third floating-point number type and optimize variable declarations between single, double and quadruple precision. The objective now consists in partitioning the full set of variables \( C \) into three subsets \( C^s \), \( C^d \) and \( C^q \) of variables whose precision can be respectively set to single, double or quadruple while still producing an accurate enough result. A problem posed by such an extension is that the \( \text{DD}_{\text{max}} \) algorithm produces configurations that are bipartitions of the variables set, whereas in our new setup, tested configurations should be tripartitions of the variables set.

The strategy used in this work to overcome this difficulty is to perform two Delta-Debugging steps in a row, as presented in Algorithm 1: in a first stage, a \( \text{DD}_{\text{max}} \) algorithm is used to test the full set of variables \( C \) for downgrade from quadruple to double precision. This allows to determine a maximal subset \( C^d_0 \) of variables which can be downgraded to double precision. Its complement \( C^q \) contains variables which must remain in quadruple precision for the result to meet accuracy requirements. In a second stage, variables in \( C^d_0 \) are tested for downgrade from double to single precision.

Algorithm 1 Two-tier Delta-Debugging algorithm

**Input:** \( C \): set of variables to optimize

**Output:** \((C^s, C^d, C^q)\): admissible configuration in \( C \)

\[
(C^d_0, C^q) = \text{DD}_{\text{max}} (C, \tau_{q\rightarrow d})
\]

\[
(C^s, C^d) = \text{DD}_{\text{max}} (C^d_0, \tau_{d\rightarrow s})
\]

return \((C^s, C^d, C^q)\)

The same Delta-Debugging algorithm is used for both stages; only the set of variables and the test functions are adjusted. A global testing function \( \tau \), shown in Algorithm 2, defines how to test the validity of any given configuration \((C^s, C^d, C^q)\). The testing functions used in each Delta-Debugging stage are simply defined as partial applications of this global test function: \( \tau_{q\rightarrow d}(C^d, C^q) = \tau (\emptyset, C^d, C^q) \) and \( \tau_{d\rightarrow s}(C^s, C^d) = \tau (C^s, C^d, C^q) \).

As for the two-precision version, this new version of PROMISE comes in two flavors: “full stochastic” and “stochastic reference”. These versions are defined in the same way as in Sect. 2.3, except reference results are now computed in quadruple precision DSA.
Algorithm 2 Global testing function \( \tau_n \)

Inputs: \( n \): requested number of exact significant digits
\((C^s, C^d, C^q)\): configuration to test

Output: admissibility of the given configuration

if \((C^s, C^d, C^q)\) was previously tested then
  return the result in cache
end if

Update the source code according to \((C^s, C^d, C^q)\)
if the source code compilation fails then
  return \( \times \)
end if

Run the modified program
if the execution fails then
  return \( \times \)
else if the result meets accuracy requirements of \( n \) significant digits then
  return \( \checkmark \)
end if

return \( \times \)

5 Numerical experiments

In this section, we first present quadruple precision results obtained with CADNA for the
computation of a chaotic sequence, the Hénon map, and the approximation of multiple roots
of polynomials. Then we show that the PROMISE software can now provide, from initial
numerical programs in C/C++, new programs with type declarations possibly mixing single,
double and quadruple precision, depending on the requested accuracy of the results.

5.1 Hénon map

The Hénon map is a discrete-time dynamical system \([12]\). It maps a point \((x_i, y_i) \in \mathbb{R}^2\) to a
new point defined by \(x_{i+1} = 1 + y_i - ax_i^2\) and \(y_{i+1} = bx_i\). Depending on the parameters \(a\) and \(b\),
the system can be regular or chaotic. Here we focus on the values of the classical Hénon map
obtained with \(a = 1.4\) and \(b = 0.3\), starting from \(x_0 = 1\) and \(y_0 = 0\). With those parameters,
the map is considered as chaotic.

Figure 2 shows the number of exact significant digits estimated by CADNA in the results
\(x_i\) obtained using single, double and quadruple precision. One can observe that the accuracy
of \(x_i\) regularly decreases with the number \(i\) of iterations performed. A similar figure, that is
not displayed here, describes the loss of accuracy of the results \(y_i\). At a certain iteration, which
depends on the precision chosen, the results have no more correct digits.

Table 3 presents at different iterations the results \((x_i, y_i)\) computed using CADNA with
single, double or quadruple precision. CADNA displays only the exact significant digits of
results and \(\oplus.0\) if they have no more correct digits. Numerical noise appears at iteration 30,
75, and 175, respectively in single, double and quadruple precision. As a remark, for any two
precisions, the ratio between these iteration numbers is close to the ratio of the corresponding
mantissa lengths.
Numerical validation in quadruple precision using stochastic arithmetic Graillat et al.

Figure 2: Accuracy estimated by CADNA of coordinates $x_i$ of the Hénon map with $a = 1.4$, $b = 0.3$, $x_0 = 1$, and $y_0 = 0$.

<table>
<thead>
<tr>
<th>iteration</th>
<th>precision</th>
<th>point $(x_i, y_i)$ computed using CADNA</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>single</td>
<td>$x_i$ @ 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y_i$ 0.2E+00</td>
</tr>
<tr>
<td>30</td>
<td>double</td>
<td>$x_i$ -0.13848191E+000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y_i$ 0.2856319104E+000</td>
</tr>
<tr>
<td>30</td>
<td>quad</td>
<td>$x_i$ -0.138481919146792462486489312E+000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y_i$ 0.2856319104003007180980589904E+000</td>
</tr>
<tr>
<td>75</td>
<td>double</td>
<td>$x_i$ @ 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y_i$ -0.1E+00</td>
</tr>
<tr>
<td>75</td>
<td>quad</td>
<td>$x_i$ -0.115649947336564503E+000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y_i$ -0.1839980672458806840E+000</td>
</tr>
<tr>
<td>175</td>
<td>quad</td>
<td>$x_i$ @ 0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y_i$ -0.2E+00</td>
</tr>
</tbody>
</table>

Table 3: At different iterations, points $(x_i, y_i)$ of the Hénon map computed using CADNA with single, double or quadruple precision ($a = 1.4$, $b = 0.3$, $x_0 = 1$, and $y_0 = 0$).

5.2 Multiple roots of polynomials

Newton’s method enables one to compute a root of a function $f$. From a root approximation $x_0$, it consists in computing the sequence $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. The stopping criterion usually relies on a threshold $\varepsilon$ and may be for instance $|x_{n+1} - x_n| < \varepsilon$ or $|\frac{x_{n+1} - x_n}{x_{n+1}}| < \varepsilon$ if $x_{n+1} \neq 0$. With DSA, the sequence can be computed until the difference between two successive approximations $x_n$ and $x_{n+1}$ is not significant. In this case, the transformation of $x_n$ into $x_{n+1}$ is due to rounding errors and further iterations are useless: the number of iterations has been optimized.

The numerical quality of an approximation provided by Newton’s method depends on the root multiplicity. The accuracy of a multiple root obtained in double precision may be unsatisfactory and quadruple precision may be required. Theorem 1 established in [7] is based on Definition 1. It gives a relation between the common digits of two successive approximations of a multiple root computed using Newton’s method and the common digits of an approxima-
tion and the root. In Theorem 1, $\sim_{\infty}$ means that quantities are asymptotically equivalent as $n \to \infty$.

**Theorem 1.** Let $x_n$ and $x_{n+1}$ be two successive approximations computed using Newton’s method of a polynomial root $\alpha$ of multiplicity $m \geq 2$. Then

$$D_{x_n,x_{n+1}} \sim_{\infty} D_{x_{n+1},\alpha} + \log_{10}(m - 1).$$

From Theorem 1, if the convergence zone is reached, then the digits common to two successive approximations $x_n$ and $x_{n+1}$ are also in common with the exact root $\alpha$, up to $\delta = \lceil \log_{10}(m - 1) \rceil$, as illustrated in Figure 3.

![Figure 3: Representation of the last iterates $x_n$ and $x_{n+1}$ of Newton’s method, and the first digits of the exact multiple root $\alpha$: if $x_n - x_{n+1}$ is not significant, then the digits of $x_{n+1}$ which are not affected by rounding errors are in common with $\alpha$, up to $\delta$.](image)

Table 4 presents for each root $\alpha_i$ of $P(x) = (3x - 1)^3$, the value $\delta_i = \lceil \log_{10}(m_i - 1) \rceil$, the number of significant digits not affected by rounding errors estimated by CADNA of its approximation by Newton’s method, and the number of significant digits in common with the exact root according to Definition 1.

<table>
<thead>
<tr>
<th>precision</th>
<th>$\alpha_1 = 1, \delta_1 = 0$</th>
<th>$\alpha_2 = 1/3, \delta_2 = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CADNA exact</td>
<td>CADNA exact</td>
</tr>
<tr>
<td>single</td>
<td>4 3.1</td>
<td>3 2.2</td>
</tr>
<tr>
<td>double</td>
<td>7 7.3</td>
<td>5 5.0</td>
</tr>
<tr>
<td>quad</td>
<td>17 16.6</td>
<td>12 11.0</td>
</tr>
</tbody>
</table>

Table 4: For each root of $P(x) = (x - 1)^2(3x - 1)^3$, number of digits not affected by rounding errors estimated by CADNA and number of digits in common with the exact root.

For each polynomial $P_m(x) = (x - 1)^m$ with $m = 6, m = 8, m = 18$, Table 5 shows the value $\delta = \lceil \log_{10}(m - 1) \rceil$, the number of significant digits not affected by rounding errors estimated by CADNA of its computed root, and the number of significant digits in common with the exact root according to Definition 1.

<table>
<thead>
<tr>
<th>Precision</th>
<th>$m = 6$</th>
<th>$m = 8$</th>
<th>$m = 18$</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>quad</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
</tbody>
</table>

As expected, from Tables 4 and 5, if the root multiplicity increases, the accuracy of the approximation obtained decreases. Depending on the root multiplicity, quadruple precision may be required to achieve satisfactory accuracy. From Tables 4 and 5, if $\delta = \lceil \log_{10}(m - 1) \rceil$, $m$ being the root multiplicity, the result accuracy estimated by CADNA is the exact accuracy, up to $\delta$ in 12 cases out of 15, and up to $\delta + 1$ in 3 cases out of 15. This remark is in accordance with Theorem 1 and the fact that the accuracy estimation by CADNA can be considered correct up to one digit, as mentioned in Sect. 2.
Numerical validation in quadruple precision using stochastic arithmetic

<table>
<thead>
<tr>
<th>Precision</th>
<th>$m = 6$, $\delta = 1$</th>
<th>$m = 8$, $\delta = 1$</th>
<th>$m = 18$, $\delta = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CADNA</td>
<td>CADNA exact</td>
<td>CADNA exact</td>
<td>CADNA exact</td>
</tr>
<tr>
<td>Single</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Double</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Quad</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5: For $P_m(x) = (x - 1)^m$ number of digits not affected by rounding errors estimated by CADNA and number of digits in common with the exact root.

5.3 Autotuning of floating-point types with three possible precisions

The PROMISE tool has already enabled tuning of floating-point types in programs to produce, taking into account accuracy requirements, mixed-precision configurations, i.e. with single and double precision variables. PROMISE has been successfully tested on programs implementing several numerical algorithms including linear system solving and also on an industrial code that solves the neutron transport equations [9]. In this section, we aim at showing the feasibility of using PROMISE to automatically provide modified programs with variable declarations in single, double, and quadruple precision. Table 6 presents results provided by the “stochastic reference” version of PROMISE, already mentioned in Sect. 4.3, with the following simple programs: matrix multiplication (MatMul), Babylonian method for square root computation (SquareRoot), and rectangle method for the computation of integrals (Rectangle). For each program, different rows in Table 6 correspond to several accuracy requirements: 4 to 20 (with step 2) exact significant digits. In the case of matrix multiplication, all elements of the resulting matrix must satisfy the accuracy criterion.

The performance of PROMISE is measured on the workstation already mentioned in Sect. 3.3 with the same compilers and the compilation option O3. The results reported in Table 6 are: the number of executions of the transformed programs (# exec); the number of variables which must be stored in quadruple precision (# quad); the number of variables which can be relaxed to double precision (# double); the number of variables which can be relaxed to single precision (# float); the total execution time of PROMISE, including the compilation and the execution of the transformed codes, as well as the time spent in PROMISE’s own routines; the speedup of the proposed configuration, when run without CADNA, w.r.t. the initial configuration (all variables in quadruple precision).

As the number of required digits decreases, the number of variables in single or in double precision increases and the speedup of the transformed program w.r.t. the quadruple precision one also increases. This speedup is up to 5.5 if 4 correct digits are requested for matrix multiplication. In the numerical experiments carried out with the two-precision version of PROMISE [9], lower speedup has been obtained (up to 1.3). In this case, speedup was measured w.r.t. the double precision configuration. The better speedup obtained with the three-precision version of PROMISE can be explained by the heavy overhead of quadruple precision w.r.t. double or single precision.

6 Conclusion and perspectives

In this article, we have shown how to control the numerical quality of quadruple precision programs using Discrete Stochastic Arithmetic (DSA). The CADNA library that implements DSA has been extended to include a new type and thus enable the numerical validation of quadruple precision computation. CADNA in this case is preferable to the SAM library that implements...
<table>
<thead>
<tr>
<th>Program</th>
<th># digits</th>
<th># exec</th>
<th># quad -</th>
<th># double -</th>
<th>Time (seconds)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MatMul</td>
<td>20</td>
<td>9</td>
<td>1 - 1 - 1</td>
<td>31.3</td>
<td>1.46</td>
<td></td>
</tr>
<tr>
<td></td>
<td>18</td>
<td></td>
<td>1 - 1 - 1</td>
<td>31.3</td>
<td>1.46</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>1 - 1 - 1</td>
<td>31.3</td>
<td>1.46</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td></td>
<td>1 - 1 - 1</td>
<td>31.3</td>
<td>1.46</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
<td>1 - 1 - 1</td>
<td>31.3</td>
<td>1.46</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td>1 - 1 - 1</td>
<td>31.3</td>
<td>1.46</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>0 - 2 - 1</td>
<td>24.7</td>
<td>3.67</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>4</td>
<td>0 - 0 - 3</td>
<td>16.0</td>
<td>5.50</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SquareRoot</td>
<td>20</td>
<td>22</td>
<td>6 - 0 - 2</td>
<td>13.1</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>18</td>
<td></td>
<td>6 - 0 - 2</td>
<td>13.1</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>6 - 0 - 2</td>
<td>13.1</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td></td>
<td>6 - 0 - 2</td>
<td>13.1</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
<td>6 - 0 - 2</td>
<td>13.1</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td>6 - 0 - 2</td>
<td>13.1</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>22</td>
<td>0 - 6 - 2</td>
<td>10.9</td>
<td>2.68</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>4</td>
<td>0 - 0 - 8</td>
<td>4.7</td>
<td>2.74</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rectangle</td>
<td>20</td>
<td>18</td>
<td>6 - 1 - 0</td>
<td>11.8</td>
<td>1.07</td>
<td></td>
</tr>
<tr>
<td></td>
<td>18</td>
<td></td>
<td>6 - 1 - 0</td>
<td>11.8</td>
<td>1.07</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td>6 - 1 - 0</td>
<td>11.8</td>
<td>1.07</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td></td>
<td>6 - 1 - 0</td>
<td>11.8</td>
<td>1.07</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
<td>6 - 1 - 0</td>
<td>11.8</td>
<td>1.07</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td>6 - 1 - 0</td>
<td>11.8</td>
<td>1.07</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>12</td>
<td>0 - 2 - 5</td>
<td>8.6</td>
<td>1.40</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>12</td>
<td>0 - 1 - 6</td>
<td>8.6</td>
<td>1.45</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>0 - 0 - 7</td>
<td>4.4</td>
<td>1.45</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Experimental results of PROMISE with three types on several test cases.

DSA in arbitrary precision. The performance cost of the CADNA library in a quadruple precision program is reasonable: in our experiments with the GCC and ICC compilers, a factor 4 to 8 has been measured if instability detection is deactivated. This new version of CADNA can be used to control accuracy in programs that require quadruple precision, such as the simulation of chaotic phenomena or the computation of multiple roots of polynomials. Thanks to the quadruple precision in CADNA, the PROMISE software has been improved. We have shown how to automatically determine in a program an appropriate configuration of types that would possibly mix single, double and quadruple precision and would respect an accuracy threshold on the results.

A straightforward extension to this work would be the control of accuracy in quadruple precision parallel programs. CADNA can be used for the numerical validation of single or double precision parallel programs based on OpenMP [5] or MPI [17]. CADNA could be improved to enable the control of quadruple precision parallel programs: the main new features would be, with OpenMP the reduction operations with quadruple precision stochastic variables,
and with MPI the exchange of this kind of variables between processors.

With the O3 compilation option, the double-double type of the QD library performs better than the binary128 type of GCC or ICC. Therefore an implementation of DSA based on the QD library is another possible extension to this work. Because double-double results are not necessarily rounded as specified in the IEEE 754 standard, this perspective will require to rethink some double-double algorithms in order to compute results with directed rounding.

7 Acknowledgement

The authors wish to thank ANR (Agence Nationale de la Recherche, France) for its financial support (project FastRelax ANR-14-CE25-0018-01).

References


