Université de Versailles - Intel - EDF - Université de Perpignan
CALMIP Training November 21st 2022, Toulouse - TREX Center of Excellence

Verificarlo Tutorial

pablo.oliveira@uvsq.fr; eric.petit@intel.com; yohan.chatelain@uvsq.fr; francois.fevotte@edf.fr; bruno.lathuiliere@edf.fr; david.defour@univ-perp.fr

`https://github.com/verificarlo/verificarlo`

Contents

1 Verificarlo Basics

To debug or optimize floating-point computation with Verificarlo, the first step is to compile your program with it. Verificarlo is built as a set of LLVM plugins; to compile a program with verificarlo you should use the `verificarlo` command instead of the usual `clang`, `icc` or `gcc`.

Once a program is compiled with Verificarlo, you can load various backends to simulate round-off noise or the effect of lower floating-point precisions. Backends are selected and configured by defining the `VFC_BACKENDS` environment variable.

This tutorial will guide you through examples on how to use Verificarlo. If necessary, you can access an online documentation at `https://github.com/verificarlo/verificarlo`.

In this tutorial, we will use the docker container available at `https://hub.docker.com/r/verificarlo/verificarlo/`. To pull the latest container with verificarlo and start working with it on the tutorial workspace type the following commands,

```
$ wget https://github.com/verificarlo/verificarlo_tutorial/files/9840599/verificarlo-tutorial.tar.gz
$ tar xzvf verificarlo-tutorial.tar.gz
$ cd verificarlo-tutorial/
$ docker pull verificarlo/verificarlo
$ docker run -v $PWD:/workdir -it verificarlo/verificarlo bash
```

Question 1

Start the docker container and ensure that you can run the `verificarlo` command,

```
$ verificarlo --help
```

2 Monte Carlo Arithmetic: Polynomial evaluation

Polynomial evaluation is a common source of computational error. Polynomials are frequently used for function interpolation in libraries or user codes. Different evaluations of the same polynomial do not have the same behavior in terms of performance or numerical accuracy.

This tutorial uses the Tchebychev polynomial from [1, pp.52-54]:

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

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

Due to catastrophic cancellations, the polynomial is difficult to evaluate near 1 as discussed in [1, pp.52-54].

2.1 Evaluation of the naive expanded form

2.1.1 First steps with Verificarlo

In this first approach, we will evaluate the polynomial in its expanded naive form and in single precision. This part of the tutorial is located in the `tchebychev/` folder.

Question 2

(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 -D FLOAT tchebychev.c -o tchebychev
```

(c) Run the program.

You should get an error as the `VFC_BACKENDS` environment variable is empty. The simplest backend is the one emulating IEEE-754 arithmetic(`libinterflop_ieee.so`). It has a `--debug` option that trace each instrumented floating-point operations.

Question 3

Run the program using the IEEE backend,

```
VFC_BACKENDS="libinterflop_ieee.so --debug" ./tchebychev 0.99 EXPANDED
```

To estimate the numerical error, we will now use the Monte Carlo Arithmetic backend (`libinterflop_mca.so`) in single precision by simulating round-off errors that could occurs at 24 bits of precision.

Question 4

(a) Run the program using the Monte Carlo Arithmetic backend with 24 bits precision for single precision variables,

```
VFC_BACKENDS="libinterflop_mca.so --precision-binary32=24" ./tchebychev 0.99 EXPANDED
```

(b) Execute the program multiple times. What can you observe?

The Monte Carlo Arithmetic backend define precision (i.e. noise level) for single and double precision variable by respectively using the `--precision-binary32=<value>` and `--precision-binary64=<value>`.

The Monte Carlo arithmetic backend supports different modes,

- `--mode=rr` is the *random round* mode that adds noise on the result of an operation only when the operation is not exactly representable at the chosen precision. This mode is useful to simulate the effect of round-off errors.
- `--mode=pb` is the *precision bound* mode that adds noise on the operands before performing the operation. It is useful to simulate the effect of cancellations errors.
- `--mode=mca` is the default mode that combines `rr` and `pb` modes.

2.1.2 Numerical quality analysis

In this section, we analyze the numerical quality of the computed results. The `run.sh` script available in the exercice directory automates the verificarlo runs. Visualization is done using the `plot.py` script.

Since we are working with a headless docker image, the `plot.py` output will be a `.pdf` file that you can open in the host machine.

Question 5

(a) Open `run.sh` and understand how it works.

(b) Modify `run.sh` to evaluate the polynomial in the interval $[0.5, 1]$ by 0.001 step (you can leave the number of execution unchanged).

(c) Open `plot.py` and understand how it works, and what are the plotted data.

Table 1 is generated with the `plot.py` script.

The lowest part of each plot shows the $T(x)$ samples and their average in dotted line. The 20 Monte Carlo samples $T(x)$ are plotted for each x value (sometimes overlapping on the graphic). The central part is the empirical standard deviation $\hat{\sigma}$ for each value of x. The upper part of the figure represents the number s of significant digits of each output defined as $s = -\log_{10}\left|\frac{\hat{\sigma}}{\hat{\mu}}\right|$ with $\hat{\sigma}$ the sample empirical standard deviation and $\hat{\mu}$ their average.

Question 6

(a) To execute the `EXPANDED` version with `float (binary32)` types and a virtual precision of 24 bits, execute the command: `./run.sh EXPANDED FLOAT 24 mca`
This command's output is given in table 1 (Left).

(b) Now execute the benchmark with `double (binary64)` types and a virtual precision of 53. You can use the command: `./run.sh EXPANDED DOUBLE 53 mca`
This command's output is given in table 1 (Right).

(c) Compare the number s of significant digits in both cases. What is the problem and is double precision a solution?

The polynomial evaluation done with 24 bits is subject to severe *cancellations* when the input value is close to 1. This drasticaly reduces the accuracy of the result. Using evaluation in double precision on the contrary seems satisfactory. But this solution forces the developer to use a larger and more expensive data type and it does not solve the problem, it only delays it.

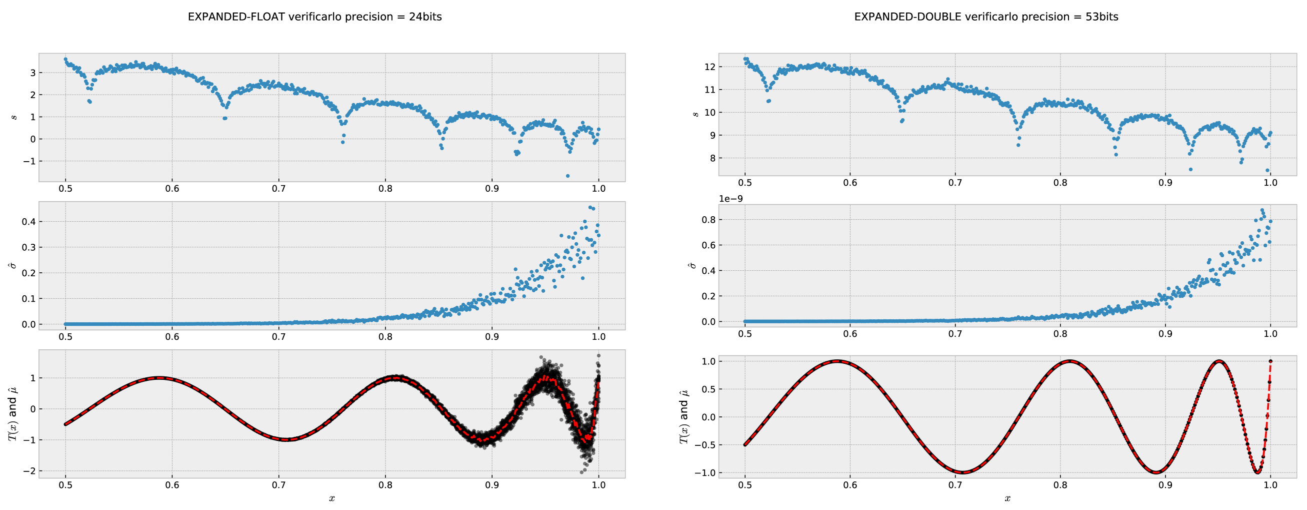


Table 1: Evaluation of T(x) in Expanded form, compiled in single/double precision, with a virtual precision of 24/53 bits

2.2 Factored form

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

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

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

Question 7

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

(b) Execute the command `./run.sh FACTORED FLOAT 24 mca`
The output of this command is given in figure 1.

(c) Compare these results to those obtained with the EXPANDED version.

Question 8

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

$\rightarrow$ We evaluate T on one of the roots of the factored right-hand terms which become zero. It is an example where the error is absorbed and the precision and accuracy of the results are improved.

3 VPREC: Variable precision backend

The VPREC backend emulates reduced floating-point formats without having to change the implementation. It can emulate any format that fits into the original type. Unlike the MCA backend, VPREC is deterministic. It mirrors what would happen with the selected reduced type. It correctly handles overflow, underflows, and rounding in the target type.

To use VPREC one can either tune manually the range and precision of the target type. The option `--precision-binary64` controls the pseudo-mantissa bit length of the new tested format for floating-point operations in double precision. The option `--range-binary64` controls the exponent bit length of the target format.

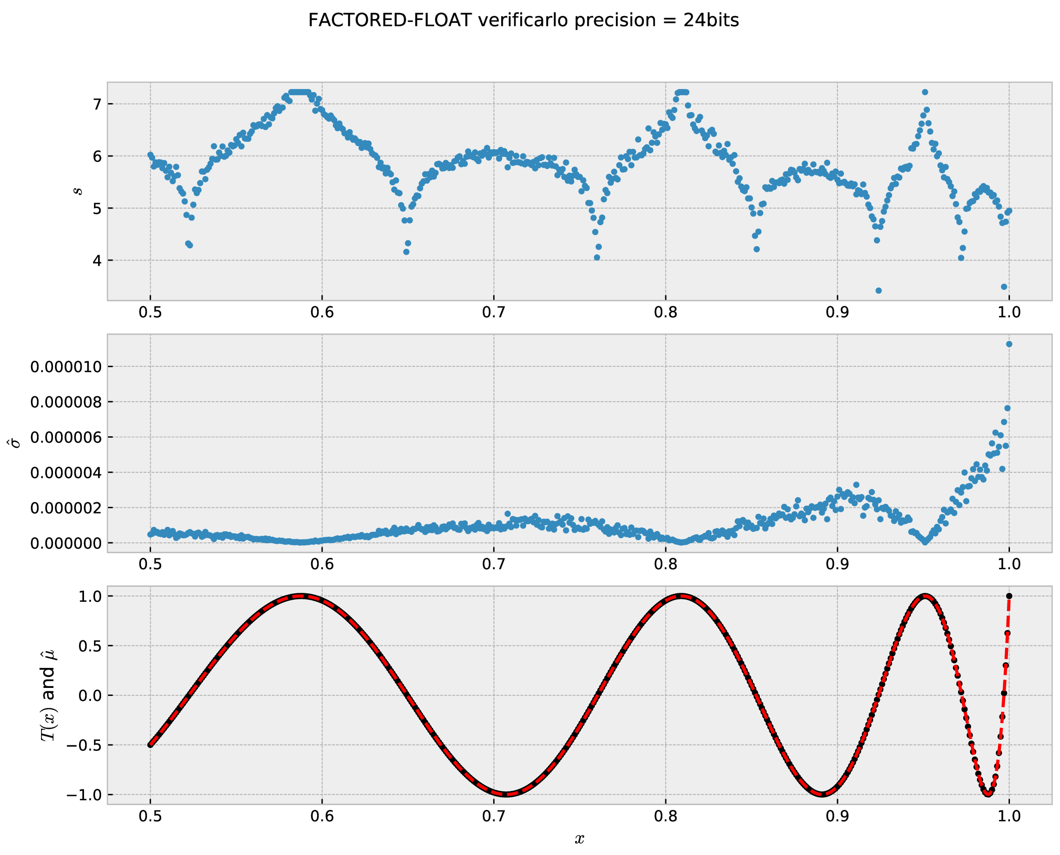


Figure 1: Evaluation of T(x) in its factored form, compiled in single precision, with a virtual precision of 24

One can also use one of the standard presets: binary16, binary32, bfloat16, tensorfloat, fp24, PXR24; which automatically sets the corresponding range and precisions.

Question 9

(a) Compile and run the benchmark using single precision without verificarlo

```
clang-7 -D FLOAT -o reference tchebychev.c
./reference 0.75 FACTORED
7.5000000e-01 -3.1220961e-01
```

(b) Now compile it using double precision with verificarlo

```
verificarlo -D DOUBLE -o tchebychev tchebychev.c
VFC_BACKENDS="libinterflop_vprec.so --preset=binary32" ./tchebychev 0.75 FACTORED
7.5000000000000000e-01 -3.1220960617065430e-01
```

(c) Compare the mantissas between the two previous runs. Observe that VPREC has accurately emulated binary32 (FLOAT) format.

Question 10

(a) You can emulate the effect of bfloat16 using

```
VFC_BACKENDS="libinterflop_vprec.so --preset=bfloat16" ./tchebychev 0.75 FACTORED
7.5000000000000000e-01 -3.1250000000000000e-01
```

(b) You can also emulate a custom precision, such as 12 bits, with

```
VFC_BACKENDS="libinterflop_vprec.so --precision-binary64=12" ./tchebychev 0.75 FACTORED
7.5000000000000000e-01 -3.1225585937500000e-01
```

(c) Try different precisions, at which threshold does the computation looses all significance ?

Question 11

(a) Analyze the script `run_vprec.sh`. This script allows you to plot the result of VPREC runs in the interval $[0.5, 1]$.

(b) Use the script to simulate and plot the effect of running FACTORED and EXPANDED versions with bfloat16.

4 Pinpointing bugs with Delta-Debug: Archimedes' method

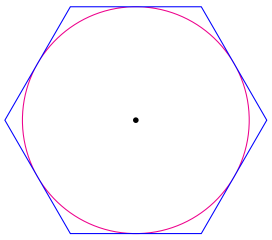


Figure 2: Archimedes method to approximate π with a 6-sided circumscribed polygon.

In this section we demonstrate how we can use Verificarlo to precisely localize a numerical bug in a program. The localization method is based on the Zeller's Delta-Debug reduction method [2]. Verificarlo uses the Interflop's stochastic Delta-Debug library [3, 4].

In 200BC Archimedes proposed the first numerical method for computing π. Archimedes method uses a $(6.n)$-sided circumscribed polygon to the unit circle whose area provides an upper bound for π and a $(6.n)$-sided inscribed polygon whose area provides a lower bound for π.

Here we will use the circumscribed polygon to approximate π. Figure 2 shows a 6-sided circumscribed polygon to the unit circle. Archimedes shows geometrically that the half-perimeter of the polygons (also converging to π) can be computed with the following recursive sequence,

$$\begin{aligned}
T_1 &= \frac{1}{\sqrt{3}} \\
T_{i+1} &= \frac{\sqrt{T_i^2+1}-1}{T_i} \\
\frac{P_i}{2} &= 6 \times 2^{i-1} \times T_i \xrightarrow[i\to\infty]{} \pi
\end{aligned}$$

In this part of the tutorial we will work inside the `archimedes/` folder.

Question 12

(a) Open the file `archimedes.c` and analyze the code.

(b) The provided makefile builds the program using Verificarlo. Run the program with the IEEE backend (`libinterflop_ieee.so`) and with the MCA backend (`libinterflop_mca.so --precision 53`).

(c) How many digits are significant in the computed result?

The previous experiment shows that computation becomes numerically unstable around iteration 15. Where is the error in the code? To help pinpointing the error, we are going to use Delta-Debug.

Delta-Debug (DD) is a general bug reduction method that allows to efficiently find a minimal set of conditions that trigger a bug. In this case, we are going to consider the set of floating-point instructions in the program. We are using DD to find a minimal set of instructions responsible for the instability in the output.

Table 2 shows a simple DD execution to find a reduced instruction set responsible for a numerical instability. By testing instructions sub-sets and their complement, DD is able to find smaller failing sets step by step. DD stops when it finds a failing set where it cannot remove any instruction. In this case, DD is able to find a minimal failing set (ddmin) of size 1 (which is therefore also minimum). However, there is no guarantee of unicity.

Step	Instructions with MCA noise	Numerically Stable
1	1 2 3 4	stable
2	. . . . 5 6 7 8	unstable
3	. . . . 5 6 . .	stable
4	. . . . . . 7 8	unstable
5	. . . . . . 7 .	unstable
Result (ddmin)	. . . . . . 7 .	

Table 2: Example of Delta-Debug bug minimization

By default, Interflop's Delta-Debug implementation iterates to find all the possible different ddmin sets. At the end, it produces the rddmin-cmp set which is the complement of the union of the ddmin sets. The rddmin-cmp set therefore includes the "stable" instructions and excludes the "unstable" instructions.

To use Delta-Debug, we need to write two scripts:

- A first script `ddRun <output_dir>`, is responsible for running the program and writing its output inside the `<output_dir>` folder.
- A second script `ddCmp <reference_dir> <current_dir>`, takes as parameter two folders including respectively the outputs from a reference run and from the current run. The `ddCmp` script must return with a success when the deviation between the two runs is acceptable, and fail if the deviation is unacceptable. To decide if a given set is unstable, DD will run the program five times (the number of times can be changed by setting the environment variable `INTERFLOP_DD_NRUNS`).

Question 13

(a) Open the files `ddRun` and `ddCmp` and analyze how they work.

`ddRun` and `ddCmp` depend on the user's application and the error tolerance of the application domain; therefore it is hard to provide a generic script that fits all cases. That is why we require the user to manually write these scripts. Once the scripts are written, the Delta-Debug session is launched using the following command:

```
VFC_BACKENDS="libinterflop_mca.so --precision-binary64=53 -m mca" vfc_ddebug ddRun ddCmp
```

where `VFC_BACKENDS` specifies the backend that will be used to capture numerical errors. Here we provide a simple Makefile target that runs this command,

```
make dd
```

Question 14

(a) Open the file `Makefile` and analyze the `make dd` target.

(b) Run the `make dd` target.

Now that you have run Delta-Debug, you can observe that it has found two minimal failing sets (ddmin0 and ddmin1). The outputs sets are located in `dd.line/ddmin0` and `dd.line/ddmin1` directory. You can locate the "unstable" instructions belonging to each set by browsing the content of the `dd.line/ddmin{0,1}/dd.line.include` files. This information is also present in the `dd.line/rddmin-cmp/dd.line.exclude` file.

```
$ cat dd.line/ddmin0/dd.line.include
0x0000000000400e5c: archimedes at archimedes.c:16
$ cat dd.line/ddmin1/dd.line.include
0x0000000000400e89: archimedes at archimedes.c:17

# We can also get the union of culprit instructions with

$ cat dd.line/rddmin-cmp/dd.line.exclude
0x0000000000400e5c: archimedes at archimedes.c:16
0x0000000000400e89: archimedes at archimedes.c:17
```

We can notice that instructions at line 16 and 17 found by DD method are responsible for the observed numerical instability. To ease the debugging process we include a script that transforms this output, into the error format used by compilers. Therefore we can use a standard IDE to pinpoint culprit instructions inside the code.

Question 15

(a) Run the `make dderrors` target. This should open a Vim session.

(b) Type `:cw` to open the error quick-fix window. Use `:cn` and `:cp` to move to the next and previous "unstable" instructions.

Line 17 points to a subtraction operation executed in double precision (doublesub) between s and 1. One can see that as T_{i+1} (`tii`) becomes smaller, s becomes closer to 1. Therefore it looks like line 17 could trigger a catastrophic cancellation. Let us do some experiments to confirm this hypothesis.

Question 16

(a) Run Delta-Debug with a RR noise model at precision 53.

```
make dd VFC_BACKENDS="libinterflop_mca.so -m rr --precision-binary64=53"
```

With RR 53, only line 16 is flagged. Indeed the operation $T_i^2 + 1$ is inexact due to a round-off error ($T_i \ll 1$). The error then propagates and is amplified by the cancellation line 17.

Interestingly, with RR the line 17 is not flagged; that is because RR 53 mode does not perturb cancellations which are exact operations. The fact that line 17 disappears with RR mode hints that there is indeed a cancellation problem.

Question 17

(a) Run Delta-Debug with the cancellation backend to confirm the analysis.

```
make dd VFC_BACKENDS="libinterflop_cancellation.so"
```

With the cancellation backend we see that only line 17 is flagged. Therefore, we conclude that this program is affected both by a round-off error at line 16 and a cancellation error at line 17. The round-off error by itself is not problematic but is amplified by the cancellation.

To fix this problem, we can try to rewrite the culprit expression in line 17. Observe that,

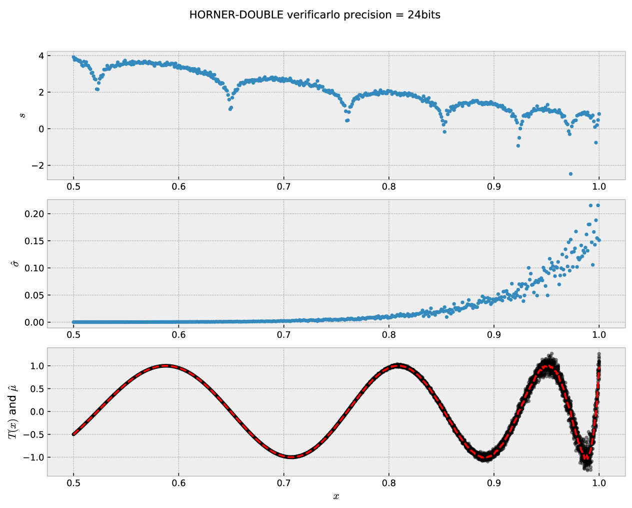
$$\begin{aligned} T_{i+1} &= \frac{\sqrt{T_i^2+1}-1}{T_i} = \frac{\sqrt{T_i^2+1}-1}{T_i} \times \frac{\sqrt{T_i^2+1}+1}{\sqrt{T_i^2+1}+1} \\ &= \frac{(T_i^2+1)-1}{T_i(\sqrt{T_i^2+1}+1)} \\ &= \frac{T_i}{\sqrt{T_i^2+1}+1} \end{aligned}$$

The new formula is interesting since it eliminates the subtraction that triggered the cancellation.

Question 18

(a) Modify `archimedes.c` to use the previous expression rewriting.

(b) Study the numerical instability of the new version. Is the problem fixed?



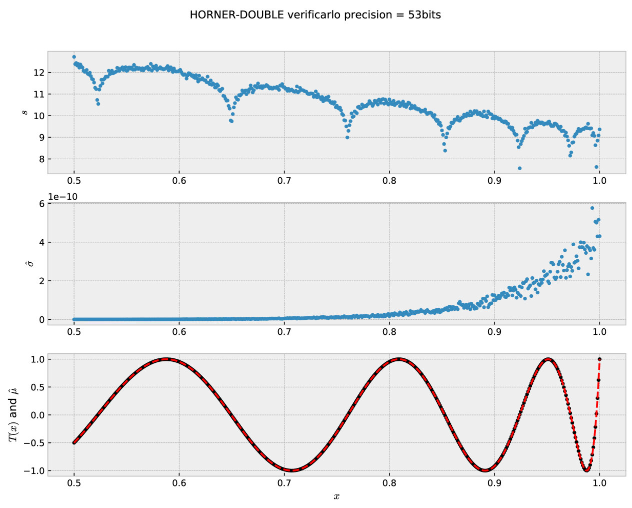


Table 3: Evaluation of T(x) with Horner's scheme, compiled in double precision, with a virtual precision of {24/53} bits

5 Bonus exercices

5.1 Polynomial evaluation using Horner's method

There exists multiple ways to evaluate polynomials, using associativity, commutativity and factorization. Each evaluation scheme has its own impact on precision and performance. One of them has already been the subject of many studies: the Horner's method which for the considered polynomial takes 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 19

(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 mca` The output of this command is given in table 3 (Left).

Question 20

Execute the command `./run.sh HORNER DOUBLE 24 mca`
The output of this command is given in table 3 (Right). What do you observe?

As shown in this experiment, the improvment brought by the Horner scheme is not significant ($\simeq 1$ additional significant bit in the result). 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.

5.2 Compensated Horner's method

Compensated algorithms belong to the class of algorithms that increase program precision without changing the internal working format. The goal is to capture for each operation an estimation of the error term and to reinject it into the result. For 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 algorithms are qualified as (*Error Free Transform*), EFT, in the literature. The algorithm for the compensated horner scheme described in [5] is:

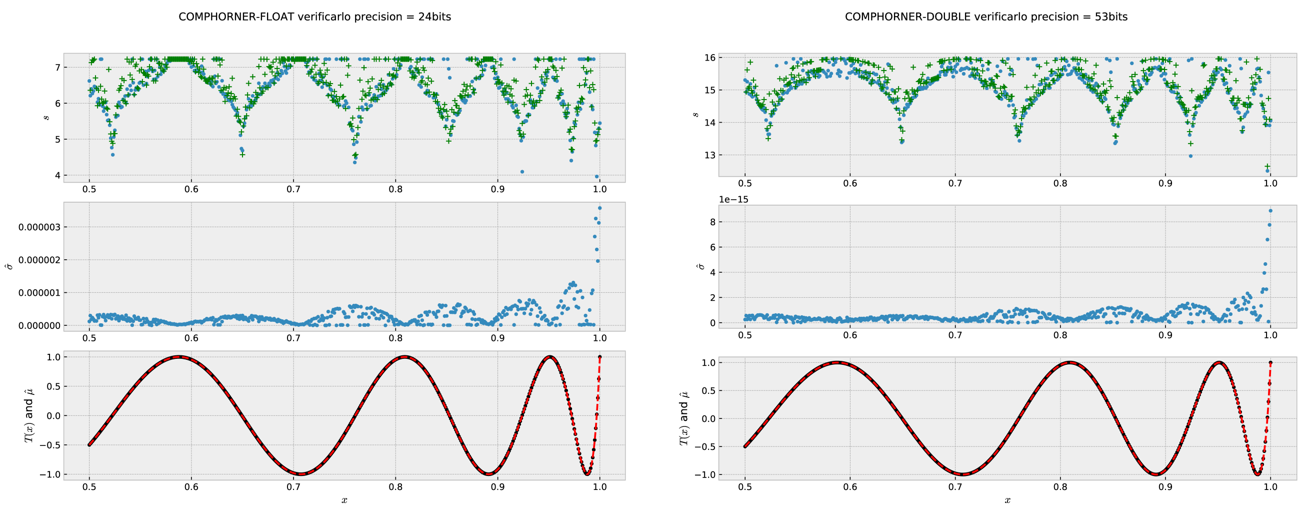


Table 4: Evaluation of T(x) using Horner and compHorner in single/double (left/right) precision: error estimated by verificarlo (blue), compared to the real error (green)

```
1: procedure COMPHORNER($x$,$\{a_1, a_2, \dots, a_n\}$)
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, evaluate HORNER with EFT calls. Line 7 accumulates the error terms, which will be added to the final result in line 9.

We provide implementations for EFT in the `libeft.c` and `libeft.h` files.

Question 21

(a) Implement comphorner algorithm in `tchebychev.c` using the EFT implementations in `eft.h`.

(b) Modify `run.sh` to also compile eft.c with verificarlo.

(c) Run comphorner with the following command: `./run.sh COMPHORNER FLOAT 24 rr`

(d) Run comphorner with the following command: `./run.sh COMPHORNER DOUBLE 53 rr`

(e) What happens if you use a precision different from 53 for program compiled in DOUBLE precision? $\Rightarrow$ WARNING, TWOPROD and TWOSUM rely 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 results shown in figure 4.

The resulting precision of this approach is shown in table 4 with verificarlo. 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 Verificarlo [3].

We can notice on these figures 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.

References

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

[2] A. Zeller, “Automated debugging: Are we close?,” *Computer*, vol. 34, no. 11, pp. 26–31, 2001.

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

[4] “Interflop Project.” `https://github.com/interflop/interflop`.

[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.