ANL/MCS-TM-129

ELEFUNT Test Results Using Titan Fortran under Ardent UNIX® 2.0 on the Titan

by

W. J. Cody

March 1989

MATHEMATICS AND
COMPUTER SCIENCE
DIVISION

Argonne National Laboratory, with facilities in the states of Illinois and Idaho, is owned by the United States government, and operated by The University of Chicago under the provisions of a contract with the Department of Energy.

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439-4801

ANL/MCS-TM--129

DE89 010233

ELEFUNT Test Results Using Titan Fortran under Ardent UNIX® 2.0 on the Titan

by

W. J. Cody

Mathematics and Computer Science Division

Technical Memorandum No. 129

March 1989

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy under Contract no. W-31-109-Eng-38.

Contents

ELEFUNT Test Results Using Titan Fortran under Ardent UNIX† 2.0 on the Titan

by

W. J. Cody

Abstract

This report discusses testing of the regular and "fast" elementary function libraries supplied with Titan Fortran on the Ardent Titan computer in the Mathematics and Computer Science Division's Advanced Computing Research Facility. Performance tests were conducted using the ELEFUNT suite of programs from the book *Software Manual for the Elementary Functions* by Cody and Waite. The quality of Titan arithmetic was checked with the MACHAR and PARANOIA programs.

1. Introduction

In August 1988 a two-processor Ardent Titan vector-register graphics computer was installed in the Mathematics and Computer Science Division's Advanced Computing Research Facility. The machine has since been upgraded to four processors sharing 32 megabytes of memory. Floating-point arithmetic is performed with proprietary chips implementing IEEE arithmetic [IEEE 1985].

This report summarizes and analyzes the results of running various programs designed to test the arithmetic and the Fortran elementary and intrinsic function packages on that machine. The programs run include MACHAR [Cody 1988a] and the ELEFUNT suite of transportable Fortran test programs from the *Software Manual for the Elementary Functions* by Cody and Waite [1980], the Fortran version of the arithmetic test program PARANOIA [Karpinski 1985], and prototype programs from the nascent INTFUNT test suite for intrinsic functions. The tests were run with two different libraries: the default library and the "fast" library invoked with the "-fast" compiler option. All tests were run using Titan Fortran under the Ardent UNIX 2.0 operating system.

The next section discusses the computer arithmetic as analyzed by MACHAR and PARANOIA. Section 3 discusses test results for the elementary and intrinsic functions. Section 4 summarizes our findings.

This report is one of a continuing series of reports on the quality of the arithmetic and Fortran libraries available on machines in the Mathematics and Computer Science Division [Cody 1986a, 1986b, 1986c, 1986d, and 1988b].

2. The Arithmetic

MACHAR is an evolving subroutine to dynamically determine fundamental parameters for the floating-point arithmetic system. Table 1 contains the parameter values determined by MACHAR for both single- and double-precision arithmetic on the Ardent Titan. Results with the fast library were identical to those for the default library, which is not surprising because MACHAR does not make extensive

† UNIX is a trademark of AT&T Bell Laboratories.

Table 1. Machine Parameters Determined by MACHAR

Parameter	Single Precision	Double Precision
β	2	2
t	24	53
rnd	2	2
ngrd	0	0
machep	-23	-52
negep	-24	-53
iexp	8	11
minexp	-126	-1022
maxexp	128	1024
eps	0.1192092895508E-06	0.2220446049250D-15
epsneg	0.5960464477539E-07	0.1110223024625D-15
xmin	0.1175494350822E-37	0.2225073858507D-307
xmax	0.3402823466385E+39	0.1797693134862D+309

use of library functions. The tabulated results reveal that Titan arithmetic is not a full implementation of the IEEE standard; it does include the IEEE representation scheme with its reserved exponents, but it lacks graceful underflow.

Definitions of the parameters are as follows:

1) β, the radix for the representation scheme;
2) *t*, the number of base-β digits in the floating-point significand;
3) *rnd*, a parameter indicating the method of rounding in addition and the type of underflow (full or partial):
   a value of 0 indicates truncation with full underflow;
   a value of 1 indicates some non-IEEE form of rounding with full underflow;
   a value of 2 indicates IEEE-style of rounding with full underflow;
   a value of 3 indicates truncation with partial underflow;
   a value of 4 indicates some non-IEEE form of rounding with partial underflow; and
   a value of 5 indicates IEEE-style rounding with partial underflow;

Table 2. Results from PARANOIA

Test	Single-Precision Result	Double-Precision Result
Integer Arithmetic	Okay	Okay
β	2	2
epsneg	5.96046448E-08	1.11022302E-16
t	24	53
Extra-Precise Subexpressions	No	No
Subtraction Normalized	Yes	Yes
Guard Digits in $\times, +, -$	Yes	Yes
Rounding in $+/-, \times, +$	Yes	Yes
Sticky Bit	Yes	Yes
Multiplication Commutative	Yes	Yes

4) *ngrd*, 0 for $rnd \neq 0$; otherwise, the number of base-β guard digits used in multiplication;
5) *machep*, the exponent for the smallest power of β (but bounded below by $t-3$) whose sum with 1.0 is greater than 1.0;
6) *negep*, the exponent for the smallest power of β (but bounded below by $t-3$) whose difference with 1.0 is less than 1.0;
7) *iexp*, the number of bits dedicated to the representation of the exponent (including bias or sign) of a floating-point number;
8) *minexp*, the smallest permissible exponent;
9) *maxexp*, the largest permissible exponent;
10) *eps*, on a binary machine, the floating-point number β^{machep};
11) *epsneg*, on a binary machine, the floating-point number β^{negep};
12) *xmin*, the floating-point number β^{minexp}; and
13) *xmax*, an approximation of the floating-point number β^{maxexp}.

Because MACHAR is intended to be used by other programs, it must avoid exceptions that will terminate execution, severely limiting what it can attempt to determine about an arithmetic system. PARANOIA (see Table 2), a second and more probing program for examining computer arithmetic, does not have that handicap. It is a self-contained program that periodically marks its progress by writing recovery information to file. Thus, if execution is terminated for any of a number of anticipated reasons,

Table 2. Results from PARANOIA (Continued)

‡ denotes results with the fast library

Test		Single-Precision Result	Double-Precision Result
$\sqrt{(i \times i)} = i$		Yes	Yes
	‡	Error	Yes
Sqrt Monotone		Yes	Yes
Sqrt Correctly Rounded or Chopped		Correctly Rounded	Neither
	‡	Neither	Neither
Error Bounds for Sqrt		-0.5 and +0.5 ULP	-0.5 and 1.0 ULP
	‡	-1.5 and +0.6E-5 ULP	-0.5 and 1.0 ULP
z^i for Small Positive i		Okay	Okay
xmin		1.17549435E-38	2.22507386E-308
$(xmin+xmin)/xmin$		2.0	2.0
$1.375 \times xmin - xmin$		0.0 w/o Underflow Signal	0.0 w/o Underflow Signal
$\beta^{-2 \times minexp}$		Okay	Okay
$x^{((x+1)/(x-1))}$ *vs* $exp(2)$, $x \rightarrow 1$		Okay	Okay
z^q for Nearly Extremal Values		Okay	Okay
Overflow		INF	INF
xmax		3.40282347E+38	1.79769313E+308
$z = xmax \times 1$, $xmax / 1$		Okay	Okay
1/0		INF	INF
0/0		NaN	NaN

the program can be restarted with the expectation that saved data will permit it to properly report the reason for its termination and to resume execution beyond the troublesome point. In this way, with possible restarts from time to time, the program is able to run tests on arithmetic characteristics that are not

possible with MACHAR.

PARANOIA was originally written in BASIC by W. Kahan at the University of California, Berkeley, and then translated to Fortran by T. Quarles and G. Taylor. It was made available to the general public by R. Karpinski at the University of California, San Francisco [Karpinski 1985]. The particular version used here was further refined at AT&T Bell Laboratories and transmitted privately by David Gay.

The results from single- and double-precision runs of PARANOIA that are reported in Table 2 are self-explanatory (except that ULP refers to Units in the Last Place of the significand). In most cases the results obtained with the fast library were identical to those obtained with the default library. Here and in subsequent tables, whenever the results for the two libraries disagree, the results for the fast library are tabulated immediately after the corresponding results for the default library, and they are marked with a double dagger (‡).

All runs were completed without restarting. Operations that have led to interrupts on other machines were handled here by returning the IEEE defaults of infinity and NaN (Not a Number) and continuing execution. PARANOIA reaffirms that Titan arithmetic is IEEE-style except for graceful underflow and the optional extended-precision arithmetic.

PARANOIA did uncover a number of problems, however, most of them associated with the *sqrt* function. Except for the single-precision routine in the default library, PARANOIA reports that the sqrt function returns neither correctly rounded nor correctly truncated results, despite the machine's using IEEE round-to-nearest even arithmetic. Indeed, ad hoc tests show that the failure in the $\sqrt{(i \times i)} = i$ test for the fast library, which PARANOIA declares to be a "defect," is incorrect rounding of the result for $i = 33$.

All PARANOIA runs reported a "flaw" because $1.375 \times xmin$ tested as not equal to $xmin$, but $1.375 \times xmin - xmin$ underflowed to zero. The lack of a trap in this case elevated the interpretation to "serious defect." The significance of this behavior is that testing for equality of x and y before division by $x-y$ can lead to division by zero.

PARANOIA summarizes its tests by declaring the arithmetic to be "satisfactory though flawed" in all cases except the single-precision fast case where the judgment is "may be acceptable despite inconvenient defects."

3. ELEFUNT/INTFUNT Results

ELEFUNT is the suite of transportable Fortran test programs from the *Software Manual for the Elementary Functions* by Cody and Waite [1980], and INTFUNT is an emerging suite of test programs extending the ELEFUNT concepts to tests of intrinsic functions. Each of the test programs exercises one or more of the elementary or intrinsic functions to estimate accuracy, check simple mathematical properties, and assess the response to improper or unusual arguments. The requirement that the test programs be portable has limited the approach in accuracy checking to determining how well the function program tested satisfies certain well-behaved identities.

The INTFUNT tests interpret results without reporting specific statistics. Table 3 summarizes single- and double-precision results for INTFUNT tests of AINT, ANINT, INT, and MOD. As with PARANOIA, the fast and default libraries gave identical results most of the time; the only disagreements between the two libraries were in the double-precision results for ANINT. The major problem uncovered is that the single-precision and fast double-precision versions of AINT, ANINT, and MOD cannot return values exceeding 2^{31} in magnitude. This strongly suggests that the algorithms used in these programs convert some intermediate quantity to an integer format, even though the value returned is to be in floating point.

Table 3. INTFUNT Test Results

‡ denotes results with the fast library

Test		Single-Precision Result	Double-Precision Result
AINT			
aint(*x*) *vs* 0, $x = 2^i$, $i = -1$,*minexp*		Okay	Okay
aint(1+*x*) *vs* 1, $x = 2^i$, $i = -1$,*minexp*		Okay	Okay
aint(*x*+1/2) *vs* *x*, $x = 2^i$, $i = 1$,*max*(35,*t*+3)		Bad for $i>31$	Okay
Parity check		Okay	Okay
$2^t - 1.0$		Okay	$2^{53}-2^{32}$
aint(±*xmax*)		Returns $\pm 2^{31}$	Okay
ANINT			
anint(*x*), $x = 2^i$, $i = -1$,*minexp*		Okay	Okay
anint(*x*+1/2) *vs* 1, $x = 2^i$, $i = -1$,*minexp*		Okay	Okay
	‡	Okay	All in error
anint(*x*+1/2) *vs* *x*+1, $x = 2^i$, $i = 1$,*max*(*t*+3,35)		Bad for $i>31$	Okay
	‡	Bad for $i>31$	All in error
Parity check		Okay	Okay
anint($2^t - 1.0$)		2^{24}	2^{53}
	‡	2^{24}	$2^{31}-1$
anint(*xmax*)		Returns 2^{31}	Okay
	‡	Returns 2^{31}	Returns $2^{31}-1$
anint(−*xmax*)		Returns -2^{31}	Okay
	‡	Returns -2^{31}	Returns -2^{31}
INT			
int(*x*) *vs* 0, $x = 2^{-i}$, $i = 1,126$		Okay	Okay
int(1+*x*) *vs* 1, $x = 2^{-i}$, $i = 1,126$		Okay	Okay
int(*x*+1/2) *vs* *x*, $x = 2^i$, $i = 1,30$		Okay	Okay
Parity check		Okay	Okay
$2^{24} - 1.0$, $n = min(t, 31)$		Okay	Okay
int(*xmax*)		Returns $2^{31}-1$	--
int(−*xmax*)		Returns -2^{31}	--
MOD			
mod(*n*×*x*+*half*,*x*), *x* and *n* random in (0,1000)		All bits correct	All bits correct
mod(*x*+½,1.0), $x = 2^i$, $i = 1$,*max*(*t*+3,35)		Bad for $i>31$	Okay
Parity check		Okay	Okay
mod(1.0,0.0)		1.0	NaN

We look at results for each of the programs tested in more detail. The single-precision versions of AINT seem to perform correctly for small arguments, but return exactly 2^{31} for arguments greater than 2^{31}. Results for the default double-precision AINT look good except for the curious return of $2^{53}-2^{32}$ for the argument $2^{53}-1$. Further investigation shows that this function returns only the most significant 31 bits whenever the argument x satisfies $2^{52} \le x < 2^{53}$. Results seem to be correct for arguments greater than 2^{53} in magnitude or less than 2^{52} in magnitude.

Tests of the ANINT routines uncovered more problems. The single-precision routines have inherited all of the problems of AINT, and have introduced an additional error. Note that ANINT($2^{24}-1$) incorrectly returns 2^{24}. We speculate that the algorithm used does a floating-point add of 1/2 to the argument and then invokes AINT. If this is the case, then $(2^{24}-1) + 1/2$ returns a result halfway between two floating-point numbers, and the IEEE round-to-even rule causes the result to be rounded up to 2^{24}. Thus an incorrect argument is generated for AINT. This hypothesis has been tested with additional arguments without contradicting the hypothesis.

The analogous algorithm appears to have been used for the default double-precision version of ANINT. All of the problems uncovered in the double-precision AINT are found here: the rounding bug shows up with the argument $2^{53}-1$, and ad hoc tests motivated by the findings with AINT reveal the same problems as before with arguments between 2^{52} and 2^{53}.

The results for the fast double-precision ANINT are even more disturbing. The program appears to return AINT instead of ANINT, except that it returns $2^{31}-1$ (where did the -1 come from?) for large arguments.

The INT routines appear to perform properly. We prefer an error indicator of some sort when the floating-point argument exceeds the largest representable integer, but that is a minor quibble compared with the problems in AINT and ANINT.

Results for the MOD functions are again curious. We did not detect a difference between the libraries, but did uncover a major difference between the single- and double-precision versions. The double-precision routines look to be perfect; the single-precision ones return bad results for large arguments and, unbelievably, return 1.0 for MOD(1.0,0.0). The double-precision programs return the IEEE NaN in this latter case, which is the preferred result.

In contrast with the INTFUNT tests, ELEFUNT tests report statistics without interpretation. A typical accuracy test from the ELEFUNT suite evaluates an identity using 2000 random arguments uniformly distributed across an interval, and reports the number of times the identity was exactly satisfied, the number of times it was not satisfied on the high side and on the low side, the maximum relative error (MRE) encountered, and the root-mean-square (RMS) relative error. To normalize results, the MRE and RMS errors are reported as an estimated number of erroneous trailing base-β digits in the significand. In general, MRE values between 1.0 and 2.0 are common with ELEFUNT on binary machines; values over 2.5 are rare and often indicate trouble.

Table 4 summarizes results for the ELEFUNT tests. We would not expect the MRE (measured in our way) for a single-precision program to exceed that for the corresponding double-precision program, and it rarely does on the Ardent Titan. Indeed, because most elementary function programs are written in C on UNIX systems and C normally does all floating-point computation in at least double precision, we would expect many of the single-precision functions to be accurate to within rounding error.

Most of the test programs also check for preservation of parity, for small argument approximations, for behavior near the boundaries of the function domain, and for response to illegal or ill-advised arguments. In all cases, proper parity was preserved, and the small argument approximations held. The following detailed discussion for each function includes appropriate comments on behavior near the boundaries and on error responses.

Table 4. ELEFUNT Test Results

‡ denotes results with the fast library

Test	Interval		Precision	Exact	MRE	RMS
ASIN						
asin (*x*) *vs Taylor Series*	(-1/8, 1/8)		Single Prec.	1998	0.10	0.00
			Double Prec.	1243	1.71	0.00
	(3/4, 1)		Single Prec.	1668	1.00	0.00
		‡	Single Prec.	1133	1.23	0.00
			Double Prec.	1370	1.23	0.00
ACOS						
acos (*x*) *vs Taylor Series*	(-1/8, 1/8)		Single Prec.	1963	0.46	0.00
			Double Prec.	1349	1.33	0.00
	(3/4, 1)		Single Prec.	1531	0.99	0.00
		‡	Single Prec.	606	1.98	0.63
			Double Prec.	1040	1.98	0.17
	(-1, -3/4)		Single Prec.	1818	0.73	0.00
	(-1, -3/4)	‡	Single Prec.	1679	0.73	0.00
			Double Prec.	1586	0.73	0.00
ATAN						
atan (*x*) *vs Taylor Series*	(-1/16, 1/16)		Single Prec.	2000	0.00	0.00
			Double Prec.	1859	0.94	0.00
$atan(x)$ *vs* $atan(1/16)+atan\left[\frac{(x-1/16)}{(1+x/16)}\right]$	$(1/16, 2-\sqrt{3})$		Single Prec.	1393	1.00	0.00
			Double Prec.	1364	1.00	0.00
$2\,atan(x)$ *vs* $atan\left[\frac{2x}{(1-x^2)}\right]$	$(2-\sqrt{3}, \sqrt{2}-1)$		Single Prec.	1455	0.93	0.00
			Double Prec.	1443	1.61	0.00
	$(\sqrt{2}-1, 1)$		Single Prec.	1752	1.00	0.00
			Double Prec.	1707	1.00	0.00

The single-precision ASIN/ACOS functions are accurate almost to within rounding error. In all cases except the final test for ACOS, the reported MRE for the double-precision version of these functions is larger than we normally see on binary machines, although not so large as to be alarming. These routines return a NaN for arguments greater than 1.0 in magnitude.

Table 4. ELEFUNT Test Results (Continued)

‡ denotes results with the fast library

Test	Interval		Precision	Exact	MRE	RMS
EXP						
$exp(x-1/16)$ *vs* $\frac{exp(x)}{exp(1/16)}$	$(1/16-ln(2)/2, ln(2)/2)$		Single Prec.	1486	1.00	0.00
			Double Prec.	1445	1.00	0.00
		‡	Double Prec.	802	2.06	0.49
$exp(x-45/16)$ *vs* $\frac{exp(x)}{exp(45/16)}$	$(-5\ ln(2), ln[2^{58} xmin])$		Single Prec.	1514	1.00	0.00
			Double Prec.	1441	1.00	0.00
		‡	Double Prec.	656	9.01	5.21
	$(10\ ln(2), ln[.9\ xmax])$		Single Prec.	1509	1.00	0.00
			Double Prec.	1451	1.00	0.00
		‡	Double Prec.	642	9.01	5.15
LOG						
$ln(x)$ *vs Taylor Series*	$(1-\varepsilon, 1+\varepsilon)$		Single Prec.	2000	0.00	0.00
			Double Prec.	2000	0.00	0.00
		‡	Double Prec.	1445	1.00	0.00
$ln(x)$ *vs* $ln(17x/16)-ln(17/16)$	$(1/\sqrt{2}, 15/16)$		Single Prec.	1434	1.00	0.00
			Double Prec.	1512	1.00	0.00
		‡	Double Prec.	551	3.67	1.56
$ln(x \times x)$ *vs* $2\ ln(x)$	$(16, 240)$		Single Prec.	1959	0.96	0.00
			Double Prec.	1930	0.94	0.00
		‡	Double Prec.	1088	1.96	0.16
LOG10						
$log(x)$ *vs* $log(11x/10)-log(11/10)$	$(1/\sqrt{10}, .9)$		Single Prec.	922	2.13	0.38
		‡	Single Prec.	911	2.13	0.39
			Double Prec.	781	2.57	0.53
		‡	Double Prec.	635	3.44	1.08

Our results indicate that the ATAN functions are good. The only fault we could find was that ATAN(0.0,0.0) returns 0.0 for all of the various versions. To our mind this should return NaN.

Table 4. ELEFUNT Test Results (Continued)

‡ denotes results with the fast library

Test	Interval		Precision	Exact	MRE	RMS
POWER						
x *vs* x^1	(1/2, 1)		Single Prec.	2000	0.00	0.00
			Double Prec.	2000	0.00	0.00
		‡	Double Prec.	1070	1.99	0.02
$(x \times x)^{1.5}$ *vs* $(x \times x) \times x$	(1/2, 1)		Single Prec.	2000	0.00	0.00
		‡	Single Prec.	1986	0.98	0.00
			Double Prec.	1815	0.90	0.00
		‡	Double Prec.	622	3.23	1.08
	$(1, xmax^{1/3})$		Single Prec.	2000	0.00	0.00
		‡	Single Prec.	1985	0.95	0.00
			Double Prec.	1783	0.96	0.00
		‡	Double Prec.	1	∞	∞
x^y *vs* $(x \times x)^{y/2}$	X: (1/10, 10), Y: $(\frac{ln[xmin]}{ln[100]}, \frac{-ln[xmin]}{ln[100]})$		Single Prec.	2000	0.00	0.00
			Double Prec.	1438	2.19	0.00
		‡	Double Prec.	1178	10.58	7.41

Both default EXP functions and the fast single-precision EXP function are also good, but the accuracy of the fast double-precision program is unacceptable. The MRE and RMS statistics are much too large. All of these functions return 0.0 for large negative arguments and INF (the IEEE infinity) for large positive arguments. Supplemental testing reveals that the fast double-precision function returns INF too soon, however, i.e., for arguments greater than 1023.5 * *ln*(2) instead of 1024 * *ln*(2) (this is the source of the unusual values for MRE and RMS in the third test).

The default and fast single-precision LOG functions look very good, although the errors in LOG10 are larger than expected. Test results for the fast double-precision routine are poor, especially those for the second test. These functions return -INF for a zero argument, and NaN (-NaN in the case of the default double-precision routine) for negative arguments.

Results for the default POWER and the fast single-precision POWER functions (the Fortran ** operator) look good, but those for the fast double-precision routine are not acceptable. Because X **Y = INF whenever Y ln(X) > 1023.5 *ln*(2), we suspect that this latter routine explicitly uses the fast EXP and LOG routines. The magnitude of the MRE figures is consistent with this conjecture. Error returns for the default power functions are designed to permit continued computation whenever possible.

Table 4. ELEFUNT Test Results (Continued)

‡ denotes results with the fast library

Test	Interval		Precision	Exact	MRE	RMS
SIN						
$sin(x)$ *vs* $3sin(x/3)-4sin(x/3)^3$	$(0, \pi/2)$		Single Prec.	1243	1.32	0.00
		‡	Single Prec.	1250	1.32	0.00
			Double Prec.	1268	1.03	0.00
		‡	Double Prec.	1120	1.82	0.00
	$(6\pi, 6.5\pi)$		Single Prec.	1221	1.17	0.00
		‡	Single Prec.	1215	1.25	0.00
			Double Prec.	1214	1.40	0.00
		‡	Double Prec.	1103	1.95	0.00
COS						
$cos(x)$ *vs* $4cos(x/3)^3-3cos(x/3)$	$(7\pi, 7.5\pi)$		Single Prec.	1250	1.32	0.00
		‡	Single Prec.	1243	1.15	0.00
			Double Prec.	1269	1.42	0.00
		‡	Double Prec.	1086	1.73	0.00
SINH						
$sinh(x)$ *vs Taylor Series*	$(0, 1/2)$		Single Prec.	1979	0.98	0.00
			Double Prec.	1323	1.15	0.00
$sinh(x)$ *vs* $\frac{[sinh(x+1)+sinh(x-1)]}{2cosh(1)}$	$(3, ln(xmax)-1/2)$		Single Prec.	956	1.57	0.15
			Double Prec.	943	1.98	0.17
COSH						
$cosh(x)$ *vs Taylor Series*	$(0, 1/2)$		Single Prec.	1967	0.96	0.00
			Double Prec.	1918	0.99	0.00
		‡	Double Prec.	1873	0.99	0.00
$cosh(x)$ *vs* $\frac{[cosh(x+1)+cosh(x-1)]}{2cosh(1)}$	$(3, ln(xmax)-1/2)$		Single Prec.	960	1.61	0.15
			Double Prec.	998	1.63	0.14
		‡	Double Prec.	798	8.83	4.31

For example, a negative argument raised to a floating-point integer value is evaluated instead of returning an error of some sort. Thus, $(-2.0)^{2.0} = 4.0$ (but the fast routines inexplicably return 0.0 for this). In

Table 4. ELEFUNT Test Results (Continued)

‡ denotes results with the fast library

Test	Interval		Precision	Exact	MRE	RMS
SQRT						
x *vs* $\sqrt{x \times x}$	$(1/\sqrt{2}, 1)$		Single Prec.	2000	0.00	0.00
		‡	Single Prec.	849	1.50	0.08
			Double Prec.	1712	0.50	0.00
		‡	Double Prec.	1707	0.50	0.00
	$(1, \sqrt{2})$		Single Prec.	2000	0.00	0.00
		‡	Single Prec.	452	2.00	0.00
			Double Prec.	1869	1.00	0.00
		‡	Double Prec.	1853	1.00	0.00
TAN						
$tan(x)$ *vs* $\frac{2tan(x/2)}{[1-tan(x/2)^2]}$	$(0, \pi/4)$		Single Prec.	1075	1.74	0.02
			Double Prec.	1071	2.02	0.06
	$(7\pi/8, 9\pi/8)$		Single Prec.	1309	1.50	0.00
			Double Prec.	1077	1.87	0.03
	$(6\pi, 6.25\pi)$		Single Prec.	1076	1.84	0.03
			Double Prec.	1066	1.90	0.05
TANH						
$tanh(x)$ *vs* $\frac{[tanh(x-1/8)+tanh(1/8)]}{[1+tanh(x+1/8)tanh(1/8)]}$	$(1/8, ln[3]/2)$		Single Prec.	1076	1.50	0.00
			Double Prec.	776	2.22	0.44
	$(1/8+ln[3]/2, 59\ ln[2]/2)$		Single Prec.	924	1.13	0.00
			Double Prec.	947	1.68	0.00

addition, $0.0^{0.0} = 1.0$ (the fast routines again return 0.0), a response popular with many numerical analysts but not with us; we prefer that a NaN be returned. In any case, the responses from the fast routines are misleading and unacceptable.

Results for the SIN, COS, SINH, and COSH routines are generally quite good. Only the reported errors for the second test of the fast double-precision COSH are unacceptably large. The magnitude of the MRE in this case is consistent with the large errors reported for the fast double-precision EXP routine, but then why is the fast double-precision SINH routine so accurate over the same range of arguments?

The SIN and COS routines attempted to compute function values for arguments in which at least half of the precision is necessarily lost during argument reduction. We personally prefer that a warning or error return be given in such situations. The SINH and COSH routines return INF for sufficiently large arguments. we did not attempt to determine the thresholds for this result.

The rounding problem detected by PARANOIA for some versions of the SQRT function is reflected by the corresponding values of MRE in Table 4. Note in particular the superb figures for the default single-precision version of SQRT. All of the SQRT functions returned zero for the argument -0.0, and returned NaNs for true negative arguments.

The tabulated results for TAN and TANH all look good. A standard auxiliary test disclosed that the default double-precision TAN function lost 10 significant bits for the argument 11.0. This would suggest a problem with the argument reduction scheme, but that problem is not apparent in other test results. As with the SIN and COS routines, TAN returns a computed value with no warning for arguments that should lose half their significance during argument reduction. There is no cotangent function, but cotangent is not one of the functions required by the Fortran standard.

4. Summary

With the exception of the double-precision MOD function, we are dismayed by the quality of the intrinsic functions AINT, ANINT, and MOD. These programs must be corrected.

In contrast, the overall quality of the default library of elementary functions is quite good. Some of the responses for unusual arguments should be cleaned up, or at least reexamined. We believe the value of ATAN(0.0,0.0) should be NaN, for example, and that the SIN, COS, and TAN routines ought to give a warning when arguments become so large that computed results are meaningless. There is also a problem with the accuracy of the double-precision SQRT, and possibly a rounding problem in LOG10. Correcting these relatively minor things would improve the library.

The fast libraries, especially the double-precision library, are bad. The motivation for having such a library is clear, but speed at the expense of accuracy and of consistency with the default library in responses for special arguments seems foolish to us. The main accuracy problem here seems to be in the double-precision EXP function. We suspect that improving the accuracy of that one function will improve many of the others to the point where the overall accuracy of the library will be acceptable. We also urge that the responses for special arguments be made consistent with those for the default library.

References

ANSI [1978]. *American National Standard Programming Language FORTRAN.* ANSI X3.9-1978. New York: American National Standards Institute, Inc.

W. J. Cody [1986a]. *An Alternative Library under 4.2 BSD UNIX on a VAX 11/780.* Argonne National Laboratory Report ANL-86-10.

W. J. Cody [1986b]. *ELEFUNT Test Results under X1.4 on the Encore Multimax.* Technical Memorandum ANL/MCS-TM-68, Argonne National Laboratory.

W. J. Cody [1986c]. *ELEFUNT Test Results under FX/FORTRAN Version 1.0 on the Alliant FX/8.* Technical Memorandum ANL/MCS-TM-78, Argonne National Laboratory.

W. J. Cody [1986d]. *ELEFUNT Test Results under NS32000 Fortran V2.5.3 on the Sequent* Balance. Technical Memorandum ANL/MCS-TM-80, Argonne National Laboratory.

W. J. Cody [1988a]. "Algorithm 665. MACHAR: A subroutine to dynamically determine machine parameters." *ACM Trans. on Math. Soft. 14*, pp. 303-311.

W. J. Cody [1988b]. *ELEFUNT Test Results under FORTRAN-PLUS on the Active Memory Technology DAP 510-8.* Technical Memorandum ANL/MCS-TM-125, Argonne National Laboratory.

W. J. Cody and W. Waite [1980]. *Software Manual for the Elementary Functions.* Englewood Cliffs, N.J.: Prentice-Hall.

IEEE [1985]. *IEEE Standard for Binary Floating-Point Numbers.* ANSI/IEEE Std 754-1985. New York, IEEE.

R. Karpinski [1985]. "PARANOIA: A floating-point benchmark." *BYTE* 10, no. 2.