PRECAST

Portability, Reproducibility and Exception Handling of Control Software on Host and Target Platforms

Contract No. 4000122343/17/NL/FE/as

Andoni Arregi

2018-12-11
TEC-ED & TEC-SW Final Presentation Day – ESTEC

GTD GmbH

Table of contents

Introduction

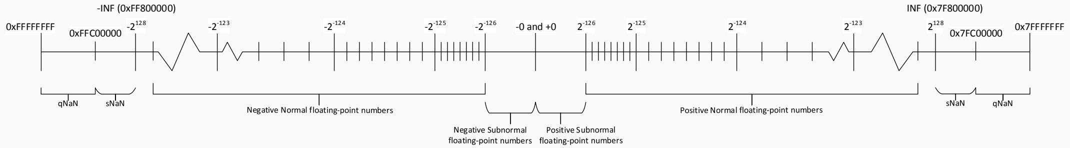
Computers and Floating Point Arithmetic

- Computers are called that way because we expect them to be able to compute
- Floating-point arithmetic is a somewhat tricky *model* for real-number arithmetic (see [2])
- One of the first floating-point arithmetics developed for the `MANIAC` and called `FLINT`, had 40 bits precision in the 50s
- Now, over 60 years later, we have commonly *only* 64 bits precision

This makes exhaustive testing of numerical computing impossible to current computers but does not provide enough accuracy to forget about accuracy problems.

Having reproducible numerical results enables inferring the correct behavior of a system based on the known correct behavior of another one.

Computers and Floating Point Arithmetic



The rational number $\frac{1}{10} = 0.1$ has no exact representation in floating-point (no, not even in 64 bits)

```
32 bits float:            0x3DCCCCCC 9.99999940395355224609375E−2
64 bits double: 0x3FB999999999999A 1.00000000000000005551115E−1
```

Computers and Floating Point Arithmetic

Some curios non-intuitive phenomena affects the computation with floating-point numbers of which the software programmer shall be aware of, such as Rump's example (see the original article [5] and how to reproduce it on IEEE 754 arithmetic in [4]):

$$f = (333.75 - a^2)b^6 + a^2(11a^2b^2 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b^2}$$

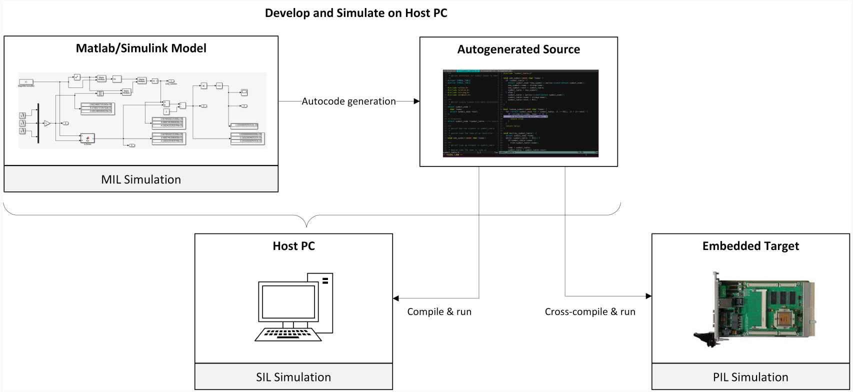
```
a = 77617
b = 33096
32 bits evaluation f = 1.172604
64 bits evaluation f = 1.1726039400531786
correct value      f = −0.827396059946821368141165095479 82
```

Validation in Model Based Software Engineering

- Can we numerically **validate** the **Model** in a *model-based software-engineering* workflow for GNC/AOCS software?
- Can we **return** to our **simulation model** once a numerical error is detected *on-target* and **reproduce** it?

Validation in Model Based Software Engineering

Example Model Based SW Engineering approach based on Matlab/Simulink including MIL, SIL, and PIL simulation modes (▸ note):



Validation in Model Based Software Engineering

No we Can't

- Numerical behavior is not the same on *host* as on *target*
- Error condition handling is not the same on *host* as on *target*

Problem!

Numerical reproducibility issues between the *host* systems used for development (normal PCs) and the embedded *target* systems (on-board processors) **impede** a proper and early **validation** on *host*, as well as the investigation of problems observed during AIT phases on *target* systems.

...but we use double precission!
...the differences are *small*
...we can neglect them

Motivation

Motivation

Algorithms developed for space-software and especially for GNC/AOCS systems base on numerical computations including among others:

- Elementary mathematical functions (trigonometric functions, logarithms, exponentials, &c)
- Other arithmetic/algebraic operations:
    - Matrix multiplications (based on the *dot product* operation)
    - Matrix inversions
    - Linear system solving
    - &c

Motivation

Standards such as

- the IEEE-754 Floating-point arithmetic standard,
- the ISO C programming language standard, and
- the POSIX standard

clarify numerical and error condition behavior.

*But the **reproducibility** problem throughout hardware/software development environments is still **not solved** because of the **common noncompliance** to these standards and **opacity** of the **simulation tools** used.*

Motivation

Common non-compliances of these standards are, among others,

- the lack of subnormal (▸ note) floating-point support (e.g., GRFPU),
- the lack of Fused Multiply Add (▸ note) operation support (e.g., SPARC V8), and
- compiler and tool-chain library non-compliances

But do we care?

Or is this only a theoretical kind of discussion...

Motivation

Numerical error analysis is a difficult task for complex numerical computations:

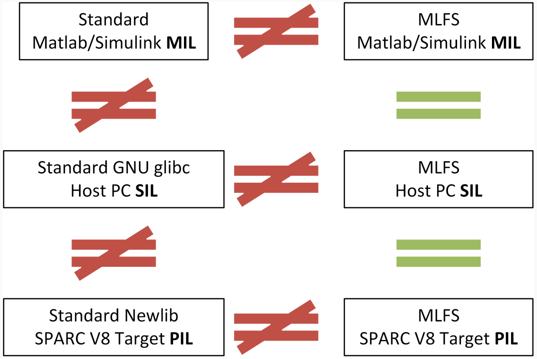
- how can an accuracy requirement of having a relative error below 10^{-8} be reliably validated?
- on which platforms?
- on which testing level? On unit tests only?
- against which reference? Matlab?

We should not be surprised if Computer Science students prefer a sliver under the fingernail to compulsory study of Numerical Analysis. It's a horrible subject.

William Kahan, [3]

Motivation

When using the **MLFS** mathematical library, which provides a subset of the `mat.h` C functions and has been qualified in the previous project together with the **PRECAST** guidelines we obtained the following numerical reproducibility results in the shown simulation modes:



Standard Matlab/Simulink **MIL**
MLFS Matlab/Simulink **MIL**
Standard GNU glibc Host PC **SIL**
MLFS Host PC **SIL**
Standard Newlib SPARC V8 Target **PIL**
MLFS SPARC V8 Target **PIL**

Our Analysis

Our Analysis

We analyzed the following aspects regarding numerical and error condition handling reproducibility:

- Implications of the use of **Matlab/Simulink** (*normal*, *accelerator*, *rapid accelerator*, and SIL modes) and autocode generation
- Implications of the use of **elementary mathematical functions** (those provided by `math.h`)
- Implications of the use of other arithmetic/algebraic operations (dot product, matrix multiplication, &c)
- Implications of the differences in **FPU architectures** (e.g., availability of *Fused Multiply Add* instructions, subnormal support, &c.)

Our Analysis

We analyzed the following aspects regarding numerical and error condition handling reproducibility:

- Implications of the use of **parallel/multicore** computing and the use of GPUs (in examples given by Intel the same binary can give different results even on the same processor in successive runs [6])
- Implications of the used **compilers** and different **tool-chains** (GCC, Clang, Intel C compiler)
- Implications of **exception** generation and **NaN** handling

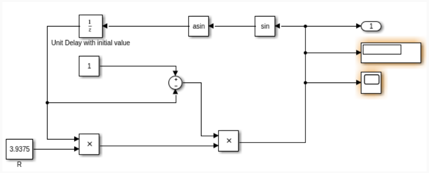
Our Analysis

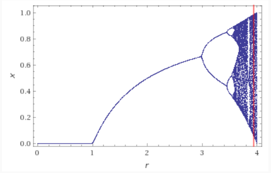
The goal of PRECAST is to produce guidelines which when applied yield reproducible numerical results (to the last bit) on Model in the loop (MIL – host), Software in the loop (SIL – host), and Processor in the loop (PIL – target) executions.

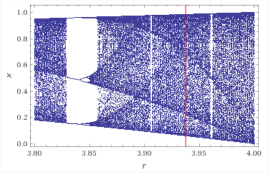
Our Analysis

We defined an **aperiodic** iterative **simulation model** that is highly **sensitive** to input data. An altered version of the **Logistic Map**:

$$x_{n+1} = r \cdot \arcsin(\sin(x_n)) \cdot (1 - \arcsin(\sin(x_n)))$$







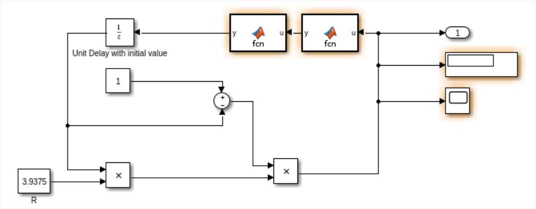
Our Analysis

This model was run in the following modes:

- Simulink *normal*, *accelerator*, and *rapid accelerator* modes.
- Simulink SIL mode (x86 host PC)
- PIL mode on a SPARC V8 LEON2 (GR-CPCI-AT697)

Our Analysis

Since a good deal of numerical discrepancies come from the elementary mathematical functions used in the model, we modified the model substituting these elementary mathematical functions by those provided in the qualified `MLFS` mathematical library:



For this purpose the *Embedded Matlab* approach or the Simulink S-Function approach can be used.

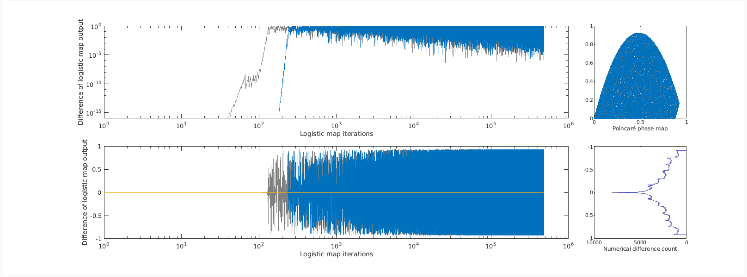
This ensures that the same `MLFS` implementations of these functions will be used throughout the different simulation modes (MIL, SIL, PIL).

Our Analysis

The following diagram shows an example of the numerical differences analysis done between:

- Standard Simulink MIL (`FDLIBM`) vs. standard Simulink SIL (`glibc`) simulation (grey)
- Standard Simulink MIL (`FDLIBM`) vs. `MLFS` MIL (blue)
- `MLFS` MIL vs. `MLFS` SIL (yellow; no numerical difference)

*All MIL simulations done in normal mode.



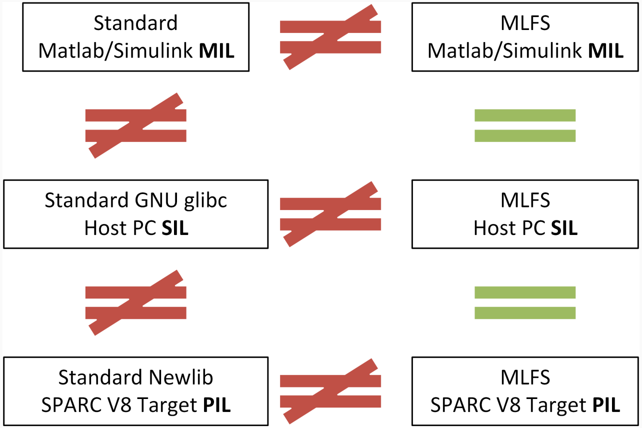
Our Analysis

In addition, analysis has also been done on the **EagleEye** AOCS Simulink model.

- We analyzed which numerical computations are being used by this model
    - Elementary mathematical functions provided by `math.h`
    - Algebraic operations (e.g., matrix multiplications, linear system solving, matrix inversions, &c.)
- We run sample tests of some of these numerical computations to assess the implications of our study.

Our Analysis

Relation of MIL, SIL, and PIL executions regarding numerical results we obtained with **MLFS** and the **PRECAST** guidelines:


Standard Matlab/Simulink **MIL**
≠
MLFS Matlab/Simulink **MIL**
≠
=
Standard GNU glibc Host PC **SIL**
≠
MLFS Host PC **SIL**
≠
=
Standard Newlib SPARC V8 Target **PIL**
≠
MLFS SPARC V8 Target **PIL**

Our Analysis

Regarding error conditions handling we analyzed the following:

- How the `MLFS` library behaves regarding the floating-point exception generation requirements of IEEE-754
- What the implications of the FPUs and the compiler tool-chains are
- How the NaN (qNaN and sNaN) generation and propagation/percolation behaves

Our Analysis

Our conclusions regarding error conditions handling are:

- The floating-point exceptions *Invalid Operation*, *Overflow*, and *Divide by Zero* should be regarded as **more important** than *Inexact* and *Underflow*, as they represent significant errors and thus, effort should be made to detect them.
- The `MLFS` library generates the *important* exceptions required by IEEE-754 for each function
- For which input values the exceptions are generated differs from platform to platform, because of the different compiler tool-chain and FPU behavior (e.g., because of unavailable subnormal support).
- The tool-chain behavior introduces differences in the exception behavior on different platforms.
- NaNs can be effectively used for pre-initializing data to invalid values (sNaN) as well as for the detection of invalid computations, via NaN propagation. But again care must be taken, as some older compilers have bugs regarding this behavior (NaN propagation through relational operators solved in GCC 8.1)

Conclusion

Conclusion

Numerical Reproducibility

It is **not too difficult** to get reproducible results on *host* and *target* systems to enable the validation of numerical computations on model level but **care has to be taken**.

Error Condition Handling

- Focus shall be set on the important exceptions: **Invalid Operation**, **Overflow**, and **Divide by Zero**
- **Pre-initialize** data to **sNaN**, so that uninitialized use can be detected via exception
- Frequently **check** for **NaN** values as the result of intermediate computations

Guidelines and Recommendations

Guidelines and Recommendations

(1) Same Mathematical Library

Always use the same mathematical library for elementary mathematical functions (e.g., `MLFS`) on all systems (*host* and *target*)

To assure that the starting point of numerical and exception behavior will be the same on all those platforms (the compilation and the hardware itself will still have an impact though, which we will try to solve with the following guidelines).

Guidelines and Recommendations

(2) Compilation Flags

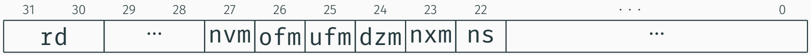
Always compile using the `-frounding-math -fsignaling-nans -fno-builtin` compiler options

To obtain an IEEE-754 compliant floating-point arithmetic behavior when using GCC, or the `-fp-model strict -fp-model source` *options when using the Intel C compiler. Clang does not support* `-frounding-math -fsignaling-nans`.

Guidelines and Recommendations

(3) Configure FSR Register

Properly configure the `FSR` resister on SPARC V8 processors



- *Always use* round to nearest tie to even *rounding mode (bits 30 and 31,* `rd`*, set to 0).*
- *Configure the FPU to trap on the* important *exceptions* (Invalid Op., Div. by 0, *and* Overflow) *while developing the software. For LEON2, LEON3, and LEON4 processors see the* `FSR` *register bits below and set the corresponding bits to 1 (*`nvm` *to 1,* `ofm` *to 1, and* `dzm` *to 1). The general trap enabling bit has also be set to 1 (*`PSR` *bit 5 to 1). For flight software the project shall decide if trapping on the exceptions is desired or not and what the trap handler shall do.*
- *Set the* nonstandard *modus on processors using the* `GRFPU` *(bit 22,* `ns` *set to 1), since the processor will otherwise trap on subnormal floating-point numbers. This modus will handle subnormals as 0, which is not IEEE-754 compliant but will not trap (see [1]).*

**on x86 systems, the MXCSR register will have to be set accordingly.*

Guidelines and Recommendations

(4) Save FPU Context

In case of using task preemption on RTEMS, make sure that the current floating-point status is saved and restored during context switching

Set the `RTEMS_FLOATING_POINT` *attribute flag when creating all RTEMS tasks with* `rtems_task_create()`.

Guidelines and Recommendations

(5) Disable Subnormal Support (In case of GRFPU)

Disable subnormal support in Matlab and Simulink setting the DAZ and FTZ modes

Guidelines and Recommendations

(6) Use MLFS S-Function Blocks in Simulink

Use the S-Function mechanism to use `MLFS` procedures within the Simulink `MIL` simulation

Guidelines and Recommendations

(7) Disable Parallel Processing in Matlab

Limit the computational threads of Matlab and limit Matlab to use only one CPU core

Guidelines and Recommendations

(8) Don’t use the 80 bit x87 FPU Registers

Always compile for SSE/AVX architecture on x86-64 platforms

e.g., using the flag `-msse2` *with GCC or Clang compilers, or* `-march=sse2` *with the Intel C compiler. The use of the 80 bit x87 registers would produce a more accurate results, but the results will not be reproducible on a SPARC V8 architecture.*

Guidelines and Recommendations

(9) Disable FMA Support

Always compile excluding `FMA` instructions on x86 platforms

Use the `-no-fma` *compiler option (this is valid for GCC, Clang, and the Intel C compiler). The SPARC V8 architecture does not support* `FMA` *either.*

Guidelines and Recommendations

(10) Configure IEEE-754 compatibility for CUDA

Configure the CUDA compiler flags for IEEE-754 support

- `-ftz=false`: *use subnormal floating-point numbers (do not flush them to zero)*
- `-prec-div=true`: *compute division to the nearest floating-point number*
- `-prec-sqrt=true`: *compute square root to the nearest floating-point number*
- `-fmad=false`: *do not merge multiply and add operations*

**If subnormal support is not desired, as for the GRFPU, set* `-ftz=true`

References

References i

Cobham-Gaisler.
Handling denormalized numbers with the grfpu, 2015.

D. Goldberg.
What every computer scientist should know about floating-point arithmetic, 1991.

W. Kahan.
Matlab's loss is nobody's gain, 1998.

E. Loh and G. Walster.
Rump's example revisited.
In *Reliable Computing*, volume 8, pages 245–248. 01 2002.

References ii

S. M. Rump.
Reliability in computing: The role of interval methods in scientific computing.
chapter Algorithms for Verified Inclusions: Theory and Practice, pages 109–126.
Academic Press Professional, Inc., San Diego, CA, USA, 1988.

G. Zitzlsberger.
Fp accuracy & reproducibility; intel c++/fortran compiler, intel math kernel library, and intel threading building blocks, September 2014.

Additional Notes

Subnormal Floating-Point Numbers

There is a special subset of floating-point numbers called *Subnormals*, representing very small numbers. These numbers have less precision that the rest of the floating-point numbers. Their range being (only shown for positive numbers, for negative ones the symmetrical ranges apply):
Double: $[2^{-1074}, 2^{-1022}) \approx [4.94066 \cdot 10^{-324}, 2.22507 \cdot 10^{-308})$
Float: $[2^{-149}, 2^{-126}) \approx [1.40130 \cdot 10^{-45}, 1.17549 \cdot 10^{-38})$

▸ back

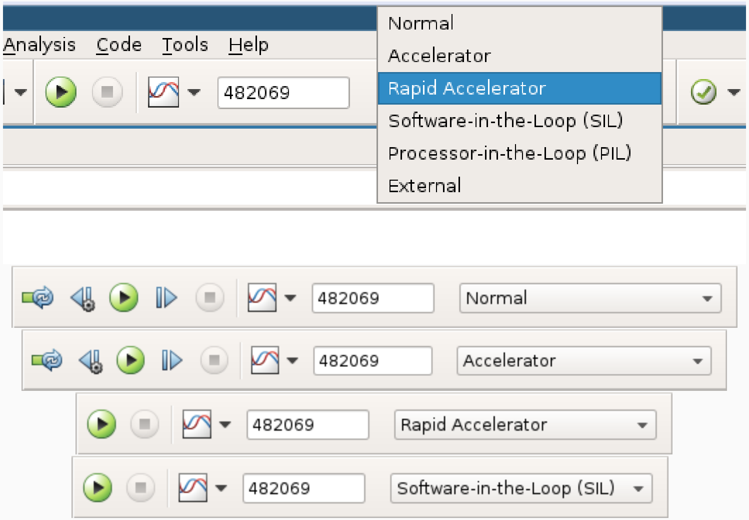
Simulation Modes

Some short notes on the different existing simulation modes and their relation to Matlab/Simulink:

- **MIL**: Model In the Loop simulation. E.g., a native Simulink simulation, run in *normal mode* within the Simulink environment, where our model is being computed by the Simulink engine.
- **SIL**: Software In the Loop simulation. Here we mean a simulation where source code has been auto-generated from our model, compiled with a compiler tool-chain available on the host PC to a standalone executable, and run on that host PC without any Simulink dependency.
- **PIL**: Processor In the Loop execution. Here we mean a run, where source code has been auto-generated from our model, cross-compiled for the embedded target processor, and run on that target.

Simulink Simulation Modes

Selection in Simulink of the different available modes:



Simulink Simulation Modes

Note on **Matlab/Simulink** simulation modes:

- ***Normal mode***: The Matlab computing environment is used. The math library is based on `FDLIBM` and BLAS/LAPACK are used for algebraic operations.
- ***Accelerator Mode***
  - *Default accelerator mode*: No compiler needed. An *acceleration target code* is generated for the Simulink model and kept in memory. Further concrete implementation aspects are unknown but this seems to be a pure MIL simulation based on the same Matlab computing environment as for the *normal mode*.
  - *Classic accelerator mode*: Compiler needed. C code is generated for the Simulink model and linked into a shared library. The Matlab own RTW (Real Time Workshop) libraries are used thus, the math library used is still the Matlab internal one based on `FDLIBM` and BLAS/LAPACK for algebraic operations.
    ```
    set_param(0,'GlobalUseClassicAccelMode','on');
    ```

Simulink Simulaton Modes

Note on **Matlab/Simulink** simulation modes:

- ***Rapid accelerator mode***: Compiler needed. C code is generated for the Simulink model and the solver, producing a standalone executable. The Matlab own RTW libraries are used thus, the math library used is still the Matlab internal one based on `FDLIBM`.
- ***SIL mode***: Compiler needed. C code is generated and a standalone executable is produced. The libraries available to the system compiler tool-chain are used (e.g., `glibc`) thus, the math library used is a different one than the Matlab internal one. *(the scope functionality is not available in this mode)*

▸ back

Fused Multiply Add Operation

A **FMA** operation represents the following arithmetic operation on three operands:

$$fma(x, y, z) = x * y + z$$

with a single rounding after the addition and not once for the multiplication and once for the addition as if performed in two separate operations. This is a required arithmetic operation of the IEEE-754 standard since its 2008 version.

FMA availability

Since about mid 2013 x86 Intel and AMD procesors (Intel Haswell and AMD Bulldozer) have this operation, Newer ARM processors (ARM v8) also have it but SPARC V8 processors (LEON2, LEON3, and LEON4) do not provide it!

▸ back