Accurate Square Root Computation

Nelson H. F. Beebe
Center for Scientific Computing
Department of Mathematics
University of Utah
Salt Lake City, UT 84112
USA

Tel: +1 801 581 5254
FAX: +1 801 581 4148

Internet: beebe@math.utah.edu

09 March 1994
Version 1.05

Contents

Abstract

These notes for introductory scientific programming courses describe an implementation of the Cody-Waite algorithm for accurate computation of square roots, the simplest instance of the Fortran elementary functions.

1 Introduction

The classic books for the computation of elementary functions, such as those defined in the Fortran and C languages, are those by Cody and Waite [2] for basic algorithms, and Hart *et al* [3] for polynomial approximations.

Accurate computation of elementary functions is a non-trivial task, witnessed by the dismal performance of the run-time libraries of many languages on many systems. Instances have been observed where 2/3 of the bits in the answer have been wrong.

The pioneering work of Cody and Waite set down accurate algorithms for floating-point and fixed-point machines, in both binary and decimal arithmetic, and in addition, provided a comprehensive test package, known as ELEFUNT, that can test the quality of an implementation of the Fortran run-time libraries. The author and a student assistant, Ken Stoner, a few years ago prepared a C translation of the ELEFUNT package which has been used to make the same tests on numerous C run-time libraries.

We must observe from the beginning that there is a limit to the accuracy that can be expected from any function. Consider evaluating $y = F(x)$. From elementary calculus, $dy = F'(x)dx$, or

$$\frac{dy}{y} = x\frac{F'(x)}{F(x)}\frac{dx}{x}$$

That is, the relative error in the argument x, which is $\frac{dx}{x}$, produces a relative error in the function result, $\frac{dy}{y}$, which is the product of the argument relative error, and the magnification factor $x\frac{F'(x)}{F(x)}$.

For the case of the square root function to be considered here, the magnification factor is $1/2$. The square root function is therefore quite insensitive to errors in its argument.

For the exponential function, **exp**(x), the magnification factor is x, or equivalently, $\frac{dy}{y} = dx$. This says that the relative error in the function is the *absolute* error in the argument. The only way to achieve a small relative error in the function value is therefore to decrease the absolute error in the argument, which can only be done by increasing the number of bits used to represent it.

2 The plan of attack

In most cases, the computation of the elementary functions is based upon these steps:

- Consider the function argument, x, in floating-point form, with a base (or radix) B, exponent e, and a fraction, f, such that $1/B \le f < 1$.

Then we have $x = \pm f \times B^e$. The number of bits in the exponent and fraction, and the value of the base, depends on the particular floating-point arithmetic system chosen.

For the now-common IEEE 754 arithmetic, available on most modern workstations and personal computers, the base is 2. In 32-bit single precision, there are 8 exponent bits, and 24 fraction bits (23 stored plus 1 implicit to the left of the binary point). In 64-bit double precision, there are 11 exponent bits, and 53 fraction bits (52 stored plus 1 implicit to the left of the binary point).

The IEEE 754 standard also defines an 80-bit temporary real format which the Intel floating-point chips implement; most others do not. This format has a 15-bit exponent, 63 fraction bits, and 1 explicit bit to the left of the binary point.

I have only encountered one Fortran compiler (Digital Research's f77 on the IBM PC) that made this format accessible, but that compiler was otherwise fatally flawed and unusable.

The 1989 ANSI C standard [1] permits a *long double* type which could be used to provide access to the IEEE temporary real format, or to 128-bit floating-point types supported on IBM and DEC VAX architectures. We may hope that future C compilers will provide access to this longer floating-point type when the hardware supports it. Fortran compilers for those architectures already provide support for it.

- Use properties of the elementary function to range reduce the argument x to a small fixed interval. For example, with trigonometric functions, which have period π, this means expressing the argument as $x = n\pi \pm r$, so that e.g. $\sin x = \pm \sin r$, where r is in $0 \le r < \pi/2$.

- Use a small polynomial approximation [3] to produce an initial estimate, y_0, of the function on the small interval. Such an estimate may be good to perhaps 5 to 10 bits.

- Apply Newton iteration to refine the result. This takes the form $y_k = y_{k-1}/2 + (f/2)/y_{k-1}$. In base 2, the divisions by two can be done by exponent adjustments in floating-point computation, or by bit shifting in fixed-point computation.

  Convergence of the Newton method is quadratic, so the number of correct bits doubles with each iteration. Thus, a starting point correct to 7 bits will produce iterates accurate to 14, 28, 56, ... bits. Since the number of iterations is very small, and known in advance, the loop is written as straight-line code.

- Having computed the function value for the range-reduced argument, make whatever adjustments are necessary to produce the function value for the original argument; this step may involve a sign adjustment, and possibly a single multiplication and/or addition.

While these steps are not difficult to understand, the programming difficulty is to carry them out rapidly, and without undue accuracy loss. The argument reduction step is particularly tricky, and is a common source of substantial accuracy loss for the trigonometric functions.

Fortunately, Cody and Waite [2] have handled most of these details for us, and written down precise algorithms for the computation of each of the Fortran elementary functions.

Their book was written before IEEE 754 arithmetic implementations became widely available, so we have to take additional care to deal with unique features of IEEE arithmetic, including infinity, not-a-number (NaN), and gradual underflow (denormalized numbers).

3 Some preliminaries

We noted above that the algorithm is described in terms of the base, exponent, and fraction of floating-point numbers. We will therefore require auxiliary functions to fetch and store these parts of a floating-point number. We will also require a function for adjusting exponents.

Because these operations require knowledge of the base, and the precise storage patterns of the sign, exponent, and fraction, they depend on the host floating-point architecture, and they also require bit-extraction primitives. They are therefore definitely machine-*dependent*.

Once they have been written for a given machine, much of the rest of the code can be made portable across machines that have similar floating-point architectures, even if the byte order is different; for example, the Intel IEEE architecture used in the IBM PC stores bytes in 'little-Endian' order, while most other IEEE systems (Convex, Motorola, MIPS, SPARC, ...) store bytes in 'big-Endian' order. It therefore would be desirable for these primitives to be a standard part of all programming languages, but alas, they are not.

All computer architectures have machine-level instructions for bit manipulation. Unfortunately, bit extraction facilities are not a standard part of Fortran, although many Fortran implementations provide (non-standard) functions for that purpose. The C language fares much better; it has a rich set of bit manipulation operators. Thus, the primitives we shall describe may not be easily implementable in Fortran on some machines. We shall give both C and Fortran versions suitable for the Sun 3 (Motorola) and Sun 4 (SPARC), but *not* the Sun 386i (Intel) systems, because of the byte-order difference noted above.

Cody and Waite postulate ten primitives that are required to implement the algorithms for the Fortran elementary functions. For the square root algorithm, only three of these primitives are needed (the wording is taken directly from [2, pp. 9–10]):

adx(x,n) Augment the integer exponent in the floating-point representation of x by n, thus scaling x by the n-th power of the radix. The result is returned as a floating-point function value. For example, **adx**(1.0,2) returns the value 4.0 on binary machines, because $4.0 = 1.0 \times 2^2$. This operation is valid only when x is non-zero, and the result neither overflows nor underflows.

intxp(x) Return as a function value the integer representation of the exponent in the normalized representation of the floating-point number x. For example, **intxp**(3.0) returns the value 2 on binary machines, because $3.0 = (0.75) \times 2^2$, and the fraction 0.75 lies between 0.5 and 1.0. This operation is valid only when x is non-zero.

setxp(x,n) Return as a function value the floating-point representation of a number whose significand (i.e. fraction) is the significand of the floating-point number x, and whose exponent is the integer n. For example, **setxp**(1.0,3) returns the value 4.0 on binary machines because $1.0 = (0.5) \times 2^1$ and $4.0 = (0.5) \times 2^3$. This operation is valid only when x is non-zero and the result neither overflows nor underflows.

Note that each of these primitives has an argument validity restriction. In particular, these functions *cannot* be used with floating-point arguments that are denormalized, Infinity, or NaN. Cody and Waite use them only with valid arguments, but in a more general environment, the functions would have to check their arguments for validity.

Many UNIX Fortran compilers, including Sun's, generate external symbols from Fortran names by appending a trailing underscore to the name, spelling it in lower-case. Not all do so; the Stardent Fortran compiler generates Fortran external names spelled in upper-case, without an extra underscore.

Therefore, to make Fortran-callable C versions on Sun OS, we require knowledge of the argument passing methods of C and Fortran, and how function values are returned by these languages, and we need to name the C function with a trailing underscore. Fortunately, all UNIX systems use similar argument passing conventions for all languages, because almost the entire operating system, compilers, and run-time libraries are written in C, and on Sun OS, function values are also returned the same way by both C and Fortran.

In the remainder of this section, we present C and Fortran versions of the three primitives needed for the square root computation. We give versions in both languages, because some Fortran implementations lack the necessary bit primitives needed to implement these functions, in which case a C implementation will be necessary.

If you are not interested in the details of how these functions are implemented, skip now to the next section.

Until ANSI C [1], the C language automatically promoted function arguments and return values from single precision to double precision. This was an economization introduced early into the C language so that the run-time library had to be implemented only in double precision. At the time, floating-point use in C was expected to be minimal, since it was developed to write the UNIX operating system and support utilities.

Consequently, for the single-precision C functions (data type `float` instead of `double`), it is necessary to compile them with an ANSI-level compiler. On Sun systems, this is presently available only in the Free Software Foundation's compiler, `gcc`, and that compiler must be used instead of the native `cc`.

Here is the function for incrementing the exponent field:

```
float
adx_(x,n)
float *x;
int *n;
{
    union
    {
        float f;
        int i;
    } w;
    int old_e;

    w.f = *x;
    old_e = (w.i >> 23) & 0xff;          /* extract old exponent */
    old_e = (old_e + *n) & 0xff; /* increment old exponent */
    w.i &= ~0x7f800000;                  /* clear exponent field */
    w.i |= old_e << 23;                  /* set new exponent */
    return (w.f);
}
```

For readers unfamiliar with C, a brief explanation is in order. Fortran passes arguments by *address*; the C notation `float *x` says that `x` is an address of a single-precision number. The C `union` structure is like the Fortran `EQUIVALENCE` statement; it overlays two or more variables in the same memory locations. C hexadecimal constants are written with a leading `0x`. The C << binary operator left-shifts the left operand by the number of bits specified by its right operand, and >> right shifts. The C & operator is a bit-wise *and*, so `expr & 0xff` clears all but the lower eight bits of the integer value `expr`. The C += assignment operator adds the right-hand side to the left-hand side. Finally, the &= and |= operators *and* and *or* the right-hand side into the left-hand side.

In other words, this function assigns x to $w.f$, extracts the exponent, adds the exponent increment n, clears the exponent field, shifts the new exponent into place, and *or*s it in, doing all of this in integer operations. The result that remains in storage is the floating-point number, `w.f`, which is returned as a function value.

Given the Sun Fortran bit primitives, **and**(), **lshift**(), **or**(), and **rshift**(), a Fortran translation of this code is straightforward. Note that we do not declare the types of the bit primitives, because they are intrinsic functions in Sun Fortran that are expanded by the compiler into efficient in-line code.

Fortunately, the Sun Fortran compiler supports hexadecimal constants wherever integer constants can be used, making the code more readable than it would be if the masks were coded in decimal. It also permits initialization of floating-point numbers by hexadecimal constants. Most other compilers are more restrictive, making Fortran versions for them much more tedious to prepare.

```
      real function  adx(x,n)
*     (Adjust exponent of x)
*     [14-Nov-1990]
      real x, wf
      integer n, olde, wi

*     Force storage overlay so we can twiddle bits
      equivalence (wf, wi)

      wf = x

*     Extract old exponent
      olde = and(rshift(wi,23),z'ff')

*     Increment old exponent
      olde = and(olde + n,z'ff')

*     Zero the exponent field
      wi = and(wi,z'807fffff')

*     Or in the new exponent
      wi = or(wi,lshift(olde,23))

      adx = wf

      end
```

Here is the function for extracting the exponent:

```
int
intxp_(x)
int *x;                 /* really, float *x  */
```

```
{
    return (((x[0] >> 23) & 0xff) - 126);
}
```

This one is quite simple. To avoid the need for a `union` declaration, we lie about the argument, telling C that we received an integer, even though it was really a floating-point number. We right-shift it 23 bits to move the exponent to the right end of the word, *and* it with a mask to clear all but the 8 exponent bits, subtract the exponent bias, and return the result.

The Fortran version is again straightforward:

```
      integer function  intxp(x)
*     (Return unbiased exponent of x)
*     [14-Nov-1990]
      real x, wf
      integer wi

*     Force storage overlay so we can twiddle bits
      equivalence (wf, wi)

      wf = x

*     Extract the exponent field and unbias
      intxp = and(rshift(wi,23),z'ff') - 126

      end
```

Finally, here is the function to set an explicit exponent on a number with a fraction, f, in the range $0.5 \le f < 1.0$:

```
float
setxp_(x,n)
float *x;
int *n;
{
    union
    {
        float f;
        int i;
    } w;

    w.f = *x;
    w.i &= ~0x7f800000;                  /* clear exponent field */
    w.i |= ((126 + *n) & 0xff) << 23;    /* set new exponent */
    return (w.f);
}

```

This function first clears out the exponent bits by *and*ing with the one's complement of the mask `0x7f800000`, then *or*'s in the left-shifted biased exponent, n, to form the new floating-point number, `w.f`, which is returned as a function value.

The Fortran version is:

```
      real function  setxp (x,n)
*     (Set exponent of x)
*     [14-Nov-1990]
      real x, wf
      integer n, wi

*     Force storage overlay so we can twiddle bits
      equivalence (wf, wi)

      wf = x

*     Zero exponent field
      wi = and(wi,z'807fffff')

*     Or in the new exponent field
      wi = or(wi,lshift(and(126 + n,z'ff'),23))

      setxp = wf

      end
```

4 Denormalized numbers

Denormalized numbers pose a special problem. Implementations of IEEE 754 arithmetic are free to handle the Standard entirely in hardware, entirely in software, or in a combination of the two. Because denormalized numbers lack the hidden bit, they require special handling in either hardware or software.

For performance reasons, some vendors have not supported denormalized numbers at all (Convex and Stardent), or require a software trap handler (DEC Alpha; MIPS R3000, R4000, R4400, and R6000; Hewlett-Packard PA-RISC; and Sun SPARC before Version 8.0 (most models except LX, SX, and 10)).

On the DEC Alpha with OSF 1.x, Fortran does not permit denormalized, NaN, or Infinity values at all; execution is irrevocably terminated if they are encountered. The DEC Alpha C language implementation does not have this flaw.

The DEC Alpha architecture carries a severe performance penalty for the handling of exceptional values. Normally, several successive floating-point instructions are in execution at once, and if an exception occurs in one of them, it is impossible to precisely identify the faulting instruction. To avoid this imprecision and allow handling of denormalized numbers, NaN, and Infinity via a software trap handler, the code must be compiled with special *trap barrier* instructions before every floating-point instruction, to ensure that no traps are outstanding; performance then plummets. The Alpha architects argue that this sacrifice in the handling of exceptional values is necessary to be able to produce chip implementations that run at very high clock rates (up to 200MHz in current models). It will require several years of experience before the correctness of that decision can be evaluated.

Denormalized numbers are handled in hardware with no performance penalty on IBM RS/6000 POWER systems, and little performance penalty (from 5% to 25%) on Sun SPARC Version 8.0 or later. Motorola 68xxx (Apple Macintosh) and Intel x86 (IBM PC) floating-point processors do all arithmetic in 80-bit extended precision; in such a case, denormalized values can be renormalized in hardware because of the wider extended-precision exponent range, and so should cause no significant performance penalty.

Timing tests on those systems that require software trap handlers for denormalized arithmetic show a serious performance hit of a factor of 25 to 50, because the trap handling involves two context switches between user and kernel processes, and software implementation of the arithmetic operation.

Since the Cody-Waite primitives cannot be used with denormalized operands, we have to handle such operands specially. What we will do is to test for a denormalized argument of **sqrt**(), and when one is found, scale the argument by an even power of the base to produce a properly-normalized value, and then in the final range adjustment, undo that scaling.

The need for an explicit test on every argument raises a performance issue. Comparison with the largest denormalized number will cause a software trap on some architectures. On those, we should therefore carry out the test with an external function that examines bits in the floating-point representation; that is slower than an in-line comparison, but still much faster than a software trap. On architectures that can handle denormalized values quickly, we can do a fast in-line comparison.

Here are single-precision and double-precision Fortran functions for testing for denormalized values on big-endian IEEE 754 arithmetic systems (IBM RS/6000, HP, MIPS (including DECstation), Motorola, and Sun). The double-precision version requires a slight modification to work on little-endian systems (Intel (IBM PC) and DEC Alpha): exchange `wi(1)` and `wi(2)` in the last executable statement. The **and**() and **rshift**() bit primitives are Sun Fortran extensions; they will likely need to be changed on other architectures.

```
      logical function  isden(x)
*     (is denormalized?)
*     Return .TRUE. if x is denormalized, and
*     .FALSE. otherwise, without causing a
*     floating-point trap.
*     (09-Mar-1994)
      real x,wf
      integer wi
      equivalence (wf, wi)

      wf = x

*     Denorm has minimum exponent, and non-zero
*     fraction
      isden = (and(rshift(wi,23),255) .eq. 0)
     x      .and. (and(wi,8388607) .ne. 0)

      end
```

```
      logical function  disden(x)
*     (is denormalized?)
*     Return .TRUE. if x is denormalized, and
*     .FALSE. otherwise, without causing a
*     floating-point trap.
*     (08-Mar-1994)
      double precision x,wf
      integer wi(2)
      equivalence (wf, wi(1))

      wf = x

*     Denorm has minimum exponent, and non-zero
*     fraction
      disden = (and(rshift(wi(1),20),2047) .eq. 0)
     x     .and.
     x     ((and(wi(1),1048575) .ne. 0) .or.
     x      (wi(2) .ne. 0))

      end
```

Decimal, rather than hexadecimal, representations of the mask values are used to reduce the number of places where the code might have to be changed for a different compiler.

5 Infinity and NaN

As with denormalized values, Infinity and NaN require a software trap handler on some architectures, and on such systems, we need to resort to external functions that can safely test for these values without causing a trap.

The bit patterns for Infinity and NaN have the largest exponent, while the fraction is zero for Infinity, and non-zero for NaN. Thus, the code for the test functions is very similar.

```
      logical function  isinf(x)
*     (is Infinity?)
*     Return .TRUE. if x is Infinity, and
*     .FALSE. otherwise, without causing a
*     floating-point trap.
*     (09-Mar-1994)
      real x,wf
      integer wi
      equivalence (wf, wi)

      wf = x

*     Inf has maximum exponent, and zero fraction
      isinf = (and(rshift(wi,23),255) .eq. 255)
     x     .and. (and(wi,8388607) .eq. 0)

      end
```

```
      logical function  isnan(x)
*     (is NaN?)
*     Return .TRUE. if x is a NaN, and
*     .FALSE. otherwise, without causing a
*     floating-point trap.
*     (09-Mar-1994)
      real x,wf
      integer wi
      equivalence (wf, wi)

      wf = x

*     NaN has maximum exponent, and non-zero fraction
      isnan = (and(rshift(wi,23),255) .eq. 255)
     x     .and. (and(wi,8388607) .ne. 0)

      end
```

The double-precision versions are similar, and as with **disden**(), on little-endian architectures, the code must be rewritten to exchange `wi(1)` and `wi(2)` in the last executable statement.

```
      logical function disinf(x)
*     (is Infinity?)
*     Return .TRUE. if x is Infinity, and
*     .FALSE. otherwise, without causing a
*     floating-point trap.
*     (09-Mar-1994)
      double precision x,wf
      integer wi(2)
      equivalence (wf, wi(1))

      wf = x

*     Inf has maximum exponent, and zero fraction
      disinf = (and(rshift(wi(1),20),2047) .eq. 2047)
     x     .and. (and(wi(1),1048575) .eq. 0) .and.
     x     (wi(2) .eq. 0)

      end
```

```
      logical function  disnan(x)
*     (is NaN?)
*     Return .TRUE. if x is NaN, and .FALSE. otherwise,
*     without causing a floating-point trap.
*     (09-Mar-1994)
      double precision x,wf
      integer wi(2)
      equivalence (wf, wi(1))

      wf = x

*     NaN has maximum exponent, and non-zero fraction
      disnan = (and(rshift(wi(1),20),2047) .eq. 2047)
     x     .and. ((and(wi(1),1048575) .ne. 0) .or.
     x           (wi(2) .ne. 0))

      end
```

6 The square root algorithm

We are now ready to demonstrate the algorithm for computing the square root [2, pp. 17–34]. Assume that $x \geq 0$, and recall that $x = f \times B^e$, $1/B \leq f < 1$, and for IEEE arithmetic, $B = 2$.

We decompose the floating-point number into its parts by

$$e = \textbf{intxp}(x) \quad (1)$$

$$f = \textbf{setxp}(x, 0) \quad (2)$$

$$B = 2 \quad (3)$$

Then we have

$$\begin{aligned}\sqrt{x} &= \sqrt{f} \times B^{e/2} \qquad (e \text{ is even}) && (4)\\ &= (\sqrt{f}/\sqrt{B}) \times B^{(e+1)/2} \qquad (e \text{ is odd}) && (5)\end{aligned}$$

These two results are the range reduction; if we compute $\sqrt{f}$, we can get $\sqrt{x}$ from it by a single multiplication, and possibly, a division. We do not require any exponentiations, because powers of the base can be constructed more rapidly with the **setxp**() function.

A starting estimate of the square root value can be obtained from [3]:

$$y_0 = 0.41731 + 0.59016 \times f \qquad (6)$$

With this approximation, Newton iterations give the following numbers of correct bits:

Iteration Number			
0	1	2	3
7.04	15.08	31.16	63.32

Thus, in single precision, two iterations suffice to generate a result correct to more than the 24 bits that we actually store in single precision. Three iterations are enough for IEEE double precision.

The Newton iteration formula for base $B = 2$ is

$$y_k = 0.5 \times (y_{k-1} + f/y_{k-1}) \qquad (7)$$

On a machine with slow multiplication, we could do the multiplication by 0.5 by calling **adx**(y_k,-1). However, on modern RISC systems, like the Sun SPARC, a floating-point multiply takes only few machine cycles, and is faster than a function call. From the table above, we only need to compute up to $k = 2$. If we use in-line code, this gives

$$y_1 = 0.5 \times (y_0 + f/y_0)$$

and

$$y_2 = 0.5 \times (y_1 + f/y_1)$$

We can save one multiply as follows:

$$\begin{aligned} z &= (y_0 + f/y_0)\\ y_1 &= 0.5 \times z\\ y_2 &= 0.5 \times (0.5 \times z + f/(0.5 \times z))\\ &= 0.25 \times z + f/z \end{aligned}$$

For the case of odd e, we need to divide by $\sqrt{B} = \sqrt{2}$. Since division is often much slower than multiplication, we can instead multiply by $\sqrt{0.5}$, which we can code in-line as a constant. This constant can be written in floating-point if the compiler's conversion of floating-point numbers to internal form is sufficiently accurate (this is the case with Sun's compilers), and otherwise, we must use an octal or hexadecimal representation so as to get every bit correct. This constant is

$$\begin{aligned}\sqrt{0.5} &= 0.70710\ 67811\ 86547\ 52440\ldots \quad \text{(decimal)}\\ &= 0.55202\ 36314\ 77473\ 63110\ldots \quad \text{(octal)}\\ &= 0.B504F\ 333F9\ DE648\ 4597E\ldots \quad \text{(hexadecimal)}\end{aligned}$$

Hint: you can compute these values on any UNIX system using the bc arbitrary-precision calculator. Here, we ask for accuracy of 30 decimal digits, and evaluate $1/\sqrt{2}$ in output bases 8, 10, and 16:

```
% bc
scale=30;
ob=8;
sqrt(0.5);
.552023631477473631102131373046633

ob=10;
sqrt(0.5);
.707106781186547524400844362104

ob=16;
sqrt(0.5);
.B504F333F9DE6484597D89B3
```

Having obtained the approximation to $\sqrt{f}$ as y_2, we could then use **adx**(y_2,M) to set the correct exponent. In the code below, we actually use **setxp**() since we know the exponent is already zero. This avoids the need for **adx**() altogether in this implementation of the **sqrt**() function.

Here then is the Fortran code for our single-precision square root routine:

```
      real function sqrt(x)
*     Cody-Waite implementation of sqrt(x)
*     [08-Mar-1994]

      integer and, e, intxp, nbias
      real f, setxp, x, xx, y, z
      logical isden, isinf, isnan

      real denmax
      real onemme
      real Inf

*     denmax = maximum positive denormalized number
      data denmax /z'007fffff'/

*     onemme = 1.0 - machine epsilon
      data onemme /z'3f7fffff'/

*     Inf = +infinity
      data Inf    /z'7f800000'/

*******************************************************
**    Use external isxxx() implementations if
**    exceptional values are handled (slowly) in
**    software, e.g. MIPS (R3000, R4000, R4400), HP
**    PA-RISC (1.0, 1.1), DEC Alpha, Sun SPARC
**    (pre-version 8: most models).
**    Use in-line statement functions if exceptional
**    values are handled in hardware, e.g. IBM RS/6000
**    (POWER), Sun SPARC (version 8 or later: LX, SX,
**    and 10/xxx)
*******************************************************
      isden(xx) = (xx .le. denmax)
      isinf(xx) = (xx .ge. Inf)
      isnan(xx) = (xx .ne. xx)

*     Generate a NaN for x <= 0.0, and Inf for x == Inf
*     and handle the special case of x == 0.0 so we do
*     not violate the assumptions that the arguments to
*     setxp() and intxp() are non-zero.

      if (x .eq. 0.0) then
           sqrt = 0.0
           return
      else if (x .lt. 0.0) then
           sqrt = 0.0
           sqrt = sqrt/sqrt
           return
      else if (isinf(x)) then
           sqrt = 0.0
           sqrt = 1.0/sqrt
           return
      else if (isnan(x)) then
           sqrt = 0.0
           sqrt = sqrt/sqrt
           return
      else if (isden(x)) then
*          scale x by 2**24 (this is exact)
           xx = x * 16 777 216.0
           nbias = -24
      else
           xx = x
           nbias = 0
      end if

      e = intxp(xx)
      f = setxp(xx,0)

*     y0 to 7.04 bits
      y = 0.41731 + 0.59016 * f

*     y1 to 15.08 bits
      z = y + f/y

*     y2 to 31.16 bits
      y = 0.25*z + f/z

*     Include sqrt(2) factor for odd exponents, and
*     ensure (0.5 <= y) and (y < 1.0).  Otherwise,
*     setxp() will give wrong answer.

      if (and(e,1) .ne. 0) then
         y = y * 0.70710 67811 86547 52440 08443 62104 E+00
         y = max(y,0.5)
         e = e + 1
      end if
      y = min(y,onemme)

*     Insert exponent to undo range reduction.
      sqrt = setxp(y,(e + nbias)/2)

      end
```

There are several very subtle points in this code that must be discussed:

- Because we are going to twiddle bits using the **intxp**() and **setxp**() functions, we must ensure that the arguments to those functions match their design requirements: they must be normalized, non-zero, and the result may not overflow or underflow, or generate a NaN. This means that we have to test for the special cases of $x < 0$, $x = 0$, $x = \text{NaN}$, $x = \infty$, and $x =$ *denormalized* since these would otherwise result later in invalid arguments to **intxp**() and **setxp**().

- The tests themselves must not generate a run-time trap from generation of denormalized, NaN, or infinity values. This means that we have to store the constants `Inf` and `denmax` in the program with hexadecimal initializations. We do not need to store a NaN, because the expression `(x .ne. x)` tests for it.

- The fast in-line arithmetic statement functions must be commented out on those architectures that trap on exceptional values so that the external functions presented earlier can do the tests safely. Good optimizing compilers will expand arithmetic statement functions in-line, so there should not be a significant penalty for their use. For some compilers, it may be desirable to replace the **isxxx**() statement function invocations with explicit comparisons.

- If the function argument is a NaN, or is less than zero, the **sqrt**() function is not defined on the real axis, so we must return a NaN. We must not return a precomputed one however, because that would not generate a trap that the user might want to handle. Instead, we generate one directly by evaluating the expression 0.0/0.0. The Sun compiler computes this at run time if the function is compiled without optimization, but with optimization, reduces it to a compile-time constant. We therefore have to generate it from a run-time expression, `sqrt/sqrt`, and verify that the compiler does not optimize this too by checking the assembly code generated (the compile-time option `-S` produces that output).

- If the function argument is infinity, then we generate an infinity for the result, being careful to code the value in such a way that a run-time division, and trap, will occur.

- A denormalized argument is rescaled into the range of normalized numbers, and the scale factor is remembered for later use in the range restoration.

- Testing for oddness of e can be done portably by

  ```
  if (mod(e,2) .ne. 0) then
  ```

  but the Hewlett-Packard PA-RISC and Sun SPARC architecture before version 8.0 have neither integer multiply, nor divide, nor modulus, instructions, so those operations have to be done by an expensive subroutine call. The **and**() function does a simple bit test, and is expanded in-line by the compiler. Testing the low-order bit to determine oddness is valid only in sign-magnitude and two's complement arithmetic systems; in a one's complement system, a number is odd if its low-order bit differs from its sign bit.

The integer divisions by 2 are handled by a fast shift instruction on the Sun SPARC, and by a fast bit-extraction instruction on the Hewlett-Packard PA-RISC.

- Because blanks are not significant in Fortran, we are permitted to embed blanks in constants to make them more readable, and I have done so for two of them.
- In the last few lines, we are ready to scale the result, $y = \sqrt{f}$ to recover the answer for $\sqrt{x}$. However, because finite arithmetic has been used to compute y, it is possible that the computed value lies slightly out of the range $0.5 \le y < 1.0$ required for the **setxp**() function. The **min**() and **max**() functions take care of moving y to the nearest endpoint if it is outside that interval, and to do so, we need to make use of the stored exact constant, `onemme`, which is the largest floating-point number below 1.0.

The double-precision version is a simple extension of this code; we only need to change the function names, data types, and constant initializations, and then add one more Newton iteration to get the answer to sufficient accuracy:

```
      double precision function  dsqrt(x)
*     Cody-Waite implementation of dsqrt(x)
*     [08-Mar-1994]

      integer and, e, dintxp, nbias
      double precision dsetxp, f, xx, x, y, z
      logical disden, disinf, disnan

      double precision denmax
      double precision onemme
      double precision Inf

*     denmax = maximum positive denormalized number
      data denmax /z'000fffffffffffff'/

*     onemme = 1.0 - machine epsilon
      data onemme /z'3fefffffffffffff'/

*     Inf = +infinity
      data Inf    /z'7ff0000000000000'/

*******************************************************
**    Use external disxxx() implementations if
**    exceptional values are handled (slowly) in
**    software, e.g. MIPS (R3000, R4000, R4400), HP
**    PA-RISC (1.0, 1.1), DEC Alpha, Sun SPARC
**    (pre-version 8: most models).
**    Use in-line statement functions if exceptional
**    values are handled in hardware, e.g. IBM RS/6000
**    (POWER), Sun SPARC (version 8 or later: LX, SX,
**    and 10/xxx)
*******************************************************
      disden(xx) = (xx .le. denmax)
      disinf(xx) = (xx .ge. Inf)
      disnan(xx) = (xx .ne. xx)

*     Generate a NaN for x <= 0.0, and Inf for x == Inf
*     and handle the special case of x == 0.0 so we do
*     not violate the assumptions that the arguments to
*     setxp() and intxp() are non-zero.

      if (x .eq. 0.0D+00) then
           dsqrt = 0.0D+00
           return
      else if (x .lt. 0.0D+00) then
           dsqrt = 0.0D+00
           dsqrt = dsqrt/dsqrt
           return
      else if (disinf(x)) then
           dsqrt = 0.0D+00
           dsqrt = 1.0D+00/dsqrt
           return
      else if (disnan(x)) then
           dsqrt = 0.0D+00
           dsqrt = dsqrt/dsqrt
           return
      else if (disden(x)) then
*          scale x by 2**54 (this is exact)
           xx = x * 18 014 398 509 481 984.0D+00
           nbias = -54
      else
           xx = x
           nbias = 0
      end if

      e = dintxp(xx)
      f = dsetxp(xx,0)

*     y0 to 7.04 bits
      y = 0.41731D+00 + 0.59016D+00 * f

*     y1 to 15.08 bits
```

```
      z = y + f/y

*     y2 to 31.16 bits
      y = 0.25D+00*z + f/z

*     y3 to 63.32 bits
      y = 0.5D+00*(y + f/y)

*     Include sqrt(2) factor for odd exponents, and
*     ensure (0.5 <= y) and (y < 1.0).  Otherwise,
*     dsetxp() will give wrong answer.

      if (and(e,1) .ne. 0) then
         y = y * 0.70710 67811 86547 52440 08443 62104 D+00
         y = max(y, 0.5D+00)
         e = e + 1
      end if
      y = min(y,onemme)

*     Insert exponent to undo range reduction.
      dsqrt = dsetxp(y,(e + nbias)/2)

      end
```

The Cody-Waite primitives needed in **dsqrt**() are straightforward extensions of the single-precision ones, but the exponent bias, bit masks, and shift counts are different:

```
double
adx_(x,n)
double *x;
int *n;
{
    union
    {
        double f;
        int i[2];
    } w;
    int old_e;

    w.f = *x;
    old_e = (w.i[0] >> 20) & 0x7ff;      /* extract old exponent */
    old_e = (old_e + *n) & 0x7ff;        /* increment old exponent */
    w.i[0] &= ~0x7ff00000;               /* clear exponent field */
    w.i[0] |= (old_e << 20);             /* set new exponent */
    return (w.f);
}
```

```
      double precision function  dadx(x,n)
*     (Increment exponent of x)
*     [14-Nov-1990]
      double precision wf, x
      integer n, olde, wi(2)

*     Force storage overlay so we can twiddle bits
      equivalence (wf, wi(1))

      wf = x

*     Extract old exponent
      olde = and(rshift(wi(1),20),z'7ff')

*     Increment old exponent
      olde = and(olde + n,x'7ff')

*     Zero exponent field
      wi(1) = and(wi(1),z'800fffff')

*     Or in the new exponent field
      wi(1) = or(wi(1),lshift(olde,20))

      dadx = wf

      end
```

```
int
dintxp_(x)
int *x;                 /* really, double *x  */
{
    return (((x[0] >> 20) & 0x7ff) - 1022);
}
```

```
      integer function  dintxp(x)
*     (Return unbiased exponent of x)
*     [14-Nov-1990]
      double precision wf, x
      integer wi(2)

*     Force storage overlay so we can twiddle bits
      equivalence (wf, wi(1))

      wf = x

*     Extract the exponent field and unbias
      dintxp = and(rshift(wi(1),20),z'7ff') - 1022

      end
```

```
double
dsetxp_(x,n)
double *x;
int *n;
{
    union
    {
        double f;
        int i[2];
    } w;

    w.f = *x;
    w.i[0] &= ~0x7ff00000;              /* clear exponent field */
    w.i[0] |= ((1022 + *n) & 0x7ff) << 20; /* set new exponent */
    return (w.f);
}
```

```
      double precision function  dsetxp (x,n)
*     (Set exponent of x)
*     [14-Nov-1990]
      double precision wf, x
      integer n, wi(2)

*     Force storage overlay so we can twiddle bits
      equivalence (wf, wi(1))

      wf = x

*     Zero exponent field
      wi(1) = and(wi(1),z'800fffff')

*     Or in the new exponent field
      wi(1) = or(wi(1),lshift(and(1022 + n,z'7ff'),20))

      dsetxp = wf

      end
```

7 Testing the square root function

The mathematical square root function is monotonic with respect to its argument; is our implementation also monotonic? This turns out to be quite difficult to test for.

We observe that square root maps a given argument interval $0 \ldots x$ onto $0 \ldots$ **sqrt**(x); for $x > 1$, that interval is shorter than the argument interval. In other words, there will be a many-to-one mapping of argument points into function points; in general, two immediately adjacent arguments will have the same square root.

You might wonder, could we just step through all floating-point numbers, generating the square root of each one, and then test the accuracy of that result?

Timing a loop that calls our single-precision square root function with random arguments in the range (0.0,1.0) shows that on a Sun SPARCstation 10/51 (the fastest SPARC processor in 1993), it takes an average of 2.67μsec per call; Sun's built-in function averages 2.17μsec per call. With an older SPARCstation 1 system, our function averages 26.7μsec per call, and Sun's, 11.0μsec per call. With older versions of the Sun Fortran compiler and library, our function was faster.

In IEEE 32-bit arithmetic, there are 254 exponent values (the exponent 255 is reserved for infinity and NaN, and the exponent 0 for zero), and 2^{23} fraction values; this gives a total of approximately 2.13×10^9 different 32-bit floating-point numbers. On the SPARC 10/51, calling the square root function for each of them would take about 1.6 hr for our version, and 1.3 hr for Sun's. Thus, it would be quite feasible to test a single-precision elementary function with every possible single-precision argument.

In double precision, there are 2046 exponent values (0 and 2047 are reserved), and 2^{53} fractional values, giving a total of about 1.84×10^{19} values. On the SPARC 10/51, the Sun **dsqrt**() function averages about 2.5μsec per call; it would then take about 1.5 million years to evaluate all possible square roots.

Testing by enumeration of function values for all possible arguments is therefore only feasible in single precision, and even then would be tedious except on the fastest processors. Clearly, some sort of argument sampling will be necessary in general.

In the ELEFUNT package, Cody and Waite developed tests that examine certain equalities that should be obeyed by particular functions. These tests examine the behavior of 2000 random arguments in selected intervals. The tests also check the behavior of the functions when presented with arguments near the floating-point limits, or with ones that would generate results near those limits.

How does our implementation of the square root function compare with the built-in one on Sun SPARC systems? The answer can be found by running the ELEFUNT tests on both functions.

The following results are extracted from the output listings of the square test program using first the native Sun Fortran **sqrt**(), and then ours. The listing files are each about 85 lines long, so we show first the listing for our implementation, and then we show the test headers and differences between Sun's and ours.

```
------------------------------------------------------------------------
TEST OF SQRT(X*X) - X

  2000 RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL
     (     0.7071E+00,     0.1000E+01)

SQRT(X) WAS LARGER    86 TIMES,
            AGREED  1841 TIMES, AND
       WAS SMALLER    73 TIMES.

THERE ARE  24 BASE   2 SIGNIFICANT DIGITS IN A FLOATING-POINT NUMBER

THE MAXIMUM RELATIVE ERROR OF     0.8422E-07 =    2 ** -23.50
   OCCURRED FOR X =     0.707740E+00
THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.50

THE ROOT MEAN SQUARE RELATIVE ERROR WAS     0.2124E-07 =    2 ** -25.49
THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.00
------------------------------------------------------------------------
TEST OF SQRT(X*X) - X

  2000 RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL
     (     0.1000E+01,     0.1414E+01)

SQRT(X) WAS LARGER     9 TIMES,
            AGREED  1684 TIMES, AND
       WAS SMALLER   307 TIMES.

THERE ARE  24 BASE   2 SIGNIFICANT DIGITS IN A FLOATING-POINT NUMBER

THE MAXIMUM RELATIVE ERROR OF     0.1192E-06 =    2 ** -23.00
   OCCURRED FOR X =     0.100022E+01
THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   1.00

THE ROOT MEAN SQUARE RELATIVE ERROR WAS     0.3990E-07 =    2 ** -24.58
THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.00
------------------------------------------------------------------------
TEST OF SPECIAL ARGUMENTS

SQRT(XMIN) = SQRT(  0.1401298E-44) =   0.3743392E-22
SQRT(1-EPSNEG) = SQRT(1-  0.5960464E-07) =   0.9999999E+00

SQRT(1.0) = SQRT(  0.1000000E+01) =   0.1000000E+01

SQRT(1+EPS) = SQRT(1+  0.1192093E-06) =   0.1000000E+01

SQRT(XMAX) = SQRT(Inf            ) = Inf

------------------------------------------------------------------------
TEST OF ERROR RETURNS

SQRT WILL BE CALLED WITH THE ARGUMENT     0.0000E+00
THIS SHOULD NOT TRIGGER AN ERROR MESSAGE

SQRT RETURNED THE VALUE     0.0000E+00

SQRT WILL BE CALLED WITH THE ARGUMENT    -0.1000E+01
THIS SHOULD TRIGGER AN ERROR MESSAGE

SQRT RETURNED THE VALUENaN

THIS CONCLUDES THE TESTS
------------------------------------------------------------------------
```

Here is a comparison between the two implementations:

```
------------------------------------------------------------------------
TEST OF SQRT(X*X) - X

   2000 RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL
      (     0.7071E+00,     0.1000E+01)

Sun: SQRT(X) WAS LARGER     0 TIMES,
Sun:             AGREED  2000 TIMES, AND
Sun:        WAS SMALLER     0 TIMES.

Us:   SQRT(X) WAS LARGER    86 TIMES,
Us:               AGREED  1841 TIMES, AND
Us:          WAS SMALLER    73 TIMES.

Sun: THE MAXIMUM RELATIVE ERROR OF     0.0000E+00 =    2 **-999.00
Sun:    OCCURRED FOR X =     0.000000E+00
Sun:  THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.00

Us:  THE MAXIMUM RELATIVE ERROR OF     0.8422E-07 =    2 ** -23.50
Us:     OCCURRED FOR X =     0.707740E+00
Us:  THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.50
```

```
Sun: THE ROOT MEAN SQUARE RELATIVE ERROR WAS 0.0000E+00 = 2 **-999.00

Us:  THE ROOT MEAN SQUARE RELATIVE ERROR WAS 0.2124E-07 = 2 ** -25.49

------------------------------------------------------------------------
TEST OF SQRT(X*X) - X

   2000 RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL
      (     0.1000E+01,     0.1414E+01)

Sun: SQRT(X) WAS LARGER     0 TIMES,
Sun:             AGREED  2000 TIMES, AND
Sun:        WAS SMALLER     0 TIMES.

Us:  SQRT(X) WAS LARGER     9 TIMES,
Us:              AGREED  1684 TIMES, AND
Us:         WAS SMALLER   307 TIMES.

Sun: THE MAXIMUM RELATIVE ERROR OF     0.0000E+00 =    2 **-999.00
Sun:    OCCURRED FOR X =     0.000000E+00
Sun: THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.00

Us:  THE MAXIMUM RELATIVE ERROR OF     0.1192E-06 =    2 ** -23.00
Us:     OCCURRED FOR X =     0.100022E+01
Us:  THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   1.00

Sun: THE ROOT MEAN SQUARE RELATIVE ERROR WAS 0.0000E+00 = 2 **-999.00
Sun: THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.00

Us:  THE ROOT MEAN SQUARE RELATIVE ERROR WAS 0.3987E-07 = 2 ** -24.58
Us:  THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.00
------------------------------------------------------------------------
```

Our implementation of **sqrt**() is reasonably good. It has a worst case error of only 1 bit in the intervals tested.

Why is Sun's implementation better? On most modern RISC machines, the fastest floating-point arithmetic is double precision, not single precision, because the hardware always works in the long precision, and additional instructions are needed to convert to single precision. Thus, Sun's **sqrt**() function is very likely written in double precision, with the final result rounded to single. This should then give completely correct results over the entire range of arguments. An internal double-precision implementation has the added benefit that no explicit test for denormalized numbers is required, since single-precision denormalized values can be represented in double-precision normalized form.

To test this prediction, I prepared a modified version of the single-precision square root routine that used the algorithm for **dsqrt**(), then converted the function result to single precision before returning. The ELEFUNT test results for it are now identical with those for Sun's implementation of **sqrt**().

Finally, we show the differences in the ELEFUNT test output between Sun's **dsqrt**() and ours:

```
-----------------------------------------------------------------------
TEST OF DSQRT(X*X) - X

   2000 RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL
      (     0.7071D+00,     0.1000D+01)

Sun:  DSQRT(X) WAS LARGER     0 TIMES,
Sun:               AGREED  2000 TIMES, AND
Sun:          WAS SMALLER     0 TIMES.

Us:   DSQRT(X) WAS LARGER    73 TIMES,
Us:                AGREED  1852 TIMES, AND
Us:           WAS SMALLER    75 TIMES.

Sun:  THE MAXIMUM RELATIVE ERROR OF     0.0000D+00 =    2 **-999.00
Sun:     OCCURRED FOR X =     0.000000D+00
Sun:  THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.00

Us:   THE MAXIMUM RELATIVE ERROR OF     0.1567D-15 =    2 ** -52.50
Us:      OCCURRED FOR X =     0.708678D+00
Us:   THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.50

-----------------------------------------------------------------------
TEST OF DSQRT(X*X) - X

   2000 RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL
      (     0.1000D+01,     0.1414D+01)

Sun:  THE ROOT MEAN SQUARE RELATIVE ERROR WAS 0.0000D+00 = 2 **-999.00

Us:   THE ROOT MEAN SQUARE RELATIVE ERROR WAS 0.3787D-16 = 2 ** -54.55

Sun:  DSQRT(X) WAS LARGER     0 TIMES,
Sun:               AGREED  2000 TIMES, AND

Us:   DSQRT(X) WAS LARGER   668 TIMES,
Us:                AGREED  1332 TIMES, AND

Sun:  THE MAXIMUM RELATIVE ERROR OF     0.0000D+00 =    2 **-999.00
Sun:     OCCURRED FOR X =     0.000000D+00
Sun:  THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   0.00

Us:   THE MAXIMUM RELATIVE ERROR OF     0.2215D-15 =    2 ** -52.00
```

```
Us:      OCCURRED FOR X =     0.100252D+01
Us:   THE ESTIMATED LOSS OF BASE   2 SIGNIFICANT DIGITS IS   1.00

Sun:  THE ROOT MEAN SQUARE RELATIVE ERROR WAS 0.0000D+00 = 2 **-999.00

Us:   THE ROOT MEAN SQUARE RELATIVE ERROR WAS 0.1082D-15 = 2 ** -53.04

------------------------------------------------------------------------
TEST OF SPECIAL ARGUMENTS

Sun:  DSQRT(1+EPS) = DSQRT(1+0.2220446D-15) = 0.10000000000000000D+01
Us:   DSQRT(1+EPS) = DSQRT(1+0.2220446D-15) = 0.10000000000000002D+01
```

As in the entirely single-precision implementation of **sqrt**(), accuracy is excellent (a maximum error of 1 bit). Note that Sun's implementation is superb; their **dsqrt**() has all bits correct.

8 Conclusions

The goal of this description has been to lead the reader through the details of the computation of the simplest elementary function, and to illustrate the care that must be taken to produce a correct, and fast, implementation in a computer program.

Achieving maximum speed means that we need to use non-standard extensions to the Fortran language, such as bit-manipulation primitives and hexadecimal constants, even though the basic algorithm remains unchanged. Such machine dependencies are regrettable, but Fortran's lack of standardization in these areas gives us no recourse. Fortunately, they need to be handled only once for each architecture, and then the code can be made available as a library routine.

It should be clear that there is a very large step from being able to write down the *mathematical* relations in equations 3, 5, 6, and 7 beginning on page 12 to the *computational algorithms* expressed in the computer programs.

The difficulty in making the transition from mathematical formulas to computer algorithms is intimately connected with the fact that in the computer, we must work with *finite precision*, and a *finite number range*, and because of this, few computations are exact.

References

[1] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language C,*

ANSI X3.159-1989, December 14 1989.

[2] William J. Cody, Jr. and William Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1980. ISBN 0-13-822064-6. x + 269 pp. LCCN QA331 .C635 1980.

[3] John F. Hart, E. W. Cheney, Charles L. Lawson, Hans J. Maehly, Charles K. Mesztenyi, John R. Rice, Henry G. Thatcher, Jr., and Christoph Witzgall. *Computer Approximations*. Robert E. Krieger Publishing Company, Huntington, NY, USA, 1968. ISBN 0-88275-642-7. x + 343 pp. LCCN QA 297 C64 1978. Reprinted 1978 with corrections.