Interval Testing Strategies Applied to COSY's Interval and Taylor Model Arithmetic

George F. Corliss and Jun Yu

Electrical and Computer Engineering
Marquette University
Milwaukee WI, USA

Abstract. The COSY Infinity software package by Berz et al. is widely used in the beam physics community. We report execution-based testing of its interval and Taylor model arithmetics. The testing strategy is careful to avoid contamination by inevitable rounding errors. Tests were ported to Sun's F95 and INTLAB. In each package, we uncovered violations of containment which have all been corrected by their authors. We encourage users of COSY and most other software packages to check author/vendor web sites regularly for possible updates and patches.

1 Testing COSY's Interval Arithmetic

During Spring 2002, the reliable computing email list `reliable_computing@interval.louisiana.edu` had an active discussion of COSY Infinity [1,9] (Berz et al., available from `http://cosy.pa.msu.edu` [2]). COSY Infinity is an arbitrary order package for multivariate automatic differentiation and interval and Taylor model arithmetic. It can be used in an interpreted version, which we tested, in a compiled version from Fortran 77 and C programs, or through objects in Fortran 90 and C++. The `reliable_computing` discussions raised concerns about the reliability of interval and Taylor model arithmetics, so Berz commissioned the execution-based testing of COSY interval arithmetic we report here. We also applied our tests to Sun Microsystems' Fortran 95 [10] and Rump's INTLAB for MATLAB [13,14,15].

Testing software is challenging. Myers summarizes testing philosophy, "The purpose of testing is to find errors" [11]. Kit [8], Kaner et al. [6], or Whittaker [16] offer best practice in industrial software quality assurance.

Authors of many packages for interval arithmetic have tested their work, but there is little literature describing those tests. In TOMS 737 [7], Kearfott et al. tested their Fortran 77 INTLIB arithmetic operations with a combination of specially constructed and randomly generated arguments. Corliss [4] gave a suite of programs for "testing" environments for interval arithmetic for usability and speed. Sun Microsystems says their Fortran 95 interval elementary function library has undergone exhaustive testing, which is confidential.

The focus of this paper is on the testing of COSY's interval and Taylor model arithmetic. Since we found little methodological discussion in the literature, we developed testing methods that could be applied more generally. Besides the testing of COSY, we applied our methods also to Sun's F95 and INTLAB primarily to validate our testing methods. The testing methods have wider utility, but our focus is execution-based testing of COSY.

2 What Is "Correct?"

The fundamental tenet of the interval community is, *"Thou shalt not lie!"* It is an error to i) violate containment or ii) assert a mathematical falsehood. Our testing exposed violations of containment for

1. COSY: power when the exponent is not an integer, but very close to it.
2. COSY: (with warning) tan when the interval argument crosses discontinuity.
3. INTLAB: sqrt for most arguments.
4. Sun F95: tanh for many negative arguments.
5. COSY Taylor models: sin, asin, and acos.

We give details of errors we found in Sects. 5 and 9. On the other hand, questions of appropriate domains for interval operations, tightness of enclosures, speed, and ease of use are not considered errors, but may represent opportunities for improved performance. We raise some of those issues in Sects. 6, 7, and 8.

3 Test Strategy

To complete the testing in a timely manner, we accepted a very narrow scope. We tested the arithmetic operations unary and binary addition and subtraction, multiplication, and division, and the intrinsic functions power, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, log, exp, sqrt. sqr, and isqrt. Our goal is to identify i) violations of containment or ii) assertions of mathematical falsehood. We developed a set of test cases consisting of an interval vector $[x]$ and an expression $f(x)$. Expected results are computed a posteriori in Maple. We did not attempt testing of other features of COSY including its linear dominated bounder, shrink-wrapping, or ODE solving.

We denote by $[\widehat{f([x])}]$ the result of challenging the interval arithmetic to evaluate f on the interval $[x]$. We seek examples $x \in [x]$ for which $f(x)$ is not in $[\widehat{f([x])}]$. We do not need to know the true containment set of $f([x])$. Instead, we use Maple as the "referee" of containment. We

1. Read each test case into a COSY driver;
2. Construct COSY intervals for the arguments;
3. Evaluate the expression using COSY interval arithmetic;
4. Write binary values of the arguments and the COSY result;
5. Read the binary arguments and COSY results into Maple;
6. Perform many point evaluations $f(x)$ for $x \in [x]$;
7. Compare Maple's $f(x)$ with COSY enclosure.

The most challenging aspect of conducting the tests was to prevent inevitable roundoff errors from contaminating our results.

3.1 Roundoff Errors

Suppose we wish to test sin on [0.1, 0.6] on an IEEE arithmetic machine. Fundamentally, it is *impossible,* since 0.1 and 0.6 are not exactly representable. We cannot even express the question, "What is sin [0.1, 0.6]?"

Roundoff errors may be introduced into tests of interval software when we

1. Read test cases into the test driver;
2. Construct interval(s) for the arguments;
3. Extract interval bounds from arguments and results;
4. Write arguments and results to a file;
5. Read arguments and results into Maple;
6. Construct Maple variable precision representations;
7. Perform Maple operations;
8. Report from Maple.

Table 1 suggests the schematic flow of the testing process. It shows the communication from COSY to Maple of both the exact (binary) argument(s) used to challenge each arithmetic operation or intrinsic function and the exact (binary) result computed by COSY. We consider each potential source of roundoff error in turn. The issue is not with Maple. Issues 1 - 3 concern COSY. Issues 4 - 6 concern communication between any pair of dissimilar software packages.

Table 1. Schematic of the flow of testing

COSY		Maple referee
Enter [0.1, 0.6]		
$\Downarrow$ round near		
[internal IEEE 754]	$\Longrightarrow$	$[\cdots]$
$\Downarrow$ INTV round out		$\Downarrow$ multiprecision
INTV($\cdots$)		$x \in [\cdots]$
$\Downarrow$ f round out		$\Downarrow$ f multiprecision
$\Downarrow$		$f(x)$
funct($\cdots$)	$\Longrightarrow$	enclosed in?

Read Test Cases into the Test Driver. We must separate the testing of the input and output routines from the testing of the operations. Our goal is to test the operations of interval arithmetic. We read files of test cases into a test driver. We view the internal binary values as *truth,* while the ASCII values in the file are viewed as approximations. In the few cases where the difference matters, we use test arguments that are exactly representable in binary, and we check whether they are read exactly.

Construct an Interval. COSY's interval constructor INTV() by default adds one ULP outward to its arguments to compensate for assumed possible inward rounding in assigning their values. We tested COSY using the INTV() constructor to model usual use. Using INTV() prevented us from testing cases such as asin([1, 1]) because INTV(1.0, 1.0) contains points at which asin is not defined.

Extract Interval Bounds. After challenging COSY's interval arithmetic, we write the challenge arguments and the COSY results to a file. We use the COSY functions INL() and INU() to extract the lower and upper bounds, respectively, of the COSY intervals. We verified by both execution testing and by direct code inspection that INL() and INU() return their respective values with no rounding.

Write Arguments and Results to a File. To avoid roundoff in writing the challenge arguments and COSY's results to a file for reading by Maple, we write them in binary form, either big endian or little endian, depending on the host testing platform.

Read Arguments and Results into Maple. We read binary representations of the challenge arguments and the COSY results into Maple. The binary representation is system dependent, so we used different functions to read binary files written in big or little endian formats. We verified the correct transmission of values by comparing HEX dumps from COSY, Maple, and DOS's debug.

Construct Maple Variable Precision Representations. We read the binary file into Maple using 900 decimal digit arithmetic. Maple's binary read and convert to decimal is not accurate (previously known), so we wrote our own binary read in Maple, reading each byte as an integer and reassembling the IEEE representation using Maple's 900 decimal digit arithmetic.

Perform Maple Operations. We used 900 decimal digits to ensure that the full dynamic range of 53 bit mantissa IEEE double precision numbers is exactly representable in the decimal form used by Maple's variable precision arithmetic. Even if Maple's variable precision arithmetic were not accurate in the last few digits, we are safe, since we are detecting violation of containment errors in about the 14 - 17 th decimal digit.

Is 900 decimal digits "large?" No. In order for our logic to hold, we must ask Maple to evaluate the sin at *exactly* the same endpoints with which we challenged COSY. We have "exact" in the form of binary values. 53 binary digit numbers can be exactly representable in a finite number (56) of decimal digits (not the other way around). Representing the full range of IEEE double precision numbers, about 10^{-308} to 10^{308}, requires another 617 decimal digits. To get Maple to evaluate sin at INF(X) and SUP(X) as evaluated by COSY, we must use at least 673 decimal digits in Maple. 900 gives a margin of error in case Maple's last few digits might be in error, of which we saw no evidence. In practice, we saw some incorrectly diagnosed "failures" using 100 decimal digits, but not with 200 digits.

Violations of containment are detected in Maple by comparing Maple's 900 digit evaluation of $f(x)$ with COSY's enclosure. If a violation of containment were due to a rounding error in Maple's evaluation, the failure of containment would be in the last few of the 900 digits, and increasing to 1000 or more digits would resolve them. In all violations of containment we observed, the failure was of approximately the accuracy of double precision computation, and increasing the number of digits had no effect.

In each violation of containment detected by Maple, careful human examination of the test case confirms that reported violations of containment truly represent a failure of the software under test. We used Maple for its arbitrary precision capabilities to detect the errors, but once found, errors are visible to the reader in this paper or in COSY execution with no need to rely on Maple.

Report from Maple. Values printed by Maple are subject to rounding error on output, but all of our conclusions have been drawn using internal Maple representations. Any Maple output rounding has no effect on our conclusions.

3.2 Test Cases

We tested 30 multi-operation expressions, but if an arithmetic package gets individual operations and intrinsic functions right, it will get complicated expressions right, too. Hence, we tested primarily 2,600+ expressions composed of a single operation or intrinsic function.

For elementary operations, no matter how wide the arguments, extrema occur at the endpoints, except for division by intervals containing zero. Similarly for intrinsic functions, extrema are always at the endpoints, except for a modest set of exceptions (e.g., sin and cos for arguments that span π or $\pi/2$), which we enumerate and test. Hence, we are most likely to find violations of containment at endpoints of the challenge arguments.

Our Maple "referee" checks interior points, but we observed no failures at interior points. For each test case, we have Maple evaluate the expression under test at 11 points in the challenge argument interval using 900 decimal digit approximate arithmetic, as illustrated in the pseudo-code below. All errors we found would have been detected using only two points in the challenge interval. If $f(x)$ is not in COSY's result interval, we have a likely violation of containment, which we verify by human inspection of results as described in Sect. 5.

```
for (i = 0; i <= 10; i++) {
  y = INF(X) + (SUP(X) - INF(X)) * i/10.0
  fx = f(y)
  ERROR if fx is outside COSY result
}
```

We might look at extrema of the function, check at randomly chosen points, or at far more points. There are separate test cases to challenge evaluation within one ULP of extrema, so checking at extrema is already covered. Random tests are rarely as effective at uncovering errors as carefully constructed challenges; our test cases uncovered all the errors we found. None of our 500,000 random tests uncovered an error. Similarly, we had checked at 10,000 points (vs. 11) early in our testing, but all the errors we found at endpoints.

Most of our test cases came from TOMS 737 [7]. Kearfott et al. tested their Fortran 77 INTLIB interval arithmetic operations with a combination of specially constructed and randomly generated arguments. We added a few specially constructed arguments of our own and 30 multi-operation expressions taken from tests of a validated quadrature package by Corliss and Rall [3]. In general, we expect interval arithmetic most likely to fail for very large or very small (in either absolute or relative terms) domain or range values, near boundaries of domains, or near underflow or overflow.

To increase the coverage of our tests of binary operations, each pair of arguments was used in several combinations. For example for addition and subtraction, argument intervals $[a]$ and $[b]$ give test cases

– $[a]+[b]$, $[a]-[b]$, $[-a]+[b]$, $[-a]-[b]$
– $[-a]+[-b]$, $[-a]-[-b]$, $[a]+[-b]$, $[a]-[-b]$
– $[b]+[a]$, $[b]-[a]$, $[-b]+[a]$, $[-b]-[a]$
– $[-b]+[-a]$, $[-b]-[-a]$, $[b]+[-a]$, $[b]-[-a]$

For multiplication, with $0 \le [\underline{a}, \overline{a}]$ and $0 \le [\underline{b}, \overline{b}]$, we test 16 combinations:

– $[\underline{a},\overline{a}] \times [\underline{b},\overline{b}]$, $[-\underline{a},\overline{a}] \times [\underline{b},\overline{b}]$, $[-\overline{a},-\underline{a}] \times [\underline{b},\overline{b}]$, $[-\overline{a},\underline{a}] \times [\underline{b},\overline{b}]$
– $[\underline{a},\overline{a}] \times [-\underline{b},\overline{b}]$, $[-\underline{a},\overline{a}] \times [-\underline{b},\overline{b}]$, $[-\overline{a},-\underline{a}] \times [-\underline{b},\overline{b}]$, $[-\overline{a},\underline{a}] \times [-\underline{b},\overline{b}]$
– $[\underline{a},\overline{a}] \times [-\overline{b},-\underline{b}]$, $[-\underline{a},\overline{a}] \times [-\overline{b},-\underline{b}]$, $[-\overline{a},-\underline{a}] \times [-\overline{b},-\underline{b}]$, $[-\overline{a},\underline{a}] \times [-\overline{b},-\underline{b}]$
– $[\underline{a},\overline{a}] \times [-\overline{b},\underline{b}]$, $[-\underline{a},\overline{a}] \times [-\overline{b},\underline{b}]$, $[-\overline{a},-\underline{a}] \times [-\overline{b},\underline{b}]$, $[-\overline{a},\underline{a}] \times [-\overline{b},\underline{b}]$

and similarly for division. In addition, we constructed more than 500,000 random tests that discovered no additional errors:

```
loops for i and j
  a = RAND(); b = RAND();
  x1 := +- 0.a * 2^+-i;
  x2 := +- 0.b * 2^+-j;
  [X] := [x1, x2];
  expr(X);
```

4 Test Environment

Our tests of COSY and INTPAK were executed on an HP notebook PC N5270 with a 700 MhZ Pentium III processor, 128 MB RAM, and a 20 GB hard disk under Microsoft Windows ME. The tests were replicated on a Toshiba Satellite 4090XDVD with an Intel Celeron at 400 Mhz, 128 MB RAM, running Windows 98. Our tests of Sun Workshop 6 were conducted on a Sun Enterprise 250, UltraSPARC 3 with one CPU at 450 Mhz with 512 Mb RAM. We tested

– COSY version 8.1 (updated June 8, 2002) downloaded from `www.cosy.pa.msu.edu` on June 25, 2002. The tests were repeated on a modified version of COSY provided on May 2, 2003.
– Sun WorkShop 6 update 1 Fortran 95 6.1 2000/09/11 (from f95 -V ...). The tests were repeated with a patched version released in September, 2002.
– INTPAK version 4.0, `www.ti3.tu-harburg.de/~rump/intlab` downloaded on January 15, 2003. The tests were repeated on Version 4.1.1 downloaded on January 22, 2003.

We used Maple 6 and MATLAB version 5.2. In Maple, we use little beyond the underlying variable precision arithmetic, so newer versions should have no effect on our tests. The error in INTPAK was traced to an anomaly in MATLAB which might be changed in a later version, although Rump observed the same anomaly in the current MATLAB version as of January, 2003.

5 Test Results

In this section, we report the results of our tests. In Sect. 3.2, we claimed to have verified suspected errors by human inspection. In this section, we offer the errors for inspection by the reader. Maple found the errors, but the reader can see them with no dependence on Maple.

5.1 COSY: POWER Near an Integer

Test case (ASCII): $[2.0, 2.0]^{1.00000000001}$
As presented to COSY: $2^{1.0000000000100000000827\ldots}$ (approximate decimal representation of binary value)
COSY result: $[1.999999999999999555\ldots, 2.000000000000000444\ldots]$ (approximate decimal representation)
Maple's $f(x)$**:** $2.0000000000138\ldots$ (approximate decimal representation), which violates containment by about 10^{-11}.
Cause: The POWER operator was intended only for internal use by COSY for integer and half-integer exponents. Exponents within 10^{-10} of an integer or a half-integer are rounded to the nearby integer or a half-integer. Exponents further from an integer or a half-integer are rounded with a warning message.
Solution: COSY authors removed the POWER operator from the list of user callable operations.

5.2 COSY: TAN Crossing Discontinuity

Test case (ASCII): $\tan([1.0, 2.0])$ or $\tan([1.0, 1.0E + 30])$
COSY result: Print a warning and return $[-1.0E{+}35, 1.0E{+}35]$, which violates containment at points very close to $\pi/2$. This is a problem if the user ignores the warning, or if the warning scrolls off the screen.
Cause: COSY correctly recognized that the challenge argument includes a singularity, but it returned finite bounds.
Solution: COSY authors modified COSY so that after the warning is printed, execution halts.

5.3 COSY: ASIN or ACOS at ± 1

Test case (ASCII): asin(1), asin($[-1.0, 1.0]$), or similarly for acos.
COSY result: Messages "asin(1) does not exist, " and "asin([-1, 1]) does not exist," respectively. These assert mathematical falsehoods.

Cause: COSY's interval constructor INTV() outwardly rounds the intervals [1, 1] and [-1, 1], even though their endpoints are exactly representable. Hence, COSY correctly detects that the challenge argument includes points outside the domain of asin. The default output routines in the test environment round endpoints as printed in the message, although other environments printed more digits, so the message was correct as printed.
Solution: COSY authors changed the formating of the message to read, "arcsin does not exist for the interval $[0.999999999999999, 1.000000000000001]$."

5.4 Sun F95: tanh (Negative)

To validate the testing methodology, we re-wrote the same test battery for Sun's F95 compiler. For challenge arguments less than about -4, e.g., tanh ([-4.879, -4.267]), containment fails by 1-2 ULP's.
Cause: There was a discrepancy between production and development versions.
Solution: Sun corrected the problem within one week, releasing an update.

5.5 INTLAB: sqrt

To further validate the testing methodology, we re-wrote the same test battery in Matlab for Rump's INTLAB. For the sqrt function, every degenerate interval fails by one ULP, and most thick intervals fail.
Cause: MATLAB's sqrt is not the IEEE sqrt. It uses round to nearest, rather than the current rounding mode.
Solution: Within a day, Rump posted a corrected version of INTLAB using its own rounding control for sqrt.

6 Domains: Opportunity for Improvement?

When a package for interval arithmetic encounters arguments outside the mathematical domain, it can respond by

1. Continue execution with empty, NAN, over/underflow, or other special value
2. Consider $f([x])$ as $f([x] \cap \text{domain of } f)$ (Sun's approach)
3. Halt execution, possibly with an error message (COSY and INTLAB)

As originally tested, COSY was not consistent in its handling of arguments outside the mathematical domain. Those inconsistencies have been corrected by the COSY authors.

COSY considers it a fatal error to evaluate outside the domain of an expression, e.g., asin(1) or sqrt(0). These examples are outside the domain because COSY enlarges the intervals on construction. Sun's F95 "handled" many cases COSY did not. For example, Sun considers sqrt ([-1, 1]) to be [0, 1].

We suggest handling of domains as an opportunity for improvement. We found no further violations of containment, and we understand why COSY treats asin(1) or sqrt(0) as fatal errors. However, we would consider it an improvement if COSY were able to evaluate such cases correctly.

Sun's csets (containment sets) represent Sun's effort to handle domains. Csets are based on an elegant theory, but their implications are not well understood by the interval community. For example, Neher has given an example $f(x) = \sqrt{x} + 1/2 = 0$ on [-4, 4]. Naive cset evaluation gives $f([-4, 4]) = [1/2, 5/2] \subset [-4, 4]$, incorrectly suggesting the existence of a fixed point. Cset evaluation appears to require independent verification of continuity, which is done implicitly in some systems for interval arithmetic.

7 Tightness: Opportunity for Improvement?

COSY makes many compromises for efficiency over tightness of the intervals. For example, the COSY interval constructor INTV() rounds endpoints outward, while Sun's F95 and Rump's INTLAB provide interval constructors that accept strings and round outward only when necessary to guarantee containment.

We compared the excess widths of the COSY, Sun F95, and INTLAB results across our test cases. Table 2 shows the number of Units in the Last Place (ULP's) the interval result is wider than the Maple result, the interval computed by Maple in 900 decimal digit arithmetic. Compared with IEEE double precision computed by COSY, the Maple result is a very good approximation to the true result. We do not have exactly the correct number of ULP's in every case, but we do have a reliable measure of excess widths. Suppose (in pseudocode)

```
$t_U$ = Maple upper bound of the result
$c_U$ = upper bound computed by COSY ($t_U \leq c_U$)
$r_U = c_U - t_U$
if $t_U = 0$ then $r_U = r_U * 2^{1022}$ else $r_U = r_U/|t_U| * 2^{52}$
Similarly for the ULP's at the lower bound $r_L$
Add $r_L + r_U$ ULP's at lower and upper bounds
```

For example, consider $[1, 2] + [3, 4] = [4, 6]$. The COSY result is

$$\begin{aligned}&[\text{FC FF FF FF FF FF 0F 40 08, 04 00 00 00 00 00 18 40 08}]\ (\text{hex})\\&= [3.999\,999\,999\,999\,998\,223\,..., \ 6.000\,000\,000\,000\,003\,552\,...]\ ,\end{aligned}$$

which is eight excess ULP's because the constructors INTV(1.0, 2.0) and INTV (3.0, 4.0) round out, and the operator ADD rounds out further. Sun's F95 and INTLAB give excess widths of zero ULP's for this example. The excess widths in ULP's can be large when the true answer is near the underflow limit.

Table 2 shows the number of test cases for which the interval result had excess widths shown. Smaller excess widths are better, so it is better to have more test cases with excess widths of 0 - 2 and fewer test cases with larger excess widths. The first row in Table 2 shows that COSY computed the tightest possible enclosure (zero excess width) in 33 test cases, while F95 and INTLAB were as tight as possible in 1277 and 1201 test cases, respectively, from the total

Table 2. Excess width in ULP's

	COSY June '02	COSY May '03	Sun F95	INTLAB ver. 4
0	33	33	1277	1201
1	1	1	697	607
2	81	79	251	292
3-4	746	746	26	147
5-8	906	906	1	9
9-16	194	190	0	2
17-32	151	129	0	0
33-64	17	15	0	0
65-128	6	6	0	0
129-256	14	14	0	0
257-512	12	12	0	0
Total	2161	2130	2252	2259

of 2,600 test cases. Test cases with no finite true result, with true result zero, or with underflow or overflow are excluded, leading to different numbers of total test cases reported for each package.

Loss of tightness is not an error, but it is an opportunity for improvement, possibly at the expense of speed or portability. The Sun and INTLAB results in Table 2 show that increased tightness is achievable.

8 Speed: Opportunity for Improvement?

We prefer fast programs to slow ones, but unbiased, comprehensive speed testing is difficult and controversial. Speed is not in the scope of our tests, but we have run programs implementing the same test cases in different environments, and we suspect some readers might wonder, "How long did each take?" We make no claim of fair testing of speed. That could be the subject of another paper, but we report what we observed.

COSY and INTLAB timings were made on a Toshiba Satellite 4090XDVD with an Intel Celeron at 400 Mhz, 128 Mb RAM, running Windows 98, denoted by (Win 98) in Table 3. The versions of COSY and INTLAB we tested both run in an interpreted mode. The Sun F95 timings were made on a Sun Enterprise 250, UltraSPARC 3, 1 CPU at 450 Mhz with 512 Mb RAM, denoted by (SPARC) in Table 3. The F95 code was compiled, linked, and run. We have not reported compile and link times.

Table 3 reports CPU time for one million evaluations of the Shekel 5 function, commonly used to measure a Standard Time Unit (STU) [5]:

$$f(x) = -\sum_{i=1}^{m=5} \frac{1}{(x - A_i)(x - A_i)^T + c_i} \ ,$$

where A_i denotes the ith row of a given 5×5 matrix A, and c is a given vector of length 5. Evaluation of the Shekel 5 function reflects arithmetic operations, so we also report CPU time for the evaluation of

$$f(x) = \log_{10}(\text{asin }(\sin^2(x) + \cos^2(x) - \exp(\text{atan }(-x^2)))) \tag{1}$$

constructed to reflect executions for intrinsic function evaluations.

Table 3. CPU times in seconds

	COSY (Win98)	INTLAB (Win98)	Sun F95 (SPARC)
1 M evaluations of Shekel 5			
Double precision	92	410	25.4
Interval	157	23289	33.2
1 M evaluations Equation (1)			
Double precision	7.3	142	2.89
Interval	25.4	41650	13.58
2,600 interval test cases	6.0	19.1	0.3

INTLAB interval times were estimated by timing 10,000 evaluations and multiplying by 100. Execution of our interval test cases is dominated by disk I/O. In this environment, interpreted COSY is significantly faster than interpreted INTLAB, although recoding either one in a style more appropriate for its environment may yield significant improvements. We did not attempt to optimize the performance, preferring to keep the code for the tests as similar as possible in each environment. For example, the INTLAB code uses loops rather than much faster vector operations. The ratio of interval / real times for COSY are comparable with Sun's F95, and significantly smaller than INTLAB. Results in other environments may be markedly different.

Regarding tightness and speed, Martin Berz responds to the results of our tests,

"COSY is designed on the two premises of portability across platforms on the one hand, and use within the Taylor model framework on the other. The desired portability is achieved by building interval intrinsics based on F77 intrinsics, with the necessary safety factors of around four ULP's because of the inherent precision (or rather lack thereof) of the intrinsics. The use in the Taylor model framework entails that in practically relevant calculations, these slight overestimations usually do not matter since the Taylor model approach is used for large domain intervals where because of dependency, conventional validated methods usually have much larger overestimations in all but the simplest cases. Furthermore, since the vast majority of effort in the Taylor model arithmetic lies in the floating point coefficient arithmetic which is highly optimized in COSY, the efficiency of the interval implementation is of secondary significance."

We repeated our tests replacing the default safety factor in COSY for inflation of F77 intrinsics by an inflation of one ULP at each end. We observed reduced excess widths and no further violations of containment.

9 Testing COSY's Taylor Model Arithmetic

After testing COSY's interval arithmetic, we turned to its Taylor model arithmetic. Revol et al. [12] provide mathematical proofs that the algorithms in COSY for multiplying a Taylor model by a scalar and for adding or multiplying two Taylor models return Taylor models satisfying the containment property. We performed broader, execution-based testing. Revol's proof of the algorithm and our execution-based testing are complementary. The proof is more general than a (large) collection of test cases in the sense that test cases can demonstrate the existence of an error, but cannot demonstrate absence of errors. Our execution-based tests might discover implementation errors of a correct algorithm, and we covered operations and intrinsic functions Revol did not consider.

Given an interval vector $[x]$ and an expression $f(x)$, a Taylor model TM_f is

1. $p(x)$, a polynomial in x with floating-point coefficients, and
2. $[I]$, an interval

such that $f(x) \in TM_f(x) = p(x) + I$ for all $x \in [x]$. The goal of our execution-based testing was to find examples for which containment of *point* evaluation failed, i.e., $x \in [x]$ for which $f(x)$ is not in $TM_f(x)$. We did not consider the weaker range bound test: $f([x]) \in TM_f([x])$. By inclusion monotonicity, if $f(x) \in TM_f(x)$ for all $x \in [x]$, then $f([x]) \in TM_f([x])$. The point evaluation challenges might discover an error which could be masked by even slight interval overestimation in the interval evaluation challenge.

9.1 Verification Process

COSY's Taylor model arithmetic can be verified using COSY's interval arithmetic to verify COSY's Taylor model arithmetic. All the comparison is done inside COSY. Alternatively, we can use Maple as a referee. Both of the tests are rigorous. The second test might detect containment failures the first one does not, but it is difficult to communicate the required information to Maple. We would have to communicate sparse structure of the Taylor model and binary values of its coefficients. The first test is much faster, and it is the approach we used.

Taylor Model Verification:

1. Evaluate the function f over the domain $[x]$.
   For example: $f = \cos(3.14 + 1.57 * x)$ on $[x] = [-1, 1]$.
2. Construct the Taylor model expression of f (TM_EXPR) in COSY.
   TM_EXPR := COS(-3.14 * TM_ONE + (1.57 * TM_ONE) * TM_INDEP);

TM_ONE is Taylor model for the constant ONE. It is used to convert constants such as -3.14 and 1.57 into Taylor models.
TM_INDEP is a Taylor model for the independent variable.
3. Construct the interval expression of f (INL_EXPR) in COSY.
IVL_EXPR := COS(INTV(-3.14, -3.14) + INTV(1.57, 1.57) * VAR1);
VAR1 is the interval independent variable.
4. Choose a point $z \in [x]$ and convert it to a tight interval $[z]$ using COSY's interval constructor.
5. Evaluate the polynomial part of the Taylor model expression (TM_EXPR) on the tight interval ($[z]$) and add the remainder bound.
6. Evaluate the interval expression (IVL_EXPR) on the tight interval ($[z]$).
7. Compare the results of 5) and 6).
If the intervals are disjoint, there is an error.

9.2 Testing Scope

We designed test cases to evaluate the COSY operations of $+$, $-$, $\times$, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, log, exp, sqrt, sqr, isqrt, and unary $+$ and $-$. Taylor model operations combine their operand polynomials and interval remainder bounds using floating point arithmetic to the extent possible and guaranteeing that the resulting Taylor model preserves containment. We tested Taylor models with both general domains for the independent variables and domains normalized to $[-1, 1]^n$ at dimension 1 (13 expressions): order 1, ..., 20; dimension 2 (20 expressions): order 1, ..., 18; and dimension 7 (21 expressions): order 1, 2, 3, and 4. "Dimension" denotes the number of independent variables, and "order" is the order of the Taylor model polynomial. The Taylor models were challenged at the corner points of n-dimensional boxes and at a few interior points. As for the interval tests, we expect errors to be most visible at the boundaries. Here is pseudo-code for these tests:

```
Loop for general and normalized domain
    Dimension = 1; Loop for order = 1, ..., 20
        Loop for 9 challenge points
            Loop for Taylor model 1 ... 13
                Pass to 149 unary operations
                Pass to 69 binary operations
    Dimension = 2; Loop for order = 1, ..., 18
        Loop for 25 challenge points
            Loop for Taylor model 1 ... 20
                Pass to 149 unary operations
                Pass to 69 binary operations
    Dimension = 7; Loop for order = 1, 2, 3, 4
        Loop for 256 challenge points
            Loop for Taylor model 1 ... 21
                Pass to 149 unary operations
                Pass to 69 binary operations
```

This represents more than 300,000 Taylor models challenged at a total of over 14 million points. That test suite required about eight hours on the 400 Mhz Intel Celeron machine described in Sect. 8. In constructing test cases, we considered order, dimension, normalization, domain, challenge points in the domain, sparsity, oscillation, simplicity, and special numbers to create at least one test case from each test case equivalence class. We adopted the same philosophy as in the interval tests that the test case is the internal binary form of the expression constructed from approximate ASCII representations.

A second test suite used 11 expressions such as

1. $\cos(-3.141592653590006 + 1.570796326794687\ x_1)$;
2. $\sin(-4.712388980384691 + 1.570796326794690\ x_1)$;
3. $\operatorname{asin}\ (0.0009999999999999983\ x_1)$;
4. $\operatorname{asin}\ (-0.4935 + 0.003499999999999997\ x_1)$;
5. $\operatorname{asin}\ (0.0004999999999999989\ x_1 + 0.0004999999999999989\ x_2 x_5)$;

```
Loop for general and normalized domain
    Dimension = 1; Loop for order = 1, 7, 15, 17, 20
        Loop for 8 challenge points
            Loop for expression 1 ... 5
    Dimension = 2; Loop for order = 1, 7, 15, 17, 18
        Loop for 25 challenge points
            Loop for expression 1 ... 9
    Dimension = 7; Loop for order = 1, 2, 3, 4
        Loop for 256 challenge points
            Loop for expression 1 ... 11
```

This represents 228 Taylor models challenged at more than 25,000 points. This test required about 90 seconds and disclosed violations of containment in sin and cos and in asin and acos.

9.3 Containment Error in sin and cos

We found a violation of containment error in sin and cos (examples 1 and 2 above) in arguments of dimensions 1 and 2 with order 17 at x_1 near -1.
Cause: In the test environment, integer arithmetic used internally by COSY overflows and wraps from positive to negative with no alert, warning, or trap.
Solution: Replace some integer arithmetic in the sin and cos modules by double precision. The remaining COSY code was carefully scanned to be sure there were no similar use of integer arithmetic. The May 2, 2003, version of COSY runs the test cases as expected.

9.4 Containment Error in asin and acos

We found several violation of containment errors in asin (examples 3 - 5 above).
Cause: In one case in the asin module, some coefficients were multiplied by $[0, h]$ instead of $[-h, h]$.
Solution: Correct the coding error. The May 2, 2003, version of COSY runs the test cases as expected.

10 Conclusions and Extensions

Testing software of this complexity is itself a complex task. One needs to develop test cases that distinguish subtle errors. For interval packages, one must present to the software under test cases free from possible roundoff, and one similarly must guard against roundoff in specifying the expected result.

Effective testing of interval and Taylor model arithmetic in COSY is difficult because the conservative outward rounding of interval arithmetic can mask subtle errors. Simple test cases were successful (found errors) where more complicated tests had failed. For example, we found Taylor model errors in sin and in asin, although extensive sin(asin(x)) and asin(sin(x)) tests had passed. Similarly, asymmetric tests seemed to be more powerful. The error in sin and cos appeared only for order 17 because the remainder has the form $[0, \delta]$ rather than $[-\delta, \delta]$. The error is present in other orders, but it is hidden by slight excess widths introduced by repeated outward roundings.

Although our test suites for both interval and Taylor model arithmetics are large, they are neither comprehensive nor exhaustive. For example, one might port Gonnet's floating point tests from `www.inf.ethz.ch/personal/gonnet/FPAccuracy/Analysis.html`. Gonnet's is a demanding test for the accuracy of double precision intrinsic functions. He uses challenge points known to be problematic or for which evaluation values are known to be problematic. Gonnet's additional values may disclose errors in interval or Taylor model evaluation.

Execution based testing cannot show the absence of errors, but can only demonstrate their presence. While we prefer to see no errors in our programs, especially in programs that claim to compute with guarantees, we think it speaks well of the authors of the COSY, Sun F95, and INTLAB packages we tested that we found relatively few errors. We cannot guarantee that they are now error-free, but our tests should appreciably raise the level of confidence in their reliability.

Complete software for the testing reported here is available from `www.eng.mu.edu/corlissg/Pubs/COSYtest`.

We encourage users of COSY and most other software packages to check author/vendor web sites regularly for possible updates and patches.

Acknowledgment. This work was funded in part from Michigan State University. The testing could not have been completed without the assistance of Martin Berz and Kyoko Makino. The article is based on a talk presented at the Dagstuhl Seminar on Numerical Software with Result Verification, January 20, 2003. We appreciate the referees' many helpful comments.

References

1. Martin Berz. COSY INFINITY Version 8 reference manual. Technical Report MSUCL–1088, National Superconducting Cyclotron Laboratory, Michigan State University, East Lansing, MI 48824, 1997.
2. Martin Berz. COSY INFINITY web page, 2000. `cosy.pa.msu.edu`.
3. George F. Corliss. Performance of self-validating quadrature. In Pat Keast and Graeme Fairweather, editors, *Proceedings of the NATO Advanced Workshop on Numerical Integration: Recent Developments, Software, and Applications*, pages 239–259. Reidel, Boston, 1987.
4. George F. Corliss. Comparing software packages for interval arithmetic, 1993. Presented at SCAN '93, September 1993, Vienna.
5. Laurence C. W. Dixon and G. P. Szegö. *Towards Global Optimization 2*. North-Holland, 1978.
6. Cem Kaner, Jack Falk, and Hung Quoc Nguyen. *Testing Computer Software, Second edition*. Wiley, New York, 1999.
7. R. Baker Kearfott, M. Dawande, K.-S. Du, and Chenyi Hu. INTLIB: A portable FORTRAN 77 interval standard function library. *ACM Transactions on Mathematical Software*, 1994.
8. Edward Kit. *Software Testing in the Real World: Improving the Process*. Addison Wesley, 1995.
9. Kyoko Makino and Martin Berz. Taylor models and other validated functional inclusion methods. *International Journal of Pure and Applied Mathematics*, 4(4):379–456, 2003. `bt.pa.msu.edu/pub/`.
10. Sun Microsystems. Sun ONE Studio 7 (formerly Forte Developer 7) Interval Arithmetic, 2002.
11. Glenford Myers. *The Art of Software Testing*. Wiley, New York, 1979.
12. Nathelie Revol, Kyoko Makino, and Martin Berz. Taylor models and floating-point arithmetic: Proof that arithmetic operations are validated in COSY. LIP report RR 2003-11, University of Lyon, France, 2003. MSU HEP report 30212, submitted, `bt.pa.msu.edu/pub/`.
13. Siegfried M. Rump. Fast and parallel interval arithmetic. *BIT*, 39(3):539–560, 1999.
14. Siegfried M. Rump. INTLAB - INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999. `www.ti3.tu-harburg.de/rump/intlab`.
15. Siegfried M. Rump. Rigorous and portable standard functions. *BIT*, 41(3):540–562, 2001.
16. James A. Whittaker. *How to Break Software: A Practical Guide to Testing*. Addison Wesley, Boston, 2003.