A Practical Approach to Verification of Floating-Point C/C++ Programs with math.h/cmath Functions

A Practical Approach to Verification of Floating-Point C/C++ Programs with math.h/cmath Functions

ROBERTO BAGNARA, University of Parma and BUGSENG srl
MICHELE CHIARI, University of Parma, BUGSENG srl, and Politecnico di Milano
ROBERTA GORI, University of Pisa
ABRAMO BAGNARA, BUGSENG srl

Verification of C/C++ programs has seen considerable progress in several areas, but not for programs that use these languages' mathematical libraries. The reason is that all libraries in widespread use come with no guarantees about the computed results. This would seem to prevent any attempt at formal verification of programs that use them: without a specification for the functions, no conclusion can be drawn statically about the behavior of the program. We propose an alternative to surrender. We introduce a pragmatic approach that leverages the fact that most `math.h`/`cmath` functions are *almost* piecewise monotonic: as we discovered through exhaustive testing, they may have *glitches*, often of very small size and in small numbers. We develop interval refinement techniques for such functions based on a modified dichotomic search, that enable verification via symbolic execution based model checking, abstract interpretation, and test data generation. Our refinement algorithms are the first in the literature to be able to handle non-correctly rounded function implementations, enabling verification in the presence of the most common implementations. We experimentally evaluate our approach on real-world code, showing its ability to detect or rule out anomalous behaviors.

CCS Concepts: • **Software and its engineering → Software verification**; • **Theory of computation → Program verification**; • **Mathematics of computing** → *Solvers*.

Additional Key Words and Phrases: Floating-point numbers, constraint propagation, model checking, abstract interpretation, program verification, symbolic execution

ACM Reference Format:
Roberto Bagnara, Michele Chiari, Roberta Gori, and Abramo Bagnara. 2019. A Practical Approach to Verification of Floating-Point C/C++ Programs with math.h/cmath Functions. *ACM Trans. Softw. Eng. Methodol.* 37, 4, Article 111 (August 2019), 52 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The use of floating-point computations for the implementation of critical systems is perceived as increasingly acceptable. This was facilitated by the widespread adoption of significant portions of the IEEE 754 standard for binary floating-point arithmetic [40]. Even in modern avionics, floating-point numbers are now used, more often than not, instead of fixed-point arithmetic [15, 56]. Thus, the development of techniques for verifying the correctness of such programs becomes imperative.

According to [56], the main goals of floating-point program verification are the following:

Authors' addresses: Roberto Bagnara, University of Parma, BUGSENG srl, roberto.bagnara@unipr.it, roberto.bagnara@bugseng.com; Michele Chiari, University of Parma, BUGSENG srl, Politecnico di Milano, michele.chiari@polimi.it; Roberta Gori, University of Pisa, roberta.gori@unipi.it; Abramo Bagnara, BUGSENG srl, abramo.bagnara@bugseng.com.

1049-331X/2019/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

(1) Proving that the program will never trigger "undefined" or "undesirable" behaviors, such as an overflow on a conversion from a floating-point type to an integer type.
(2) Pin-pointing the sources of roundoff errors in the program; proving an upper bound on the amount of roundoff error in some variable.
(3) Proving that the program implements such or such numerical computation up to some specified error bound.

An infamous example of the issues addressed by goal (1) is the crash of the Ariane 5 rocket, which was caused by an overflow in a conversion from a 64-bit floating-point to a 16-bit integer in the software embedded into its control system [46]. Other examples of such "undesirable" behaviors are the generation of infinities or Not-A-Numbers (NaNs). An empirical study on real-world numerical software [26] found out that behaviors of this kind are the cause of 28 % of numerical bugs in the considered programs. [26] advocates for the development of tools for the detection of such bugs, and the automated generation of test cases triggering them. In this paper, we present a technique to achieve such goals, leaving (2) and (3) to the extensive literature concerned with evaluating the *precision* of computations, such as [22, 35, 63].

To illustrate the concrete problem raised by the use of floating-point computations in program verification settings, consider the code reproduced in Figure 1. It is a reduced version of a real-world example extracted from a critical embedded system. The purpose of function `latlong_utm_of` is to convert the latitude and longitude received from a drone to UTM coordinates, which are stored in the two global variables at lines 45-46. For the moment, let us just notice that this code consists in a large and varied amount of floating-point computations, many of them non-linear. Many calls to mathematical functions are made (highlighted in red): any kind of analysis of this code must be able to take them into account. Some of the questions to be answered for each one of the floating-point operations in this code are:

(i) Can infinities and NaNs be generated?
(ii) Can `sin`, `cos`, and `tan` be invoked on ill-conditioned arguments?
(iii) If anomalies of any one of these kinds are possible, which inputs to the given functions may cause them?

Concerning question (ii), we call the argument of a floating-point periodic trigonometric function *ill-conditioned* if its absolute value exceeds some application-dependent threshold. Ideally, this threshold should be just above π. To understand this often-overlooked programming error, consider that the distance between two consecutive floating-point numbers (i.e., their ULP)[1] increases with their magnitude, while the period of trigonometric functions remains constant. Thus, if x is an IEEE 754 single-precision number and $x \geq 2^{23}$, then the smallest single-precision range containing $[x, x + 2\pi)$ contains no more than three floating-point numbers. Current implementations of floating-point trigonometric functions, such as those of CR-LIBM,[2] libmcr[3], and GNU libc [51], contain precise range reduction algorithms that compute a very accurate result even for numbers much higher than 2^{23}. The point is that the function inputs at this magnitude are so distant form each other that the graph of the function becomes practically indistinguishable from that of a pseudo-random number generator, potentially becoming useless for the application. Substitute 2^{23} with 2^{52}, and the same holds for IEEE 754 double-precision numbers.

In order to answer questions (i)–(iii), we need a precise characterization of the semantics of all operations involved in the program. Most implementations of the C and C++ programming languages

[1] *ULP* stands for *unit in the last place*: if x is a finite floating-point number, $\mathrm{ulp}(x)$ is the distance between the two finite floating-point numbers nearest x [58].
[2] See https://gforge.inria.fr/scm/browser.php?group_id=5929&extra=crlibm, last accessed on July 16th, 2020.
[3] See https://github.com/simonbyrne/libmcr/blob/master/README, last accessed on July 16th, 2020.

```
1  #include <math.h>
2  #include <stdint.h>
3
4  #define RadOfDeg(x) ((x) * (M_PI/180.))                              ▷ Convert degrees to radians
5  #define E 0.08181919106 /* Computation for the WGS84 geoid only */
6  #define LambdaOfUtmZone(utm_zone) RadOfDeg((utm_zone-1)*6-180+3)     ▷ Origin longitude of UTM zone
7  #define CScal(k, z) { z.re *= k; z.im *= k; }                           ▷ Multiply complex z by k
8  #define CAdd(z1, z2) { z2.re += z1.re; z2.im += z1.im; }                      ▷ Complex addition
9  #define CSub(z1, z2) { z2.re -= z1.re; z2.im -= z1.im; }                   ▷ Complex subtraction
10 #define CI(z) { float tmp = z.re; z.re = - z.im; z.im = tmp; }                  ▷ Multiply by i
11 #define CExp(z) { float e = exp(z.re); z.re = e*cos(z.im); \              ▷ Exp of complex number
12   z.im = e*sin(z.im); }
13 #define CSin(z) { CI(z); struct complex _z = {-z.re, -z.im}; \            ▷ Sine of complex number
14   float e = exp(z.re); float cos_z_im = cos(z.im); z.re = e*cos_z_im; \
15   float sin_z_im = sin(z.im); z.im = e*sin_z_im; _z.re = cos_z_im/e; \
16   _z.im = -sin_z_im/e; CSub(_z, z); CScal(-0.5, z); CI(z); }
17
18 static inline float isometric_latitude(float phi, float e) {
19   return log^p1(tan^p2(M_PI_4 + phi / 2.0))
20     - e / 2.0 * log((1.0 + e * sin^p3(phi)) / (1.0 - e * sin(phi)));
21 }
22
23 static inline float isometric_latitude0(float phi) {
24   return log^p4(tan(M_PI_4 + phi / 2.0));
25 }
26
27 void latlong_utm_of(float phi, float lambda, uint8_t utm_zone) {
28   float lambda_c = LambdaOfUtmZone(utm_zone);           ▷ Function arguments:
29   float ll = isometric_latitude(phi, E);                ▷  phi: latitude in radians
30   float dl = lambda - lambda_c;                         ▷  lambda: longitude in radians
31   float phi_ = asin(sin^p5(dl) / cosh(ll));             ▷  utm_zone: UTM zone of the location
32   float ll_ = isometric_latitude0(phi_);                ▷ Output:
33   float lambda_ = atan(sinh(ll) /^p6 cos^p7(dl));       ▷  latlong_utm_x: easting of the location
34   struct complex z_ = { lambda_, ll_ };                 ▷  latlong_utm_y: northing of the location
35   CScal(serie_coeff_proj_mercator[0], z_);              ▷ Function latlong_utm_of converts the
36   uint8_t k;                                            ▷ coordinates of a location from WGS84
37   for(k = 1; k < 3; k++) {                              ▷ latitude and longitude to UTM coordinates.
38     struct complex z = { lambda_, ll_ };
39     CScal(2*k, z);
40     CSin(z);
41     CScal(serie_coeff_proj_mercator[k], z);
42     CAdd(z, z_);
43   }
44   CScal(N, z_);
45   latlong_utm_x = XS + z_.im;
46   latlong_utm_y = z_.re;
47 }
```

Fig. 1. Code excerpted from a real-world avionic library. The original source code is available at http://paparazzi.enac.fr, Paparazzi UAV (Unmanned Aerial Vehicle), v5.14.0_stable release, file sw/misc/satcom/tcp2ivy.c, last accessed on July 16th, 2020. The annotations p_i are referred to in Section 7.2. Comments preceded by ▷ were not present in the original code.

provide floating-point data types that conform to IEEE 754 as far as basic arithmetic operations and conversions are concerned. The C and C++ floating-point mathematical functions are part of the standard libraries of the respective languages (see, e.g., [42, 43]). Access to such functions requires inclusion of the `math.h` header file in C, or the `cmath` header file in C++. The library implementing them is called `libm`. Very few C/C++ implementations comply to the *recommendation* of the IEEE 754 standard [40, Section 9.2] that such functions be correctly rounded.[4] One of them is CR-LIBM, a remarkable research prototype that, however, has still not found industrial adoption, possibly because of the worst-case performance of some functions (the average case being usually very good). Another freely available correctly-rounded library is libmcr, by Sun Microsystems, Inc. It provides some double-precision functions (`exp`, `log`, `pow`, `sin`, `cos`, `tan`, `atan`), but its development stopped in 2004. Even though we cannot exclude the existence of proprietary implementations of `libm` providing formalized precision guarantees, we were not able to find one. In the most popular implementations, such guarantees are usually not available. For example, the documentation of GNU libc [51] contains:

> *"Therefore many of the functions in the math library have errors. The table lists the maximum error for each function which is exposed by one of the existing tests in the test suite. The table tries to cover as much as possible and list the actual maximum error (or at least a ballpark figure) but this is often not achieved due to the large search space."*

This provides nothing that can really be trusted in a safety-critical context. In the embedded world, we checked the documentation of all major toolchain providers in our possession: for four of them we found that the lack of guarantees is explicitly mentioned (e.g., [2, page 2-338], [36, page 591], [37, page 665], [28, page 180], [65, page 87]), whereas for all the others (incuding Arm, CodeWarrior/Freescale/NXP, CrossWorks, HighTec, IAR, Keil, Microchip, NEC, Renesas, TASKING, Wind River) the lack of guarantees is left implicit.

We do not have a precise specification for the library functions that are assumed in the code of Figure 1. Its sources refer to GNU libc and to Newlib,[5] but no specific versions are mentioned. We can probably assume a POSIX-compliant behavior with respect to special values. E.g., if `log()` is called with a negative number, a NaN is returned. If `atan()` is called with ±1, then an infinity is returned [41]. This information is not sufficient to provide a general answer to the verification questions (i)–(iii). Things change if we fix a specific implementation of the mathematical library.

In this respect, we propose a practical approach that enables verification of C/C++ programs using `math.h`/`cmath` functions, even with minimal or no specification in addition to the special cases mandated by standards such as POSIX [41]. Our main contribution is the extension of constraint satisfaction problem solving techniques based on interval consistency [10, 11] to programs using `libm` functions, by providing interval refinement algorithms for such constraints. We use such techniques to solve the constraint systems generated for each program path by symbolic execution [13], and perform symbolic-execution based model checking [31]. Symbolic-execution based test data generation [4] may be performed as well. The application to abstract interpretation based on (multi-) interval domains is also straightforward [19].

We conducted an investigation by means of exhaustive testing on the most common implementations of `libm` (cf. Section 9.2). We observed that, for all the implementations we tested, the piecewise monotonicity property of the corresponding real functions is "almost" preserved. The results of this investigation are presented in Section 4. Consider, for instance the `tanhf()` function, which is meant to approximate the hyperbolic tangent function over IEEE 754 single-precision

[4] A function is said to be *correctly rounded* if its result is as if computed with infinite precision, and then rounded to the floating-point format in use.

[5] See https://sourceware.org/newlib/, last accessed on July 16th, 2020.

floats. While $y = \tanh(x)$ is monotonically (strictly) increasing over $(-\infty, +\infty)$, `y = tanhf(x)` can be monotonically non-decreasing over the full range of IEEE 754 single-precision floats, or it can be "almost monotonically non-decreasing." By this we mean that, going from $-\infty$ to $+\infty$, there may be an occasional drop in the graph of `tanhf()`, but this is quickly recovered from, that is, the function starts increasing again. We use the term *glitches* to name such occasional drops: we observed that glitches are often shallow (most often just one ULP), narrow (most often just 2 ULPs), and, on average, not very frequent. We leverage this fact to provide general interval refinement algorithms that enable software verification and testing. Such algorithms are based on an efficient dichotomic search of the intervals to be refined. While traditional dichotomic search can only be applied to monotonic domains, our version is modified to work despite the presence of glitches, by just exploiting some minimal data about them. As we explain in Section 8, our algorithms are the first in the literature that do not require the function implementations to be correctly rounded, thus enabling their use with the most common `libm` implementations.

If we have approximate but correct information about the maximal depth and width of glitches and, possibly, their number and their localization,[6] then we can guarantee the refined intervals are conservative. This allows performing formal verification via abstract interpretation, symbolic model checking or automatic theorem proving. With quite precise (correct) information and small/few glitches (or if there are no glitches at all, which includes the case of correctly rounded implementations), the refinement results in tight intervals, and verification is computationally cheaper and with fewer "don't knows". With less precise but still correct information, verification is still possible, but slower and with more "don't knows". With incorrect information about glitches, we can still automatically generate test inputs with much greater coverage than random testing.

For single-precision and half-precision IEEE 754 functions, collection of precise data about glitches can be obtained by analysing each function on every possible input. This is perfectly feasible since glitch data must be collected only once for each implementation of `libm`. For double-precision functions, when the mathematical library comes equipped with guarantees on the maximum errors, they can be used as correct approximations of the glitch parameters required by our algorithm. This is the case for the HA and LA accuracy modes of the Intel Math Kernel Library.[7] Techniques for automatically proving the correctness of error bounds in `math.h`/`cmath` implementations have been recently developed [38, 39, 49]. When provably correct bounds are available, we can verify programs by proving that bad things cannot happen. On the other hand, when the target mathematical library comes equipped with merely empirical information on the maximum errors, as is the case for GNU libc [51], such information can be used to obtain (possibly incorrect) bounds for glitches, which enable the automatic generation of test inputs.

The approach presented in this paper has been fully implemented in a commercial tool (ECLAIR, developed and commercialized by BUGSENG) and is used for verification and testing of real C code. We used this tool to experimentally evaluate the effectiveness of our approach. As we detail in Section 7, ECLAIR was able to answer questions (i)–(iii) by detecting some dangerous bugs affecting the code of Figure 1, including the generation of a NaN when certain coordinates are received as inputs. Moreover, ECLAIR has been able to answer the same questions for a large and heterogeneous benchmark that we assembled with both real-world and self-developed code, without timing out in at least 96 % of the cases. In Section 8, we compare our techniques with the state of the art, finding out that they outperform other approaches in the ability to detect anomalous behaviors, most of the times offering even shorter analysis times.

[6] For details, see the requirements of the algorithms in Section 5.2.

[7] See https://software.intel.com/sites/products/documentation/doclib/mkl/vm/vmdata.htm, last accessed on July 16th, 2020.

Plan of the paper. Section 2 recalls basic definitions and introduces the required notation; Section 3 explains the verification framework we use; Section 4 introduces the notions of *monotonicity glitch* and *quasi-monotonicity*; Section 5 describes direct and indirect propagation algorithms that are able to deal with (at least) 75 of the `math.h`/`cmath` functions; Section 6 explains how trigonometric functions, which are periodic, can be treated by partitioning a subset of their graph into a set of quasi-monotonic branches; Section 7 briefly describes the implementation in the context of the ECLAIR software verification platforms and illustrates the experimental results; Section 8 compares our approach with the state of the art, and other related work; Section 9 discusses the problems that remain to be solved and sketches several ideas for future work; Section 10 concludes the main part of the paper. Appendix A presents additional data about glitches in single-precision functions for several implementations of `libm`, and Appendix B contains more details on the algorithms of Section 5. The proofs of the results of Sections 5 and 6 are reported in [6].

2 BACKGROUND: FLOATING-POINT NUMBERS AND INTERVALS

We denote by $\mathbb{R}_+$ and $\mathbb{R}_-$ the sets of strictly positive and strictly negative real numbers, respectively.

Definition 2.1. **(IEEE 754 binary floating-point numbers.)** A set of IEEE 754 binary floating-point numbers [40] is uniquely identified by: $p \in \mathbb{N}$, the number of significant digits (precision); $e_{\max} \in \mathbb{N}$, the maximum exponent, the minimum exponent being $e_{\min} := 1 - e_{\max}$. The set of binary floating-point numbers $\mathbb{F}(p, e_{\max}, e_{\min})$ includes:

- all signed zero and non-zero numbers of the form $(-1)^s \cdot 2^e \cdot m$, where
  - s is the *sign bit*;
  - the *exponent* e is any integer such that $e_{\min} \le e \le e_{\max}$;
  - the *mantissa* m, with $0 \le m < 2$, is a number represented by a string of p binary digits with a "binary point" after the first digit:

$$m = (d_0 \,.\, d_1 d_2 \ldots d_{p-1})_2 = \sum_{i=0}^{p-1} d_i 2^{-i};$$

- the *infinities* $+\infty$ and $-\infty$.

The smallest positive *normal* floating-point number is $f_{\min}^{\text{nor}} := 2^{e_{\min}}$ and the largest is $f_{\max} := 2^{e_{\max}}(2 - 2^{1-p})$. The non-zero floating-point numbers whose absolute value is less than $2^{e_{\min}}$ are called *subnormals*: they always have fewer than p significant digits. Every finite floating-point number is an integral multiple of the smallest subnormal magnitude $f_{\min} := 2^{e_{\min}+1-p}$. Note that the *signed zeroes* $+0$ and -0 are distinct floating-point numbers.

Each IEEE 754 binary floating-point format also includes the representation of symbolic data called *NaNs*, from "Not a Number." There are *quiet NaNs*, which are propagated by most operations without signaling exceptions, and *signaling NaNs*, which cause signaling invalid operation exceptions. The unintended and unanticipated generation of NaNs in a program (e.g., by calling the `log` function on a negative number) is a serious programming error that could lead to catastrophic consequences.

In the rest of the article we will only be concerned with IEEE 754 binary floating-point numbers, excluding NaNs, and we will write simply $\mathbb{F}$ for $\mathbb{F}(p, e_{\max}, e_{\min})$ when there is no risk of confusion.

Definition 2.2. **(Floating-point symbolic order.)** Let $\mathbb{F}$ be any IEEE 754 floating-point format. The relation $\prec \subseteq \mathbb{F} \times \mathbb{F}$ is such that, for each $x, y \in \mathbb{F}$, $x \prec y$ if and only if either: $x = -\infty$ and $y \neq -\infty$, or $x \neq +\infty$ and $y = +\infty$, or $x = -0$ and $y \in \{+0\} \cup \mathbb{R}_+$, or $x \in \mathbb{R}_- \cup \{-0\}$ and $y = +0$, or $x, y \in \mathbb{R}$ and $x < y$.

The partial order $\preccurlyeq \subseteq \mathbb{F} \times \mathbb{F}$ is such that, for each $x, y \in \mathbb{F}$, $x \preccurlyeq y$ if and only if $x \prec y$ or $x = y$.

Note that $\mathbb{F}$ is linearly ordered with respect to '$\prec$'.

For $x \in \mathbb{F}$, we sometimes confuse the floating-point number with the extended real number it represents, the floats -0 and $+0$ both corresponding to the real number 0. Thus, when we write, e.g., $x < y$ we mean that x is numerically less than y (for example, we have $-0 \prec +0$ although $-0 \not\prec +0$, but note that $x \preccurlyeq y$ implies $x \leq y$ if x and y are finite).

Definition 2.3. **(Floating-point intervals.)** Let $\mathbb{F}$ be any IEEE 754 floating-point format. The set $\mathcal{I}_{\mathbb{F}}$ of floating-point intervals with boundaries in $\mathbb{F}$ is

$$\mathcal{I}_{\mathbb{F}} := \{\varnothing\} \cup \left\{ [l,u] \;\middle|\; l,u \in \mathbb{F}, l \preccurlyeq u \right\}.$$

$[l,u]$ denotes the set $\{ x \in \mathbb{F} \mid l \preccurlyeq x \preccurlyeq u \}$. $\mathcal{I}_{\mathbb{F}}$ is a bounded meet-semilattice with least element $\varnothing$, greatest element $[-\infty, +\infty]$, and the meet operation, which is induced by set-intersection, will be simply denoted by $\cap$. (A *bounded meet-semilattice* is a partially ordered set that has a meet, or greatest lower bound, for any nonempty finite subset and a greatest element.)

Half-open floating-point intervals are defined similarly, so that $[l,u)$ denotes the set $\{ x \in \mathbb{F} \mid l \preccurlyeq x \prec u \}$.

Floating-point intervals with boundaries in $\mathbb{F}$ allow us to capture the extended numbers in $\mathbb{F}$: NaNs should be tracked separately.

Given a floating-point interval $[l,u] \in \mathcal{I}_{\mathbb{F}}$, we denote by $\#[l,u]$ the cardinality of the set $\{ x \in \mathbb{F} \mid l \preccurlyeq x \preccurlyeq u \}$.

Definition 2.4. **(Floating-point successors and predecessors.)** The function $\mathrm{succ}\colon \big(\mathbb{F}\backslash\{+\infty\}\big) \to \mathbb{F}$ is defined, for each $x \in \mathbb{F} \setminus \{+\infty\}$, by $\mathrm{succ}(x) := \min\{ y \in \mathbb{F} \mid x \prec y \}$. Similarly, function $\mathrm{pred}\colon \big(\mathbb{F} \setminus \{-\infty\}\big) \to \mathbb{F}$ is defined, for each $y \in \mathbb{F} \setminus \{-\infty\}$, by $\mathrm{pred}(y) := \max\{ x \in \mathbb{F} \mid x \prec y \}$. We will iteratively apply these functions, so that, e.g., for each $n \in \mathbb{N}$, we will refer to the partial function $\mathrm{succ}^n\colon \mathbb{F} \rightarrowtail \mathbb{F}$ given, for each $x \in \mathbb{F}$, by

$$\begin{cases} \mathrm{succ}^0(x) := x; \\ \mathrm{succ}^{n+1}(x) := \mathrm{succ}\big(\mathrm{succ}^n(x)\big), \text{if } \mathrm{succ}^n(x) \neq +\infty. \end{cases}$$

The definition of the iterated $\mathrm{pred}^n\colon \mathbb{F} \rightarrowtail \mathbb{F}$ function is analogous.

Note that the notation $\#[x,y) > n$ is equivalent to $x \prec \mathrm{succ}^n(y)$.

When the result $\hat{x}$ of a floating-point operation is not representable exactly in the current format, its floating-point approximation is chosen according to the *rounding mode* in use. The IEEE 754 Standard defines rounding modes 'near', 'up', 'down' and 'zero', that round $\hat{x}$ with, respectively:
near : the number $x \in \mathbb{F}$ minimizing $|x - \hat{x}|$; if two such values exists, the even one is chosen.
up : the minimum number $x \in \mathbb{F}$ such that $\hat{x} \prec x$.
down : the maximum number $x \in \mathbb{F}$ such that $\hat{x} \succ x$.
zero : the same as 'down' if $x > 0$, the same as 'up' if $x \leq 0$.

3 BACKGROUND: APPROACHES TO PROGRAM VERIFICATION

In this section, we recall the verification and correctness-ensuring techniques that our interval refinement algorithms enable for floating-point programs.

3.1 Symbolic Execution

Symbolic execution is a technique originally introduced for test-data generation [44], but that has found numerous applications also in the field of program verification [17, 18, 31]. It consists of the evaluation of each execution path in the program by using symbolic values for variables, treating all assignments and guards of conditional statements as constraints on such symbolic values. The

so obtained constraint systems characterize the program variables' values for which the execution path is feasible. Thus, solving the constraint system for a path can either prove it unfeasible, if the system has no solution, or yield a set of assignments for program variables, including input variables, that causes the execution of such path.

We perform symbolic execution of floating-point computations as described in [13] and [4]. We briefly illustrate this approach by means of an example taken from line 31 of the listing in Figure 1:

```
31  float phi_ = asin(sin(dl) / cosh(ll));
```

Program analysis starts by translating the code into *static single assignment form* (SSA) [1]. In this intermediate code representation, complex expressions are decomposed into sequences of assignment instructions where at most one operator is applied, and new variable names are introduced so that each variable is assigned to only once. Thus, assignments can be considered as if they were equality constraints. The above expression is transformed into

```
1  float phi_; double z1, z2, z3, z4, z5, z6;
2  z1 = (double) dl;
3  z2 = sin(z1);
4  z3 = (double) ll;
5  z4 = cosh(z3);
6  z5 = z2 / z4;
7  z6 = asin(z5);
8  phi_ = (float) z6;
```

Then, we can directly regard the assignments as a system of constraints over floating-point numbers. When `if` statements are involved, the execution flow is split in two different paths, each of which results in a different constraint system. Loops are dealt with by unrolling them, and function calls by inlining. For more details, we refer the reader to [13].

Once the constraint system for a path has been generated, it can be immediately solved to generate test input data that causes its execution, or to prove it is unfeasible. Additionally, it is possible to augment the constraint system with assertions, whose truth can be evaluated by solving the system. This is how we perform model checking by means of symbolic execution. We support any kind of assertion formulated as a Boolean combination of constraints on the ranges of program variables. To prove or disprove the truth of an assertion, we generate the constraint systems related to each execution path leading to it. Then, we augment such system with the negation of the assertion, and try to solve it. If no solution is found, we can conclude that the assertion is never violated. Otherwise, the solution to the constraint system produces a counter-example.

For example, suppose we want to know whether a NaN can be generated by the invocation to `asin` in line 7. Thus, we want to prove the assertion $\texttt{z5} \geq -1 \wedge \texttt{z5} \leq 1$, stating that the argument of `asin` is always in its domain. We first generate the constraint system for the execution path leading to line 7. Then, we negate the assertion, obtaining two new constraint systems, one with the addition of $\texttt{z5} < -1$, and the other one with $\texttt{z5} > 1$. As we shall see in Section 3.3, both systems are unsatisfiable, proving that a NaN can never be generated in line 7.

3.2 Constraint Solving over Floating-Point Variables

To solve constraint systems, we employ an approach called *interval-based consistency*, which amounts to iteratively narrowing the floating-point intervals associated with each variable in a process called *filtering* [10, 11]. In the literature on constraint propagation, the unary constraints associated with variables, e.g., intervals, are also called *labels* [24]. A *projection* $\text{proj}(x_i, C, I_1, \ldots, I_n)$, is a function that, given a constraint C with n variables $x_1, \ldots x_n$ and the intervals $I_1, \ldots I_n$ associated with them, computes a possibly refined interval I_i' for one of the variables x_i, that is tighter

Algorithm 1 Constraint Propagation

Require: Constraint store S, Variables $x_1, \ldots, x_n$, Intervals $I_1, \ldots I_n$, $i_{\max} \in \mathbb{N}$.
Ensure: Refined Intervals $I'_1 \subseteq I_1, \ldots, I'_n \subseteq I_n$.
1: $I'_1 := I_1; \ldots; I'_n := I_n; Q := S; i := 0;$
2: **while** $Q \neq \emptyset \wedge i < i_{\max}$ **do**
3: $\quad \text{proj}(x_i, C) := \text{dequeue}(Q);$
4: $\quad I''_i := \text{proj}(x_i, C, I_{i_1}, \ldots, I_{i_k});$
5: $\quad$ **if** $I''_i = \emptyset$ **then break**
6: $\quad$ **else if** $I''_i \neq I'_i$ **then**
7: $\quad\quad I'_i := I''_i;$
8: $\quad\quad \forall C' \in S, x_i \in C' : \forall x_j \in C' : \text{enqueue}(Q, \text{proj}(x_j, C'))$
9: $\quad$ **end if**
10: **end while**

than or equal to the original interval I_i associated with that variable. Ternary constraints of the form `x = y ∘ z`, where ∘ is one of `+`, `-`, `*`, `/`, and `x`, `y`, `z` are three program variables, result in the *direct projection* $\text{proj}(\mathtt{x}, \mathtt{x} = \mathtt{y} \circ \mathtt{z}, I_\mathtt{x}, I_\mathtt{y}, I_\mathtt{z})$, and the *indirect projections* $\text{proj}(\mathtt{y}, \mathtt{x} = \mathtt{y} \circ \mathtt{z}, I_\mathtt{x}, I_\mathtt{y}, I_\mathtt{z})$, and $\text{proj}(\mathtt{z}, \mathtt{x} = \mathtt{y} \circ \mathtt{z}, I_\mathtt{x}, I_\mathtt{y}, I_\mathtt{z})$. Given a constraint `x = f(y)`, where `f` is any unary `math.h`/`cmath` library function, we get the direct projection $\text{proj}(\mathtt{x}, \mathtt{x} = \mathtt{f(y)}, I_\mathtt{x}, I_\mathtt{y})$ and the indirect projection $\text{proj}(\mathtt{y}, \mathtt{x} = \mathtt{f(y)}, I_\mathtt{x}, I_\mathtt{y})$. Binary constraints of the form `x = (type) y`, where `type` is either `float` or `double`, and binary relations found in Boolean expressions, such as `==`, `!=`, `<`, `<=`, `>`, `>=`, result in similar projections. For example, considering `z5 = z2 / z4`, the projection $\text{proj}(\mathtt{z5}, \mathtt{z5} = \mathtt{z2/z4}, I_\mathtt{z5}, I_\mathtt{z2}, I_\mathtt{z4})$ over `z5` is called direct projection (it goes in the same sense of the TAC assignment it comes from) while the projections $\text{proj}(\mathtt{z2}, \mathtt{z5} = \mathtt{z2/z4}, I_\mathtt{z5}, I_\mathtt{z2}, I_\mathtt{z4})$, and $\text{proj}(\mathtt{z4}, \mathtt{z5} = \mathtt{z2/z4}, I_\mathtt{z5}, I_\mathtt{z2}, I_\mathtt{z4})$ over `z2` and `z4` are called indirect projections.

In constraint propagation, both direct and indirect projections are repeatedly applied in order to refine the intervals associated with the variables. We define a *propagator* as the actual implementation of a projection, that possibly refines an interval. For example, a propagator for the projection $\text{proj}(\mathtt{z5}, \mathtt{z5} = \mathtt{z2/z4}, I_\mathtt{z5}, I_\mathtt{z2}, I_\mathtt{z4})$, with $I_\mathtt{z2} = [\mathtt{z2}_l, \mathtt{z2}_u]$ and $I_\mathtt{z4} = [\mathtt{z4}_l, \mathtt{z4}_u]$ could be

$$I'_\mathtt{z5} = [\min(\mathtt{z2}_l/\mathtt{z4}_l, \mathtt{z2}_l/\mathtt{z4}_u, \mathtt{z2}_u/\mathtt{z4}_l, \mathtt{z2}_u/\mathtt{z4}_u), \max(\mathtt{z2}_l/\mathtt{z4}_l, \mathtt{z2}_l/\mathtt{z4}_u, \mathtt{z2}_u/\mathtt{z4}_l, \mathtt{z2}_u/\mathtt{z4}_u)] \cap I_\mathtt{z5}.$$

Propagators for basic floating-point operations are already present in the literature [3, 5, 13], and are out of the scope of this paper. Moreover, integer variables can also be treated with this approach, by employing appropriate propagators.

The application of projections by executing the corresponding propagators is governed by heuristic algorithms that go beyond the scope of this paper. For our purposes, it suffices to show Algorithm 1. Whenever the interval associated to a variable is refined, all the propagators are inserted into a data structure Q (in our case a FIFO queue) of propagators that are ready to run (line 1). Heuristics are used to select and remove from the data structure one of the ready propagators (line 3), which is then run. If that results in the refinement of the interval of one variable, all the propagators that depend on that variable are inserted into the same data structure, unless they are already present (line 8). This process continues until one of the intervals becomes empty (line 5), in which case propagation can be stopped as unsatisfiability of the given system of constraints has been proved, or the data structure becomes empty ($Q = \emptyset$ at line 2), i.e., propagation has reached *quiescence* as no projection is able to infer more information, or propagation is artificially stopped, e.g., because a timeout has expired ($i = i_{\max}$ at line 2).

When the constraint system reaches quiescence, the *labeling* procedure comes into play: a variable is chosen and the corresponding interval, its label, is divided into two or more parts and each part is searched independently. The way this partition is made is determined by heuristics that go beyond the scope of this paper, although we will return on this topic in Section 9.5. For each one of such parts, a new "child" propagation process is started, in which the interval for the chosen variable is instantiated to such part. Once all children processes reach a new quiescent state, labeling is performed again and this procedure repeated. This procedure stops when either one of the children propagation processes reaches quiescence with a singleton interval, in which case the singleton value can be used as a test-case or counterexample, or an interval becomes empty in all children processes, in which case the system is unsatisfiable.

3.3 Examples of Constraint Solving

Let us see how this can be used for program verification. As a first example, let us consider the question of whether the division `z5 = z2 / z4` can give rise to a division by zero. Assume all the intervals are initially full, i.e., they contain all possible numerical floating-point values and all propagators are ready to run. We modify the interval associated to `z4` to $[-0, +0]$ and start propagation. At some stage the indirect propagator for `cosh` will be called to possibly refine the interval for `z3` starting from the interval of `z4`: a propagator correctly capturing a passable implementation of `cosh` will refine the label of `z3` to the empty interval, thus proving that division by zero is indeed not possible. As we will see, all the implementations of `cosh()` we have examined are far from perfect, but none of them has a zero in its range.

As another example, let us consider `z4 = cosh(z3)`, and suppose the intervals associated to `z3` and `z4` are $[1, +\infty]$ and $[-\infty, +\infty]$, respectively. The direct projection for `cosh`, described in Section 5.1, would compute, on a machine we will later call `xps`,[8] the refining interval $[18B07551D9F550_{16} \cdot 2^{-52}, +\infty]$ for `z4`, where $18B07551D9F550_{16} \cdot 2^{-52} \approx 1.543$. Now suppose we want to determine for which values of `z3` the computation of `z4 = cosh(z3)` results in an overflow, thereby binding `z4` to $+\infty$. To answer this question we artificially refine the interval of `z4` to the singleton $[+\infty, +\infty]$ and let the indirect propagator for `cosh`, described in Section 5.2, do its job: this will result in the refining interval $[1633CE8FB9F87E_{16} \cdot 2^{-43}, +\infty]$ for `z3`, where $1633CE8FB9F87E_{16} \cdot 2^{-43} \approx 710.5$.

Coming back to the listing in SSA form at the beginning of this section, suppose now we want to know whether a NaN can be generated by the invocation to `asin` in line 7, i.e., whether we can have $z5 < -1$ or $z5 > 1$. Let us concentrate on the latter constraint, which we impose together with the constraints saying that `dl` and `ll` are neither NaNs nor infinities. All the other variables can take any value. We indicate with d_ℓ and i_ℓ the direct and the indirect projections for the constraint at line ℓ, respectively. The related propagators are either those described in Section 5, or in [3, 5]. Here is what happens on a selected constraint propagation process on machine `xps`, where the numbers have been rounded for increased readability:

$$
\begin{aligned}
&\xrightarrow{z5>1} z5 \in [1.0000000000000002, 1.798 \cdot 10^{308}] \\
&\xrightarrow{i_6} z4 \in [-1.798 \cdot 10^{308}, 1.798 \cdot 10^{308}] \\
&\xrightarrow{d_5} z4 \in [1, 1.798 \cdot 10^{308}] \\
&\xrightarrow{i_4} z3 \in [-710.5, 710.5] \\
&\xrightarrow{d_3} z2 \in [-1, 1]
\end{aligned}
$$

[8] On `xps`, `float` and `double` are 32-bit and 64-bit IEEE 754 floating point numbers, respectively; see Table 4 for more details.

$$\xrightarrow{\mathrm{d}_6} \mathtt{z5} \in \varnothing = [-1, 1] \cap [1.0000000000000002, 1.798 \cdot 10^{308}]$$

As the last constraint is unsatisfiable, the original constraint system is unsatisfiable. The same happens if $\mathtt{z5} < -1$ is imposed, thereby proving that NaNs cannot be generated on line 7.

As a final example, in order to show the indirect projections for `asin` and `sin` at work, we consider a partial constraint propagation starting from state

$$\begin{aligned} &\mathtt{z1} \in [-16, 16], && \mathtt{z5} \in [-1, 1], \\ &\mathtt{z2} \in [-1, 1], && \mathtt{z6} \in [-1.571, 1.571], \\ &\mathtt{z4} \in [1, 1.798 \cdot 10^{308}], && \mathtt{phi\_} \in [+0, 1.571]. \end{aligned}$$

A possible sequence of propagation steps is the following:

$$\begin{aligned} &\xrightarrow{\mathrm{i}_8} \mathtt{z6} \in [+0, 1.571] \\ &\xrightarrow{\mathrm{i}_7} \mathtt{z5} \in [-2^{-1074}, 1] \\ &\xrightarrow{\mathrm{i}_6} \mathtt{z2} \in [-1.332 \cdot 10^{-15}, 1], \\ &\xrightarrow{\mathrm{i}_3} \mathtt{z1} \in [-16, 15.71]. \end{aligned}$$

The constraint system has reached quiescence and labeling starts: after 7 labeling steps, a test-case is generated that falls off line 8 without generating NaN or infinities. This test-case is very simple:

$$\begin{aligned} &\mathtt{dl} = +0, && \mathtt{ll} = +0, && \mathtt{z1} = +0, && \mathtt{z2} = +0, \\ &\mathtt{z3} = +0, && \mathtt{z4} = 1, && \mathtt{z5} = +0 && \mathtt{z6} = +0, \\ &\mathtt{phi\_} = +0. \end{aligned}$$

3.4 Integration into Abstract Interpreters

In this section, we assume familiarity with Abstract Interpretation [19, 20], a static analysis technique that enables verification of program properties by soundly approximating their semantics. We consider the concrete domain $C = \wp(\mathbb{F})$, where $\mathbb{F}$ is any IEEE 754 floating-point format, and $\wp$ denotes the power-set operation, and the abstract domain $A = \mathcal{I}_{\mathbb{F}} \times B$, where $\mathcal{I}_{\mathbb{F}}$ is the set of intervals with boundaries on $\mathbb{F}$, and B is a Boolean domain, that captures the possibility that a value is NaN. In particular, the Boolean domain of an abstract value is `true` if it *may be NaN*, and `false` if it *cannot be NaN*. The *concretization function* $\gamma\colon A \to C$ is defined as

$$\gamma\Big(\big([x_l, x_u], b\big)\Big) = \{\, x \in \mathbb{F} \mid x_l \preccurlyeq x \preccurlyeq x_u \,\} \cup \big\{\, \mathrm{nan}(p) \;\big|\; b = \mathtt{true}, p \text{ is the NaN payload} \,\big\}.$$

We deal with `math.h`/`cmath` functions of the form $f\colon \mathbb{F} \to \mathbb{F}$, so we need abstract functions of the form $f^{\#}\colon A \to A$ to perform abstract interpretation, with the *correctness condition*

$$\forall a \in A : f^{\flat}\big(\gamma(a)\big) \subseteq \gamma\big(f^{\#}(a)\big),$$

where $f^{\flat}\colon \wp(\mathbb{F}) \to \wp(\mathbb{F})$ is the trivial extension of f to subsets of $\mathbb{F}$. The correctness condition comprises two parts: the numeric-symbolic part on $\mathcal{I}_{\mathbb{F}}$, and the NaN part on B. The NaN part is simple: if f may return NaN on any element of $\gamma(a)$, $a \in A$, then the Boolean part of $f^{\#}(a)$ must be `true`. For all the functions we treat, the implication also holds in the other direction, since the POSIX standard specifically defines for which values any function may return a NaN. For the numeric-symbolic part, we must ensure that, if $f^{\#}\big([x_l, x_u], b_x\big) = \big([y_l, y_u], b_y\big)$, then $\forall x \in [x_l, x_u] : f(x) = \mathrm{NaN} \vee f(x) \in [y_l, y_u]$. Our contribution consists in determining $f^{\#}$ on actual implementations of f, and studying the conditions under which we can guarantee soundness and precision of the

approximation. The direct projections we describe in Section 5 can be immediately used, verbatim, as $f^{\#}$ in *forward analysis* (from the initial states to the target states). The inverse projections can also be immediately used in *backward analysis* (from the target states, e.g., erroneous states, back to the initial states).[9] For all floating-point arithmetic operations, projections that can be used as their abstract versions are already present in the literature [3, 13].

4 (QUASI-) MONOTONICITY AND GLITCHES

A real-valued partial function $\hat{f}\colon \mathbb{R} \rightarrowtail \mathbb{R}$ is called *monotonic* if it is order preserving (i.e., $\hat{f}(x) \leq \hat{f}(y)$ whenever $x \leq y$ and both $\hat{f}(x)$ and $\hat{f}(y)$ are defined) in which case we call it *isotonic*, or if it is order reversing (i.e., $\hat{f}(x) \geq \hat{f}(y)$ whenever $x \leq y$ and both $\hat{f}(x)$ and $\hat{f}(y)$ are defined) in which case we say $\hat{f}$ is *antitonic*.

Definition 4.1. **(Quasi-monotonicity.)** Let $I \subseteq \mathbb{F}$ be a floating-point interval and $f\colon \mathbb{F} \to \mathbb{F}$ be a floating-point function meant to approximate a real-valued partial function $\hat{f}\colon \mathbb{R} \rightarrowtail \mathbb{R}$. We say that f *is quasi-monotonic/quasi-isotonic/quasi-antitonic on* I if $\hat{f}$ is always defined and monotonic/isotonic/antitonic on I.

Let $f\colon \mathbb{F} \to \mathbb{F}$ be a quasi-monotonic function. The best we can hope for is that f be monotonic over $(\mathbb{F}, \preccurlyeq)$ for all rounding modes. While this is *often* the case, it is not *always* the case: monotonicity is occasionally violated at spots we call *monotonicity glitches*.

Definition 4.2. **(Monotonicity glitches.)** Let $f\colon \mathbb{F} \to \mathbb{F}$ be a quasi-isotonic function on $I \subseteq \mathbb{F}$. An *isotonicity glitch of* f *in* I is an interval $[l, u] \subseteq I$ such that:

$$u \succ \mathrm{succ}(l) \quad \wedge \quad \forall x \in (l, u) : f(l) \succ f(x) \quad \wedge \quad f(l) \preccurlyeq f(u).$$

If f is quasi-antitonic, an *antitonicity glitch of* f *in* I is an isotonicity glitch of $-f$ in I. Isotonicity and antitonicity glitches are collectively called *monotonicity glitches* or, simply, *glitches*.

Let $G = [l, u]$ be a monotonicity glitch of f in I. The *width* and the *depth* of G are given, respectively, by

$$\begin{aligned} \mathrm{width}(G) &:= \#[l, u] - 1, \\ \mathrm{depth}(G) &:= \#\big[m, f(u)\big] - 1, \qquad\qquad \text{where } m = \min_{x \in (l,u)} f(x). \end{aligned}$$

Note that, for each glitch G, we have $\mathrm{width}(G) \geq 2$ and $\mathrm{depth}(G) \geq 1$. A glitch of f in I is called *maximal* if none of its proper supersets is a glitch of f in I. Non-maximal glitches are also called *sub-glitches*.

See Figure 2 for an exemplification of these concepts: G_1 is a maximal glitch; G_2, being contained into G_1 is non-maximal; $\mathrm{width}(G_1) = 5$, $\mathrm{depth}(G_1) = 4$, $\mathrm{width}(G_2) = \mathrm{depth}(G_2) = 2$.

We gathered the relevant statistics about glitches for 25 functions provided by `libm` on several implementations. We report in Table 1 the data for a machine we call `xps`, which features an x86_64 CPU and runs Ubuntu 19.10, with GCC 9.2.1, and `libm` is provided by GNU libc 2.30. The data for other platforms are reported in Appendix A. Table 1 presents, for each function, its name and the minimum and maximum of the considered domain interval. For most of the functions, this interval is the natural one. The exceptions are the following: for `lgammaf` we start at 2, which is where monotonicity theoretically begins; for `tgammaf` we also start at 2 because the considered implementations are neither monotonic nor periodic for arguments less than 2;[10] for

[9] Forward and backward analysis are usually alternated in abstract-interpretation-based static analyses.

[10] The real Γ function is strictly increasing in the interval $[\mu, +\infty)$ for $1 < \mu < 2$.

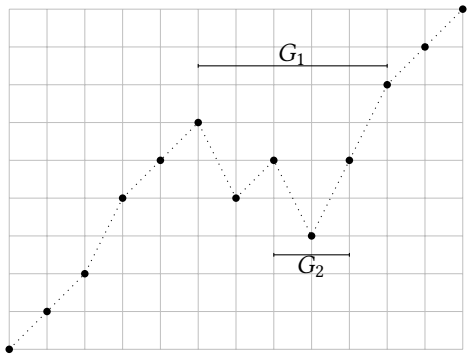

Fig. 2. An example of monotonicity glitches

Table 1. Glitch data for the xps machine

function	$D_{\min}$	D_{M}	near			up			down			zero		
			n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}
acosf	-1	1												
acoshf	1	∞							1	1	2	1	1	2
asinf	-1	1												
asinhf	$-\infty$	∞							2	1	2	2	1	2
atanf	$-\infty$	∞				1	1	10^8						
atanhf	-1	1							2	1	2	2	1	2
cbrtf	$-\infty$	∞	10^6	1	2	10^6	1	2	10^6	1	2	10^6	1	2
coshf	$-\infty$	∞	454	1	2	466	1	2	442	1	2	448	1	2
erff	$-\infty$	∞												
expf	$-\infty$	∞				1	1	10^9						
exp10f	$-\infty$	∞												
exp2f	$-\infty$	∞				1	1	10^9						
expm1f	$-\infty$	∞												
lgammaf	2	∞	168	1	2	168	1	2	169	1	2	169	1	2
logf	0	∞												
log10f	0	∞												
log1pf	-1	∞							1	1	2	1	1	2
log2f	0	∞												
sinhf	$-\infty$	∞												
sqrtf	0	∞												
tanhf	$-\infty$	∞				1	1	2	2	1	3			
tgammaf	2	∞	10^5	4	3	10^5	4	3	10^5	4	3	10^5	4	3
cosf	-2^{23}	2^{23}												
sinf	-2^{23}	2^{23}												
tanf	-2^{23}	2^{23}												

the trigonometric functions, at the bottom of the tables, we restrict the domain to a region where there are at least 12 floats per period, for reasons that will be discussed in Section 6.1.

For each function and each rounding mode (near, up, down, zero), Table 1 gives the number of glitches, n_{g}, their maximum depth, d_{M}, and their maximum width, w_{M}. For the trigonometric functions we report the cumulative results concerning all the quasi-isotonic and quasi-antitonic branches in the given range. In the columns labeled n_{g}, we report for these functions the maximum number of glitches in any such quasi-monotonic branch (we will refer to this quantity in Section 6.2 as n_{gM}). All the data above were collected by exhaustively testing the functions in their domains, which is computationally feasible on single-precision floating-point numbers (cf. Section 9.2 for more details).

The following observations can be made:

(1) there are few glitches: many functions have no glitch at all, several functions have just a few glitches, a few functions have many glitches;
(2) most glitches are very shallow;
(3) with a notable exception, glitches are also very narrow.

It is important to observe that glitches are not simply bugs that will surely be fixed at the next release. For instance, the implementation of `tgammaf` in Ubuntu 19.10/x86_64 has more numerous and deeper glitches than the one in Ubuntu 14.04/x86_64. Moreover, the implementation of `cosf()` in Ubuntu 18.04/x86_64 contains one glitch in all rounding modes that was not present in previous versions. The point is that monotonicity is not one of the objectives of most implementations of `math.h`/`cmath` functions. For instance, both the manual [51] and the FAQ of GNU libc explicitly exclude monotonicity from the accuracy goals of the library, so that bug reports about violated monotonicity are closed as invalid.[11]

We will now see how quasi-monotonicity can be exploited for the purposes of interval refinement and, in turn, software verification. Afterwards, we will deal with the special case of the trigonometric functions, as they pose the additional problem of periodic slope inversions.

5 PROPAGATION ALGORITHMS

Let $\mathbb{F}$ be any IEEE 754 floating-point format and let $\mathcal{S} \subseteq \wp(\mathbb{F})$ be a bounded meet-semilattice. A floating-point unary constraint over $\mathcal{S}$ is a formula of the form $\mathbf{x} \in S$ for $S \in \mathcal{S}$.

Let $f : \mathbb{F} \to \mathbb{F}$ be a function and consider a constraint of the form $\mathbf{y} = f(\mathbf{x})$ along with the unary constraints $\mathbf{x} \in S_x$ and $\mathbf{y} \in S_y$ with $S_x, S_y \in \mathcal{S}$.

Direct propagation amounts to computing a possibly refined set $S'_y \in \mathcal{S}$, such that

$$S'_y \subseteq S_y \land \forall x \in S_x : f(x) \in S_y \implies f(x) \in S'_y. \tag{1}$$

Of course this is always possible by taking $S'_y = S_y$, but the objective of the game is to compute a "small" (possibly the smallest) S'_y satisfying (1), compatible with the available information on f and computing resources. The smallest $S'_y \in \mathcal{S}$ that satisfies (1) is such that

$$\forall S''_y \in \mathcal{S} : S''_y \subset S'_y \implies \exists x \in S_x \,.\, f(x) \in S_y \setminus S''_y. \tag{2}$$

Indirect propagation for the same constraints, $\mathbf{y} = f(\mathbf{x})$, $\mathbf{x} \in S_x$ and $\mathbf{y} \in S_y$, is the computation of a possibly refined set for $\mathbf{x}$, S'_x, such that

$$S'_x \subseteq S_x \land \forall x \in S_x : f(x) \in S_y \implies x \in S'_x.$$

Again, taking $S'_x = S_x$ is always possible and sometimes unavoidable. The best we can hope for is to be able to determine the smallest such set, i.e., satisfying

$$\forall S''_x \in \mathcal{S} : S''_x \subset S'_x \implies \exists x \in S_x \setminus S''_x \,.\, f(x) \in S_y. \tag{3}$$

[11] See, e.g., bug reports https://sourceware.org/bugzilla/show_bug.cgi?id=15898 and https://sourceware.org/bugzilla/show_bug.cgi?id=15899, last accessed on July 16th, 2020.

Satisfying predicates (2) and (3) corresponds to enforcing and obtaining *domain consistency* [66] on our constraint set. This goal is often difficult to reach, especially if the underlying variable domains are large. A less demanding approach is to seek *interval consistency*: we associate an interval $[x_l, x_u]$ to variable $\mathbf{x}$ and an interval $[y_l, y_u]$ to $\mathbf{y}$, and we try to obtain new intervals whose bounds satisfy $\mathbf{y} = f(\mathbf{x})$.

If f is isotonic, direct propagation can be reduced to finding a new interval $[y'_l, y'_u]$ such that

$$y'_l \geqslant y_l \wedge \forall x \in [x_l, x_u] : f(x) \geqslant y_l \implies f(x) \geqslant y'_l,$$
$$y'_u \leqslant y_u \wedge \forall x \in [x_l, x_u] : f(x) \leqslant y_l \implies f(x) \leqslant y'_u.$$

Taking $y'_l = y_l$, $y'_u = y_u$ trivially satisfies these predicates, but we aim to find an interval such that

$$\forall y''_l \in \mathbb{F} : y''_l \succ y'_l \implies \exists x \in [x_l, x_u] \,.\, y_l \preccurlyeq f(x) \prec y''_l,$$
$$\forall y''_u \in \mathbb{F} : y''_u \prec y'_u \implies \exists x \in [x_l, x_u] .\, y''_u \prec f(x) \preccurlyeq y_u.$$

Indirect propagation consists now in finding an interval $[x'_l, x'_u]$ such that

$$x'_l \geqslant x_l \wedge \forall x \in [x_l, x_u] : f(x) \geqslant y_l \implies x \geqslant x'_l,$$
$$x'_u \leqslant x_u \wedge \forall x \in [x_l, x_u] : f(x) \leqslant y_u \implies x \leqslant x'_u.$$

An optimal result would satisfy

$$\forall x''_l \in \mathbb{F} : x''_l \succ x'_l \implies \exists x \in [x_l, x''_l) \,.\, f(x) \geqslant y_l,$$
$$\forall x''_u \in \mathbb{F} : x''_u \prec x'_u \implies \exists x \in (x''_u, x_u] .\, f(x) \leqslant y_u.$$

A possible compromise between domain and interval consistency is the use of multi-intervals. It achieves further granularity by splitting domains into multiple intervals. The predicates given above can be easily extended to "multi-interval consistency."

Unfortunately, the functions we are concerned with are neither isotonic nor antitonic, because of glitches. Yet, we devised algorithms that, given the implementation of a quasi-monotonic library function $f : \mathbb{F} \to \mathbb{F}$, an interval $[x_l, x_u]$ for $\mathbf{x}$ and an interval $[y_l, y_u]$ for $\mathbf{y}$, compute refined bounds for both intervals, satisfying the correctness predicates defined above and, in some cases, even optimality predicates. These algorithms exploit simple data describing the glitches of a specific function to overcome the issues generated by its quasi-monotonicity. Such data consist in safe approximations n_{g}, d_{M} and w_{M} of, respectively, the total number of glitches n_{g}^f, their maximal depth d_{M}^f and width w_{M}^f. Moreover, a safe approximation α of where the first glitch starts, α^f, and a safe approximation ω of where the last glitch ends inside the function's domain, ω^f, are needed.

If the values of such data are conservative, then the refined intervals computed by our algorithms contain all solutions to the constraint $\mathbf{y} = f(\mathbf{x})$. We call this property *correctness*. If all projections involved in constraint solving are also correct, then no solution is mistakenly eliminated, and the process yields no false negatives.

In general, the refined intervals may also contain values that are not solutions to the constraint, which could potentially lead to false positives in the verification process. This is avoided at the constraint-solving level by the labeling process, which splits variable domains until they become singletons. All projections are made so that they always discard singletons iff they do not contain a solution. For mathematical functions, this is done by the direct projection. Thus, if all projections have this property, the constraint solving process never yields false positives. This has, however, the drawback that the labeling process may lead to the enumeration of all values in the variable domains, if the projections fail to further refine them. If such domains are large, this likely results in a time-out. Thus, as we shall see in Section 7, the verification process yields no false positives or

negatives, but *don't knows* when it times out. Projections that fail to satisfy such requirements can, in fact, lead to false positives or negatives (cf. Section 8).

5.1 Direct Propagation

Given an interval $[x_l, x_u]$ for $\mathbf{x}$ and a function f, finding a refined interval $[y'_l, y'_u]$ for $\mathbf{y}$ satisfying constraint $\mathbf{y} = f(\mathbf{x})$ is trivial if f is monotonic: computing $[y'_l, y'_u] \equiv \big[f(x_l), f(x_u)\big]$ suffices. However, the presence of glitches in quasi-monotonic functions raises two main issues:

- there may be glitches in $[x_l, x_u]$ in which the value of the function is lower than $f(x_l)$;
- x_u may be inside a glitch, and there may be values of x outside it such that $f(x) \succ f(x_u)$.

If $[x_l, x_u]$ and $[\alpha, \omega]$ do not intersect, f can be treated as if it was monotonic. Otherwise, we exploit the information about the glitches of f to tackle these issues.

Lower bound y'_l : if $x_l \in [\alpha, \omega]$, the worst-case scenario is that there is a glitch starting right after x_l, where the graph of f goes lower than $f(x_l)$. Such a glitch cannot be deeper than $\mathrm{pred}^{d_{\mathrm{M}}}(f(x_l))$: we take this value as y'_l, the lower bound of the refined interval. If x_l is not in the "glitch-area," then we consider the value of $f(\alpha)$: no glitch in $[\alpha, x_u]$ can go lower than $\mathrm{pred}^{d_{\mathrm{M}}}(f(\alpha))$. In this case, we set $y'_l = \min\big\{f(x_l), \mathrm{pred}^{d_{\mathrm{M}}}(f(\alpha))\big\}$.

Upper bound y'_u : if $x_u \notin [\alpha, \omega]$, then it cannot be in a glitch, and $y'_u = f(x_u)$. Otherwise, it may be in a glitch, which cannot be deeper than d_{M}: the actual maximum value of the function, outside of the glitch, cannot be higher than $\mathrm{succ}^{d_{\mathrm{M}}}(f(x_u))$. We set y'_u to this value.

If the actual range of f in its whole domain is known, y'_l and y'_u can be compared with it to make sure they do not fall outside.

5.2 Indirect Propagation

Assume function f is quasi-isotonic. Indirect propagation, i.e., the process of inferring a new interval $[x'_l, x'_u] \subseteq [x_l, x_u]$ for $\mathbf{x}$ starting from the interval $[y_l, y_u]$ for $\mathbf{y}$, is carried out by separately looking for a lower bound for the values of x satisfying $y_l = f(\mathbf{x})$, and an upper bound for the values of x satisfying $y_u = f(\mathbf{x})$. We use such bounds to refine correctly interval $[x_l, x_u]$ into $[x'_l, x'_u]$. We designed two different algorithms, `lower_bound` and `upper_bound`, that carry out such tasks for the equation $y = f(\mathbf{x})$, where y is a given single value of $\mathbf{y}$. They extend the well known dichotomic search method to quasi-isotonic functions. For brevity, we describe in detail algorithm `lower_bound` in the next section, leaving `upper_bound`, which is symmetric, to Appendix B.

Function `lower_bound` (Algorithm 2) returns a value l satisfying one of the following predicates:

$$
\begin{aligned}
p_0(y, x_l, x_u, l) &\equiv \forall x \in [x_l, x_u] : y \succ f(x),\\
p_1(y, x_l, x_u, l) &\equiv \forall x \in [x_l, l] : y \prec f(x),\\
p_2(y, x_l, x_u, l) &\equiv \forall x \in [x_l, l] : y \succ f(x),\\
p_3(y, x_l, x_u, l) &\equiv f(l) \prec y \prec f(\mathrm{succ}(l)) \wedge \forall x \in [x_l, l) : y \succ f(x),\\
p_4(y, x_l, x_u, l) &\equiv y = f(l) \wedge \forall x \in [x_l, l) : y \succ f(x).
\end{aligned}
$$

Such predicates express properties on whether the new bound for $\mathbf{x}$ satisfies equation $y = f(\mathbf{x})$. When condition $p_0(y, x_l, x_u, l)$ holds, $y = f(\mathbf{x})$ has no solution over $[x_l, x_u]$. Also if $p_1(y, x_l, x_u, l)$ holds with $l = x_u$, $y = f(\mathbf{x})$ has no solution there. When $p_1(y, x_l, x_u, l)$ holds with $l \prec x_u$ or $p_2(y, x_l, x_u, l)$ holds, $y = f(\mathbf{x})$ may have a solution and choosing $x'_l = \mathrm{succ}(l)$ gives a correct refinement for x_l. When $p_3(y, x_l, x_u, l)$ holds, we identified the leftmost point in $[x_l, x_u]$ where f crosses y without touching it and we can set $x'_l = \mathrm{succ}(l)$. Finally, when $p_4(y, x_l, x_u, l)$ holds, we identified the leftmost solution $x = l$ of $y = f(\mathbf{x})$ and we can set $x'_l = l$.

Function upper_bound returns a value u that satisfies one of the predicates below:

$$p_5(y, x_l, x_u, u) \equiv \forall x \in [x_l, x_u] : y \prec f(x),$$
$$p_6(y, x_l, x_u, u) \equiv \forall x \in [u, x_u] : y \succ f(x),$$
$$p_7(y, x_l, x_u, u) \equiv \forall x \in [u, x_u] : y \prec f(x),$$
$$p_8(y, x_l, x_u, u) \equiv f(\text{pred}(u)) \prec y \prec f(u) \wedge \forall x \in (u, x_u] : y \prec f(x),$$
$$p_9(y, x_l, x_u, u) \equiv y = f(u) \wedge \forall x \in (u, x_u] : y \prec f(x).$$

These predicates are the counterparts of p_0-p_4 for upper_bound. If $p_5(y, x_l, x_u, u)$ or $p_6(y, x_l, x_u, u)$ hold, the latter only with $u = x_l$, then $y = f(\mathbf{x})$ has no solution in $[x_l, x_u]$. If $p_6(y, x_l, x_u, u)$ holds with $u \prec x_u$, or $p_7(y, x_l, x_u, u)$ holds with any $u \in [x_l, x_u]$, then $y = f(\mathbf{x})$ has no solution in interval $[u, x_u]$, but it might have a solution somewhere in $[x_l, u)$. So, setting $x'_u = \text{pred}(u)$ is correct. If $p_8(y, x_l, x_u, u)$ holds, then we identified the rightmost point in $[x_l, x_u]$ where the graph of f crosses y without touching it. Setting $x'_u = \text{pred}(u)$ is correct. Finally, when $p_9(y, x_l, x_u, u)$ holds, we found the rightmost solution $x = u$ of equation $y = f(\mathbf{x})$, and setting $x'_u = u$ is correct.

When function f is quasi-isotonic, the results of invoking lower_bound on y_l and upper_bound on y_u are combined to refine the interval $[x_l, x_u]$ into $[x'_l, x'_u]$, as follows.

- If p_0 or p_5 hold, then there is no solution, because the entire graph of the function is either below or above the interval for $\mathbf{y}$. Note that p_0 implies p_6 and p_5 implies p_1.
- If p_1 holds, we set $x'_l = x_l$ even if $l \succ x_l$. In fact, although the graph of the function is entirely above y_l, there might be a value y'_l such that $y_l \prec y'_l \preccurlyeq y_u$ that satisfies $y'_l = f(\mathbf{x})$ for some $x \in [x_l, l]$, so $[x_l, l]$ cannot be excluded. However, if $\mathbf{y}$ is a singleton, or if p_5 holds, then there are no solutions. The upper bound is treated similarly: unless $\mathbf{y}$ is a singleton or p_0 holds, x'_u must be set to x_u if p_6 holds.
- If p_3 and p_8 both hold and $l \succ u$, then there is no solution for $\mathbf{y} = f(\mathbf{x})$.
- For all other predicates, x'_l and x'_u can be set as stated below the definitions of the predicates.

The same algorithms are used for the quasi-antitonic functions, because if f is quasi-antitonic, then $-f$ is quasi-isotonic. They are called with $f' = -f$, $f^{\text{i}\prime} = f \circ (-\text{id})$, and $-y_u$ instead of y_l for lower_bound, and $-y_l$ in place of y_u for upper_bound. When they terminate, $p_i(-y_u, x_l, x_u, l)$ and $p_j(-y_l, x_l, x_u, u)$, $0 \le i \le 4$ and $5 \le j \le 9$, hold for $-f$. Since $-y \prec -f(x) \iff y \succ f(x)$ and $-y \succ -f(x) \iff y \prec f(x)$, they do not hold on f directly, but since y_l and y_u are switched, the same case analysis can be done to obtain x'_l and x'_u depending on the values of i and j.

5.2.1 Computation of the Lower Bound for $y = f(\mathbf{x})$. Given a value for y and equation $y = f(\mathbf{x})$, Algorithm 2 computes a correct lower bound refining the interval of $\mathbf{x}$. Its preconditions are listed in the **Require** statement and demand a quasi-isotonic function $f : \mathbb{F} \to \mathbb{F}$, a value $y \in \mathbb{F}$, and an interval $[x_l, x_u]$ for $\mathbf{x}$ to be refined. To be as precise as possible, the algorithm needs safe approximations of the glitch data listed previously in this section. It also uses an inverse function $f^{\text{i}} : \mathbb{F} \to \mathbb{F}$, if available. To avoid complexity issues, parameter $t \in \mathbb{N}$ fixes the maximum length of the linear searches the algorithm performs in some cases, and $s \in \mathbb{N}$ is the maximum number of times function logsearch_lb in Algorithm 4 can return an interval too wide to ensure the logarithmic complexity of the dichotomic search.

The algorithm ends guaranteeing the post-conditions in the **Ensure** statement, where predicates $p_r(y, x_l, x_u, l)$ for $r \in \{0, \ldots, 4\}$ are those described previously. The post-conditions are divided in two parts: the *correctness* part is preceded by (c) and the *precision* part by (p). The algorithm determines r and l by performing a number of calls to library function f bounded by a small constant k (e.g., $k = 3$), times the logarithm of $\#[x_l, x_u]$.

Algorithm 2 Indirect propagation: `lower_bound`$(f, y, [x_l, x_u], n_g, d_M, w_M, \alpha, \omega, f^i, s, t)$

Require: $f\colon \mathbb{F} \to \mathbb{F}$, $y \in \mathbb{F}$, $[x_l, x_u] \in \mathcal{I}_{\mathbb{F}}$, $n_g \geq n_g^f$, $d_M \geq d_M^f$, $w_M \geq w_M^f$, $\alpha \preccurlyeq \alpha^f$, $\omega \succcurlyeq \omega^f$, $n_g > 0 \implies (x_l \preccurlyeq \alpha \preccurlyeq \omega \preccurlyeq x_u)$, $f^i\colon \mathbb{F} \to \mathbb{F}$, $s, t \in \mathbb{N}$.

Ensure: (c) $l \in \mathbb{F}, r \in \{0,1,2,3,4\} \implies p_r(y, x_l, x_u, l)$

(p) $\Big(f(x_l) \preccurlyeq y \preccurlyeq f(x_u) \wedge \big(n_g = 0 \vee w_M < t \vee (n_g = 1 \wedge \alpha = \alpha^f)\big)\Big) \implies r \in \{3,4\}$

1: $i := \texttt{init}(y, [x_l, x_u], f^i)$; ▷ $x_l \preccurlyeq i \preccurlyeq x_u$
2: $(\mathrm{lo}, \mathrm{hi}) := \texttt{gallop\_lb}(f, y, [x_l, x_u], d_M, i)$;
3: ▷ $(x_l \preccurlyeq \mathrm{lo} \preccurlyeq \mathrm{hi} \preccurlyeq x_u) \wedge (x_l \prec \mathrm{lo} \implies \#[f(\mathrm{lo}), y) > d_M) \wedge (x_u \succ \mathrm{hi} \implies f(\mathrm{hi}) \succcurlyeq y)$
4: **if** $f(\mathrm{lo}) \succ y$ **then**
5: **if** $n_g = 0 \vee \#[y, f(\alpha)) > d_M$ **then**
6: $l = x_u$; $r := 1$; **return**
7: **else**
8: $l := \alpha$; $r := 1$; **return**
9: **end if**
10: **else if** $f(\mathrm{lo}) = y$ **then**
11: $l := \mathrm{lo}$; $r := 4$; **return**
12: **end if**;
13: **if** $f(\mathrm{hi}) \prec y$ **then**
14: $(r, l, \mathrm{hi}) := \texttt{findhi\_lb}(f, y, [x_l, x_u], n_g, d_M, w_M, \alpha, \omega, t)$;
15: **if** $r \in \{0, 2\}$ **then return**
16: **end if**
17: **end if**;
18: $\mathrm{lo} := \texttt{bisect\_lb}(f, y, [x_l, x_u], n_g, d_M, w_M, \alpha, \omega, s, t, \mathrm{lo}, \mathrm{hi})$;
19: **while** $f(\mathrm{succ}(\mathrm{lo})) \prec y \wedge t > 0$ **do**
20: $\mathrm{lo} := \mathrm{succ}(\mathrm{lo})$;
21: $t := t - 1$
22: **end while**;
23: **if** $f(\mathrm{succ}(\mathrm{lo})) \succ y$ **then** $l := \mathrm{lo}$; $r := 3$
24: **else if** $f(\mathrm{succ}(\mathrm{lo})) = y$ **then** $l := \mathrm{succ}(\mathrm{lo})$; $r := 4$
25: **else** $l := \mathrm{lo}$; $r := 2$
26: **end if**

The code in lines 1-17 sorts out all edge cases and chooses two values lo and hi suitable to start the dichotomic search, carried out by function `bisect_lb`. First, it calls function `init`, that takes a value y, interval $[x_l, x_u]$, and an inverse function $f^i\colon \mathbb{F} \to \mathbb{F}$, if available, and returns a point inside $[x_l, x_u]$. `init` returns $f^i(y)$ if $x_l \preccurlyeq f^i(y) \preccurlyeq x_u$, and the middle point between x_l and x_u, otherwise.

Next, function `gallop_lb` finds values lo and hi, satisfying the precondition of algorithm `bisect_lb`, i.e., so that $x_l \preccurlyeq \mathrm{lo} \preccurlyeq \mathrm{hi} \preccurlyeq x_u$, $x_l \prec \mathrm{lo} \implies \#[f(\mathrm{lo}), y) > d_M$ and $x_u \succ \mathrm{hi} \implies f(\mathrm{hi}) \succcurlyeq y$. `gallop_lb` starts with $\mathrm{hi} = i$ and increases it (e.g., by multiplying it by 2) until it finds a value such that $\mathrm{hi} \prec x_u$ and $f(\mathrm{hi}) \succcurlyeq y$. If no such value can be found, it sets $\mathrm{hi} = x_u$. Similarly, it finds a value lo such that $x_l \prec \mathrm{lo}$ and $\#[f(\mathrm{lo}), y) > d_M$, or it sets $\mathrm{lo} = x_l$.

The case in which $f(\mathrm{lo}) \succ y$ is then handled by lines 4-9. We need to determine if $[x_l, x_u]$ really does not contain any solution for y, i.e. if, for each $x \in [x_l, x_u]$, $y \succ f(x)$ holds. If glitches are too deep, a suboptimal value for l is returned, and the algorithm terminates. If $f(\mathrm{lo}) = y$ at line 10, the exact solution for the lower bound was found, and it is returned.

Algorithm 3 Indirect propagation: `findhi_lb`$(f, y, [x_l, x_u], n_g, d_M, w_M, \alpha, \omega, t)$

Require: $f\colon \mathbb{F} \to \mathbb{F}$, $y \in \mathbb{F}$, $[x_l, x_u] \in \mathcal{I}_{\mathbb{F}}$, $n_g \geq n_g^f$, $d_M \geq d_M^f$, $w_M \geq w_M^f$, $\alpha \preccurlyeq \alpha^f$, $\omega \succcurlyeq \omega^f$, $n_g > 0 \implies (x_l \preccurlyeq \alpha \preccurlyeq \omega \preccurlyeq x_u)$, $t \in \mathbb{N}$, $f(x_u) \prec y$.
Ensure: $l \in \mathbb{F}, r \in \{0, 2\} \implies p_r(y, x_l, x_u, l), r = 1 \implies \big(\mathrm{hi} \in [x_l, x_u] \wedge f(\mathrm{hi}) \succcurlyeq y\big)$.

```
1:  l = x_l;
2:  if n_g = 0 ∨ x_u ≻ ω ∨ #[f(x_u), y) > d_M then r := 0
3:  else if n_g = 1 ∧ (w_M > t ∨ f(succ(α)) ≺ f(α)) then
4:      if y ≻ f(α) then
5:          if f(succ(α)) ≺ f(α) then r := 0
6:          else l := α; r := 2
7:          end if
8:      else hi := α; r := 1
9:      end if
10: else
11:     (b, hi) := linsearch_geq(f, y, [x_l, x_u], w_M, t);
12:                     ▷ (b = 1 ∧ hi ∈ [x_l, x_u] ∧ f(hi) ≽ y) ∨ (b = 0 ∧ ∀x ∈ [x̂, x_u] : f(x) ≺ y)
13:                                 ▷ where x̂ = max{x_l, pred^v(x_u)} and v = min{t, w_M}
14:     if b = 1 then r := 1
15:     else if t ≥ w_M then r := 0
16:     else l := x_l; r := 2
17:     end if
18: end if
```

In line 14, function `findhi_lb` (Algorithm 3) handles the case where $\mathrm{hi} = x_u$ and $f(x_u) \prec y$. This may arise if either for no $x \in [x_l, x_u]$ we have $f(x) \succcurlyeq y$, or if there is a value $x' \in [x_l, x_u]$ such that $f(x') \succcurlyeq y$, but x_u is in a glitch. In the latter case, f is not monotonic in $[x_l, x_u]$, and $f(x_u)$ is not a safe upper bound to the value of f in it. `findhi_lb` discriminates quickly between these two cases and tries to find a value of hi suitable for `bisect_lb`. In this case, it returns $r = 1$.

- If x_u might be in a glitch wider than t (lines 3-9), for the sake of efficiency we do not perform an exhaustive search. By inspecting $f(\alpha)$, we may still be able to set $r = 0$, to signify that $\forall x \in [x_l, x_u] : f(x) \prec y$, or $r = 1$ and hi to α. If we do not have enough information to chose one of these options, we just set $r = 2$ and l to a valid (but suboptimal) lower bound.
- Otherwise, `linsearch_geq`$(f, y, [x_l, x_u], w_M, t)$ performs a backward, float-by-float search for no more than $\min(t, w_M)$ steps, starting from $\mathrm{hi} = x_u$, looking for the first value hi such that $f(\mathrm{hi}) \succcurlyeq y$. The search stops in two cases, discriminated by the value of variable b.
  $b = 0$: no value for hi was found within t search steps. If they were enough to cover the glitch, we set $r = 0$ since $\forall x \in [x_l, x_u] : f(x) \prec y$. Otherwise we set $r = 2$ and return x_l.
  $b = 1$: a value of hi appropriate for `bisect_lb` was found.

Line 16 of `lower_bound` is reached if `findhi_lb` returns $r = 1$, so $x_l \preccurlyeq \mathrm{hi} \preccurlyeq x_u$ and $f(\mathrm{hi}) \succcurlyeq y$.

Before the invocation of function `bisect_lb` (Algorithm 4) at line 18, we have $f(\mathrm{lo}) \prec y \preccurlyeq f(\mathrm{hi})$, so $\mathrm{lo} \neq \mathrm{hi}$. `bisect_lb` adapts the dichotomic method to refine interval [lo, hi] when f is a quasi-isotonic function. Each iteration of the **while** loop on line 1 uses function `split_point` to pick the middle point, mid, of interval [lo, hi], so that the cardinalities of [lo, mid] and [mid, hi] differ at most by 1. Then, $f(\mathrm{mid})$ is compared with y. If $f(\mathrm{mid}) \succcurlyeq y$, hi is updated with the value of mid. The critical case is when $f(\mathrm{mid}) \prec y$. Function `bisect_lb` further discriminates whether lo can be updated with the value of mid or other refinements of [lo, hi] are possible.

Algorithm 4 Indirect propagation: `bisect_lb`$(f, y, [x_l, x_u], n_g, d_M, w_M, \alpha, \omega, s, t, \mathrm{lo}, \mathrm{hi})$

Require: $x_l \preccurlyeq \mathrm{lo} \prec \mathrm{hi} \preccurlyeq x_u$. $f(\mathrm{lo}) \prec y \preccurlyeq f(\mathrm{hi})$, $\forall x \in [x_l, \mathrm{lo}] : f(x) \prec y$ $f\colon \mathbb{F} \to \mathbb{F}$, $y \in \mathbb{F}$, $[x_l, x_u] \in \mathcal{I}_{\mathbb{F}}$, $n_g \geq n_g^f$, $d_M \geq d_M^f$, $w_M \geq w_M^f$, $\alpha \preccurlyeq \alpha^f$, $\omega \succcurlyeq \omega^f$, $n_g > 0 \implies (x_l \preccurlyeq \alpha \preccurlyeq \omega \preccurlyeq x_u)$, $s, t \in \mathbb{N}$.

Ensure: (c) $x_l \preccurlyeq \mathrm{lo} \prec \mathrm{hi} \preccurlyeq x_u$, $f(\mathrm{lo}) \prec y \preccurlyeq f(\mathrm{hi})$, $\forall x \in [x_l, \mathrm{lo}] : f(x) \prec y$,
(p) $\big(n_g = 0 \vee w_M < t \vee (n_g = 1 \wedge \alpha = \alpha^f)\big) \implies f(\mathrm{succ}(\mathrm{lo})) \succcurlyeq y$

1: **while** $\#[\mathrm{lo}, \mathrm{hi}) > 1$ **do**
2: mid := `split_point`(lo, hi);
3: ▷ $\exists m, m' > 0 \,.\, |m - m'| \leq 1 \wedge \mathrm{mid} = \mathrm{pred}^m(\mathrm{hi}) = \mathrm{succ}^{m'}(\mathrm{lo})$
4: **if** $y \preccurlyeq f(\mathrm{mid})$ **then** hi := mid
5: **else if** $n_g = 0 \vee \mathrm{mid} \preccurlyeq \alpha \vee \mathrm{mid} \succcurlyeq \omega \vee \#[f(\mathrm{mid}), y) > d_M$ **then** lo := mid
6: **else if** $n_g = 1 \wedge \big(w_M > t \vee f(\mathrm{succ}(\alpha)) \prec f(\alpha)\big)$ **then**
7: **if** $f(\omega) \succcurlyeq y$ **then**
8: **if** $f(\alpha) \succcurlyeq y$ **then** hi := α
9: **else if** $f(\mathrm{succ}(\alpha)) \prec f(\alpha)$ **then** lo := mid
10: **else if** lo $\prec \alpha$ **then** lo := α
11: **else break**
12: **end if**
13: **else** lo := ω
14: **end if**
15: **else if** $w_M \leq t$ **then**
16: b := `findfmax`$(f, w_M, \mathrm{lo}, \mathrm{mid})$;
17: ▷ $b \in \big[\max\{\mathrm{lo}, \mathrm{pred}^{w_M}(\mathrm{mid})\}, \mathrm{mid}\big] \wedge \forall x \in \big[\max\{\mathrm{lo}, \mathrm{pred}^{w_M}(\mathrm{mid})\}, \mathrm{mid}\big] : f(x) \preccurlyeq f(b)$
18: **if** $f(b) \succcurlyeq y$ **then** hi := b
19: **else** lo := mid
20: **end if**
21: **else**
22: z := `logsearch_lb`$(f, d_M, \mathrm{lo}, \mathrm{mid}, y, s)$;
23: ▷ $z \in [\mathrm{lo}, \mathrm{mid}] \wedge \big((\mathrm{lo} \prec z) \implies \#[f(z), y) > d_M\big)$
24: **if** lo $\prec z$ **then** lo := z
25: **else break**
26: **end if**
27: **end if**
28: **end while**

- If mid is not in a glitch, (**if**-guard on line 5) lo can be updated with the value of mid.
- Otherwise, if there is only one glitch, wider than t ($w_M > t$), or starting exactly at α (i.e., $f(\mathrm{succ}(\alpha)) \prec f(\alpha)$), and mid *may be* in it, the algorithm compares $f(\alpha)$ and $f(\omega)$ with y in order to set lo to the greatest correct value. If $f(\omega) \prec y$, then the function cannot reach y before ω, and we set lo to ω. Otherwise, we set hi to α if $f(\alpha) \succcurlyeq y$, and continue searching for y in the lower part of the interval. If $f(\alpha) \prec y$ and the glitch starts at α, then even if mid is in the glitch, there cannot be values of f reaching y before mid. Otherwise, we set lo to α.
- If mid *may be* in a glitch narrower than t, function `findfmax` finds the value b inside interval $\big[\max\{\mathrm{lo}, \mathrm{pred}^{w_M}(\mathrm{mid})\}, \mathrm{mid}\big]$ where $f(b)$ is the maximal. b is then used to refine $[\mathrm{lo}, \mathrm{hi}]$.

- If mid *may be* in a glitch wider than t, `bisect_lb` refrains from running the expensive float-by-float search performed by `findfmax` and calls `logsearch_lb`. If it exists, this function finds a value $z \in [\text{lo}, \text{mid}]$ such that $\#[f(z), y) > d_{\text{M}}$. If z is found, it is used to refine $[\text{lo}, \text{hi}]$.

`logsearch_lb` performs a logarithmic search to find a value z as above. Its argument $s \in \mathbb{N}$ is (for efficiency reasons) a limit to the number of times `logsearch_lb` can return an excessively wide interval as a refinement of $[x_l, x_u]$. If s has not been reached yet, `logsearch_lb` starts with $z = \text{mid}$ and decreases it (e.g., by dividing it by 2) until $\#[f(z), y) > d_{\text{M}}$. Otherwise, it sets z to lo.

The post-condition of `bisect_lb` ensures $x_l \preccurlyeq \text{lo} \prec \text{hi} \preccurlyeq x_u$, $f(\text{lo}) \prec y \preccurlyeq f(\text{hi})$ and $\forall x \in [x_l, \text{lo}] : f(x) \prec y$ hold when it terminates. At line 19 of `lower_bound`, a **while** loop performs a float-by-float search (for at most t iterations) to approach the exact solution of $y = f(\mathbf{x})$. Afterwards, the loop invariant $\forall x \in [x_l, \text{lo}] : f(x) \prec y$ holds. An **if** block (lines 23-25) tests if an optimal solution of $y = f(\mathbf{x})$ was found. If the **else** statement is reached, then the **while** loop terminated because t reached 0. In this case, l is set to lo, a suboptimal solution of $y = f(\mathbf{x})$.

The pseudocode of functions `init`, `gallop_lb`, `linsearch_geq`, `findfmax` and `logsearch_lb` is not shown, because they are straightforward.

The next results state that Algorithms 2, 3 and 4 are correct.

Lemma 5.1. *Whenever function* `findhi_lb` *of* Algorithm 3 *is called on actual parameters satisfying the* **Require** *condition, all values computed by* `findhi_lb` *satisfy the* **Ensure** *condition.*

Proof. (Sketch) The proof begins by assuming the precondition for `findhi_lb`$(f, y, [x_l, x_u], n_{\text{g}}, d_{\text{M}}, w_{\text{M}}, \alpha, \omega, t)$ is satisfied: in particular, $f(x_u) \prec y$. Then, a case analysis on the values of $n_{\text{g}}, w_{\text{M}}, f(x_u), f(\alpha)$ and $\text{succ}(\alpha)$ determines if a value for hi suitable for bisection can be computed with at most t iterations. In these cases, function `findhi_lb` returns $r = 1$. If, at least, a new value l such that $\forall x \in [x_l, l] : y \succ f(x)$ can be found, function `findhi_lb` returns either $r = 0$ or $r = 2$. When $r = 0$ the value of l is set to x_u. □

Lemma 5.2. *Whenever function* `bisect_lb` *of* Algorithm 4 *is called on actual parameters satisfying the* **Require** *condition, the values computed by* `bisect_lb` *satisfy the* **Ensure** *conditions.*

Proof. (Sketch) We assume that the precondition for `bisect_lb`$(f, y, [x_l, x_u], n_{\text{g}}, d_{\text{M}}, w_{\text{M}}, \alpha, \omega, n_g, s, t)$ is satisfied and consider the following **while** loop invariant:

$$\text{Inv} \equiv (x_l \preccurlyeq \text{lo} \prec \text{hi} \preccurlyeq x_u) \wedge (f(\text{lo}) \prec y \preccurlyeq f(\text{hi})) \wedge (\forall x \in [x_l, \text{lo}] : f(x) \prec y).$$

The schema of the proof consists in proving the following properties of the **while** loop of `bisect_lb`.

Initialization: Inv holds prior to the first loop iteration. Note that this is true since it is entailed by the **Require** statement.

Maintenance: assuming that Inv holds at the beginning of an arbitrary loop iteration, we prove, by case analysis, that Inv holds at the end of that iteration, as well.

Termination: we prove that $\#[\text{lo}, \text{hi}]$ decreases at each iteration. Since the guard of the **while** loop at line 1 tests the condition $\#[\text{lo}, \text{hi}) > 1$, that is equivalent to $\#[\text{lo}, \text{hi}] > 2$, it is guaranteed that the loop always terminates.

Correctness: the *correctness* post-condition coincides with invariant Inv. To prove the *precision* post-condition we show that, at the exit of the loop, $\text{succ}(\text{lo}) = \text{hi}$ holds. Therefore, by Inv, we have $y \preccurlyeq f(\text{hi})$, which implies $f(\text{succ}(\text{lo})) \succcurlyeq y$.

□

As a consequence of the *precision* post-condition, the following result also shows that when function f is isotonic or it has glitches narrower than t, Algorithm 2 finds a precise solution, i.e., it returns either $r = 3$ or $r = 4$.

Theorem 5.3. *Whenever function* `lower_bound` *of* Algorithm 2 *is called on actual parameters satisfying the* **Require** *condition, the values computed by* `lower_bound` *satisfy the* **Ensure** *conditions.*

Proof. (Sketch) We assume the precondition for `lower_bound`$(y, [x_l, x_u], n_g, d_M, w_M, \alpha, \omega, f^i, t)$ holds. In order to prove the *correctness* post-condition we proceed as follow. First, we prove that, when calling function `gallop_lb` at line 2, the actual parameters satisfy the **Require** conditions of `gallop_lb`. Then, we know that function `gallop_lb`$(f, y, [x_l, x_u], d_M, i)$ returns values for lo and hi satisfying the post-condition of line 3. Now, a case analysis on the comparison between the value of $f(\text{lo})$ and y on line 4 and on line 10 directly proves the post-condition for $r = 1$ and $r = 4$. The next step is to prove that the precondition of `findhi_lb` is satisfied. Afterwards, by Lemma 5.1, the post-condition of `findhi_lb` holds. Therefore, when `findhi_lb` terminates, the post-conditions for $r = 0$ and $r = 2$ are proved. Then, the last step is proving that for $r \neq 0$ and $r \neq 2$ the preconditions of function `bisect_lb` are met. By Lemma 5.2, after it returns, $x_l \preccurlyeq \text{lo} \prec \text{hi} \preccurlyeq x_u$, $f(\text{lo}) \prec y \preccurlyeq f(\text{hi})$ and $\forall x \in [x_l, \text{lo}] : f(x) \prec y$ hold, for the new values of lo and hi. At line 19 a **while** loop is entered. This loop performs a float-by-float search (for a maximum of t iterations) to approach the exact solution of $y = f(\mathbf{x})$. We prove that the predicate $\forall x \in [x_l, \text{lo}] : f(x) \prec y$, which is also the loop invariant, holds at line 22. To this aim, we prove the following loop properties:

Initialization: the invariant $\forall x \in [x_l, \text{lo}] : f(x) \prec y$ holds prior to the first loop iteration because it is entailed by the post-condition of function `bisect_lb`.

Maintenance: we assume $\forall x \in [x_l, \text{lo}] : f(x) \prec y$ holds at the beginning of an arbitrary loop iteration. This assumption, together with the guard of the loop $f(\text{succ}(\text{lo})) \prec y$ and the assignment $\text{lo}' := \text{succ}(\text{lo})$ in the body of the loop, allows us to conclude that $\forall x \in [x_l, \text{lo}'] : f(x) \prec y$ holds also at the end of the iteration.

Termination: the loop terminates because, by the post-condition of `bisect_lb`, there exists a value hi such that $\text{lo} \prec \text{hi}$ and $y \preccurlyeq f(\text{hi})$. Moreover, the loop can end before reaching such value, because the parameter $t \in \mathbb{N}$ is decremented inside the loop, until it reaches 0.

Correctness: as a consequence, at the end of the loop, the property $\forall x \in [x_l, \text{lo}] : f(x) \prec y$ holds and either $f(\text{succ}(\text{lo})) \succcurlyeq y$ or $t = 0$.

Finally, the test on $f(\text{succ}(\text{lo})) \succ y$ at line 23 allows us to prove the post-condition for $r \in \{2, 3, 4\}$ by case analysis. □

Therefore, algorithm `lower_bound` ensures the optimality of the bound in the following cases:

- $n_g = 0$: the function is monotonic, or
- $w_M < t$: the glitches are not too large to perform linear searches, or
- $n_g = 1$ and $\alpha = \alpha^f$: f has one glitch only, and the position where it begins is known exactly.

For most functions, the worst-case computational complexity of `lower_bound` in Algorithm 2, measured as the number of calls to function f, has the form $k \log_2(\#[x_l, x_u]) + k + c$, for small constants k and c that are related to w_M. This follows from `lower_bound` being based on a dichotomic search, with occasional linear searches limited by a constant.

Theorem 5.4. *If* $f : \mathbb{F} \to \mathbb{F}$ *is an isotonic function, i.e.,* $n_g = 0$*, then, for each* $[x_l, x_u] \in \mathcal{I}_\mathbb{F}$*,* d_M*,* w_M*,* α*,* ω*,* $f^i : \mathbb{F} \to \mathbb{F}$*,* $s, t \in \mathbb{N}$*, computing* `lower_bound` *as per* Algorithm 2 *evaluates* f *at most* $2 \log_2(\#[x_l, x_u]) + 4$ *times.*

Moreover, if f has at least one glitch and $w_M \leq t$, k is strictly related to w_M.

Theorem 5.5. *If* $f : \mathbb{F} \to \mathbb{F}$ *has short glitches, that is,* $n_g > 0$ *but* $w_M < t$*, then, for each* $[x_l, x_u] \in \mathcal{I}_\mathbb{F}$*,* d_M*,* α*,* ω*,* $f^i : \mathbb{F} \to \mathbb{F}$*,* $s \in \mathbb{N}$*, computing* `lower_bound` *as per* Algorithm 2 *evaluates* f *at most* $(w_M + 1) \log_2(\#[x_l, x_u]) + w_M + 6$ *times.*

The formal proofs of all results can be found in [6].

6 TRIGONOMETRIC FUNCTIONS

The propagators for trigonometric functions (i.e., the floating-point approximations of sin, cos and tan) require a more complex approach to quasi-monotonicity. The underlying (partial) functions in $\mathbb{R} \rightarrowtail \mathbb{R}$ change their monotonicity periodically. In particular, the sin function changes its monotonicity in odd multiples of $\frac{\pi}{2}$ (of the form $(2k+1)\frac{\pi}{2}$, $k \in \mathbb{R}$) while the cos functions does so in even multiples (of the form $2k\frac{\pi}{2}$). The tan function has asymptotes in odd multiples of $\frac{\pi}{2}$, and in the intervals between them it is isotonic. Because of this behaviour, Definition 4.2 fails to distinguish glitches caused by the implementation from "legitimate" monotonicity changes. However, if we separately consider a quasi-monotonic branch of the periodic function that is significantly wider than the widest glitch in terms of ULPs, we can locally apply Definition 4.2. If we limit our reasoning to each quasi-monotonic branch separately, all the statements we made for quasi-monotonic functions locally hold, and we can use the same methods we developed for them.

To properly identify monotonicity glitches in trigonometric functions, we need:

- an interval in which the density of floating point values in every quasi-monotonic branch of the functions' graphs is sufficiently high;
- a way to split their domain into quasi-monotonic branches.

6.1 An Appropriate Domain for Trigonometric Functions Analysis

To choose a domain suitable for the search of monotonicity glitches, we must consider that, while the period of a trigonometric function is constant, the distance between two consecutive floating-point numbers increases with the exponent. Such distance is expressed by the value of the ulp: $\mathbb{R} \rightarrow \mathbb{R}$ function: we will use the definition of ulp given in [58, Definition 5]. A floating-point interval in which the idea of a monotonicity glitch is well defined should have a sufficient cardinality to allow for glitches that do not cover the entire interval.

Let $x \in \mathbb{F}$ be a positive normal floating-point number: then $\text{ulp}(x) = x - \text{pred}(x) = 2^{e_{\text{pred}(x)}-p+1}$, where p is the precision of the format, and $e_{\text{pred}(x)}$ the exponent of $\text{pred}(x)$. For trigonometric functions, the size in $\mathbb{R}$ of the intervals in which the function has constant monotonicity is π. If we consider an interval $[-\ell_{\max}, \ell_{\max}]$ in which for each $x \in [-\ell_{\max}, \ell_{\max}]$ we have $\text{ulp}(x) \leq 0.5$, then each monotonic branch contains at least 6 or 7 floats, which is acceptable for the propagators described in Section 5, if glitches have a width of 1 or 2 floats. In intervals with a higher ulp value, the notion of glitches would be hardly meaningful. So, we use a maximum exponent $e_{\ell_{\max}} = p - 1$, leading to a domain $[-\ell_{\max}, \ell_{\max}]$ with $\ell_{\max} = 2^{p-1}$. In conclusion, $\ell_{\max} = 2^{23}$ for the IEEE 754 single-precision format, and $\ell_{\max} = 2^{52}$ for double-precision.

As we noted in Section 1, domains like these are still excessively large for most real-world applications. For the experiments reported in Section 7.2, we used a bound $\ell_{\max} = 16$, for example.

6.2 Outline of the Propagation Algorithms

In this section, we describe the projection algorithms we have devised for trigonometric functions.

6.2.1 Direct Propagation. The periodicity of trigonometric functions poses a fundamental issue: in each monotonic branch the function can cover its whole range. Therefore, if the interval $[x_l, x_u]$ to be used to refine $\mathbf{y}$ spans multiple branches, almost no refinement can be performed. However, since floating-point numbers become sparser as the exponent grows, there is the possibility that some branches do not reach the ends of the range, because there are no points where the function takes those values in such branches.

Our algorithm takes advantage of these facts. First, it identifies the branches of the graph of function f to which x_l and x_u belong. Let $c \in \mathbb{R}$: with $[c]_\uparrow$ we will denote the upper floating-point approximation of c, i.e. $[c]_\uparrow = \min\{\, x \in \mathbb{F} \mid x \geq c \,\}$. Similarly, with $[c]_\downarrow$ we will denote the lower

floating-point approximation of c, so that $[c]_\downarrow = \max\{\, x \in \mathbb{F} \mid x \le c \,\}$. Given $x \in \mathbb{F}$, to identify the branch it belongs to, we compute $k = \left\lceil x \frac{2}{\pi} \right\rceil$, with sufficient precision. This can be achieved with a range reduction algorithm, as described in [6], or [59, 60]. Then, if k is odd and $f = \cos$, or if k is even and $f = \sin$ or $f = \tan$, k is incremented. The value of k is then such that f changes its monotonicity or has a discontinuity in $k\frac{\pi}{2}$ and $(k-2)\frac{\pi}{2}$, and $x \in \left[\left[(k-2)\frac{\pi}{2}\right]_\uparrow, \left[k\frac{\pi}{2}\right]_\downarrow\right]$.

If x_l and x_u are both in the same monotonic branch, the refinement algorithm for regular functions described in Section 5.1 is called. Otherwise, the refinement function should be called separately for each branch, and then the minimum value of y_l and the maximum value of y_u should be returned. Since the number of branches to be separately inspected can be high, a threshold g is imposed on the maximum number of branches to be analyzed. If $[x_l, x_u]$ spans more than g branches, the bounds of the function's range are returned.

Algorithm 5 Indirect propagation: `bounds_trig`$(f, f^{\mathrm{i}}, [y_l, y_u], [x_l, x_u], n_{\mathrm{gM}}, d_{\mathrm{M}}, w_{\mathrm{M}}, \alpha, \omega, g, s, t)$

Require: $f\colon \mathbb{F} \to \mathbb{F}$, $f^{\mathrm{i}}\colon \mathbb{F} \to \mathbb{F}$, $[x_l, x_u], [y_l, y_u] \in \mathcal{I}_{\mathbb{F}}$, $n_{\mathrm{gM}} \ge n^f_{\mathrm{gM}}$, $d_{\mathrm{M}} \ge d^f_{\mathrm{M}}$, $w_{\mathrm{M}} \ge w^f_{\mathrm{M}}$, $\alpha \preccurlyeq \alpha^f$, $\omega \succcurlyeq \omega^f$, $n_{\mathrm{gM}} > 0 \implies (x_l \preccurlyeq \alpha \preccurlyeq \omega \preccurlyeq x_u)$, $g, s, t \in \mathbb{N}$.

Ensure: $|I_r| \le g$, $\bigcup_{0 \le i \le |I_r|} [x^i_{l1}, x^i_{u2}] = [x_l, x_u]$, $\forall 0 \le i \le |I_r| : l^i, u^i \in \mathbb{F}$
$\quad \wedge \big(0 \le r^i_l \le 4 \wedge (p_{r^i_l}(y_l, x^i_{l1}, x^i_{u2}, l^i) \vee p_{r^i_l}(-y_u, x^i_{l1}, x^i_{u2}, l^i))\big)$
$\quad \wedge \big(5 \le r^i_u \le 9 \wedge (p_{r^i_u}(y_u, x^i_{l1}, x^i_{u2}, u^i) \vee p_{r^i_u}(-y_l, x^i_{l1}, x^i_{u2}, u^u))\big)$.

1: $I := \mathtt{split\_interval}(f, x_l, x_u, g)$; $I_r := \varnothing$;
2: **for all** $([x^i_{l1}, x^i_{u1}], [x^i_{l2}, x^i_{u2}]) \in I$ **do**
3: **if** isotonic$(f, [x^i_{l1}, x^i_{u1}])$ **then**
4: $(r^i_l, l^i) := \mathtt{lower\_bound}(f, y_l, [x^i_{l1}, x^i_{u1}], n_{\mathrm{gM}}, d_{\mathrm{M}}, w_{\mathrm{M}}, \alpha, \omega, f^{\mathrm{i}}, s, t)$
5: **else**
6: $(r^i_l, l^i) := \mathtt{lower\_bound}(-f, -y_u, [x^i_{l1}, x^i_{u1}], n_{\mathrm{gM}}, d_{\mathrm{M}}, w_{\mathrm{M}}, \alpha, \omega, f^{\mathrm{i}} \circ (-\mathrm{id}), s, t)$
7: **end if**
8: **if** isotonic$(f, [x^i_{l2}, x^i_{u2}])$ **then**
9: $(r^i_u, u^i) := \mathtt{upper\_bound}(f, y_u, [x^i_{l2}, x^i_{u2}], n_{\mathrm{gM}}, d_{\mathrm{M}}, w_{\mathrm{M}}, \alpha, \omega, f^{\mathrm{i}}, s, t)$
10: **else**
11: $(r^i_u, u^i) := \mathtt{upper\_bound}(-f, -y_l, [x^i_{l2}, x^i_{u2}], n_{\mathrm{gM}}, d_{\mathrm{M}}, w_{\mathrm{M}}, \alpha, \omega, f^{\mathrm{i}} \circ (-\mathrm{id}), s, t)$
12: **end if**
13: $I_r := I_r \cup ([x^i_{l1}, x^i_{u1}], [x^i_{l2}, x^i_{u2}], [l^i, u^i], r^i_l, r^i_u)$
14: **end for**

6.2.2 Indirect Propagation. Algorithm 5, for indirect propagation, refines the interval for $\mathbf{x}$ by splitting its initial domain $[x_l, x_u]$ into intervals in which the function graph is monotonic, and then applies `lower_bound` and `upper_bound` locally, to obtain a refined multi-interval. Interval $[x_l, x_u]$ can be wide, and the number of intervals it has to be split into can be excessively large. So, Algorithm 5 limits their number to parameter g. The intervals closest to 0 are split with maximum granularity (each monotonic branch is considered separately), because floating-point numbers in this area are denser, and the probability of finding an optimal solution is higher. The remaining branches are gathered into two larger intervals, one to the left of the domain and starting with x_l, and one to the right, ending with x_u. These intervals are only refined in branches at the boundaries. Function `split_interval` returns this split of the intervals, which depends on function f. If f has g or more monotonic branches in $[x_l, x_u]$, then `split_interval` returns a list of pairs $([x^i_{l1}, x^i_{u1}], [x^i_{l2}, x^i_{u2}])$, $1 \le i \le g$. If $f = \cos$, those with $2 \le i \le g-1$ are such that

$x^i_{l1} = x^i_{l2} = \max(x_l, \lceil 2k_i \frac{\pi}{2} \rceil_\uparrow)$ and $x^i_{u1} = x^i_{u2} = \min(x_u, \lceil (2k_i + 2)\frac{\pi}{2} \rceil_\downarrow)$, with $\lceil 2k_i \frac{\pi}{2} \rceil_\uparrow \geq x_l$ and $\lceil (2k_i + 2)\frac{\pi}{2} \rceil_\downarrow \leq x_u$, where the distinct values $k_i \in \mathbb{Z}$ are those closest to 0 in magnitude, and $k_{i+1} = k_i + 1$. For $i = 1$, we have $[x^1_{l1}, x^1_{u1}] = [x_l, \lceil (2k_1 + 2)\frac{\pi}{2} \rceil_\downarrow]$, where $k_1 \in \mathbb{R}$ is the highest value such that $\lceil (2k_1 + 2)\frac{\pi}{2} \rceil_\downarrow \geq x_l$, and $[x^1_{l2}, x^1_{u2}] = [\lceil 2k_1 \frac{\pi}{2} \rceil_\uparrow, \lceil (2k_1 + 2)\frac{\pi}{2} \rceil_\downarrow]$. The last pair, with $i = g$, is symmetric. If f has less than g monotonic branches, exactly those sub-intervals are returned. For functions sin and tan, which change monotonicity in odd multiples of $\frac{\pi}{2}$, replace $2k_i$ with $2k_i + 1$. The multiplications by $\frac{\pi}{2}$ are done with enough digits of π to be correctly rounded.

Functions `lower_bound` and `upper_bound` are called at lines 3–12, distinguishing whether f is isotonic or antitonic in each interval. The arguments required by Algorithm 5 are essentially the same as those for `lower_bound`, except for n_{gM}, which is a safe approximation of the maximum number of glitches in each quasi-monotonic branch of the function.

`lower_bound` and `upper_bound` can only be called if $[x_l, x_u]$ is a subset of $[-\ell_{\max}, \ell_{\max}]$ because they cannot operate on intervals too narrow (see section 6.1). If Algorithm 5 is applied repeatedly on the same interval, however, a value $g > 1$ could cause complexity issues: if sub-intervals are discarded because no solutions are found in them, a repeated call of this refinement algorithm would split the domain again and again. To avoid this problem, the algorithm should be called on a further reduced domain. Note that, while this algorithm often succeeds in finding a refined interval if the function has the desired values $\mathbf{y}$ in one of the analyzed sub-intervals, it can say nothing if such values are not found. In this case, we cannot exclude the possibility that the function reaches them somewhere outside $[-\ell_{\max}, \ell_{\max}]$ (or a smaller domain, if chosen). This prevents us from being able to tell when the equation $\mathbf{y} = \mathbf{x}$ has no solution, which we could do for the regular functions. However, since floating-point numbers are denser near 0, the functions take most of the values of their image in its vicinity. This allows our algorithm to find a solution very often (when present), making it useful for automatic test-data generation.

7 IMPLEMENTATION AND EXPERIMENTS

In this section we first describe the implementation of the algorithms introduced in this paper. Then, we show the results of its experimental evaluation.

The main research question that we aim to answer in this section regards the feasibility of our approach. This aim can be split into the following research questions, which concern different aspects of the problem:

- RQ1: In what way can our approach be useful to a programmer?
- RQ2: How effective is our approach in proving or disproving the possible occurrence of unwanted behaviors in floating-point computations?
- RQ3: What are the performances of our approach? How long does it take to perform the activities investigated by the previous questions?

We answer to RQ1 in Section 7.2, by means of a case study, and to RQ2 and RQ3 in Section 7.3.

7.1 Implementation

All the algorithms presented in this paper have been implemented and included in the ECLAIR software verification platform for C/C++ source code, Java source code and bytecode.[12] For the analysis of integers and floating-point values, ECLAIR mainly uses multi-intervals with a judicious use of polyhedral approximations made available by the Parma Polyhedra Library (PPL) [7]. For reasoning on the floating-point arithmetic operations, ECLAIR uses:

[12]http://bugseng.com/products/eclair, last accessed on July 16th, 2020.

- algorithms realizing optimal direct projections as well as correct and precise indirect projections: the result is similar to the projections defined in [53], but the ECLAIR algorithms never require working with precision greater than the operation data type;
- algorithms that exploit properties of the binary floating-point representations in order to obtain enhanced precision [5];
- dynamic linear relaxation techniques [9, 25] using the PPL to enhance constraint propagation with the relational information provided by convex polyhedra.

The algorithms defined in Section 5 have been implemented in C++ and extensively tested on a variety of implementations with different characteristics in terms of the presence and nature of monotonicity glitches. These algorithms are now used in three components of the platform instantiation for C/C++: the semantic analysis engine based on abstract interpretation [20], the automatic generator of test inputs, and the symbolic model checker, the latter being both based on constraint solving [32, 33]. All components use multi-interval refinement, though in different ways: the test generator and symbolic model checker are driven by labeling and backtracking search. As these have a negative interaction with the searches controlled by the s and t parameters of the algorithms, their setting needs to be controlled more carefully (and they are better set to 0 when glitch data is precise) by these components, whereas they are used with values in the range 5–20 in the semantic analysis engine. Here, we report on experiments with the symbolic model checker and automatic generator of test inputs. One of its interesting features is that it optionally produces a transformed source program that contains the original program, suitably instrumented, and a driver that runs one of the generated tests (or model checking counterexamples) at a time. The instrumented code checks that each one of the generated tests achieves its target, e.g., it reaches a certain program point, or it causes an integer overflow or the generation of a floating-point NaN or infinite value. The validation of test inputs is thus completely automatic.

7.2 Case Study

To illustrate better the potential that algorithms in Sections 5 and 6 have, let us consider again the introductory example of Figure 1. We show how our tool can support the workflow of a programmer in checking whether such code presents unwanted behaviors or not. Let us pretend we know nothing about the code (which is realistic, as there are no comments besides the one at line 5). So, we initially assume that the entry point is `latlong_utm_of()`; as there are no assertions, we also assume all inputs are possible. For an exploratory analysis, we use ECLAIR's symbolic model checker in order to detect the possible presence of run-time anomalies: overflow, division by zero and other sources of undefined and implementation-defined behavior over the integers, inexact integer-to-floating conversions, finite-to-infinite and numeric-to-NaN transitions over floating point numbers. A *finite-to-infinite* (resp., *numeric-to-NaN*) transition is a computation whereby the inputs to a floating-point operation or `math.h`/`cmath` function is finite (resp., numeric) and the output is infinite (resp, NaN). We also set an analysis parameter asking ECLAIR to flag all the invocations of trigonometric functions whose argument has an absolute value greater than, say, 16. Not surprisingly, we obtain three test inputs showing that this is indeed possible. They concern the following program points:

p_2 : $(-$`0x864880.p − 18F`$, +$`0.0F`$, 1)$, where $-$`0x864880.p − 18F` ≈ -33.570801,
p_3 : $(-$`0x8a3ae7.p − 19F`$, +$`0.0F`$, 1)$, where $-$`0x8a3ae7.p − 19F` ≈ -17.278761,
p_5 : $(+$`0x96d12f.p − 21F`$, -$`0x98b6c1.p − 19F`$, 1)$, where $+$`0x96d12f.p − 21F` ≈ 4.713035
and $-$`0x98b6c1.p − 19F` ≈ -19.089235.

Of course, the latter input causes the same phenomenon at program point p_7 as well. Perhaps `latlong_utm_of()` callers only pass smaller values for `phi` and `lambda`. Even if that is not the

case, then perhaps the only problem is a slight precision issue. But ECLAIR produces two other test inputs, with the specification that they trigger number-to-NaN transitions:

p_1 : (+0xc90fdb.p − 23F, +0.0F, 1), where +0xc90fdb.p − 23F ≈ 1.570796,
p_4 : (−0xb63223.p − 35F, +0xcfbb98.p − 23F, 1), where we can give the approximations −0xb63223.p − 35F ≈ $-3.48 \cdot 10^{-4}$, and +0xcfbb98.p − 23F ≈ 1.622912.

As +0xc90fdb.p − 23F ≈ 1.570796 converted to double precision is slightly greater than M_PI_2, the round-to-nearest, double-precision approximation of $\frac{\pi}{2}$ defined in math.h, we make the hypothesis that phi has to be less than or equal to M_PI_2. Indeed, looking at the function callers (there is only one in the program), we come to the realization that phi and lambda are a latitude and longitude in radians, respectively. This part of the analysis took 27.81 seconds. We attempt validation of this hypothesis by adding the assertions

```
assert(-M_PI_2 <= phi && phi <= M_PI_2);
assert(-M_PI <= lambda && lambda <= M_PI);
```

at the beginning of latlong_utm_of() and repeat the analysis. After 22.51 seconds, we obtain another ill-conditioned trigonometric function argument test input for program point p_5:

p_5 : (−0xc8f7db.p − 23F, +0xc90fda.p − 22F, 255), where −0xc8f7db.p − 23F ≈ −1.570064 and +0xc90fda.p − 22F ≈ 3.141593

(surely utm_zone = 255 is not among the expected inputs) and another numeric-to-NaN transition:

p_4 : (−0xb63223.p − 35F, +0xcfbb98.p − 23F, 1), where we have the approximations −0xb63223.p − 35F ≈ $-3.48 \cdot 10^{-4}$, and +0xcfbb98.p − 23F ≈ 1.622912.

To understand the intended inputs for latlong_utm_of(), we take into account its calling context:

```
1       nav_utm_zone0 =* (gps_lon/10000000+180) / 6 + 1;
2       latlong_utm_of(RadOfDeg(gps_lat/1e7), RadOfDeg(gps_lon/1e7), nav_utm_zone0);
```

The inputs to latlong_utm_of() depend on two 32-bit signed integers, gps_lat and gps_lon, that are received from a communication channel: no check is made upon them after reading the values out of the input buffer. Taking into account the caller context, in 25.73 seconds ECLAIR generates three reports. If gps_lat = 0 and gps_lon = −1920000000 at line 1, then the conversion in the assignment marked with '∗' on the same line causes an unsigned wraparound (−1 mod 256 = 255, so that, yes, latlong_utm_of() can be called with utm_zone = 255). The same input also generates an ill-conditioned trigonometric function argument for program point p_5 in Figure 1. Most importantly, if gps_lon = 900000059 and gps_lat = −1920000000, then we have a numeric-to-NaN transition at program point p_1. This probably means that if the equipment at the other end of the communication channel is defective or if there is a communication error, things can go horribly wrong. However, let us now suppose that there are no problems of this kind and that we have $|\texttt{gps\_lat} \cdot 10^{-7}| \leq 90$ and $|\texttt{gps\_lon} \cdot 10^{-7}| \leq 180$ as the code seems to assume. In 27.18 seconds, the analysis with ECLAIR shows this is not enough: the numeric-to-NaN transition at program point p_1 is still possible with gps_lat = −899999991 and gps_lon = −1800000000 (this point is roughly 10 cm from the Geographic South Pole). In a couple more iterations we add the assertions

```
-1    assert( -899999990 <= gps_lat && gps_lat <=  899999990);
 0    assert(-1800000000 <= gps_lon && gps_lon <= 1800000000);
```

before line 1 of the calling context, and the final ECLAIR run shows no report. This, per se, does not mean much. However, this experiment was performed on the xps machine, for which we have precise glitch data for the single-precision functions (which are not used in the code considered) and we have the maximum known errors provided by the GNU libc manual for the double-precision

functions [51]. As explained in Section 9.2.2, this data provides imprecise and possibly incorrect information about glitches that our algorithms can exploit. In turn, all this means that:

- *if* the numbers in [51] do really provide upper bounds to the maximum errors of the used functions, and
- *if* the caller guarantees that the values of `gps_lat` and `gps_lon` do satisfy the "stay away from the poles" assertions at lines −1, 0
- *then*, in the context of such a call, all the 154 potential run-time anomalies in the 90 potentially problematic program points of Figure 1 cannot occur on `xps`.

More precisely, these anomalies consist of 4 integer overflows, 4 inexact conversions, 10 ill-conditioned trigonometric function arguments, 70 finite-to-infinity and 66-numeric-to-NaN transitions. Just to mention one potential problem, division by zero and consequent finite-to-infinite transition at program point p_6, cannot happen on that implementation.

7.3 Feasibility Evaluation

To better assess the capabilities of our approach, we analyzed with ECLAIR a benchmark that we assembled by taking code from the GNU Scientific Library (GSL),[13] AxBench [69], a benchmark popular in approximate computing, and a test suite that we created to evaluate the performances of our algorithms in the most disparate ways. We also included the code from the Paparazzi UAV avionics library from Figure 1. Since our purpose is to evaluate mainly the propagators for the supported `math.h`/`cmath` functions, we selected only code that contains at least two calls to such functions, and we excluded code containing unsupported functions, such as `pow`. We tried to assemble a rather varied benchmark suite, containing code from different application domains. Indeed, GSL is a library for scientific computation, while from AxBench we picked code from the finance (blackscholes.c) and robotics (inversek2j.c) application domains. While other benchmarks are aimed at evaluating our approach on real-world code, our self-made test suite contains various computations aimed at generating constraint systems that are difficult to solve.

For each floating-point operation, ECLAIR tries to prove that such operation may not generate any finite-to-infinite or numeric-to-NaN transition. If it fails, it generates a counterexample, i.e. a program input that causes such transition. We limited the maximum number of iterations of the constraint solving process (i.e., $i_{\max}$ of Algorithm 1, Section 3.2) to 200. If such threshold is reached, ECLAIR times out. Each generated input is automatically validated by executing the original program. This showed that all inputs generated by ECLAIR actually trigger the intended behaviour in the original code, so we can claim that no false positives were generated.

The analyses were executed on machine `xps`, a high-end laptop with an x86_64 CPU (6 cores @2.20GHz) and 16 GB of RAM, running Ubuntu 19.10. This could be a typical hardware setting for a software developer. ECLAIR does not yet support multi-threaded constraint solving, so only one CPU core is used at a time. We report the results in Table 2.

7.3.1 RQ2: Detection of anomalous behaviors. RQ2 asks how effective our approach is in proving or disproving the possible occurrence of unwanted behaviors in floating-point computations. In particular, we want to assess the proportion of floating-point operations for which ECLAIR is able to generate an answer without timing out.

Table 2 shows that a significant number of operations generating infinities or NaNs were found, but in most cases, ECLAIR was able to prove that no such behavior may occur. The number of timeouts is generally limited, ad it is higher in files containing long sequences of floating-point operations with data-flow dependencies, that lead to the generation of large constraint systems.

[13] https://www.gnu.org/software/gsl/, last accessed on July 16th, 2020.

Table 2. Benchmark data for the xps machine. For each file we report the number of lines of code (# LOC), the number of finite-to-infinite and numeric-to-NaN transitions, the total time taken by the analysis (T) in seconds, the time taken by propagators for `math.h`/`cmath` functions (T_m) in milli-seconds. For each kind of transition, ECLAIR identified all operations that may potentially trigger them. We report the total number of such operations that ECLAIR proved feasible by generating a test-case (g), those that ECLAIR proved unfeasible (u), and those for which ECLAIR timed out (t).

Benchmark	# LOC	finite to $+\infty$			finite to $-\infty$			numeric to NaN			Time	
		g	u	t	g	u	t	g	u	t	T (s)	T_m (ms)
GSL												
bessel.c	191	6	74	1	5	75	1	1	90	1	23.79	531.24
bessel_i.c	144	4	96	0	2	92	0	0	94	0	9.83	6.32
bessel_j.c	158	0	109	0	0	109	0	0	114	0	85.18	109.89
bessel_olver.c	185	49	305	7	11	349	1	14	363	0	201.99	267.13
exp.c	426	18	195	14	8	200	2	0	210	0	342.94	7566.29
gegenbauer.c	181	33	68	5	14	84	8	4	104	0	700.79	10.75
lambert.c	219	17	41	3	6	55	3	2	65	0	20.27	56.77
sincos_pi.c	163	4	35	0	2	37	0	2	39	0	0.83	0.32
cauchy.c	57	2	8	0	2	8	0	2	8	0	0.63	6.55
cauchyinv.c	73	5	9	0	7	7	0	4	10	0	1.46	422.39
exponential.c	56	1	0	1	1	0	0	1	0	0	2.36	0.35
exponentialinv.c	36	1	1	0	2	1	0	1	2	0	0.61	1.11
gauss.c	337	31	59	8	3	91	2	0	95	1	308.89	70.51
gaussinv.c	286	11	73	14	6	85	9	2	97	3	125.57	8.69
gumbel1.c	47	4	6	0	4	3	0	2	5	0	0.94	7.58
gumbel1inv.c	59	2	1	0	3	3	0	3	3	0	0.68	3.30
laplace.c	56	2	11	0	2	7	0	2	7	0	0.63	1.72
laplaceinv.c	73	4	6	0	8	6	0	4	10	0	0.95	4.44
logistic.c	56	2	14	0	2	8	0	2	8	0	0.63	2.64
logisticinv.c	59	3	3	0	3	5	0	2	6	0	0.78	1.69
lognormal.c	38	33	61	8	7	93	2	4	97	1	308.58	72.09
lognormalinv.c	65	11	79	14	6	89	9	2	101	3	126.49	8.74
paretoinv.c	59	3	2	0	2	2	0	1	3	0	0.69	3.00
rayleigh.c	36	2	2	1	3	1	0	2	2	0	0.69	0.67
rayleighinv.c	59	2	0	0	2	1	0	2	3	0	0.66	3.50
AxBench												
blackscholes.c	292	6	111	8	8	107	7	4	81	2	556.67	699.89
inversek2j.c	26	5	27	0	1	31	0	3	33	0	1.10	3.38
paparazzi.c	93	2	81	1	2	83	2	5	79	1	27.22	759.97
Test suite	3370	234	421	26	125	483	17	105	582	2	135.30	4274.81
Total	6900	497	1898	111	247	2115	63	176	2311	14	2987.16	14905.72

Overall, ECLAIR was able to either prove or disprove, without timing out, the occurrence of 2395 finite to $+\infty$ transitions out of 2506 (96 %), 2362 finite to $-\infty$ transitions out of 2425 (97 %), and 2487 numeric-to-NaN transitions out of 2501 (99 %). Such a low timeout rate makes our approach useful in practice to analyze code bases such as those considered in the benchmark.

7.3.2 RQ3: Performances. RQ3 asks what are the performances of our approach, i.e. how long it takes to perform the activities investigated by the previous questions.

The total time taken by the analysis ranges from less than a second for files with tens of lines of code, to up to 12 minutes for files containing hundreds of lines of code. Thus, we can claim

that this kind of analysis is most convenient for small computational kernels, but still practically feasible for medium-large ones. We can also observe that, in general, the time taken by propagators for `math.h`/`cmath` functions, i.e., the algorithms of Sections 5 and 6, is negligible with respect to the total time taken by the analysis. This can be partially explained with the fact that, while such propagators have been implemented in C++, the rest of the program analysis and constraint propagation infrastructure has been implemented in (a small subset of) the Prolog programming language. A full implementation in C++ or another language of comparable performances could significantly improve the overall execution times.

8 COMPARISON WITH THE STATE OF THE ART

In this section, we compare the techniques presented in this paper with the state of the art, which we summarize below.

Interval consistency techniques have been extended to floating-point computations in [13], with the purpose of symbolic execution. More advanced techniques for refining arithmetic floating-point operations have been proposed in [5]. All such works only deal with basic arithmetic operations, and do not provide any technique to tackle `math.h`/`cmath` mathematical functions.

Interval refinement algorithms for mathematical functions, which take into account all IEEE 754 rounding modes, were introduced in [53], but they require excessively stringent features for the functions' implementations: they must be correctly rounded, and strictly monotonic. As we report in Sections 4 and 9.2, most implementations are far from meeting any of such requirements.

The detection of floating-point anomalies such as the ones we consider in Section 7.2 has been previously tackled in [55] by means of abstract interpretation, using linear real-valued approximations of floating-point constraints. This work has been implemented in the commercial tool ASTREÉ.[14] [55] only deals with basic arithmetic operations, and not with `math.h`/`cmath` functions. We could find no evidence in the literature of the addition of such features to ASTREÉ afterwards.

The tool Ariadne [8] performs symbolic execution of floating-point computations by approximating them with real numbers, and solves the resulting constraint systems with a SMT solver. Mathematical functions are also supported, but the fact that they are being approximated with reals makes this approach unsound, even if their implementations are correctly rounded and strictly monotonic, because rounding is not taken into account.

Canalyze-fp [68] also uses symbolic execution to detect floating-point exceptions, and uses the floating-point theory supported by the SMT solver Z3.[15] `math.h`/`cmath` functions are approximated by just considering their (theoretical) ranges, e.g., a constraint such as `y = exp(x)` is approximated to $y \geq 0 \land y \leq +\infty$. Clearly, this approach is trivially sound, but fails to exploit the peculiarities of the functions to effectively refine variable domains.

[67] combines symbolic execution with value range analysis to speed up the floating-point exception detection process. While a SMT solver is used for symbolic execution, value range analysis is performed with interval arithmetic. The indirect propagators for `math.h`/`cmath` functions employed to refine variable ranges assume correct rounding of their implementations, and do not take into account the effects of rounding, as [53] does. Thus, this approach is also unsound.

To the best of our knowledge, none of the tools above is available to the public. Anyways, as we detailed above, the most advanced treatment of `math.h`/`cmath` functions is the one of [53], despite its being less recent than other approaches. To show what kind of issues arise when using such unsound techniques to attempt software verification, we implemented the projections of [53], and

[14] https://www.absint.com/astree/index.htm, last accessed on July 16th, 2020.
[15] https://github.com/Z3Prover/z3, last accessed on July 16th, 2020.

integrated them into ECLAIR. Then, we evaluated the differences between [53] and our approach experimentally. The research questions that we seek to answer are the following:

- RQ4: What kind of issues may be caused by an unsound treatment of `math.h`/`cmath` functions?
- RQ5: How often do such issues occur in practice?
- RQ6: What is the impact of the two different approaches on the performances of the analyses?

We answer to RQ4 in Section 8.1, and to RQ5 and RQ6 in Section 8.2.

8.1 RQ4: issues caused by an unsound treatment of `math.h`/`cmath` functions

In this section, we demonstrate what is the type and severity of issues caused by an unsound treatment of `math.h`/`cmath` functions by means of another case study.

The following C function is an implementation of the Gauss Error Function, taken from [61].

```
1 float custom_ferf(float x) {
2   float sgn_x = signbit(x) ? -1.0F : 1.0F;
3   return (2/sqrtf(M_PI[16])) * sgn_x * sqrtf(1 - expf(-(x*x)))
4           * (sqrtf(M_PI)/2
5              + (31.0/200.0)*expf(-(x*x))
6              + (3481.0/8000.0)*expf(-2*(x*x)))
7 }
```

A natural question that arises by looking at this code is whether it may generate NaNs. Consider, e.g., the term `sqrtf(1 - expf(-(x*x)))`. Since the square root function is undefined for negative arguments, its `math.h`/`cmath` implementation `sqrtf` returns a NaN in such a case. But can the result of `1 - expf(-(x*x))` be negative? A very elementary property of the real exponential function $\exp(x)$ is that $\exp(x) \leq 1$ for $x \leq 0$, so, considering that `-(x*x)` ≤ 0 for any finite `x`, the answer should be no. The same argument holds for a correctly rounded implementation of the `expf` function, even with rounding mode 'up' (i.e., towards positive infinity). This is, indeed, the conclusion reached by applying constraint solving based on the propagators of [53] on this code.

Unfortunately, the implementation of the `expf` function on the `xps` machine presents a large glitch surrounding 0, when executed with rounding mode 'up'. At the left border of this glitch, the function returns a value greater than 1, even with a negative argument. E.g., it returns 1.000001 when evaluated on -2^{-149}. Our filtering algorithms take this imprecision into account, and ECLAIR correctly points out that an input value of `x` $= -2^{-149}$ causes the call to the `sqrtf` function to return a NaN, which is propagated in the subsequent computations, and returned by the above function.

Our approach handles the quirks of `math.h`/`cmath` function implementations soundly and precisely, enabling the discovery of subtle bugs, which are nearly impossible to find manually, and that result in false negatives with other state-of-the-art approaches.

8.2 Comparison on real-world code

We run ECLAIR equipped with both our interval refinement algorithms and those of [53] on the self-assembled benchmark we described in Section 7.3. Again, the purpose of the analysis is to prove or disprove the possible occurrence of finite-to-infinite and numeric-to-NaN transitions, and it has been run for rounding mode 'near'. We report the results of this evaluation in Table 3, and we comment on them with respect to the research questions below.

8.2.1 RQ5: Frequency of issues caused by unsoundness. Table 3 shows that, in all benchmark groups, our propagators find more anomalies than those of [53]. Relatively few anomalies are missed by [53] in GSL and AxBench, while many more are missed in paparazzi.c and our test suite. Most

[16] `M_PI` is a floating-point approximation of π in `math.h`/`cmath`.

Table 3. Comparison of benchmark data for the xps machine. For each file and benchmark group we report the number of potential anomalies discovered, the total time taken by the analysis (T), the time taken by propagators for `math.h`/`cmath` functions (T_m), for both our filters, and those of [53]. We report the number of potential anomalies that have been proved possible (g), those that were proved unfeasible (u), and those for which there was a timeout (t).

Benchmark	Our propagators					[53]				
	g	u	t	T (s)	T_m (ms)	g	u	t	T (s)	T_m (ms)
GSL	**420**	4205	**122**	2266.86	9167.68	419	4205	123	**2255.85**	**4841.98**
AxBench	**27**	390	**17**	**557.78**	**703.27**	24	390	20	679.80	1279.15
paparazzi.c	**9**	**243**	**4**	**27.22**	759.97	6	244	6	30.58	**663.01**
Test suite	**464**	**1486**	**45**	**135.30**	4274.81	445	1503	47	138.35	**3309.77**
Total	**920**	**6324**	**188**	**2987.16**	14905.72	894	6342	196	3104.59	**10093.91**

importantly, in the two latter cases 18 anomalies are mistakenly declared impossible by the analysis based on [53]. Such anomalies may, instead, occur, because the test inputs generated by ECLAIR with our propagators actually trigger them. Overall, the analysis based on [53] misses 26 anomalies, deeming 18 of them unfeasible, and timing out on 8 of them.

Since the proportion of missed anomalies is relatively low, the propagators of [53] may be used for the purpose of test-case generation. However, even this little miss-rate is unacceptable in the context of program verification, especially for safety-critical code.

8.2.2 RQ6: Performance comparison. The time taken by the propagators for `math.h`/`cmath` functions only (T_m) is consistently lower for the approach of [53]. Interestingly, the total time of the analyses (T) shows the opposite. For the GSL group, [53] performs better, but only for a few seconds. For all other groups, the analyses based on our algorithms are faster than those based on [53]. For the AxBench benchmarks, the difference reaches one minute.

We manually inspected the way the constraint solving process converges for a few cases in which this difference in analysis time is most pronounced. Our explanation for this behavior is that the higher precision of our algorithms favorably influences the overall constraint solving process, which converges in fewer iterations. In particular, the fact that our indirect propagation algorithms are based on a dichotomic search allows them to prune more values from variable domains in each iteration, decreasing the total number of iterations needed. Of course, this insight is limited to the cases that we analyzed, but we believe it can be generalized consistently.

8.3 Related Work

Automated test-case generation for floating-point computations has also been widely studied. The work of [54], which originated the field of *search-based* testing, searches the input space of the program by numerically maximizing an objective function that represents a given test adequacy criterion. CoverMe [27] performs its input-space exploration by minimizing a function representing the code path to be tested through *constrained programming*. Symbolic execution [44] is also widely used for structure-based test data generation. KLEE [16, 50] is a LLVM-based symbolic execution engine that leverages several SMT solvers to generate test-cases with high code coverage. CORAL [12] solves constraints generated by symbolic execution with several heuristic strategies combined with interval-based solving. Dynamic Symbolic Execution [29] (DSE) combines symbolic execution with concrete execution of the program. Runtime values gathered from the concrete executions are used when constraint solving fails, e.g. when it timeouts or encounters unsupported expressions. CUTE [62] uses DSE to generate test data. Other tools combine search-based approaches with DSE. FloPSy [48] combines DSE with search-based techniques such as the Alternating Variable

Method [45], and evolution strategies. Austin [47] uses symbolic execution combined with heuristic search-based strategies.

The main drawbacks of pure symbolic execution come from the limitations of the underlying constraint solver, which may timeout when excessively complex non-linear constraint systems are involved. Search-based methods such as [54] and CoverMe generally perform better in this respect, and tools such as CORAL, FloPSy and Austin combine them in different ways with symbolic execution to overcome such issues. Moreover, constraint solvers often do not fully support floating-point arithmetic, and approximate it to real arithmetic, which is unsound [13]. This is the case for CORAL. Only recently, SMT solvers acquired the ability to soundly solve floating-point constraints [14], which can be exploited by tools based on them, such as KLEE. However, `math.h`/`cmath` library functions are always treated as uninterpreted functions, which hinders the accuracy of constraint solving. Tools based on DSE, namely CUTE, FloPSy and Austin, use actual program executions to provide concrete values for such functions. The approach presented in this paper enables solving floating-point constraints soundly, and in a fully static way, i.e. without the need to concretely execute the program, even in the presence of `math.h`/`cmath` functions. Combining it with search-based techniques to reduce timeouts due to constraint complexity may be an interesting line of future work, although only adequate for testing, and not verification.

Much work has been done on the complementary goal of statically determining the accuracy of floating-point computations. Fluctuat [34] and PRECiSA [57] estimate error bounds by means of abstract interpretation, FPTaylor [63] uses symbolic Taylor expansions, Real2Float [52] uses semidefinite programming, Rosa [22] uses a SMT solver combined with a novel technique based on Lipschitz continuity, and Daisy [21] combines many of the earlier approaches. Gappa [23] uses interval arithmetic and forward error analysis to prove error bounds. All such tools are not concerned with the detection of floating-point exceptions or the proof of arbitrary assertions, but rather with estimating the error affecting floating-point with respect to real-valued computations. Among them, only Fluctuat and Daisy support the direct analysis of C code. The main similarity between the work presented in this paper and the above tools is the need for an estimation of program variable domains. However, none of such tools does a treatment of `math.h`/`cmath` function implementations as precise as the one presented in this paper, as they use real-valued approximations thereof.

9 DISCUSSION AND FURTHER WORK

In this section we discuss some aspects of the applicability of our proposal, which immediately suggest directions for further work.

9.1 Access to the Target Library

For the purposes of true verification, our approach requires execution access to the mathematical library used by the target. When the host and the target computer coincide, i.e. when the target can run the verifier code, this is no problem. Alternatively, the host computer might provide an implementation that is fully equivalent to the one used on the target: this is the case, e.g., on targets where floating-point support is implemented in software. In other cases, an emulator must be used.

This can be seen as the major drawback of our approach. However, in our opinion the question should be put in the following terms: in order to verify a piece of code properly using library functions against, say, the absence of run-time anomalies, the library functions have to be fully specified. If a specification of the form “all functions are POSIX-compliant [41] and compute correctly-rounded results” is available, then we have no problem. Otherwise there really is no other way than supplementing the partial specification available with the missing bits: providing

execution access to the library during the analysis along with correct bounds on the size of the glitches might well be the less expensive option.

9.2 Obtaining Glitch Data

The other requirement of the approach concerns the availability of (possibly imprecise) information about glitches. Some ways to obtain such information are the topics of the next sections.

9.2.1 Brute Force. For single-precision IEEE 754 (unary) functions, collection of precise glitch data by brute force is perfectly feasible. The glitch data presented in this paper have been obtained by running a program that computes each function on each value of its domain, in ascending order. If the function is quasi-isotonic, each time a value lower than the previous one is found, the program marks the beginning of a glitch, and measures its width and depth by incrementing appropriate counters until the end of the glitch is found (i.e., until the function yields a value greater than or equal to the one recorded at the beginning of the glitch). The program only keeps track of the maximum width and depth of the encountered glitches, of their number, of the input value in which the first glitch starts, and the one in which the last one ends. This is all the data needed by the algorithms of Section 5.

For the 25 functions studied in this paper, this procedure takes less than two hours on ordinary hardware. With less powerful CPUs used on embedded systems, it might take ten or twenty times as much. This is not really a problem as glitch data must be collected only once for each implementation of the `math.h`/`cmath` functions. And, especially in safety-critical sectors, the mathematical (and other) libraries will rarely if ever be changed once they have been selected. Of course, this method cannot be used for double-precision or extended-precision implementations of the functions.

9.2.2 Precision Guarantees. When the mathematical library comes equipped with information on the maximum errors for each function (see, e.g., the HA and LA accuracy modes of the Intel Math Kernel Library), such information can be used to determine safe approximations of the required glitch parameters. Recent developments enable the automatic proof of error bounds of `math.h`/`cmath` implementations [38, 39, 49]. We can thus expect more and more implementations will provide provably correct error bounds that we can directly exploit for program verification. In fact, given a function and an architecture, the maximum error is measured in ULP and can be used as an upper bound for the maximal depth of the glitches w_M. Given an interval $[x_l, x_u]$, the cardinality of the floating-point interval $[x_l, x_u]$ is an upper bound to the maximum width of the glitches and, of course x_l and x_u are safe approximations of where the glitches begin and end. Finally, setting $n_g > 1$ allows us to call the indirect propagation algorithms to refine the interval $[x_l, x_u]$ of the function domain with respect to a given interval $[y_l, y_u]$ of the function range. Even with such rough information on the glitches, the algorithms would allow us to refine the interval $[x_l, x_u]$ using the logarithmic searches (`logsearch_lb`) and returning $[x'_l, x'_u] \subseteq [x_l, x_u]$ such that $f(x'_l)$ is smaller than y_l by more than w_M ULPs and $f(x'_u)$ is bigger than y_u by more than w_M ULPs. Therefore, the cardinality of the resulting refined interval is related to the growth speed of the considered function. For future work, we intend to investigate how information on the maximum error, coupled with the knowledge of the function and of the interval to be refined, allows computing sound and tight bounds to the width of glitches on that interval.

9.2.3 Analysis of the Implementation. Transcendental functions are usually implemented with polynomial approximations. When speed is more important than precision, such computations are carried out in the same floating-point format as the the function being approximated; otherwise extended precision can be used to reduce the error. Whereas the total error accumulation can be bounded, the ordinary techniques used do not allow us to relate the rounding errors for different

input values to one another. So, if the error bound is small enough to imply monotonicity, fine. Otherwise, as things stand today, we are left with the approach of the previous section. However, we conjecture that (some of) the implementation algorithms can be analyzed with other techniques in order to obtain mode precise glitch data: this is another direction for future work.

9.3 Supported Functions in `math.h/cmath`

For functions that are quasi-monotonic, namely `acos`, `acosh`, `asin`, `asinh`, `atan`, `atanh`, `cbrt`, `cosh`, `erf`, `exp`, `exp10`, `exp2`, `expm1`, `log`, `log10`, `log1p`, `log2`, `sinh`, `sqrt`, `tanh`, our approach enables verification in their full domain, provided that the required data on glitches, described in Section 4, are correct and conservative for the `math.h/cmath` implementation in use. Such data can be gathered as described in Section 9.2. If glitches are sufficiently narrow and shallow, the algorithms of Section 5 are able to refine variable domains in a very precise manner, often optimal, which guarantees a fast convergence of the constraint solving process. Correctly rounded implementations of library functions also fall into this case. If glitches are too large, the results may be less precise, causing slower convergence of constraint solving, but they are still correct. In practice, this may result in having more timeouts, i.e. *don't knows*, during verification, but may never cause false positives or false negatives. Whether glitches are sufficiently narrow and shallow depends on the parameter t of the algorithms of Section 5, which controls the maximum length of linear searches. If their maximum width, w_{M}, is lower than t, then our algorithms return maximally precise results. The choice of the value of t is thus a trade-off between the computational efficiency of the algorithms and the precision of their results. In our experiments, we found that a value of $t = 20$ is satisfactory, as it allows us to get precise results on most implementations, while maintaining acceptable performances.

If glitch data is not precise, our approach cannot be used for verification, as it cannot reliably state that a certain assertion is always satisfied. It can, however, prove that an assertion does not hold, by finding a counter-example, and it can also be used for test-data generation (cf. Section 9.4).

For functions presenting natural monotonicity changes, namely `tgamma`, `lgamma`, `sin`, `cos`, and `tan`, verification applies only for a restricted part of the domain. Inside this domain, the same considerations about glitch data we made for the rest of the functions apply. For `tgamma`, `lgamma`, such part of the domain is fixed, and it is $[2, +\infty]$. For trigonometric functions, this part of the domain can be chosen by the user, and such choice is mostly influenced by performance considerations. In fact, each quasi-monotonic branch of the graphs of such functions must be analyzed separately by the propagation algorithms, as described in Section 6. Thus, this verification domain must be chosen to contain a reasonable amount of quasi-monotonic branches. In our experiments, we found that a reasonable value for such domain is $[-16, +16]$. Once such domain has been chosen, verification can be carried out by proving that the inputs to trigonometric functions never fall out of this domain, by introducing appropriate assertions. As we noted in Section 1, this is generally not a significant limitation, as in most applications the use of trigonometric functions with arguments of excessive magnitude is discouraged, due to their ULP getting excessively large. In general, since floating-point numbers are most dense around 0, most of the values they return can be found in a limited domain, which makes test generation always feasible with such functions.

So far, we described how to deal with 75 (considering `float`, `double`, and `long double` versions) of the standard C/C++ mathematical functions: but there are many others. Several of them are not problematic, as they are fully specified and their treatment poses no problem (e.g., `round`, `trunc`, `floor`, `ceil`, `fma`, `fabs`, `next`, ...). In future work we will focus on the remaining functions, i.e. functions with two inputs, such as `atan2` and `pow`, and complex functions.

9.4 Verification vs. Test-Data Generation

In our experimental evaluation we performed model checking, proving important properties of the code at hand. For such results to be achieved, a the conditions analyzed in Section 9.3 must all apply. When such conditions do not hold, the correctness of constraint propagation is not guaranteed, leading to the following issues:

(1) when the variable domains reach quiescence, they do not contain all existing solutions, but some are missing;
(2) the final variable domains contain values that are not solutions.

Due to issue (1), it is not possible to rule out some program behavior when a domain becomes empty during the constraint solving process. However, even when issue (2) occurs, as far as at least one solution is contained in the domains, another important correctness-ensuring technique is possible: automated test-data generation.

The approach we use enables white-box program testing, in the form of symbolic execution-based test data generation [4]. First, a constraint system is built for each execution path selected by a code-coverage criterion [71]. Then, the constraint-solving engine is launched. When all variable domains reach quiescence, their contents may be used as test data that cause the execution of the path. An instrumented version of the code to test can be executed with such input values, in order to make sure they actually cause the requested execution path to be followed, ruling out issue (2).

This procedure can also be employed in testing approaches that mix constraint solving with other techniques: *concolic* testing [62], or white-box *fuzzing* [29, 30, 64]. Indeed, such techniques present an improvement with respect to pure, black-box random testing, due to their greater capability of finding test inputs that trigger specific parts of the code.

9.5 Better Labeling Strategies for Constraint-Based Reasoning

The constraint-solving algorithms of Section 3.2 operate by interleaving *constraint propagation*, in which constraints are used to refine variable domains (intervals or multi-intervals in our case), and *labeling*, whereby a variable is chosen and its domain is partitioned into two or more subsets, each of which is explored separately. It is the second process that drives the first one: when constraint propagation goes to *quiescence*, i.e., when no further refinement of the domains can be achieved, labeling splits the domain of a chosen variable, triggering a new phase of constraint propagation. This goes on until a solution has been found or one of the domains becomes empty.

In this paper we only dealt with constraint propagation, but different labeling strategies have an enormous influence on performance. Unfortunately, there is no such a thing as *the* good labeling strategy: it is a matter of heuristics, and strategies that work well for one problem may still work badly for another. Test input generation and model checking give rise to constraint problems of a different nature: while the latter is very often over-constrained (i.e., there are few or no solutions at all, as the program exhibits very few or no run-time anomalies), this is not the case for the former (e.g., a function made of a single basic block can be covered by a single test input chosen more or less at random). Thus, different tasks can profit from the choice of different labeling strategies.

During the experimental evaluation, we strongly felt that the current labeling strategy employed by ECLAIR can be significantly enhanced by defining heuristics that take into account how variables are constrained by invocations to such functions. Work on these new heuristics is ongoing.

10 CONCLUSION

There is a popular quotation in the software verification and validation community, whereby "Without a specification, a system cannot be right or wrong, it can only be surprising!"[17] This

[17] Paraphrased from [70].

captures quite well the current state of affairs for C/C++ software that uses the functions declared in the standard `math.h`/`cmath` header files. Despite the progress made on the development of correctly rounded functions,[18] all implementations in widespread use, especially in the world of embedded systems, offer little or no guarantees about the computed results. As a consequence, the verification of programs using such functions is always painful and expensive and, for these reasons, more often than not it is only partially performed through testing. As the search space can be huge, testing can only cover a tiny fraction of all the possible value combinations: this cannot exclude the manifestation of unexpected results, certainly not with the level of confidence that is required for mission- and safety-critical applications.

The aim of this work is to improve upon the current situation now, i.e., without waiting for the wider adoption of correctly-rounded implementations. While such adoption is generally desirable and will certainly take place, at some stage and in some application domains, it is not clear whether correctly-rounded implementations can meet the efficiency criteria of all application domains, particularly in the field of embedded systems. Studying different implementations of the standard C/C++ mathematical functions, we realized that what they have in common is a piecewise quasi-monotonicity property: monotonicity is either preserved or only perturbed by small and, on average, not too frequent "glitches." Based on this observation, we developed direct and indirect propagation algorithms for interval refinement. These algorithms can be integrated into abstract interpreters, model checkers and automatic test input generators based on constraint propagation.

The techniques proposed here are now used in the C/C++ semantic analysis components of the ECLAIR software verification platform and the initial experiences are quite positive. We can now properly verify the absence of run-time anomalies for code using the C/C++ standard functions that, before, was completely out of reach. Verification in the strong sense is only feasible modulo the possibility of bounding the size of glitches (this can always be done for the single-precision functions) and the ability to query the underlying implementation of the functions during the analysis. For the cases where the first condition cannot be guaranteed, we can still detect many definite program issues, even though we cannot draw conclusions from the fact issues have not been found. When the second condition cannot be met, it may still be possible to use a reference implementation with significant commonalities with the target implementation (the case where libraries for different architectures are derived from the same code base is quite common), and we can nonetheless detect high-severity, possible program issues.

We cannot yet claim that the problem of the verification of C/C++ programs using the standard mathematical functions has been solved, as much remains to be done. However, we believe the present work is a definite step in the right direction, and one that has the potential of improving, starting from today, the current state of the art.

ACKNOWLEDGMENTS

We are grateful to Arnaud Gotlieb (Simula Research Laboratory, Norway, and INRIA – Rennes, France) and Claude Michel (INRIA – Sophia Antipolis, France) for the fruitful discussions we had on the subject of this paper. We are are also grateful to Marcel Beemster (Solid Sands, The Netherlands) for the discussions we had on the subject of testing mathematical libraries for the C language. Alessandro Zaccagnini (University of Parma, Italy) helped us defining rough inverses for the Gamma functions. Thanks also to Patricia M. Hill (BUGSENG srl, Italy) for her work on the ECLAIR platform that greatly facilitated the experimental evaluation of this work. We also thank Dino Mandrioli (Politecnico di Milano) for his careful reading of a preliminary version of this paper, and Michele Guerriero (Politecnico di Milano) for the discussions. Finally, we would like

[18] See, e.g., the very interesting MetaLibm project at http://www.metalibm.org/, last accessed on July 16th, 2020.

to express our gratitude to the editor and to the anonymous reviewers for helping us improving the readability of the paper.

REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. 2007. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson, London, UK.

[2] Analog Devices 2015. *C/C++ Compiler and Library Manual for Blackfin Processors.* Analog Devices. Revision 1.5.

[3] R. Bagnara, A. Bagnara, F. Biselli, M. Chiari, and R. Gori. 2019. Correct Approximation of IEEE 754 Floating-Point Arithmetic for Program Verification. Report `arXiv:1903.06119 [cs.PL]`. Available at http://arxiv.org/.

[4] R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb. 2013. Symbolic Path-Oriented Test Data Generation for Floating-Point Programs. In *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation.* IEEE Press, Luxembourg City, Luxembourg, 1–10.

[5] R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb. 2016. Exploiting Binary Floating-Point Representations for Constraint Propagation. *INFORMS Journal on Computing* 28, 1 (2016), 31–46.

[6] R. Bagnara, M. Chiari, R. Gori, and A. Bagnara. 2016. A Practical Approach to Interval Refinement for `math.h`/`cmath` Functions. Report `arXiv:1610.07390 [cs.PL]`. Available at http://arxiv.org/.

[7] R. Bagnara, P. M. Hill, and E. Zaffanella. 2008. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *Science of Computer Programming* 72, 1–2 (2008), 3–21. http://bugseng.com/products/ppl/documentation/BagnaraHZ08SCP.pdf

[8] E. T. Barr, T. Vo, V. Le, and Z. Su. 2013. Automatic detection of floating-point exceptions. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*, R. Giacobazzi and R. Cousot (Eds.). ACM Press, Rome, Italy, 549–560.

[9] M. S. Belaid, C. Michel, and M. Rueher. 2012. Boosting Local Consistency Algorithms over Floating-Point Numbers. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (Lecture Notes in Computer Science, Vol. 7514)*, M. Milano (Ed.). Springer-Verlag, Berlin, Québec City, Canada, 127–140.

[10] F. Benhamou, D. McAllester, and P. Van Hentenryck. 1994. CLP(Intervals) Revisited. In *Logic Programming: Proceedings of the 1994 International Symposium (MIT Press Series in Logic Programming)*, M. Bruynooghe (Ed.). The MIT Press, Ithaca, NY, USA, 124–138.

[11] F. Benhamou and W. J. Older. 1997. Applying Interval Arithmetic to Real, Integer, and Boolean Constraints. *Journal of Logic Programming* 32, 1 (1997), 1–24.

[12] M. Borges, M. d'Amorim, S. Anand, D. Bushnell, and C. S. Pasareanu. 2012. Symbolic Execution with Interval Solving and Meta-heuristic Search. In *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation.* IEEE Computer Society, Montreal, Canada, 111–120.

[13] B. Botella, A. Gotlieb, and C. Michel. 2006. Symbolic Execution of Floating-point computations. *Software Testing, Verification and Reliability* 16, 2 (2006), 97–121.

[14] Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. 2013. Interpolation-Based Verification of Floating-Point Programs with Abstract CDCL. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7935).* Springer-Verlag, Berlin, Berlin, Heidelberg, 412–432. https://doi.org/10.1007/978-3-642-38856-9_22

[15] L. Burdy, J.-L. Dufour, and T. Lecomte. 2012. The B Method Takes Up Floating-Point Numbers. In *Proceedings of the 6th International Conference & Exhibition on Embedded Real Time Software and Systems (ERTS 2012).* SEE, Toulouse, France, Article 5C.2, 7 pages. Available at http://web1.see.asso.fr/erts2012/Site/0P2RUC89/5C-2.pdf.

[16] C. Cadar, D. Dunbar, and D. R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, R. Draves and R. van Renesse (Eds.). USENIX Association, San Diego, California, USA, 209–224.

[17] L. A. Clarke and D. J. Richardson. 1985. Applications of symbolic evaluation. *Journal of Systems and Software* 5, 1 (1985), 15–35.

[18] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzè. 2001. Using symbolic execution for verifying safety-critical systems. In *Proceedings of the 8th European Software Engineering Conference.* ACM Press, Vienna, Austria, 142–151.

[19] P. Cousot and R. Cousot. 1976. Static Determination of Dynamic Properties of Programs. In *Proceedings of the Second International Symposium on Programming*, B. Robinet (Ed.). Dunod, Paris, France, Paris, France, 106–130. http://www.di.ens.fr/~cousot/publications.www/CousotCousot-ISOP-76-Dunod-p106--130-1976.pdf

[20] P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages.* ACM Press, Los Angeles, CA, USA, 238–252.

[21] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian. 2018. Daisy — Framework for Analysis and Optimization of Numerical Programs. In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2018) (Lecture Notes in Computer Science, Vol. 10805)*, D. Beyer and M. Huisman (Eds.). Springer, Thessaloniki, Greece, 270–287. Tool paper.

[22] E. Darulova and V. Kuncak. 2017. Towards a Compiler for Reals. *ACM Transactions on Programming Languages and Systems* 39, 2 (2017), 8:1–8:28.

[23] M. Daumas and G. Melquiond. 2010. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Software* 37, 1 (2010), 2:1–2:20.

[24] E. Davis. 1987. Constraint Propagation with Interval Labels. *Artificial Intelligence* 32, 3 (1987), 281–331.

[25] T. Denmat, A. Gotlieb, and M. Ducassé. 2007. Improving Constraint-Based Testing with Dynamic Linear Relaxations. In *Proceedings of the 18th IEEE International Symposium on Software Reliability (ISSRE 2007)*. IEEE Computer Society, Trollhättan, Sweden, 181–190.

[26] A. Di Franco, H. Guo, and Cindy C. Rubio-González. 2017. A Comprehensive Study of Real-world Numerical Bug Characteristics. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Urbana-Champaign, IL, USA, 509–519.

[27] Z. Fu and Z. Su. 2017. Achieving high coverage for floating-point code via unconstrained programming. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*, A. Cohen and M. T. Vechev (Eds.). ACM Press, Barcelona, Spain, 306–319.

[28] Fujitsu Semiconductor 2013. *FR Family SOFTUNE C/C++ Compiler Manual for V6* (eighth ed.). Fujitsu Semiconductor.

[29] P. Godefroid, N. Klarlund, and K. Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005)*, V. Sarkar and M. W. Hall (Eds.). ACM, Chicago, IL, USA, 213–223.

[30] P. Godefroid, M. Y. Levin, and D. A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2008)*. The Internet Society, San Diego, California, USA, Article 4.1, 16 pages.

[31] P. Godefroid and K. Sen. 2018. Combining Model Checking and Testing. In *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem (Eds.). Springer, Cham, Switzerland, 613–649.

[32] A. Gotlieb, B. Botella, and M. Rueher. 1998. Automatic Test Data Generation Using Constraint Solving Techniques. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98)*. ACM Press, Clearwater Beach, Florida, USA, 53–62.

[33] A. Gotlieb, B. Botella, and M. Rueher. 2000. A CLP Framework for Computing Structural Test Data. In *Proceedings of the First International Conference on Computational Logic (CL 2000) (Lecture Notes in Artificial Intelligence, Vol. 1861)*, J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey (Eds.). Springer, London, UK, 399–413.

[34] E. Goubault, M. Martel, and S. Putot. 2002. Asserting the Precision of Floating-Point Computations: A Simple Abstract Interpreter. In *Pogramming Languages and Systems — 11th European Symposium on Programming (ESOP 2002) (Lecture Notes in Computer Science, Vol. 2305)*, D. Le Métayer (Ed.). Springer, Grenoble, France, 209–212.

[35] E. Goubault and S. Putot. 2006. Static Analysis of Numerical Algorithms. In *Static Analysis: Proceedings of the 13th International Symposium (Lecture Notes in Computer Science, Vol. 4134)*, K. Yi (Ed.). Springer-Verlag, Berlin, Seoul, Korea, 18–34.

[36] Green Hills 2013. *MULTI: Building Applications for Embedded Power Architecture.* Green Hills. Version 2013.1.

[37] Green Hills 2013. *MULTI: Building Applications for Embedded V850 and RH850.* Green Hills. Version 2013.5.

[38] J. Harrison. 2000. Floating Point Verification in HOL Light: The Exponential Function. *Formal Methods in System Design* 16, 3 (2000), 271–305.

[39] J. Harrison. 2000. Formal Verification of Floating Point Trigonometric Functions. In *Formal Methods in Computer-Aided Design*, W. A. Hunt and S. D. Johnson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 254–270.

[40] IEEE Computer Society 2008. *IEEE Standard for Floating-Point Arithmetic* (IEEE Std 754-2008 (revision of IEEE Std 754-1985) ed.). IEEE Computer Society.

[41] IEEE Computer Society and The Open Group 2013. *Standard for Information Technology— Portable Operating System Interface (POSIX®), Base Specifications, Issue 7* (IEEE Std 1003.1, 2013 (incorporates IEEE Std 1003.1-2008, and IEEE Std 1003.1-2008/Cor 1-2013) ed.). IEEE Computer Society and The Open Group, New York, NY, USA.

[42] International Organization for Standardization. 2011. *ISO/IEC 9899:2011: Programming Languages — C.* International Organization for Standardization, Geneva, Switzerland. 701 pages.

[43] International Organization for Standardization. 2014. *ISO/IEC 14882:2014(E): Programming languages — C++.* International Organization for Standardization, Geneva, Switzerland.

[44] J. C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

[45] B. Korel. 1990. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering* 16, 8 (1990), 870–879.

[46] P. Lacan, J. N. Monfort, L. V. Q. Ribal, A. Deutsch, and G. Gonthier. 1998. ARIANE 5 — The Software Reliability Verification Process. In *DASIA 98: Data Systems in Aerospace (ESA Special Publication, Vol. 422)*, B. H. Kaldeich-Schürmann (Ed.). European Space Agency, Athens, Greece, 201.

[47] K. Lakhotia, M. Harman, and H. Gross. 2010. AUSTIN: A Tool for Search Based Software Testing for the C Language and Its Evaluation on Deployed Automotive Systems. In *Proceedings of the 2nd International Symposium on Search Based Software Engineering (SSBSE '10)*. IEEE Computer Society, Benevento, Italy, 101–110.

[48] K. Lakhotia, N. Tillmann, M. Harman, and J. De Halleux. 2010. FloPSy: Search-Based Floating Point Constraint Solving for Symbolic Execution. In *Proceedings of the 22nd IFIP WG 6.1 International Conference on Testing Software and Systems (Lecture Notes in Computer Science, Vol. 6435)*. Springer-Verlag, Berlin, Heidelberg, Natal, Brazil, 142–157.

[49] W. Lee, R. Sharma, and A. Aiken. 2018. On Automatically Proving the Correctness of `math.h` Implementation. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 47:1–47:32.

[50] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zähl, and K. Wehrle. 2017. Floating-point symbolic execution: a case study in n-version programming. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Computer Society, Urbana-Champaign, IL, USA, 601–612.

[51] S. Loosemore, R. M. Stallman, R. McGrath, A. Oram, and U. Drepper. 2016. *The GNU C Library Reference Manual* (version 2.23 ed.). Free Software Foundation, Inc., Boston, MA, USA.

[52] V. Magron, G. A. Constantinides, and A. F. Donaldson. 2017. Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Trans. Math. Software* 43, 4 (2017), 34:1–34:31.

[53] C. Michel. 2002. Exact Projection Functions for Floating Point Number Constraints. In *Proceedings of the 7th International Symposium on Artificial Intelligence and Mathematics*. Rutgers University, Fort Lauderdale, FL, USA, 11.

[54] W. Miller and D. L. Spooner. 1976. Automatic Generation of Floating-Point Test Data. *IEEE Transactions on Software Engineering* 2, 3 (1976), 223–226.

[55] A. Miné. 2004. Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors. In *Programming Languages and Systems: Proceedings of the 13th European Symposium on Programming (Lecture Notes in Computer Science, Vol. 2986)*, D. Schmidt (Ed.). Springer-Verlag, Berlin, Barcelona, Spain, 3–17.

[56] D. Monniaux. 2008. The Pitfalls of Verifying Floating-point Computations. *ACM Transactions on Programming Languages and Systems* 30, 3 (2008), 12:1–12:41.

[57] M. Moscato, L. Titolo, A. Dutle, and C. A. Mu noz. 2017. Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In *Proceedings of the 36th International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2017) (Lecture Notes in Computer Science, Vol. 10488)*, S. Tonetta and E. Schoitsch end F. Bitsch (Eds.). Springer, Trento, Italy, 213–229.

[58] J.-M. Muller. 2005. *On the Definition of* $\mathrm{ulp}(x)$. Rapport de recherche 5504. INRIA.

[59] J.-M. Muller. 2016. *Elementary Functions: Algorithms and Implementation* (3rd ed.). Birkhäuser, Basel, Switzerland.

[60] K. C. Ng. 1992. *Argument Reduction for Huge Arguments: Good to the Last Bit.* Technical Report. SunPro – A Sun Microsystems, Inc. Business.

[61] H. Schöpf and P. Supancic. 2014. On Bürmann's Theorem and Its Application to Problems of Linear and Nonlinear Heat Transfer and Diffusion. *The Mathematica Journal* 16 (2014), 44.

[62] K. Sen, D. Marinov, and G. Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM Press, Lisbon, Portugal, 263–272.

[63] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan. 2018. Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Transactions on Programming Languages and Systems* 41, 1 (2018), 2:1–2:39.

[64] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS 2016)*, Vol. 16. The Internet Society, San Diego, CA, USA, 1–16.

[65] Texas Instruments 2018. *ARM Optimizing C/C++ Compiler v18.1.0.LTS User's Guide.* Texas Instruments. Literature Number: SPNU151R.

[66] P. Van Hentenryck, V. Saraswat, and Y. Deville. 1998. Design, Implementation, and Evaluation of the Constraint Language `cc(FD)`. *The Journal of Logic Programming* 37, 1–3 (1998), 139 – 164.

[67] X. Wu, L. Li, and J. Zhang. 2017. Symbolic Execution with Value-Range Analysis for Floating-Point Exception Detection. In *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC 2017)*, J. Lv, H. J. Zhang, M. Hinchey, and X. Liu (Eds.). IEEE Computer Society, Nanjing, China, 1–10.

[68] X. Wu, Z. Xu, D. Yan, T. Wu, J. Yan, and J. Zhang. 2016. The Floating-Point Extension of Symbolic Execution Engine for Bug Detection. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC 2016)*. IEEE Computer

Society, Hamilton, New Zealand, 265–272.
[69] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. 2017. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design & Test* 34, 2 (2017), 60–68.
[70] W. D. Young, W. E. Boebert, and R. Y. Kain. 1985. Proving a Computer System Secure. *The Scientific Honeyweller* 6, 2 (1985), 18—-27.
[71] H. Zhu, P. A. V. Hall, and J. H. R. May. 1997. Software unit test coverage and adequacy. *Comput. Surveys* 29, 4 (1997), 366–427.

A GLITCH DATA FOR OTHER IMPLEMENTATIONS OF `libm`

In this section, we provide additional data about the glitches in the single-precision functions for other implementations of the `math.h`/`cmath` mathematical functions. Table 4 lists, for each implementation, its identification code, which is used in the other tables, the CPU architecture, the operating system, and, where known, the `libm` version.

Note that what appears as a large glitch in Table 9 for function `tanf` rounded down on the `macbook` machine, is actually due to a clear bug in the range reduction algorithm used there.

Table 4. Glitch data: main characteristics of the tested implementations

id	CPU	OS	compiler	`libm` version
`alpha`	x86_64	Ubuntu 14.04	GCC 4.8.4	EGLIBC 2.19
`gcc110`	POWER7	Fedora 20	GCC 4.8.1	GNU libc 2.18
`gcc111`	POWER7	AIX 7	GCC 4.8.1	
`gcc112`	POWER7	Fedora 21	GCC 4.9.2	GNU libc 2.20
`gcc113`	AArch64	Ubuntu 14.04	GCC 4.8.4	EGLIBC 2.19
`igor`	x86_64	Fedora 12	GCC 4.4.4	GNU libc 2.11.2
`macbook`	x86_64	Mac OS X 10.10.5	LLVM 6.1.0	Libm-3086.1
`raspi`	ARMv6 + VFPv2	Raspbian Jessie	GCC 4.9.2	GNU libc 2.19
`xps`	x86_64	Ubuntu 19.10	GCC 9.2.1	GNU libc 2.30
`zoltan`	x86_64	Ubuntu 16.04	GCC 5.4.0	GLIBC 2.23

Table 5. Glitch data for the `alpha` machine

function	$D_{\min}$	D_{M}	near			up			down			zero		
			n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}
`acosf`	-1	1												
`acoshf`	1	∞							1	1	2	1	1	2
`asinf`	-1	1												
`asinhf`	$-\infty$	∞							2	1	2	2	1	2
`atanf`	$-\infty$	∞				1	1	10^8						
`atanhf`	-1	1							2	1	2	2	1	2
`cbrtf`	$-\infty$	∞	10^6	1	2	10^6	1	2	10^6	1	2	10^6	1	2
`coshf`	$-\infty$	∞	454	1	2	466	1	2	442	1	2	448	1	2
`erff`	$-\infty$	∞												
`expf`	$-\infty$	∞												
`exp10f`	$-\infty$	∞												
`exp2f`	$-\infty$	∞	1	1	2	3	1	2	2	1	2	1	1	2
`expm1f`	$-\infty$	∞												
`lgammaf`	2	∞	163	1	2	164	1	2	166	1	2	161	1	2
`logf`	0	∞												
`log10f`	0	∞												
`log1pf`	-1	∞							1	1	2	1	1	2
`log2f`	0	∞												
`sinhf`	$-\infty$	∞												
`sqrtf`	0	∞												
`tanhf`	$-\infty$	∞				1	1	2	2	1	3			
`tgammaf`	2	∞	10^4	2	3	10^4	2	4	10^4	3	3	10^4	3	4
`cosf`	-2^{23}	2^{23}												
`sinf`	-2^{23}	2^{23}												
`tanf`	-2^{23}	2^{23}												

Table 6. Glitch data for the gcc110/2/3 machines

function	$D_{\min}$	D_{M}	near			up			down			zero		
			n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}
acosf	-1	1												
acoshf	1	∞							1	1	2	1	1	2
asinf	-1	1												
asinhf	$-\infty$	∞							2	1	2	2	1	2
atanf	$-\infty$	∞				1	1	10^8						
atanhf	-1	1							2	1	2	2	1	2
cbrtf	$-\infty$	∞	10^6	1	2	10^6	1	2	10^6	1	2	10^6	1	2
coshf	$-\infty$	∞	456	1	2	462	1	2	442	1	2	448	1	2
erff	$-\infty$	∞												
expf	$-\infty$	∞												
exp10f	$-\infty$	∞												
exp2f	$-\infty$	∞	2	1	2									
expm1f	$-\infty$	∞												
lgammaf	2	∞												
logf	0	∞												
log10f	0	∞												
log1pf	-1	∞							1	1	2	1	1	2
log2f	0	∞												
sinhf	$-\infty$	∞												
sqrtf	0	∞												
tanhf	$-\infty$	∞				1	1	2	2	1	3			
tgammaf	2	∞	10^4	2	3	10^4	4	4	10^4	2	3	10^4	3	4
cosf	-2^{23}	2^{23}	10^4	1	3	10^4	1	3	10^4	1	3	10^4	1	3
sinf	-2^{23}	2^{23}												
tanf	-2^{23}	2^{23}												

Table 7. Glitch data for the gcc111 machine

function	$D_{\min}$	D_{M}	near			up			down			zero		
			n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}
acoshf	1	∞												
asinhf	$-\infty$	∞												
atanf	$-\infty$	∞												
atanhf	-1	1												
coshf	$-\infty$	∞												
erff	$-\infty$	∞												
lgammaf	2	∞												
sinhf	$-\infty$	∞												
tanhf	$-\infty$	∞				2	1	10^7						
tgammaf	2	∞												
cosf	-2^{23}	2^{23}												
sinf	-2^{23}	2^{23}												
tanf	-2^{23}	2^{23}												

Table 8. Glitch data for the igor machine

function	$D_{\min}$	D_{M}	near			up			down			zero		
			n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}
acosf	-1	1							1	1	10^{10}	1	1	10^{10}
acoshf	1	∞							1	1	2	1	1	2
asinf	-1	1												
asinhf	$-\infty$	∞							2	1	2	2	1	2
atanf	$-\infty$	∞												
atanhf	-1	1							2	1	2	2	1	2
cbrtf	$-\infty$	∞	10^6	1	2	10^6	1	2	10^6	1	2	10^6	1	2
coshf	$-\infty$	∞	454	1	2	466	1	2	442	1	2	448	1	2
erff	$-\infty$	∞												
expf	$-\infty$	∞												
exp10f	$-\infty$	∞												
exp2f	$-\infty$	∞	1	1	2	3	1	2	2	1	2	1	1	2
expm1f	$-\infty$	∞												
lgammaf	2	∞	163	1	2	164	1	2	166	1	2	161	1	2
logf	0	∞												
log10f	0	∞												
log1pf	-1	∞							1	1	2	1	1	2
log2f	0	∞												
sinhf	$-\infty$	∞												
sqrtf	0	∞												
tanhf	$-\infty$	∞				1	1	2	2	1	3			
tgammaf	2	∞	155	109	2	155	122	2	157	119	2	153	119	2
cosf	-2^{23}	2^{23}	10^4	1	3	10^4	1	3	10^4	1	3	10^4	1	3
sinf	-2^{23}	2^{23}												
tanf	-2^{23}	2^{23}												

Table 9. Glitch data for the macbook machine

function	$D_{\min}$	D_{M}	near			up			down			zero		
			n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}
acosf	-1	1												
acoshf	1	∞												
asinf	-1	1												
asinhf	$-\infty$	∞												
atanf	$-\infty$	∞				2	1	10^9	2	1	10^9	2	1	10^9
atanhf	-1	1												
cbrtf	$-\infty$	∞												
coshf	$-\infty$	∞												
erff	$-\infty$	∞												
expf	$-\infty$	∞												
exp10f	$-\infty$	∞												
exp2f	$-\infty$	∞												
expm1f	$-\infty$	∞												
lgammaf	2	∞												
logf	0	∞												
log10f	0	∞												
log1pf	-1	∞												
log2f	0	∞												
sinhf	$-\infty$	∞												
sqrtf	0	∞												
tanhf	$-\infty$	∞				1	1	10^7						
tgammaf	2	∞												
cosf	-2^{23}	2^{23}												
sinf	-2^{23}	2^{23}												
tanf	-2^{23}	2^{23}							1	10^{10}	10^6			

Table 10. Glitch data for the raspi machine

function	$D_{\min}$	D_{M}	near			up			down			zero		
			n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}
acosf	-1	1												
acoshf	1	∞							1	1	2	1	1	2
asinf	-1	1												
asinhf	$-\infty$	∞							2	1	2	2	1	2
atanf	$-\infty$	∞				1	1	10^8						
atanhf	-1	1							2	1	2	2	1	2
cbrtf	$-\infty$	∞	10^6	1	2	10^6	1	2	10^6	1	2	10^6	1	2
coshf	$-\infty$	∞	454	1	2	466	1	2	442	1	2	448	1	2
erff	$-\infty$	∞												
expf	$-\infty$	∞												
exp10f	$-\infty$	∞												
exp2f	$-\infty$	∞	1	1	2									
expm1f	$-\infty$	∞												
lgammaf	2	∞	163	1	2	164	1	2	166	1	2	161	1	2
logf	0	∞												
log10f	0	∞												
log1pf	-1	∞							1	1	2	1	1	2
log2f	0	∞												
sinhf	$-\infty$	∞												
sqrtf	0	∞												
tanhf	$-\infty$	∞				1	1	2	2	1	3			
tgammaf	2	∞	10^4	2	3	10^4	2	4	10^4	3	3	10^4	3	4
cosf	-2^{23}	2^{23}	10^4	1	3	10^4	1	3	10^4	1	3	10^4	1	3
sinf	-2^{23}	2^{23}												
tanf	-2^{23}	2^{23}												

Table 11. Glitch data for the zoltan machine

function	$D_{\min}$	D_{M}	near			up			down			zero		
			n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}	n_{g}	d_{M}	w_{M}
acosf	-1	1												
acoshf	1	∞							1	1	2	1	1	2
asinf	-1	1												
asinhf	$-\infty$	∞							2	1	2	2	1	2
atanf	$-\infty$	∞				1	1	10^8						
atanhf	-1	1							2	1	2	2	1	2
cbrtf	$-\infty$	∞	10^6	1	2	10^6	1	2	10^6	1	2	10^6	1	2
coshf	$-\infty$	∞	454	1	2	466	1	2	442	1	2	448	1	2
erff	$-\infty$	∞												
expf	$-\infty$	∞												
exp10f	$-\infty$	∞												
exp2f	$-\infty$	∞	1	1	2	3	1	2	2	1	2	1	1	2
expm1f	$-\infty$	∞												
lgammaf	2	∞	163	1	2	164	1	2	166	1	2	161	1	2
logf	0	∞												
log10f	0	∞												
log1pf	-1	∞							1	1	2	1	1	2
log2f	0	∞												
sinhf	$-\infty$	∞												
sqrtf	0	∞												
tanhf	$-\infty$	∞				1	1	2	2	1	3			
tgammaf	2	∞	10^5	4	3	10^5	4	3	10^5	4	3	10^5	4	3
cosf	-2^{23}	2^{23}												
sinf	-2^{23}	2^{23}												
tanf	-2^{23}	2^{23}												

B COMPUTATION OF UPPER BOUNDS

The algorithm we conceived for the computation of the upper bounds is substantially similar to the one for the lower bounds in its structure and functioning. It employs the same arguments to obtain glitch data, and it ends ensuring the post-condition predicates listed in Section 5.2.

Algorithm 6 consists of a first phase in which it tries to find a sub-interval inside the initial one, $[x_l, x_u]$, that is suitable for the bisection process. If such an interval cannot be found, it tries to determine quickly whether the equation $y = f(\mathbf{x})$ has a solution or not, compatible with the available glitch information.

Otherwise, the obtained interval is searched for an admissible upper bound by Algorithm 9, a dichotomic search that takes into account the possible presence of glitches. This algorithm is similar to `bisect_lb`, except for the fact that it needs to ensure that the function is strictly greater than y in the whole interval between the found upper bound and x_u. Another significant difference between the two algorithms is the behavior in the case where the function evaluated at the mid-point mid is greater than y. The computation should continue in the first half of the original interval, discarding the second one and making sure that the graph of the function is entirely above y in the latter. This means asserting that there are no glitches after mid which are deep enough to let the function reach y. Data such as α, ω and the maximum glitch depth d_{M} are almost always helpful in excluding this circumstance. Theoretically speaking, this is not always the case, e.g., if d_{M} is very high, or y is very close to $f(\mathrm{mid})$. The former case seldom occurs in practice, as noted in Section 4. The latter can occur in the last stages of the bisection process if the function increases very slowly. The experimental evaluation we performed, however, showed that this is not a substantial problem in practice. Anyway, should this circumstance occur, if the function has only one glitch and it is sufficiently narrow, it can be searched float-by-float. Otherwise, the whole right interval should be searched for glitches, which is clearly unfeasible, unless the intervals are very small.

The analogous issue with `bisect_lb` was making sure that mid was not inside a glitch, in order to exclude the left half of the interval. This situation could always be clarified if $w_{\mathrm{M}} < t$, by means of a linear search that could analyze the entire glitch. The same approach cannot clearly solve the analogous issue for the upper-bound algorithm since all the glitches after mid would need to be analyzed. This is the reason why the *correctness* post-condition is more demanding for the upper bound than for the lower bound. In particular, it ensures optimality of the bound if the function is monotonic, i.e., $n_{\mathrm{g}} = 0$. Otherwise, it finds an optimal upper bound if

- $n_{\mathrm{g}} = 1$: the function has one glitch only, and
- $w_{\mathrm{M}} < t$: it is not too large to perform a linear search, and
- the position of the glitch is known exactly, i.e., one of conditions $\alpha = \alpha^f$, $\omega = \omega^f$, or $\#[\alpha, \omega) > k \wedge k \leq t$ is true.

Nevertheless, the algorithm for the upper bound has the same order of complexity as `bisect_lb`.

An in-depth analysis of these algorithms is available in [6], where we give the proof of their correctness and more precise claims about complexity.

Algorithm 6 Indirect propagation: upper_bound$(f, y, [x_l, x_u], n_g, d_M, w_M, \alpha, \omega, f^i, s, t)$

Require: $f\colon \mathbb{F} \to \mathbb{F}$, $y \in \mathbb{F}$, $[x_l, x_u] \in \mathcal{I}_{\mathbb{F}}$, $n_g \geq n_g^f$, $d_M \geq d_M^f$, $w_M \geq w_M^f$, $\alpha \preccurlyeq \alpha^f$, $\omega \succcurlyeq \omega^f$, $n_g > 0 \implies (x_l \preccurlyeq \alpha \preccurlyeq \omega \preccurlyeq x_u)$, $f^i\colon \mathbb{F} \to \mathbb{F}$, $s, t \in \mathbb{N}$.

Ensure: (c) $u \in \mathbb{F}, r \in \{5,6,7,8,9\} \implies p_r(y, x_l, x_u, u)$

(p) $\Big(f(x_l) \preccurlyeq y \preccurlyeq f(x_u)$

$\wedge\Big(n_g = 0 \vee \big(n_g = 1 \wedge w_M < t \wedge \big(\alpha = \alpha^f \vee \omega = \omega^f \vee (\#[\alpha, \omega) > k \wedge k \leq t)\big)\big)\Big)$

$\implies r \in \{8, 9\}$

1: $i := \texttt{init}(y, [x_l, x_u], f^i)$; ▷ $x_l \preccurlyeq i \preccurlyeq x_u$
2: $(\mathrm{lo}, \mathrm{hi}) := \texttt{gallop\_ub}(f, y, [x_l, x_u], d_M, i)$;
3: ▷ $(x_l \preccurlyeq \mathrm{lo} \preccurlyeq \mathrm{hi} \preccurlyeq x_u) \wedge (x_l \prec \mathrm{lo} \implies y \succcurlyeq f(\mathrm{lo})) \wedge (x_u \succ \mathrm{hi} \implies \#[y, f(\mathrm{hi})) > d_M)$
4: **if** $f(\mathrm{hi}) \prec y$ **then**
5: $u := \texttt{findhi\_ub}(f, y, [x_l, x_u], n_g, d_M, w_M, \alpha, \omega, t)$;
6: $r := 6$; **return**
7: **else if** $f(\mathrm{hi}) = y$ **then**
8: $u := \mathrm{hi}$; $r := 9$; **return**
9: **end if**;
10: **if** $f(\mathrm{lo}) \succ y$ **then**
11: **if** $n_g = 0 \vee \#[y, f(\alpha)) > d_M$ **then**
12: $r := 5$; **return**
13: **else**
14: $(b, z) := \texttt{check\_glitch}(f, y, [x_l, x_u], n_g, d_M, w_M, \alpha, \omega, t, \mathrm{lo}, \mathrm{lo}, \mathrm{hi})$;
15: **if** $b = 0$ **then**
16: $r := 5$; **return**
17: **else if** $b = 1$ **then**
18: **if** $f(z) = y$ **then**
19: $u = z$; $r = 9$; **return**
20: **else**
21: $u = \mathrm{succ}(z)$; $r = 8$; **return**
22: **end if**
23: **else**
24: $u := \min\{\mathrm{hi}, \omega\}$; $r := 7$; **return**
25: **end if**
26: **end if**
27: **end if**;
28: $\mathrm{hi} := \texttt{bisect\_ub}(f, y, n_g, d_M, w_M, \alpha, \omega, n_g, s, t, \mathrm{lo}, \mathrm{hi})$;
29: **while** $f(\mathrm{pred}(\mathrm{hi})) \succ y \wedge t > 0$ **do**
30: $\mathrm{hi} := \mathrm{pred}(\mathrm{hi})$;
31: $t := t - 1$
32: **end while**;
33: **if** $f(\mathrm{pred}(\mathrm{hi})) \prec y$ **then**
34: $u := \mathrm{hi}$; $r := 8$
35: **else if** $f(\mathrm{pred}(\mathrm{hi})) = y$ **then**
36: $u := \mathrm{pred}(\mathrm{hi})$; $r := 9$
37: **else**
38: $u := \mathrm{hi}$; $r := 7$
39: **end if**

Algorithm 7 Indirect propagation: `findhi_ub`$(f, y, [x_l, x_u], n_{\mathrm{g}}, d_{\mathrm{M}}, w_{\mathrm{M}}, \alpha, \omega, t)$

Require: $f \colon \mathbb{F} \to \mathbb{F}$, $y \in \mathbb{F}$, $[x_l, x_u] \in \mathcal{I}_{\mathbb{F}}$, $n_{\mathrm{g}} \geq n_{\mathrm{g}}^f$, $d_{\mathrm{M}} \geq d_{\mathrm{M}}^f$, $w_{\mathrm{M}} \geq w_{\mathrm{M}}^f$, $\alpha \preccurlyeq \alpha^f$, $\omega \succcurlyeq \omega^f$, $n_{\mathrm{g}} > 0 \implies (x_l \preccurlyeq \alpha \preccurlyeq \omega \preccurlyeq x_u)$, $t \in \mathbb{N}$, $f(x_u) \prec y$.
Ensure: $u \in \mathbb{F}$, $p_6(y, x_l, x_u, u)$.

if $n_{\mathrm{g}} = 0 \vee x_u \succ \omega \vee \#[f(x_u), y) > d_{\mathrm{M}}$ **then**
  $u := x_l$
else if $n_{\mathrm{g}} = 1 \wedge \big(w_{\mathrm{M}} > t \vee (f(\mathrm{succ}(\alpha)) \prec f(\alpha) \wedge y \succcurlyeq f(\alpha))\big)$ **then**
  if $y \prec f(\alpha) \vee f(\mathrm{succ}(\alpha)) \succcurlyeq f(\alpha)$ **then**
    $u := x_u$
  else if $y = f(\alpha)$ **then**
    $u := \mathrm{succ}(\alpha)$
  else
    $u := x_l$
  end if
else
  $(b, \mathrm{hi}, \hat{x}) := \texttt{linsearch\_geq}(f, y, [x_l, x_u], w_{\mathrm{M}}, t)$;
    ▷ $\big(b = 1 \wedge \mathrm{hi} \in [x_l, x_u] \wedge f(\mathrm{hi}) \succcurlyeq y \wedge \forall x \in [\mathrm{succ}(\mathrm{hi}), x_u] : f(x) \prec y\big)$
    ▷ $\vee \big(b = 0 \wedge \forall x \in [\hat{x}, x_u] : f(x) \prec y\big)$
    ▷ where $v = \min\{t, w_{\mathrm{M}}\}$ and $\hat{x} = \max\{x_l, \mathrm{pred}^v(x_u)\}$
  if $b = 1$ **then**
    $u := \mathrm{succ}(\mathrm{hi})$
  else if $t \geq w_{\mathrm{M}}$ **then**
    $u := x_l$
  else
    $u := \hat{x}$
  end if
end if

Algorithm 8 Indirect propagation: `check_glitch`$(f, y, [x_l, x_u], n_\mathrm{g}, d_\mathrm{M}, w_\mathrm{M}, \alpha, \omega, t, \mathrm{lo}, \mathrm{m}, \mathrm{hi})$

Require: $x_l \preccurlyeq \mathrm{lo} \preccurlyeq \mathrm{m} \preccurlyeq \mathrm{hi} \preccurlyeq x_u$, $f(\mathrm{m}) \succ y$, $f(\mathrm{hi}) \succ y$, $f : \mathbb{F} \to \mathbb{F}$, $y \in \mathbb{F}$, $[x_l, x_u] \in \mathcal{I}_\mathbb{F}$, $n_\mathrm{g} \geq n_\mathrm{g}^f$, $d_\mathrm{M} \geq d_\mathrm{M}^f$, $w_\mathrm{M} \geq w_\mathrm{M}^f$, $\alpha \preccurlyeq \alpha^f$, $\omega \succcurlyeq \omega^f$, $n_\mathrm{g} > 0$, $x_l \preccurlyeq \alpha \preccurlyeq \omega \preccurlyeq x_u$, $\alpha \leq \mathrm{hi} \wedge \omega \geq \mathrm{m}$, $t \in \mathbb{N}$.
Ensure: $b \in \{0, 1, 2\}$,
$n_\mathrm{g} = 1 \wedge w_\mathrm{M} < t \wedge \big(\alpha = \alpha^f \vee \omega = \omega^f \vee (\#[\alpha, \omega) > k \wedge k \leq t)\big) \implies b \in \{0, 1\}$,
$b = 0 \implies \forall x \in [\mathrm{m}, \mathrm{hi}] : f(x) \succ y$,
$b = 1 \implies z \in \mathbb{F} \wedge \mathrm{lo} \preccurlyeq z \preccurlyeq \mathrm{hi} \wedge \forall x \in (z, \mathrm{hi}] : f(x) \succ y \wedge f(z) \preccurlyeq y$.

1: **if** $n_\mathrm{g} = 1 \wedge w_\mathrm{M} \leq t$
$\wedge \big(f(\omega^-) \prec f(\omega) \vee f(\alpha^+) \prec f(\alpha) \vee (\#[\alpha, \omega) > k \wedge k \leq t)\big)$ **then**
2: $s_l := \max\{\alpha, \mathrm{lo}\}$;
3: **if** $f(\omega^-) \prec f(\omega) \vee (\#[\alpha, \omega) > k \wedge k \leq t)$ **then**
4: $s_u := \min\{\omega, \mathrm{hi}\}$
5: **else**
6: $s_u := \min\{\alpha^{+w_\mathrm{M}}, \mathrm{hi}\}$
7: **end if**;
8: $(b, z) :=$ `linsearch_leq`$(f, y, w_\mathrm{M}, s_l, s_u)$
9: ▷ $(b = 0 \wedge z = \hat{x} \wedge \forall x \in [z, s_u] : f(x) \succ y)$
10: ▷ $\vee (b = 1 \wedge z \in [\hat{x}, s_u] \wedge f(z) \preccurlyeq y \wedge \forall x \in (z, s_u] : f(x) \succ y)$
11: ▷ where $\hat{x} = \max\{s_l, s_u^{-w_\mathrm{M}}\}$
12: **else**
13: $b = 2$
14: **end if**

Algorithm 9 Indirect propagation: `bisect_ub`$(f, y, [x_l, x_u], n_g, d_M, w_M, \alpha, \omega, s, t, \mathrm{lo}, \mathrm{hi})$

Require: $x_l \preccurlyeq \mathrm{lo} \prec \mathrm{hi} \preccurlyeq x_u$, $f(\mathrm{lo}) \preccurlyeq y \prec f(\mathrm{hi})$, $\forall x \in [\mathrm{hi}, x_u] : f(x) \succ y$ $f \colon \mathbb{F} \to \mathbb{F}$, $y \in \mathbb{F}$, $[x_l, x_u] \in \mathcal{I}_{\mathbb{F}}$, $n_g \geq n_g^f$, $d_M \geq d_M^f$, $w_M \geq w_M^f$, $\alpha \preccurlyeq \alpha^f$, $\omega \succcurlyeq \omega^f$, $n_g > 0 \implies (x_l \preccurlyeq \alpha \preccurlyeq \omega \preccurlyeq x_u)$, $s, t \in \mathbb{N}$.

Ensure: (c) $x_l \preccurlyeq \mathrm{lo} \prec \mathrm{hi} \preccurlyeq x_u$, $f(\mathrm{lo}) \preccurlyeq y \prec f(\mathrm{hi})$, $\forall x \in [\mathrm{hi}, x_u] : f(x) \succ y$

(p) $\Big(n_g = 0 \vee \big(n_g = 1 \wedge w_M < t \wedge \big(\alpha = \alpha^f \vee \omega = \omega^f \vee (\#[\alpha, \omega) > k \wedge k \leq t)\big)\big)\Big)$ $\implies f(\mathrm{pred}(\mathrm{hi})) \preccurlyeq y$

```
1:  while #[lo, hi) > 1 do
2:      mid := split_point(lo, hi);
3:                                   ▷ ∃m, m' > 0, |m − m'| ≤ 1, mid = pred^m(hi) = succ^m'(lo)
4:      if f(mid) ≼ y then
5:          lo := mid
6:      else if n_g = 0 ∨ hi ≼ α ∨ mid ≽ ω ∨ #[y, f(mid)) > d_M then
7:          hi := mid
8:      else
9:          (b, z) := check_glitch(f, y, n_g, d_M, w_M, α, ω, t, lo, mid, hi);
10:         if b = 0 then
11:             hi := mid
12:         else if b = 1 then
13:             hi := succ(z); break
14:         else
15:             z := logsearch_ub(f, d_M, mid, hi, y, s);
16:                                   ▷ z ∈ [mid, hi] ∧ ((z ≺ hi) ⟹ #[y, f(z)) > d_M)
17:             if z ≺ hi then
18:                 hi := z
19:             elsebreak
20:             end if
21:         end if
22:     end if
23: end while
```