IXPUG tutorial – 25–28 September 2018
Intel - Université de Versailles Saint-Quentin en Yvelines - EDF

Verificarlo Tutorial

eric.petit@intel.com ; yohan.chatelain@uvsq.fr ; pablo.oliveira@uvsq.fr; francois.fevotte@edf.fr; bruno.lathuiliere@edf.fr

Verificarlo is an open-source project under GPL v3. It is freely available at the following address :
`https://github.com/verificarlo/verificarlo`

A Debian package and a Docker container are also available:

- `https://github.com/verificarlo/verificarlo/releases/tag/v0.2.0-debian`
- `https://hub.docker.com/r/verificarlo/verificarlo/`

1 Verificarlo Basics

1.1 Start a verificarlo project

VERIFICARLO is overloading the LLVM compiler. It will be used instead of your usual compiler using the command `verificarlo`.

To start a project, the first thing is to modify your build system to use the `verificarlo` compiler.

Since the fundamental usage of Verificarlo is based on multiple execution sample study, it is necessary to provide easy execution and data of interest collection script for your application. Since all applications have different need, you are responsible to provide *ad hoc* implementation. However, you will find some example in the tutorial code and the Verificarlo `test` directory.

The MPI support exists but has not been released yet (patch of the `mpicc/mpifort script`). In case you need it please contact the author of this tutorial.

1.2 Dealing with Verificarlo configurations

1.2.1 From a terminal

Verificarlo is controlled through three environment variables

1. `VERIFICARLO_MCAMODE`:
   (a) MCA: (default value) *Monte Carlo Arithmetic* with inbound and outbound error
   (b) IEEE: noiseless execution equivalent to IEEE (useful to check that the instrumentation didn't 'break' the code).
   (c) PB: *Precision Bounding* only inbound error
   (d) RR: *Random Rounding* only outbound error

2. `VERIFICARLO_PRECISION` control the virtual precision, *i.e.* the magnitude of the noise introduction. The default value is 53, the less significant bit *bit* of double precision floating point number. The equivalent in single precision is 24. [1, 2].
3. `VERIFICARLO_BACKEND` In its github released version, Verificarlo supports two main *backends* with theoretically equivalent behavior:

   `MPFR` : This is the original reference implementation using the multi precision library MPFR [3] and extrapolated from Frechtling [4] work.

   `QUAD` : Since doubling the mantissa number of bits is enough to represent all the noise that can influence the rounding, `Quad` uses *ad-hoc* quadruple precision (binary128) implementation of the stochastic arithmetic for double precision (binary64) computation, and double (binary64) for single precision (binary32) computation.

To change these variable values, you can use the `export` Unix command, or do an affectation in your shell script running the application.

In order to be sure of the execution setup during your experiment, we advise you to fix the desired parameters in the execution scripts.

1.2.2 From the source code

Using the verificarlo API, it is possible for advanced user to control verificarlo at runtime from the application by inserting calls into the program.

This functionality is undocumented and we encourage users who want to try to do it to contact us directly from github.

2 Practical exercise: About polynomial evaluation

Polynomial evaluation is a common source of computational error. In particular, they are used for function interpolation in libraries or user codes. As we will see, the various representation of the same polynomial do not have the same behavior in terms of performance, but also in term of numerical accuracy!

This tutorial is using the following Tchebychev polynomial from [2, pp.52-54]:

$$T(x) = \sum_{i=0}^{10} a_i \times x^{2i}$$

With: $a_i \in [1, -200, 6600, -84480, 549120, -2050048, 4659200, -6553600, 5570560, -2621440, 524288]$

We are interested in evaluating T near 1. This example is discussed with details in [2, pp.52-54].

2.1 Expanded form evaluation

2.1.1 Your first step with Verificarlo

In this first approach, we will evaluate the polynomial in its expanded (monomial) form as given in the previous section. We first evaluate it in single precision.

Question 1

(a) Open the `tchebychev.c` file and observe the function `REAL expanded(REAL x)`.

(b) Compile `tchebychev.c` with `verificarlo` using the following command:

```
verificarlo --verbose -D FLOAT tchebychev.c -o tchebychev
```

(c) Update the environment variable to use the `QUAD` backend with `MCA` mode and virtual precision of 24.

(d) Execute the program multiple time with $x = 0.99$ using the command :
```
./tchebychev 0.99 EXPANDED
```
What can you observe?

Question 2

(a) Recompile with verificarlo the program in double precision using the command

```
verificarlo -D DOUBLE tchebychev.c -o tchebychev
```

Check that the environment variable are still in the same configuration than in question 1.

(b) Execute the program multiple time in $x = 0.99$ using the command : `./tchebychev 0.99 EXPANDED`. What can you observe?

No recompilation is required to change the analysis mode and another virtual precision.

Question 3

(a) Update the environment variable to use the `QUAD` backend with `MCA` mode and virtual precision `53`.

(b) Execute the program again in $x = 0.99$ using the command : `./tchebychev 0.99 EXPANDED`. What can you observe?

2.1.2 First numerical quality analysis of the polynomial evaluation

In this section, we propose you to analyze the numerical quality of the results computed by the expanded evaluation of the polynomial. To simplify this task, a large part of the verificarlo command to execute is automated in the script `run.sh`. The visualization is done using `plot.py` script.

Question 4

(a) Open `run.sh` and analyze how it works. We will emulate the single precision using the virtual precision of verificarlo.

(b) Modify `run.sh` to evaluate the polynomial in the interval $[0.5, 1]$ by 0.001 step.

(c) Open `plot.py` and analyze how it works, in particular the data that will be plotted.

The `plot.py` script generates plot similar to figure 1. The upper part of the figure represents the number s of significant digits of the results: $s = -\log_{10}\left|\frac{\hat{\sigma}}{\hat{\mu}}\right|$ with $\hat{\sigma}$ the sample empirical standard deviation $\hat{\mu}$ their average.

The central part is the empirical standard deviation $\hat{\sigma}$ for each value of x.

Finally the lowest parts are the $T(x)$ samples and their average in dotted line. The 20 Monte Carlo samples $T(x)$ are plotted for each x value (sometime overlapping on the graphic)

Question 5

(a) To execute the `EXPANDED` version with `DOUBLE` and a virtual precision of 24 bits, execute the command: `./run.sh EXPANDED DOUBLE 24 .`
This command's output is given in figure 1.

(b) With a virtual precision of 53, execute the command: `./run.sh EXPANDED DOUBLE 53`
This command's output is given in figure 2..

Close to 1, the polynomial evaluation is subject to *cancellations* which rapidly decrease the result precision. The double precision on the contrary seems satisfactory.

However using double precision is just moving the problem closer to 1 and it forces the programmer to use a larger and more costly data type.

Nevertheless if the user is already using double precision number, and if the precision is still not satisfactory, how to solve the issue? Or what if we need to use single precision?

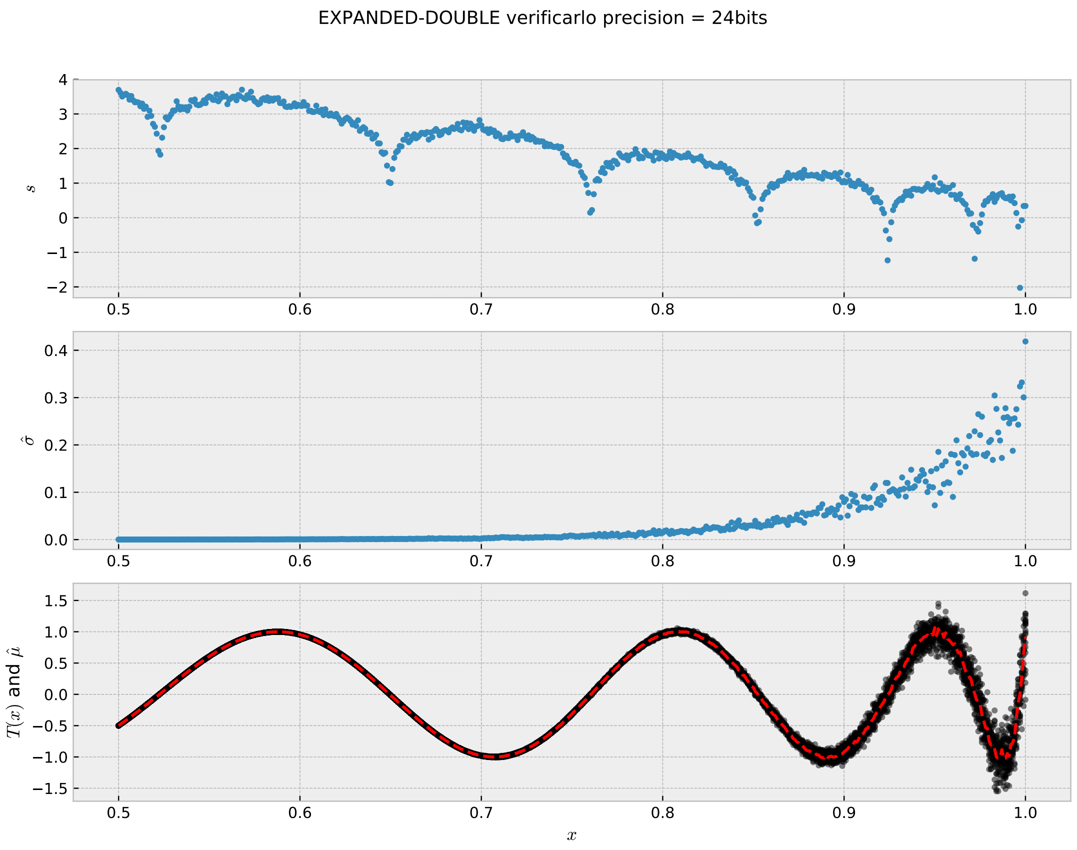

Figure 1: Evaluation of T(x) in its expanded form, compiled in double precision, with a virtual precision of 24

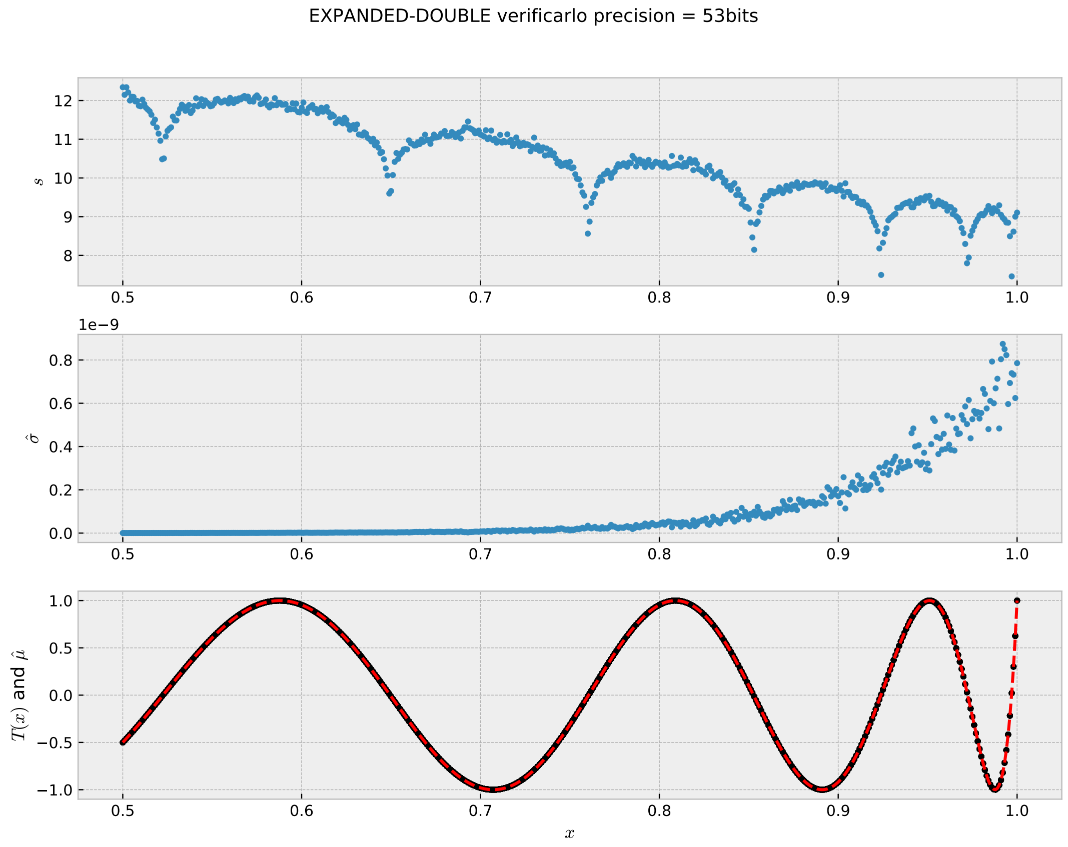

Figure 2: Evaluation of T(x) in its expanded form, compiled in double precision, with a virtual precision of 53

2.2 Evaluation using Horner scheme

It exists many other ways to evaluate polynomials, using associativity, commutativity and factorization. More often they are explored for sake of performance, but they also greatly influence the precision of the evaluation. One of them reputed to both performance with good numerical behavior is the Horner scheme which for our polynomial correspond to the following form:

$$T(x) = (\dots((a_n \times x^2 + a_{n-1}) \times x^2 + a_{n-2})\dots) \times x^2 + a_0$$

$$\begin{gathered} T(x) = ((((((((524288 * x^2 - 2621440) * x^2 + 5570560) * x^2 - 6553600)* \\ x^2 + 4659200) * x^2 - 2050048) * x^2 + 549120) * x^2 - 84480)* \\ x^2 + 6600) * x^2 - 200) * x^2 + 1 \end{gathered}$$

Question 6

(a) Open the file `tchebychev.c` and have a look to the function `REAL horner(REAL x)`

(b) While keeping previous execution parameters, execute the command `./run.sh HORNER DOUBLE 53`. The output of this command is given in figure 3.

Question 7

Modify the `run.sh` script to evaluate the polynomial from 0.5 to 1 by 0.001.

Execute the command `./run.sh HORNER DOUBLE 24`
The output of this command is given in figure 4.

As shown in this experiment, the Horner scheme has a limited influence on the result precision ($\simeq$ 1 more bit). However, it minimizes the number of operations and allows to use the FMA (*Fused Multiply Add*). For a polynomial of degree n, it produces $n-1$ FMA. Moreover, when doing multiple independent evaluations it can be vectorized.

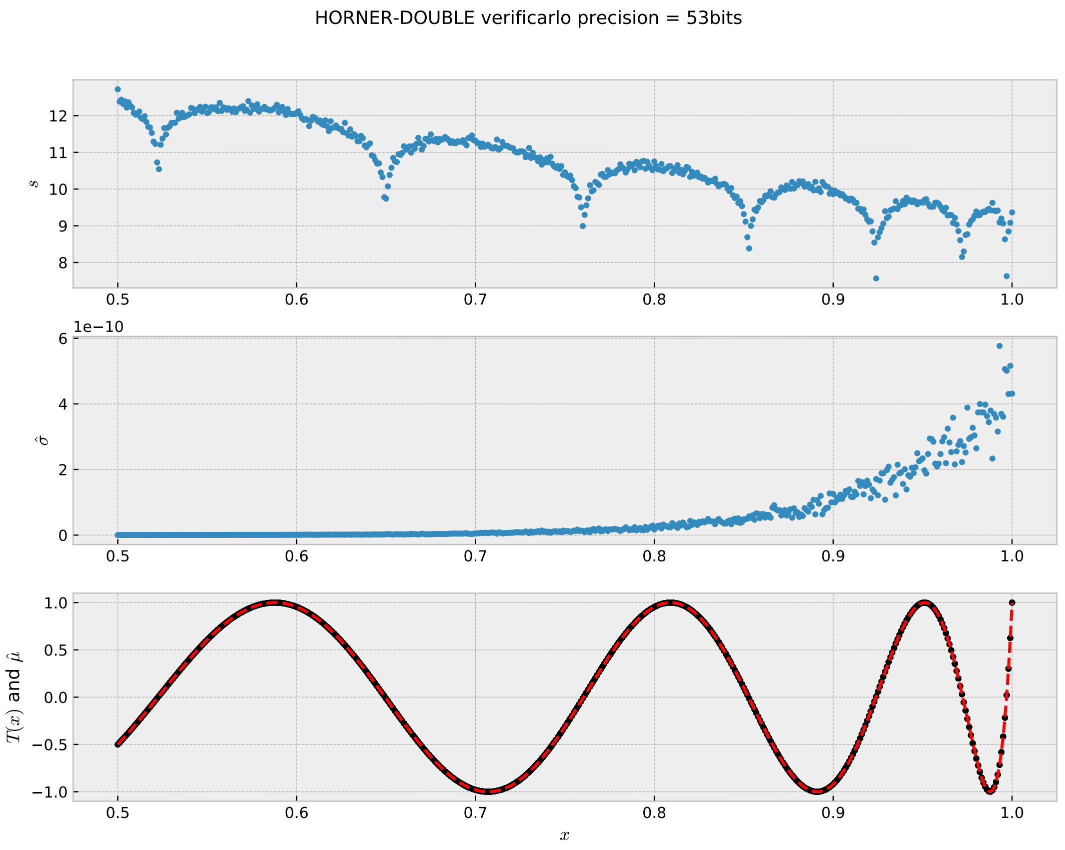

Figure 3: Evaluation of T(x) using Horner scheme, compiled in double precision, with a virtual precision of 53

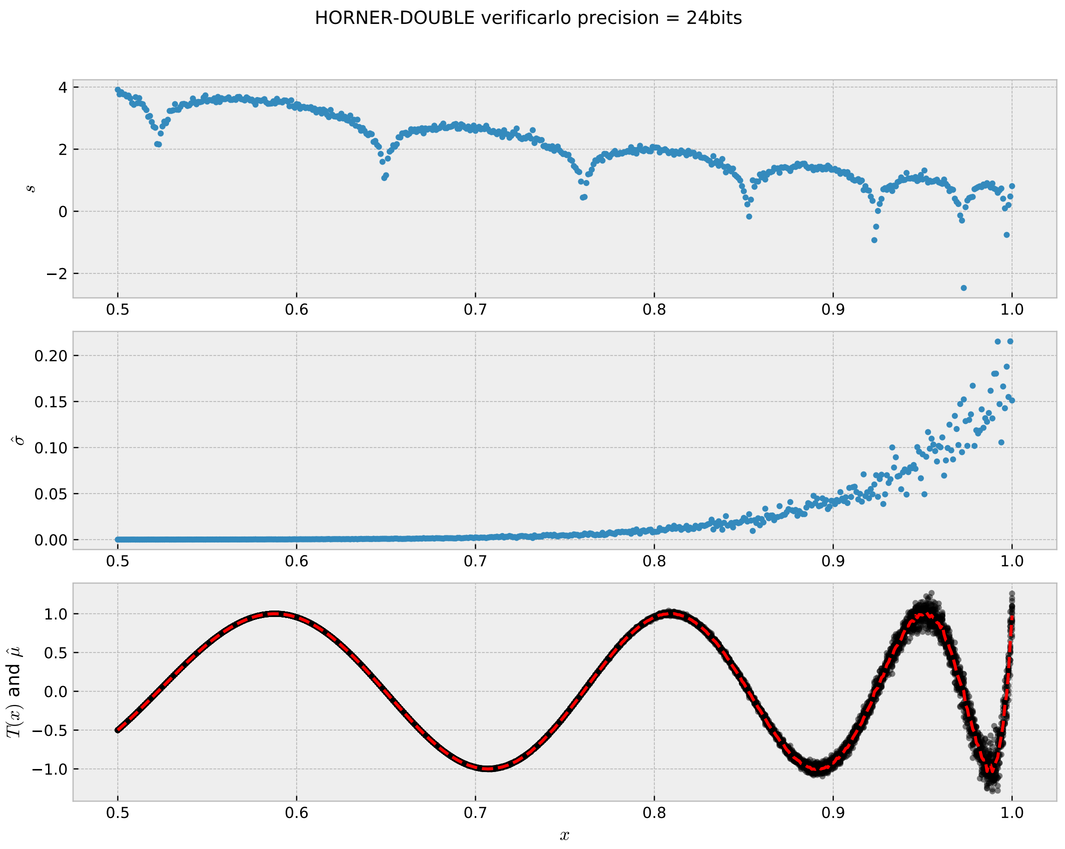

Figure 4: Evaluation of T(x) using Horner scheme, compiled in double precision, with a virtual precision of 24

2.3 Factored form

We will now evaluate the evaluation precision of the following factored rewriting:

$$T(x) = 1 + 8x^2\,(x-1)\,(x+1)\,(4x^2+2x-1)^2\,(4x^2-2x-1)^2\,(16x^4-20x^2+5)^2$$

$$\begin{aligned} T(x) &= 8.0 * x^2 * (x-1.0) * (x+1.0) \\ &\quad *(4.0*x^2+2.0*x-1.0)*(4.0*x^2+2.0*x-1.0) \\ &\quad *(4.0*x^2-2.0*x-1.0)*(4.0*x^2-2.0*x-1.0)* \\ &\quad *(16.0*x^4-20.0*x^2+5.0)*(16.0*x^4-20.0*x^2+5.0)+1 \end{aligned}$$

Question 8

(a) Open the file `tchebychev.c` and have a look to the function `REAL factored (REAL x)`

(b) Execute the command `./run.sh FACTORED DOUBLE 24`
The output of this command is given in figure 5.

(c) Compare these results to those obtained with EXPANDED and HORNER versions.

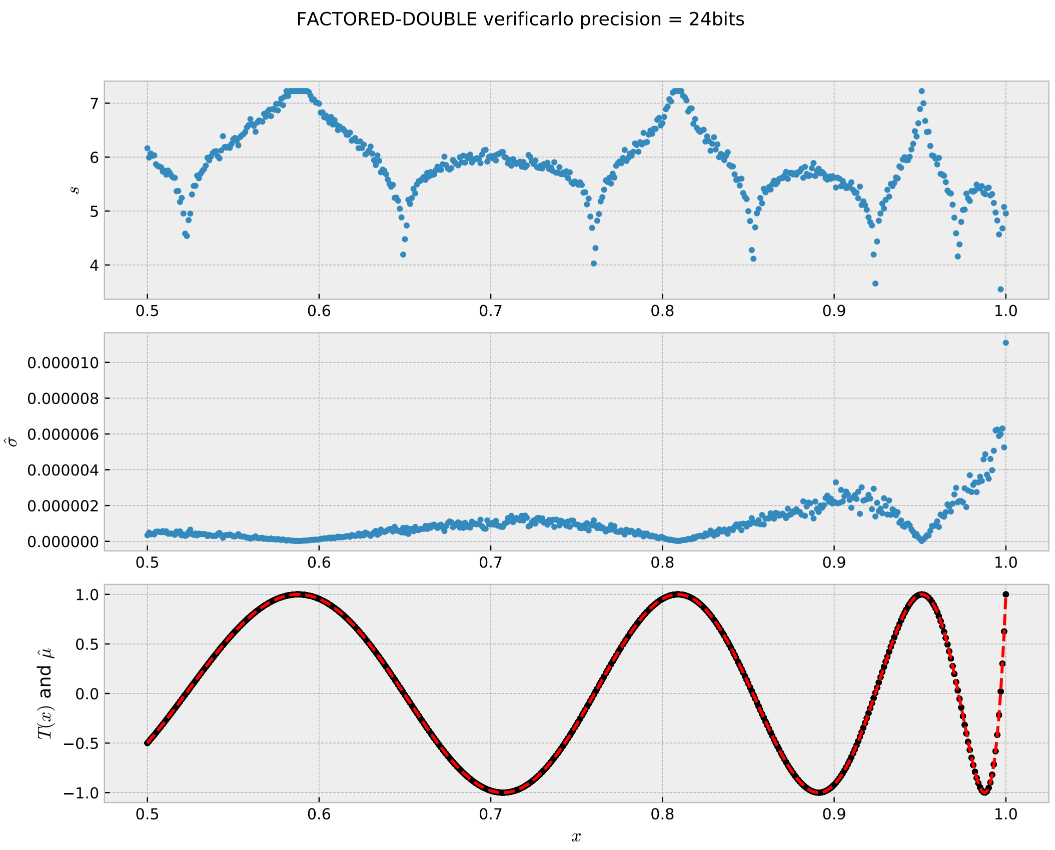

Figure 5: Evaluation of T(x) in its factored form, compiled in double precision, with a virtual precision of 24

Question 9

Explain what happens when $T(x) = 1$ for $x \simeq 0.6$, $x \simeq 0.8$ et $x \simeq 0.95$.

$\rightarrow$ It is an example where the error is absorbed and the precision and accuracy of the results are improved.

Question 10

(a) Modify the `run.sh` script to evaluate the polynomial between 0.99 and 1 by 0.00001 step.

(b) Run the scripts to execute and visualize the results for FACTORED, EXPANDED and HORNER with a virtual precision of 53. The results are respectively presented in figure 6,7 and 8.

(c) Reproduce the result with a virtual precision of 24. The results are respectively presented in figure 9,10 and 11

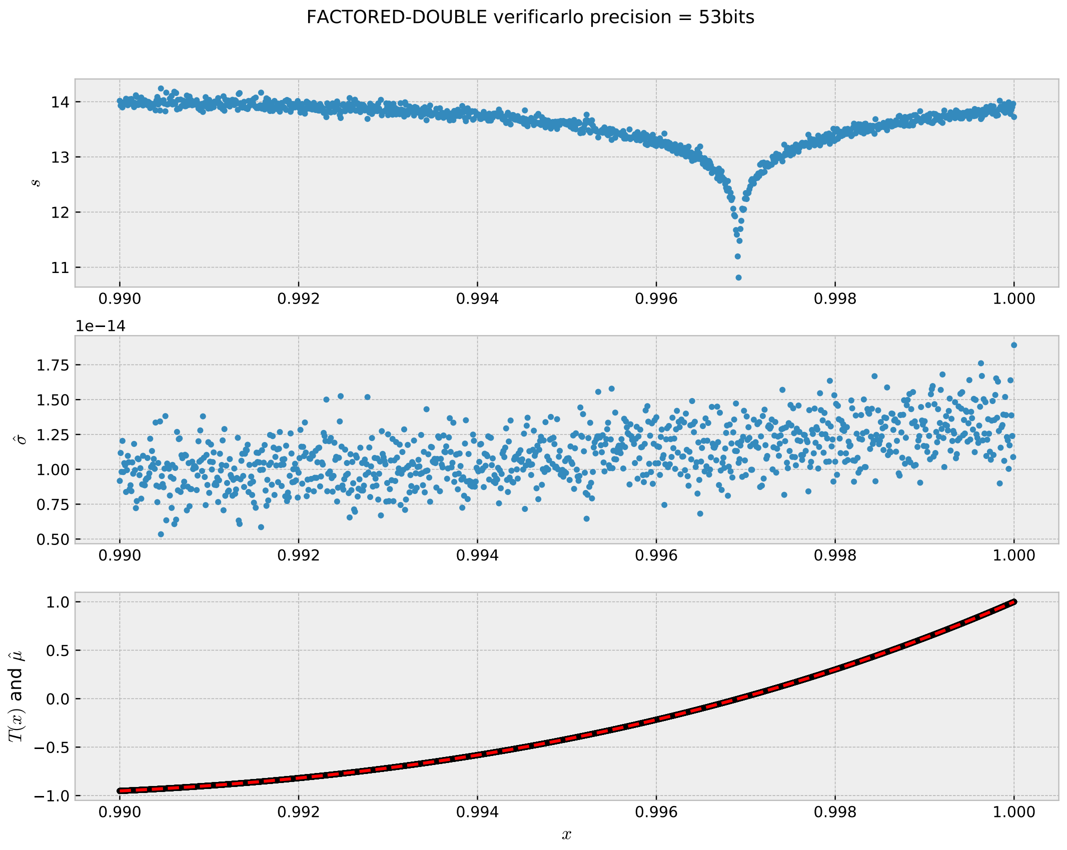

Figure 6: Evaluation of T(x) in its factored form, compiled in double precision, with a virtual precision of 53

2.4 Conclusion

From a general stand point, for every arithmetic expression in a program, it exists many valid rewriting. They are not all equivalent in terms of performance, precision, and accuracy!

Question 11

What is the best approach?

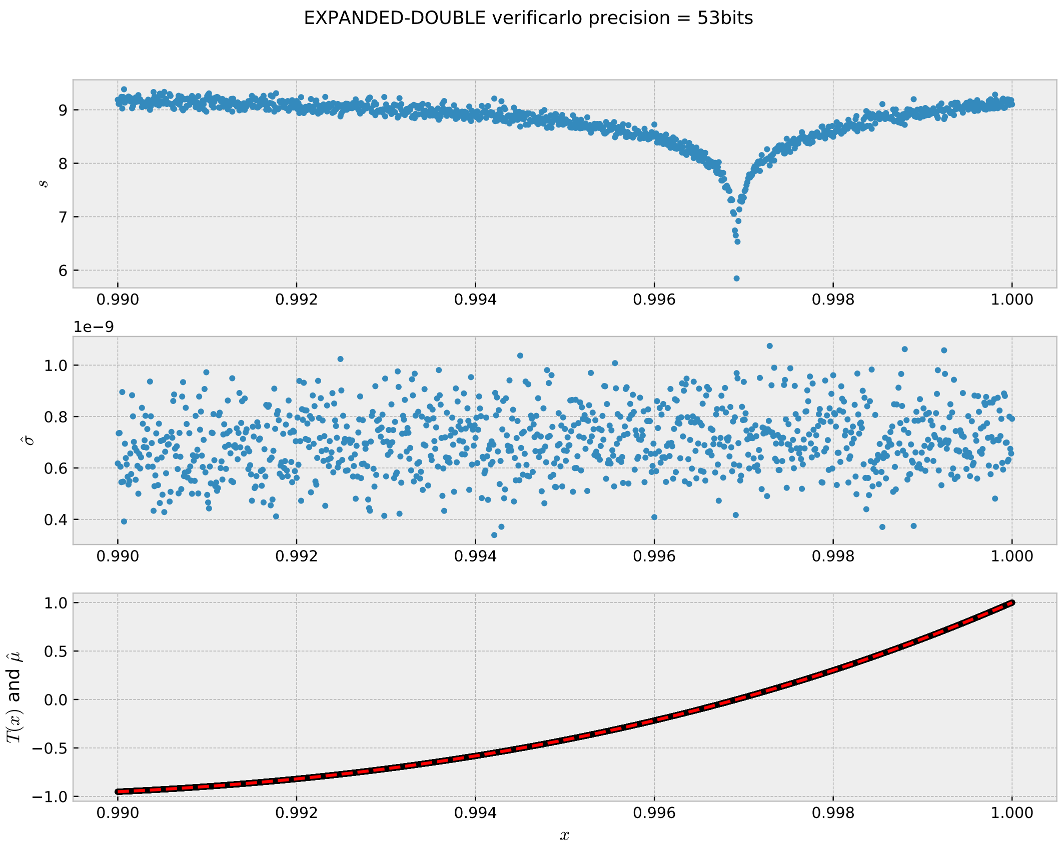

Figure 7: Evaluation of T(x) in its expanded form, compiled in double precision, with a virtual precision of 53

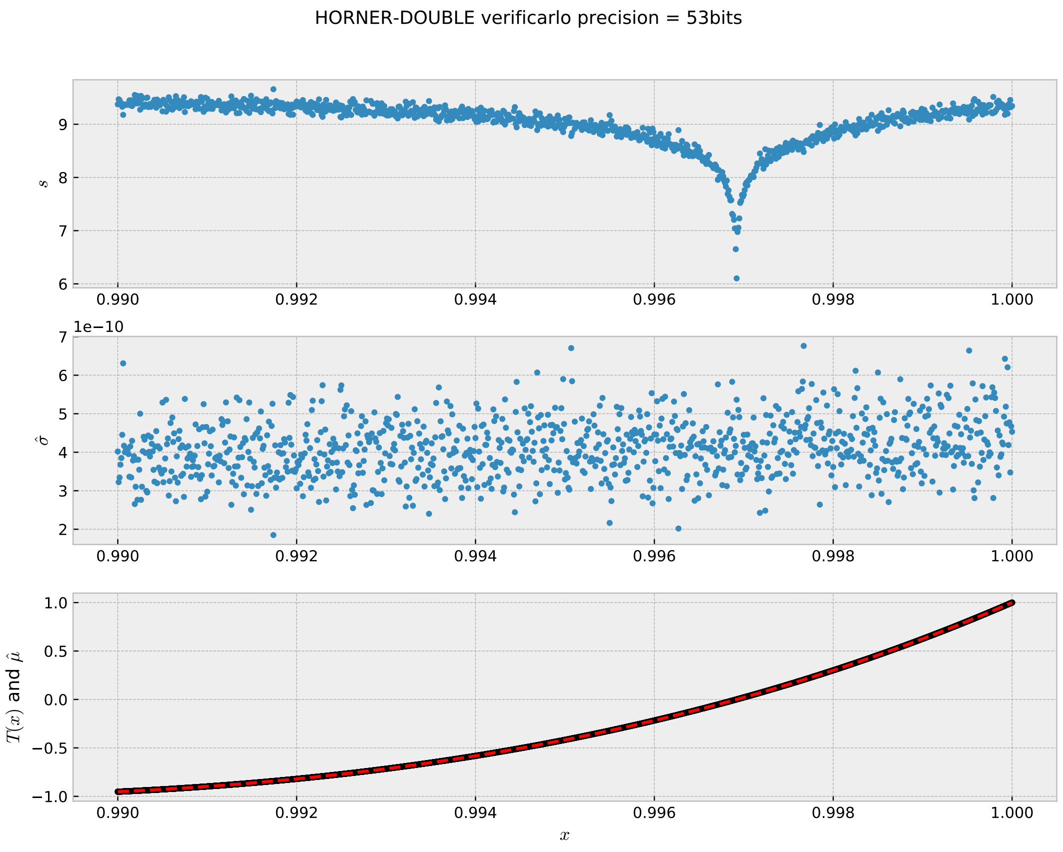

Figure 8: Evaluation of T(x) using Horner scheme, compiled in double precision, with a virtual precision of 53

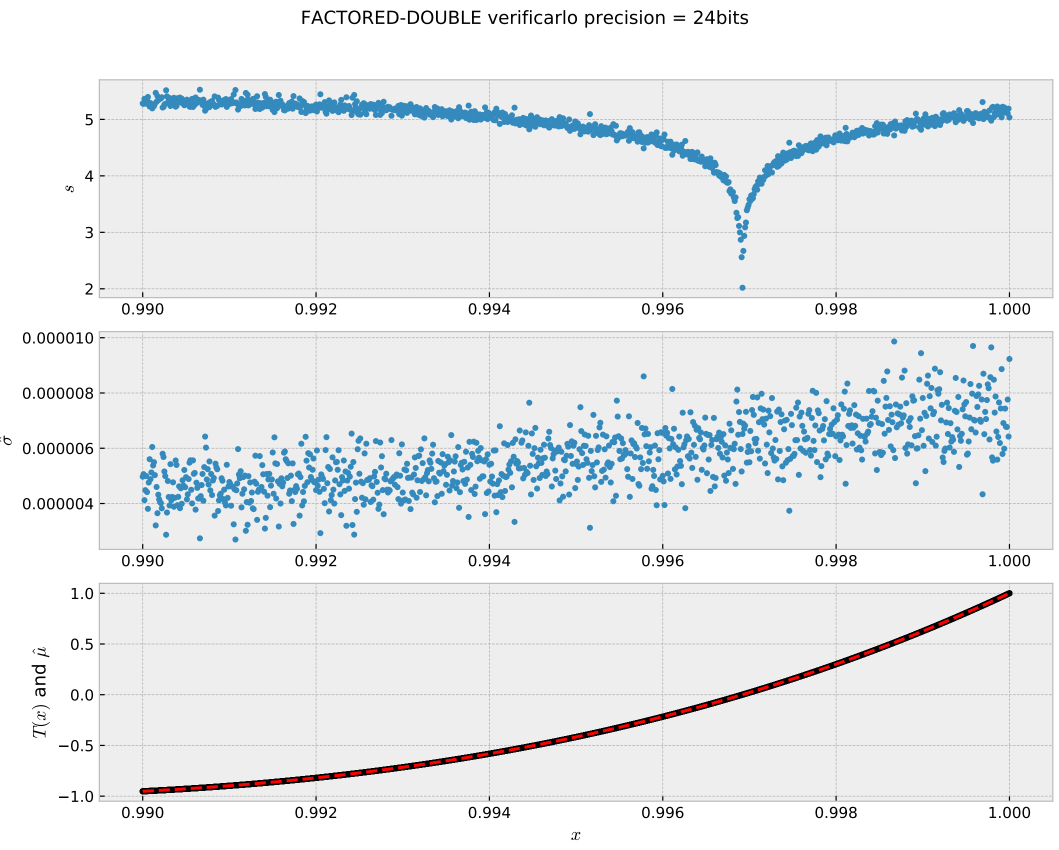

Figure 9: Evaluation of T(x) in its factored form, compiled in double precision, with a virtual precision of 24

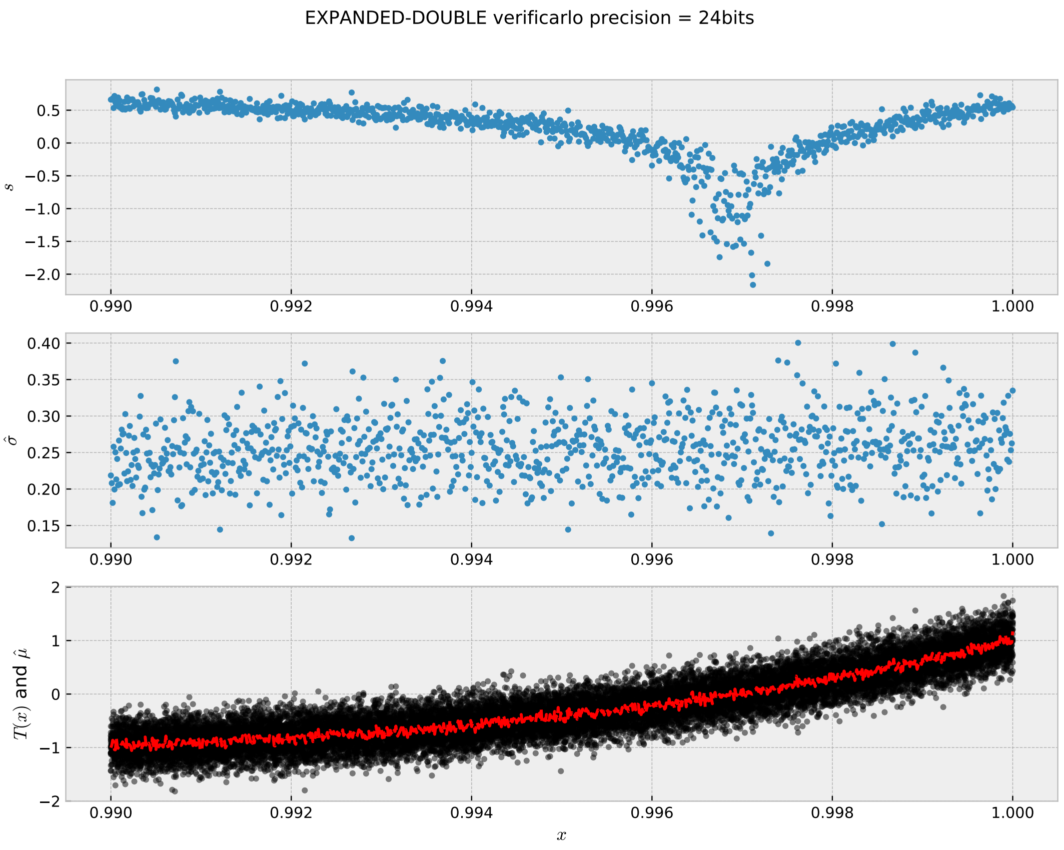

Figure 10: Evaluation of T(x) in its expanded form, compiled in double precision, with a virtual precision of 24

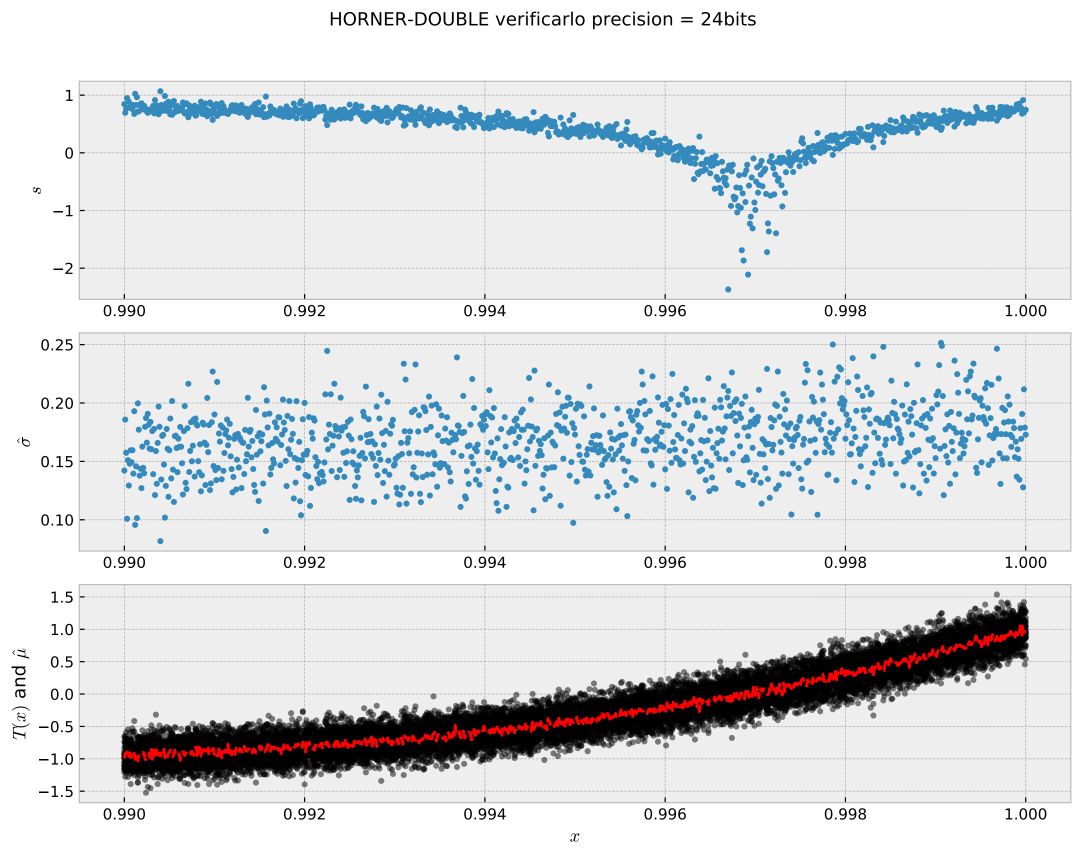

Figure 11: Evaluation of T(x) using Horner scheme, compiled in double precision, with a virtual precision of 24

3 Using Veritracer

During the first public presentation, J.M. Muller asked us if our tool could handle the following case illustrated in his book:

$$u_n = 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}u_{n-2}} \tag{1}$$

The fixed points of this sequence are the roots of the polynomial:

$u^3 - 111u^2 + 1130u - 3000 = (u-5)(u-6)(u-100)$

With the chosen initial values, $u_0 = 2$ and $u_1 = -4$, the mathematical answer is 6.

3.1 Running the code with Veritracer

We will first reproduce the experiment of D. Stott Parker [2].

Question 12

(a) Compile and test `muller.c`

(b) Modify `Makefile` to compile with `verificarlo`

(c) Collect 32 results with virtual precision 53.

(d) Compute the number of significant digits s.

⇒ As you can see the program is converging to the dominant root of the polynomial *i.e.* 100, with maximum precision! Therefore, by looking only to the final precision one could conclude that this is the correct answer

We will now do the experiment with veritracer to better understand this result.

Question 13

1. Type the command `verificarlo --help` to print how to call veritracer usage
2. Type the command, and check the output and the `.map` generated files
   `verificarlo --verbose --tracer muller.c -o muller --function muller1 -g`
3. Remove the current location map and launch veritracer in backtrace mode with the following command:
   `verificarlo --verbose --tracer muller.c -o muller --function muller1 -g --tracer-backtrace`
4. Run the program in the tracer environment with the following command:
   `veritracer launch --force --binary muller --jobs 29`
5. Check the content of the `.vtrace` directory
6. To launch the trace analysis run the command: `veritracer analyze` and check the result in the file `.vtrace/veritracer.000bt`
7. Plot the result with the provided *ad-hoc* script with the following command:
   `veritracer plot .vtrace/veritracer.000bt`
8. It is possible to add invocation information with the following command:
   `veritracer plot .vtrace/veritracer.000bt --invocation-mode`
9. and basic statistics: `veritracer plot .vtrace/veritracer.000bt --invocation --mean --std`

Bonus : do the same experiment with `-O3`. What is happening?
Use the `--tracer-level temporary` to fix the issue.

⇒ NEW update: a pre-release GUI to plot and navigate in the trace is available on github for a more friendly user experience

⇒ Figure 16 has been automatically generated thanks to veritracer. It shows the evolution of the significant digits with the iteration. We can observe a gradual degradation of the precision until it reaches no significant digits. Then the precision is gradually improving to reach the maximum attainable in double precision format, *i.e.* 17. This plot allow us to conclude that the generated results, while being precise, as lost all its accuracy.

For the final experiment of this section we will use veritracer on the Tchebychev polynomial from this tutorial.

Question 14

(a) Experiment veritracer on the Tchebychev Polynomial evaluation between 0 and 1 by 0.001. ⇒ The result is presented in figure 17.

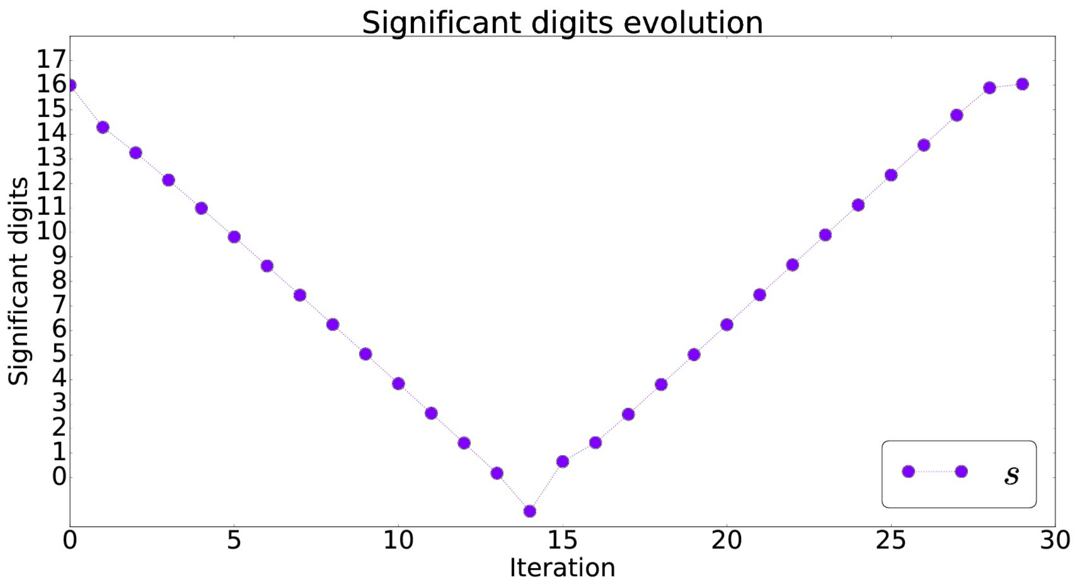

Figure 12: The evolution of the number of significant decimal digits (s) over time for the sequence u_n in equation 1. For $n = 14$, s is below 0 means that u_{14} has no correct decimal digits. Only checking the final results is not enough to detect accuracy loss.

⇒ Veritracer contextualizes and traces variable precision over time: it helps to understand the arithmetic behavior of a program. It allows to focus program analysis on a limited set of functions, variables and inputs context.

4 Open questions

4.1 Compensated Horner scheme

The "compensated" algorithms are an usual class of algorithm to increase the program precision without changing the format. The rational is to capture at every operation the accurate error term and to reinject it into the result.

In the Horner scheme, it is possible to retrieve at every step the error in x^2 and the addition of the next coefficient by using respectively the $Veltkamp - Dekker$ (`twoProd`) for the product and *twoSum* for the sum. These algorithm are qualified as (*Error Free Transform*), EFT, in the literature.

To better illustrate the concept, we can briefly analyze the *twoSum* algorithm:

```
void twoSum (REAL a, REAL b,
             REAL &x, REAL &e) {
  x = a + b;
  const REAL z = x - a;
  e = (a - (x-z)) + (b-z);
}
```

The `x` variable contain the FP sum of `a` and `b` which can be written as $a + b + epsilon(a + b)$.
This epsilon is the error term corresponding to the absorption error on the result.
Therefore, we want to extract $-epsilon(a + b)$ to capture the error term.
The `z` variable contains $b + epsilon(a + b) + epsilon(x - a)$.

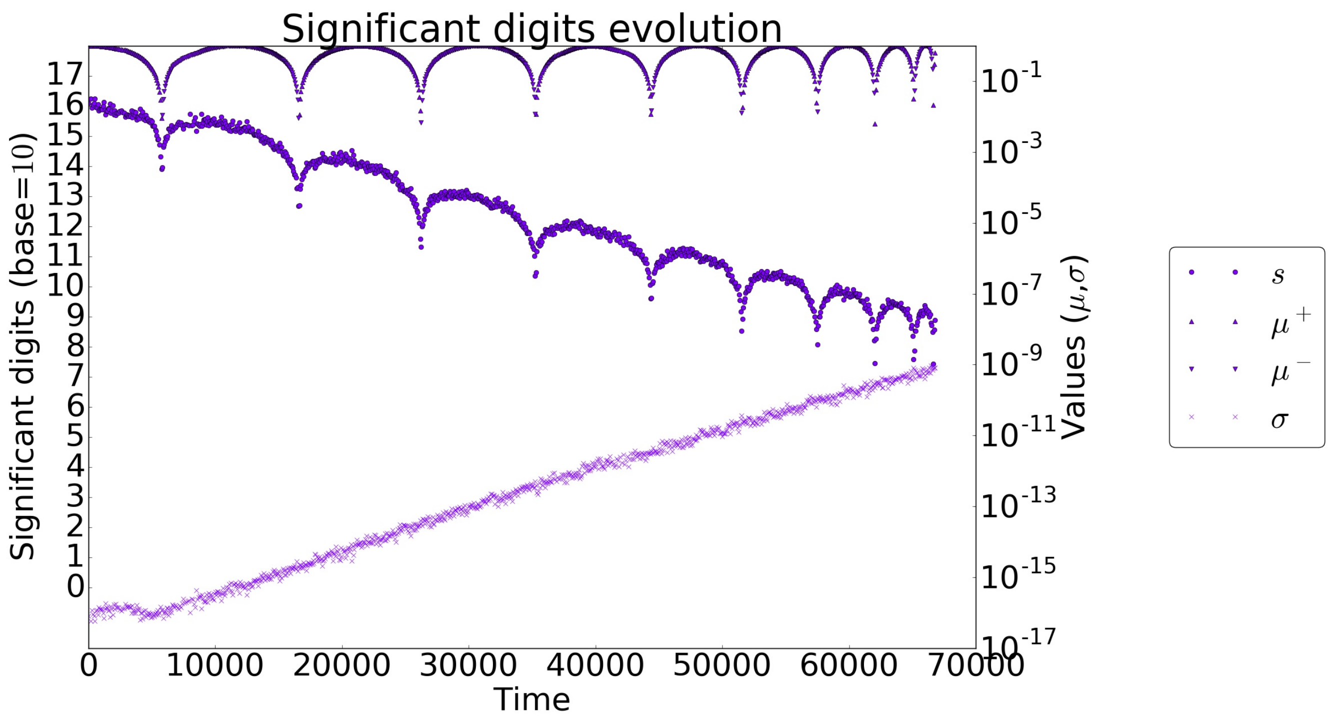

Figure 13: The evolution of the number of significant decimal digits (s) over time for the evaluation of the tchebytchev polynomial used in this tutorial, from 0 to 1 by 0.001.

Which makes `a-(x-z)` contains $epsilon(x-a)+er^2$ and `b-z` contains $-epsilon(a+b)-epsilon(x-a)+er^2$, er^2 is a term regrouping all error of the second order.
Finally `e` will contain $-epsilon(a+b)$ with negligible second order terms.

`twoProd` relies either on Dekker's algorithm, or on the IEEE FMA properties. Dekker's algorithm will work on any architecture implementing IEEE floating point operators. The listing is the following:

```
void Dekker (REAL x, Real y, Real &xy,  Real &er) {
  //extrapolated from http://toccata.lri.fr/gallery/Dekker.en.html
  #ifdef FLOAT
  REAL C=pow(2,12)+1
  #else
  REAL C==pow(2,27)+1
  #endif

  xy=x*y;

  //split x
  px=x*C;
  qx=x-px;
  hx=px+qx;
  tx=x-hx;

  //split y
  py=y*C;
  qy=y-py;
  hy=py+qy;
  ty=y-hy;

  //extract product error
  er=-xy+hx*hy;
  er+=hx*ty;
  er+=hy*tx;
  er+=tx*ty;
}
```

The algorithm for compensated horner scheme described in [5] is the following:

```
 1: procedure COMPHORNER($x$,$\{a_1, a_2, \ldots, a_n\}$)
 2:     $s_n \leftarrow a_n$
 3:     $r_n \leftarrow 0$
 4:     for $i \in [n-1:0]$ do
 5:         $[p_i, pe_i] \leftarrow$ TWOPROD($s_{i+1}, x^2$)
 6:         $[s_i, se_i] \leftarrow$ TWOSUM($p_i, a_i$)
 7:         $r_i \leftarrow r_{i+1} \times x^2 + (pe_i + se_i)$
 8:     end for
 9:     return $s_0 + r_0$
10: end procedure
```

The lines 5 and 6, evaluates HORNER with EFT calls. Line 7 accumulate the error terms, which will be add to the final result on line 9.

Question 15

Modify `run.sh` to call comphorner with the following command: `./run.sh COMPHORNER FLOAT 24` .

Instead of (carefully) recoding EFTs TwoProd and TwoSum manually, we suggest to use `libeft` [6] available and documented at the following address: `https://github.com/ffevotte/libeft`.

Question 16

(a) Modify `run.sh` to add libeft to the linker command. For this, just add `-left` to verificarlo command and ensure that libeft path is in your `LIBRARY_PATH`.

(b) Modify `tchebychev.c` following that example:

```
#include <libeft.h>

/* Define real type and format string */
#ifdef DOUBLE
#define REAL double
#define FMT "%.16e %.16e"
#define TWOPROD twoprod_d
#define TWOSUM  twosum_d
#else
#define REAL float
#define FMT "%.7e %.7e"
#define TWOPROD twoprod_s
#define TWOSUM  twosum_s
#endif
```

These modifications allow defining two macros corresponding to `TWOPROD` and `TWOSUM` calling the EFT version according to the floating point format you are using.

(c) Implement `REAL compHorner(REAL x)` in `tchebychev.c` according to the comphorner algorithm provided in this tutorial. Modify the `main` function to allow calling comphorner.

Question 17

Evaluate `compHorner` precision with Verificarlo. What happens if you use a precision different from 53 for program compiled in DOUBLE precision?

⇒ WARNING, TwoProd and TwoSum relies on exact operations; it is essential to use RR 53 (Random Rounding with precision 53) mode of verificarlo for `double` or RR 24 for `float`.

You should get the results of figures 14 and 15.

The precision of this approach is given in figures 14 and 15 with verificarlo and 18 et 19 with Verrou.

We notice on the figures 14 and 15 that CompHorner compensate precision losses in double and single precision. We retrieve a behavior similar to the factored form, in particularly for points $T(x) = 1$. However, knowing the polynomial's roots for using the Horner scheme is not required.

In figures 18 and 19, filled circles represent the real error value (evaluating in rational arithmetic in Python); circles represent the quality of the result computed in Monte Carlo Arithmetic with Verrou [7].

First, at the cost of an increase number of operations, but, generally, in the same complexity class, it is possible to recover a part (or the full) precision lost. It exists algorithms called "accurate" that compute result without loss of precision, by, for example, recursively keeping errors terms until they can not be represented in the final result and that rounding be correct (*e.g. accSum* of S. Hump).

Secondly, some algorithms, especially in mathematical libraries (libmath, Intel MKL, Intel VML, libeft) used particularity of the floating point format. By using Monte Carlo Arithmetic, it can be difficult even impossible to analyze them. In the random rounding specific case (as implemented by verificarlo RR mode with a precision length(mantissa)+1 or by Verrou), a large amount of compensated algorithms can be analyzed (including compHorner as seen before).

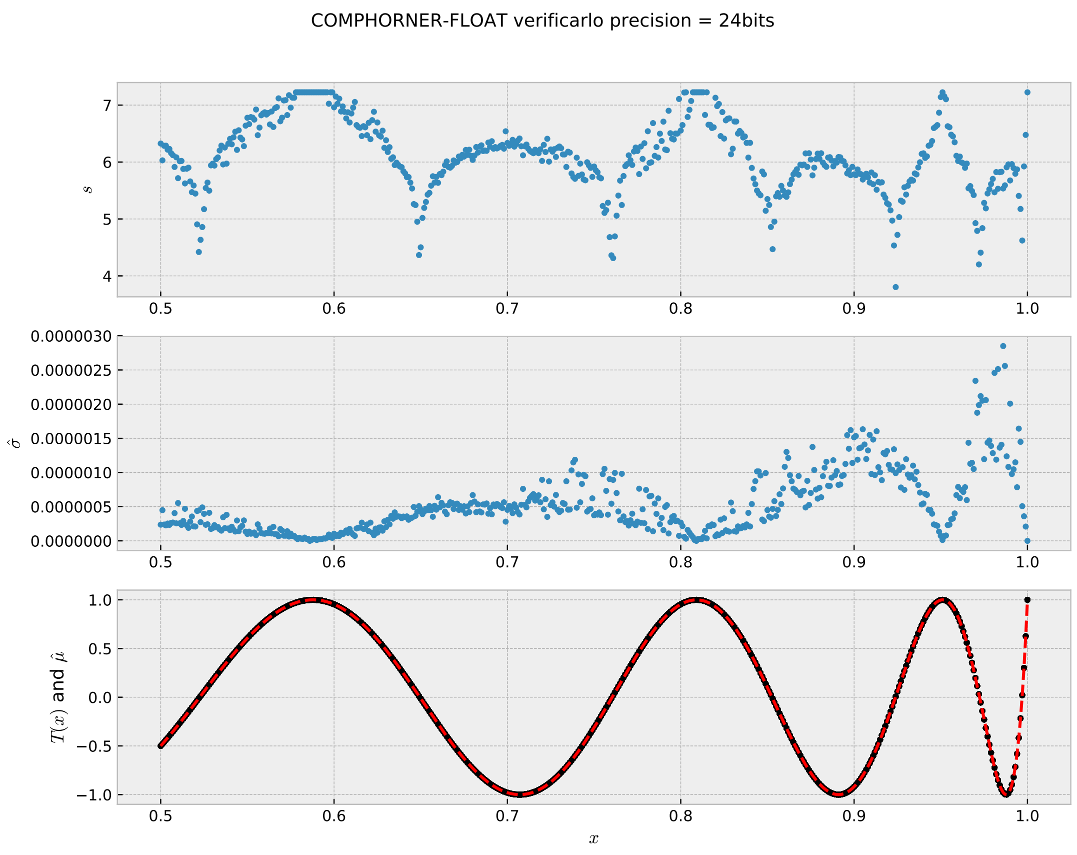

Figure 14: Evaluation of T(x) using compHorner in single precision: error estimated by Verificarlo

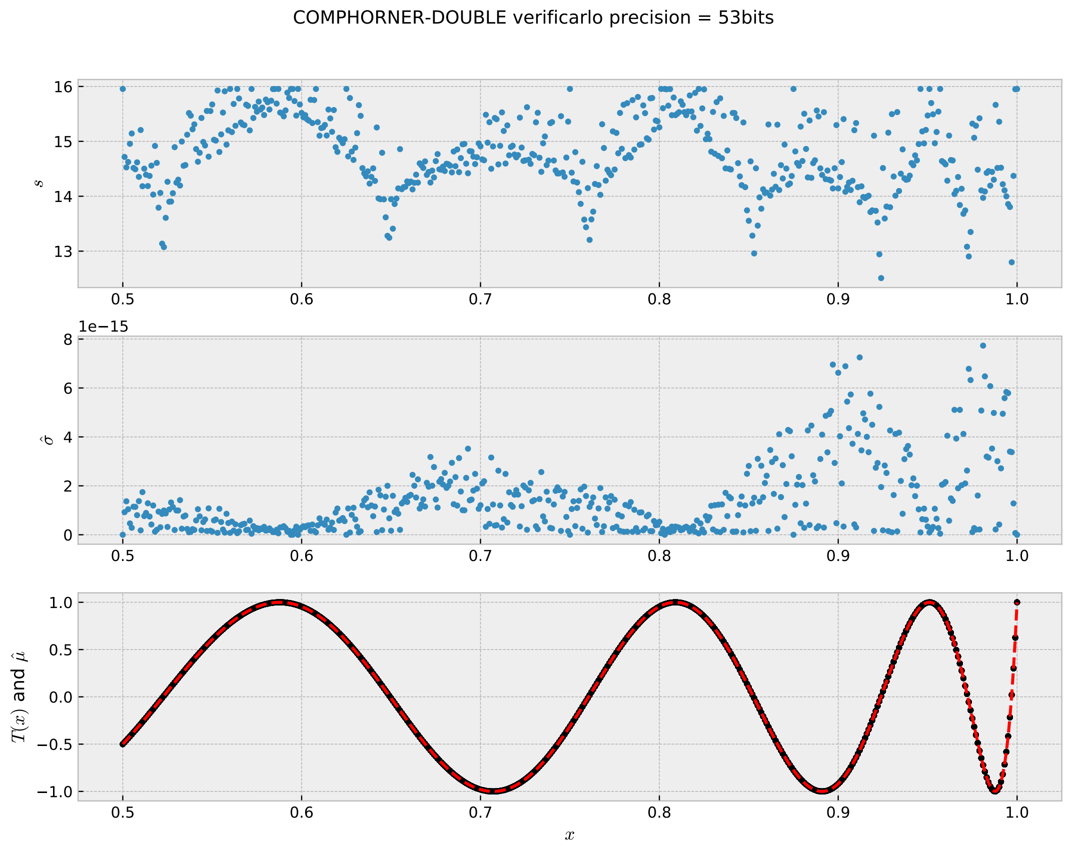

Figure 15: Evaluation of T(x) using compHorner in double precision: error estimated by Verificarlo

In addition, by their design, these algorithms have a proof of their level's precision correctness, which makes their evaluation by empirical methods useless.

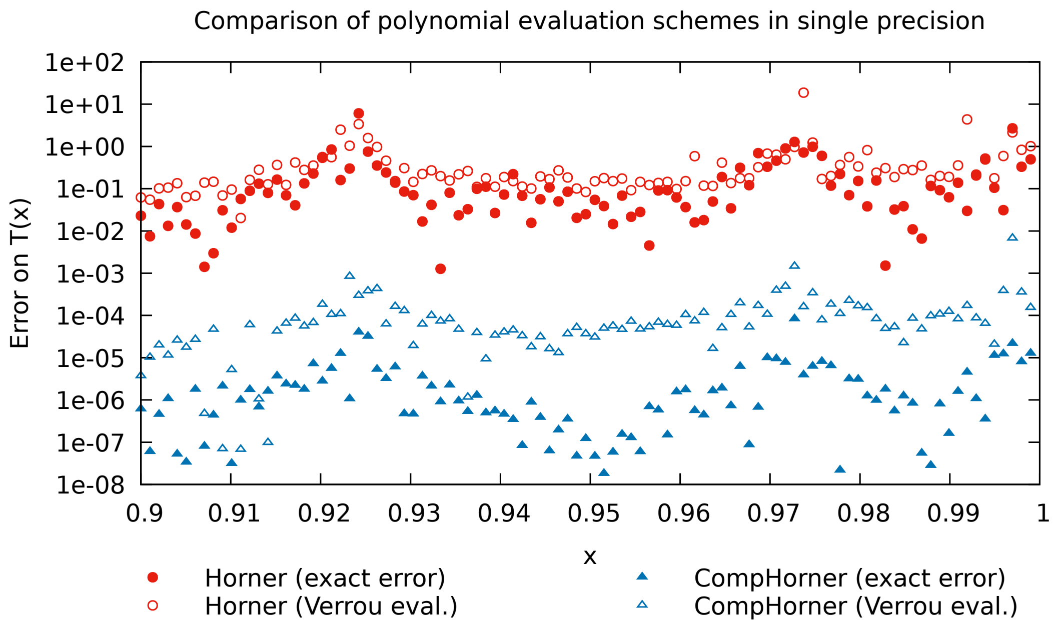

Figure 16: Evaluation of T(x) using Horner and compHorner in single precision: error estimated by Verrou, compared to the real error (courtesy of F. Févotte)

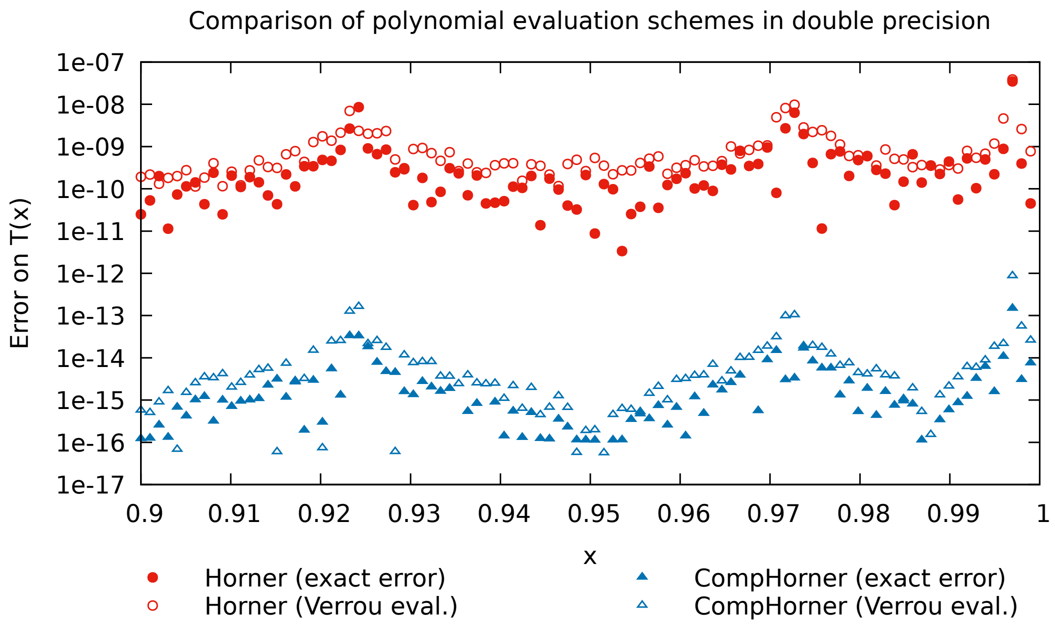

Figure 17: Evaluation of T(x) using Horner and compHorner in double precision: error estimated by Verrou, compared to the real error (courtesy of F. Févotte)

.

5 Bibliography

References

[1] C. Denis, P. de Oliveira Castro, and E. Petit, "Verificarlo: checking floating point accuracy through monte carlo arithmetic," in *Computer Arithmetic (ARITH), 23nd Symposium on*, pp. 55–62, IEEE, 2016.

[2] D. Stott Parker, *Monte Carlo Arithmetic: exploiting randomness in floating-point arithmetic.* University of California. Computer Science Department, 1997.

[3] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "Mpfr: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, June 2007.

[4] M. K. Frechtling, "Automated dynamic error analysis methods for optimization of computer arithmetic systems," 2015.

[5] S. Graillat, P. Langlois, and N. Louvet, "Compensated horner scheme," in *Algebraic and Numerical Algorithms and Computer-Assisted Proofs, B. Buchberger, S. Oishi, M. Plum and SM Rump (eds.), Dagstuhl Seminar Proceedings*, no. 05391, 2005.

[6] F. Févotte and B. Lathuilière, "LibEFT: a library implementing Error-Free transformations," 2017.

[7] F. Févotte and B. Lathuilière, "Verrou: Assessing floating-point accuracy without recompiling." `https://hal.archives-ouvertes.fr/hal-01383417`.